

C++ CÓMO PROGRAMAR

– Sexta edición –

El CD incluye
Microsoft® Visual C++®
Express Edition



Introducción a la
Programación de juegos
y las **Bibliotecas Boost**



PEARSON
Prentice Hall

DEITEL®

P. J. DEITEL
H. M. DEITEL





Harvey M. Deitel

Deitel & Associates, Inc.

Paul J. Deitel

Deitel & Associates, Inc.

TRADUCCIÓN

Alfonso Vidal Romero Elizondo

Ingeniero en Electrónica y Comunicación

Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Monterrey

REVISIÓN TÉCNICA

Jesús Muñoz Bauza

Departamento de Computación

División de Ingeniería y Arquitectura

Tecnológico de Monterrey

Campus Ciudad de México

Sergio Fuenlabrada Velázquez

Oskar A. Gómez C.

Edna M. Miranda Ch.

Mario Oviedo G.

Adalberto Robles V.

Mario A. Sesma M.

Departamento de Computación

Unidad Profesional Interdisciplinaria de Ingeniería

y Ciencias Sociales y Administrativas

Instituto Politécnico Nacional



Datos de catalogación bibliográfica

DEITEL, HARVEY M. Y PAUL J. DEITEL

Cómo programar en C++. Sexta edición

PEARSON EDUCACIÓN, México 2008

ISBN: 978-970-26-1273-5

Área: Computación

Formato: 20 × 25.5 cm

Páginas: 1112

Authorized translation from the English language edition, entitled *C++ How to program, 6th edition*, by Deitel & Associates (Harvey & Paul) published by Pearson Education, Inc., publishing as Prentice Hall, Copyright © 2008. All rights reserved.

ISBN 0136152503

Traducción autorizada de la edición en idioma inglés titulada *C++ How to program, 6^a edición*, por Deitel & Associates (Harvey & Paul), publicada por Pearson Education, Inc., publicada como Prentice Hall, Copyright © 2008. Todos los derechos reservados.

Esta edición en español es la única autorizada.

Edición en español

Editor: Luis Miguel Cruz Castillo
e-mail: luis.cruz@pearsoned.com
Editor de desarrollo: Bernardino Gutiérrez Hernández
Supervisor de producción: Juan José García Guzmán

Edición en inglés

Vice President and Editorial Director, ECS: Marcia J. Horton

Associate Editor: Carole Snyder

Supervisor/Editorial Assistant: Dolores Mars

Director of Team-Based Project Management: Vince O'Brien

Senior Managing Editor: Scott Disanno

Managing Editor: Bob Engelhardt

Production Editor: Marta Samsel

A/V Production Editor: Greg Dulles

Art Studio: Artworks, York, PA

Art Director: Kristine Carney

Cover Design: Abbey S. Deitel, Harvey M. Deitel, Francesco Santalucia, Kristine Carney

Interior Design: Harvey M. Deitel, Kristine Carney

Manufacturing Manager: Alexis Heydt-Long

Manufacturing Buyer: Lisa McDowell

Director of Marketing: Margaret Waples

SEXTA EDICIÓN, 2008

D.R. © 2008 por Pearson Educación de México, S.A. de C.V.

Atlacomulco 500-5o. piso

Col. Industrial Atoto

53519, Naucalpan de Juárez, Estado de México

Cámara Nacional de la Industria Editorial Mexicana. Reg. núm. 1031.

Prentice Hall es una marca registrada de Pearson Educación de México, S.A. de C.V.

Reservados todos los derechos. Ni la totalidad ni parte de esta publicación pueden reproducirse, registrarse o transmitirse, por un sistema de recuperación de información, en ninguna forma ni por ningún medio, sea electrónico, mecánico, fotoquímico, magnético o electroóptico, por fotocopia, grabación o cualquier otro, sin permiso previo por escrito del editor.

El préstamo, alquiler o cualquier otra forma de cesión de uso de este ejemplar requerirá también la autorización del editor o de sus representantes.

ISBN 10: 970-26-1273-X

ISBN 13: 978-970-26-1273-5

Impreso en México. Printed in Mexico.

1 2 3 4 5 6 7 8 9 0 - 11 10 09 08



*A las victimas de la tragedia de Virginia Tech,
sus familias, sus amigos y a toda la comunidad de
Virginia Tech.*

Paul y Harvey Deitel

Contenido

Prefacio	xviii
Antes de empezar	xxxix
I Introducción a las computadoras, Internet y World Wide Web	I
1.1 Introducción	2
1.2 ¿Qué es una computadora?	3
1.3 Organización de una computadora	4
1.4 Los primeros sistemas operativos	4
1.5 Computación personal, distribuida y cliente/servidor	5
1.6 Internet y World Wide Web	5
1.7 Web 2.0	6
1.8 Lenguajes máquina, lenguajes ensambladores y lenguajes de alto nivel	6
1.9 Historia de C y C++	7
1.10 Biblioteca estándar de C++	8
1.11 Historia de Java	9
1.12 FORTRAN, COBOL, Pascal y Ada	9
1.13 BASIC, Visual Basic, Visual C++, C# y .NET	10
1.14 Tendencia clave de software: la tecnología de los objetos	10
1.15 Entorno de desarrollo típico en C++	11
1.16 Generalidades acerca de C++ y este libro	13
1.17 Prueba de una aplicación en C++	14
1.18 Tecnologías de software	19
1.19 Programación de juegos con las bibliotecas Ogre	20
1.20 Futuro de C++: Bibliotecas Boost de código fuente abierto, TR1 y C++0x	20
1.21 Ejemplo práctico de Ingeniería de Software: introducción a la tecnología de objetos y el UML	21
1.22 Repaso	25
1.23 Recursos Web	25
2 Introducción a la programación en C++	35
2.1 Introducción	36
2.2 Su primer programa en C++: imprimir una línea de texto	36
2.3 Modificación de nuestro primer programa en C++	39
2.4 Otro programa en C++: suma de enteros	40
2.5 Conceptos acerca de la memoria	44
2.6 Aritmética	45
2.7 Toma de decisiones: operadores de igualdad y relacionales	47
2.8 (Opcional) Ejemplo práctico de Ingeniería de Software: cómo examinar la especificación de requerimientos del ATM	52
2.9 Repaso	59

3	Introducción a las clases y los objetos	67
3.1	Introducción	68
3.2	Clases, objetos, funciones miembro y miembros de datos	68
3.3	Generalidades acerca de los ejemplos del capítulo	69
3.4	Definición de una clase con una función miembro	70
3.5	Definición de una función miembro con un parámetro	72
3.6	Miembros de datos, funciones <i>establecer</i> y funciones <i>obtener</i>	75
3.7	Inicialización de objetos mediante constructores	81
3.8	Colocar una clase en un archivo separado para fines de reutilización	84
3.9	Separar la interfaz de la implementación	87
3.10	Validación de datos mediante funciones <i>establecer</i>	93
3.11	(Opcional) Ejemplo práctico de Ingeniería de Software: identificación de las clases en la especificación de requerimientos del ATM	97
3.12	Repaso	102
4	Instrucciones de control: parte 1	109
4.1	Introducción	110
4.2	Algoritmos	110
4.3	Seudocódigo	111
4.4	Estructuras de control	112
4.5	Instrucción de selección <i>if</i>	115
4.6	Instrucción de selección doble <i>if...else</i>	116
4.7	Instrucción de repetición <i>while</i>	120
4.8	Cómo formular algoritmos: repetición controlada por un contador	121
4.9	Cómo formular algoritmos: repetición controlada por un centinela	126
4.10	Cómo formular algoritmos: instrucciones de control anidadas	135
4.11	Operadores de asignación	139
4.12	Operadores de incremento y decremento	139
4.13	(Opcional) Ejemplo práctico de Ingeniería de Software: identificación de los atributos de las clases en el sistema ATM	142
4.14	Repaso	146
5	Instrucciones de control: parte 2	159
5.1	Introducción	160
5.2	Fundamentos de la repetición controlada por contador	160
5.3	Instrucción de repetición <i>for</i>	162
5.4	Ejemplos acerca del uso de la instrucción <i>for</i>	165
5.5	Instrucción de repetición <i>do...while</i>	169
5.6	Instrucción de selección múltiple <i>switch</i>	171
5.7	Instrucciones <i>break</i> y <i>continue</i>	179
5.8	Operadores lógicos	180
5.9	Confusión entre los operadores de igualdad (==) y de asignación (=)	184
5.10	Resumen sobre programación estructurada	185
5.11	(Opcional) Ejemplo práctico de Ingeniería de Software: cómo identificar los estados y actividades de los objetos en el sistema ATM	189
5.12	Repaso	193
6	Funciones y una introducción a la recursividad	203
6.1	Introducción	204
6.2	Componentes de los programas en C++	205
6.3	Funciones matemáticas de la biblioteca	206

6.4	Definiciones de funciones con varios parámetros	207
6.5	Prototipos de funciones y coerción de argumentos	211
6.6	Archivos de encabezado de la Biblioteca estándar de C++	213
6.7	Ejemplo práctico: generación de números aleatorios	215
6.8	Ejemplo práctico: juego de probabilidad, introducción a las enumeraciones	220
6.9	Clases de almacenamiento	223
6.10	Reglas de alcance	225
6.11	La pila de llamadas a funciones y los registros de activación	228
6.12	Funciones con listas de parámetros vacías	231
6.13	Funciones en línea	232
6.14	Referencias y parámetros de referencias	233
6.15	Argumentos predeterminados	237
6.16	Operador de resolución de ámbito unario	239
6.17	Sobrecarga de funciones	240
6.18	Plantillas de funciones	243
6.19	Recursividad	245
6.20	Ejemplo sobre el uso de la recursividad: serie de Fibonacci	247
6.21	Comparación entre recursividad e iteración	250
6.22	(Opcional) Ejemplo práctico de Ingeniería de Software: identificación de las operaciones de las clases en el sistema ATM	253
6.23	Reparo	258

7 Arreglos y vectores 277

7.1	Introducción	278
7.2	Arreglos	279
7.3	Declaración y creación de arreglos	280
7.4	Ejemplos acerca del uso de los arreglos	280
7.4.1	Declaración de un arreglo y uso de un ciclo para inicializar los elementos del arreglo	281
7.4.2	Inicialización de un arreglo en una declaración mediante una lista inicializadora	281
7.4.3	Especificación del tamaño de un arreglo con una variable constante y establecimiento de los elementos de un arreglo con cálculos	283
7.4.4	Suma de los elementos de un arreglo	285
7.4.5	Uso de gráficos de barra para mostrar los datos de un arreglo en forma gráfica	286
7.4.6	Uso de los elementos de un arreglo como contadores	287
7.4.7	Uso de arreglos para sintetizar los resultados de una encuesta	288
7.4.8	Uso de arreglos tipo carácter para almacenar y manipular cadenas	291
7.4.9	Arreglos locales estáticos y arreglos locales automáticos	292
7.5	Paso de arreglos a funciones	294
7.6	Ejemplo práctico: la clase <code>LibroCalificaciones</code> que usa un arreglo para almacenar las calificaciones	298
7.7	Búsqueda de datos en arreglos mediante la búsqueda lineal	304
7.8	Ordenamiento de arreglos mediante el ordenamiento por inserción	305
7.9	Arreglos multidimensionales	307
7.10	Ejemplo práctico: la clase <code>LibroCalificaciones</code> que usa un arreglo bidimensional	310
7.11	Introducción a la plantilla de clase <code>vector</code> de la Biblioteca estándar de C++	316
7.12	(Opcional) Ejemplo práctico de Ingeniería de Software: colaboración entre los objetos en el sistema ATM	320
7.13	Reparo	326

8 Apuntadores y cadenas basadas en apuntadores 341

8.1	Introducción	342
8.2	Declaraciones e inicialización de variables apuntadores	342
8.3	Operadores de apuntadores	343
8.4	Paso de argumentos a funciones por referencia mediante apuntadores	346

x	Contenido	
8.5	Uso de <code>const</code> con apuntadores	349
8.6	Ordenamiento por selección mediante el uso del paso por referencia	355
8.7	Operador <code>sizeof</code>	358
8.8	Expresiones y aritmética de apuntadores	360
8.9	Relación entre apuntadores y arreglos	362
8.10	Arreglos de apuntadores	366
8.11	Ejemplo práctico: simulación para barajar y repartir cartas	367
8.12	Apuntadores a funciones	372
8.13	Introducción al procesamiento de cadenas basadas en apuntador	376
8.14	Repaso	384

9 Clases: un análisis más detallado, parte I

9	Clases: un análisis más detallado, parte I	407
9.1	Introducción	408
9.2	Ejemplo práctico con la clase <code>Tiempo</code>	409
9.3	Alcance de las clases y acceso a los miembros de una clase	414
9.4	Separar la interfaz de la implementación	416
9.5	Funciones de acceso y funciones utilitarias	416
9.6	Ejemplo práctico de la clase <code>Tiempo</code> : constructores con argumentos predeterminados	419
9.7	Destructores	423
9.8	Cuándo se hacen llamadas a los constructores y destructores	424
9.9	Ejemplo práctico con la clase <code>Tiempo</code> : una trampa sutil (devolver una referencia a un miembro de datos <code>private</code>)	427
9.10	Asignación predeterminada a nivel de miembros	429
9.11	(Opcional) Ejemplo práctico de Ingeniería de Software: inicio de la programación de las clases del sistema ATM	431
9.12	Repaso	437

10 Clases: un análisis más detallado, parte 2

10	Clases: un análisis más detallado, parte 2	443
10.1	Introducción	444
10.2	Objetos <code>const</code> (constantes) y funciones miembro <code>const</code>	444
10.3	Composición: objetos como miembros de clases	452
10.4	Funciones <code>friend</code> y clases <code>friend</code>	458
10.5	Uso del apuntador <code>this</code>	461
10.6	Administración dinámica de memoria con los operadores <code>new</code> y <code>delete</code>	466
10.7	Miembros de clase <code>static</code>	467
10.8	Abstracción de datos y ocultamiento de información	472
10.8.1	Ejemplo: tipo de datos abstracto arreglo	473
10.8.2	Ejemplo: tipo de datos abstracto cadena	474
10.8.3	Ejemplo: tipo de datos abstracto cola	474
10.9	Clases contenedoras e iteradores	474
10.10	Clases proxy	475
10.11	Repaso	477

11 Sobrecarga de operadores: objetos String y Array

11	Sobrecarga de operadores: objetos String y Array	483
11.1	Introducción	484
11.2	Fundamentos de la sobrecarga de operadores	485
11.3	Restricciones acerca de la sobrecarga de operadores	485
11.4	Las funciones de operadores como clase miembro vs. funciones globales	487
11.5	Sobrecarga de los operadores de inserción de flujo y extracción de flujo	488
11.6	Sobrecarga de operadores unarios	491

11.7	Sobrecarga de operadores binarios	491
11.8	Ejemplo práctico: la clase <code>Array</code>	492
11.9	Conversión entre tipos	502
11.10	Ejemplo práctico: la clase <code>String</code>	502
11.11	Sobrecarga de <code>++</code> y <code>--</code>	513
11.12	Ejemplo práctico: una clase <code>Fecha</code>	514
11.13	La clase <code>string</code> de la Biblioteca estándar	518
11.14	Constructores <code>explicit</code>	521
11.15	Repaso	524

12 Programación orientada a objetos: herencia **535**

12.1	Introducción	536
12.2	Clases base y clases derivadas	537
12.3	Miembros <code>protected</code>	539
12.4	Relación entre las clases base y las clases derivadas	539
12.4.1	Creación y uso de una clase <code>EmpleadoPorComision</code>	540
12.4.2	Creación de una clase <code>EmpleadoBaseMasComision</code> sin usar la herencia	544
12.4.3	Creación de una jerarquía de herencia <code>EmpleadoPorComision-Empleado-BaseMasComision</code>	549
12.4.4	La jerarquía de herencia <code>EmpleadoPorComision-EmpleadoBaseMasComision</code> mediante el uso de datos <code>protected</code>	553
12.4.5	La jerarquía de herencia <code>EmpleadoPorComision-EmpleadoBaseMasComision</code> mediante el uso de datos <code>private</code>	559
12.5	Los constructores y destructores en las clases derivadas	566
12.6	Herencia <code>public</code> , <code>protected</code> y <code>private</code>	573
12.7	Ingeniería de software mediante la herencia	573
12.8	Repaso	574

13 Programación orientada a objetos: polimorfismo **579**

13.1	Introducción	580
13.2	Ejemplos de polimorfismo	581
13.3	Relaciones entre los objetos en una jerarquía de herencia	582
13.3.1	Invocación de funciones de la clase base desde objetos de una clase derivada	583
13.3.2	Cómo orientar los apuntadores de una clase derivada a objetos de la clase base	589
13.3.3	Llamadas a funciones miembro de una clase derivada a través de apuntadores de la clase base	590
13.3.4	Funciones virtuales	591
13.3.5	Resumen de las asignaciones permitidas entre objetos y apuntadores de la clase base y de la clase derivada	596
13.4	Tipos de campos e instrucciones <code>switch</code>	597
13.5	Clases abstractas y funciones <code>virtual</code> puras	597
13.6	Ejemplo práctico: sistema de nómina mediante el uso de polimorfismo	599
13.6.1	Creación de la clase base abstracta <code>Empleado</code>	600
13.6.2	Creación de la clase derivada concreta <code>EmpleadoAsalariado</code>	603
13.6.3	Creación de la clase derivada concreta <code>EmpleadoPorHoras</code>	605
13.6.4	Creación de la clase derivada concreta <code>EmpleadoPorComision</code>	607
13.6.5	Creación de la clase derivada concreta indirecta <code>EmpleadoBaseMasComision</code>	608
13.6.6	Demostración del procesamiento polimórfico	610
13.7	(Opcional) Polimorfismo, funciones virtuales y vinculación dinámica “detrás de las cámaras”	614
13.8	Ejemplo práctico: sistema de nómina mediante el uso de polimorfismo e información de tipos en tiempo de ejecución con conversión descendente, <code>dynamic_cast</code> , <code>typeid</code> y <code>type_info</code>	617
13.9	Destructores virtuales	620
13.10	(Opcional) Ejemplo práctico de Ingeniería de Software: incorporación de la herencia en el sistema ATM	620
13.11	Repaso	627

14 Plantillas	631
14.1 Introducción	632
14.2 Plantillas de funciones	632
14.3 Sobrecarga de plantillas de funciones	635
14.4 Plantillas de clases	636
14.5 Parámetros sin tipo y tipos predeterminados para las plantillas de clases	641
14.6 Notas acerca de las plantillas y la herencia	642
14.7 Notas acerca de las plantillas y funciones friend	642
14.8 Notas acerca de las plantillas y miembros static	643
14.9 Repaso	643
15 Entrada y salida de flujos	648
15.1 Introducción	649
15.2 Flujos	650
15.2.1 Comparación entre flujos clásicos y flujos estándar	650
15.2.2 Archivos de encabezado de la biblioteca iostream	651
15.2.3 Clases y objetos de entrada/salida de flujos	651
15.3 Salida de flujos	653
15.3.1 Salida de variables char *	653
15.3.2 Salida de caracteres mediante la función miembro put	653
15.4 Entrada de flujos	654
15.4.1 Funciones miembro get y getline	654
15.4.2 Funciones miembro peek, putback e ignore de istream	657
15.4.3 E/S con seguridad de tipos	657
15.5 E/S sin formato mediante el uso de read, write y gcount	657
15.6 Introducción a los manipuladores de flujos	658
15.6.1 Base de flujos integrales: dec, oct, hex y setbase	658
15.6.2 Precisión de punto flotante (precision, setprecision)	659
15.6.3 Anchura de campos (width, setw)	660
15.6.4 Manipuladores de flujos de salida definidos por el usuario	662
15.7 Estados de formato de flujos y manipuladores de flujos	663
15.7.1 Ceros a la derecha y puntos decimales (showpoint)	663
15.7.2 Justificación (left, right e internal)	664
15.7.3 Relleno de caracteres (fill, setfill)	665
15.7.4 Base de flujos integrales (dec, oct, hex, showbase)	667
15.7.5 Números de punto flotante: notación científica y fija (scientific, fixed)	667
15.7.6 Control de mayúsculas/minúsculas (uppercase)	668
15.7.7 Especificación de formato booleano (boolalpha)	669
15.7.8 Establecer y restablecer el estado de formato mediante la función miembro flags	670
15.8 Estados de error de los flujos	671
15.9 Enlazar un flujo de salida a un flujo de entrada	673
15.10 Repaso	673
16 Manejo de excepciones	682
16.1 Introducción	683
16.2 Generalidades acerca del manejo de excepciones	684
16.3 Ejemplo: manejo de un intento de dividir entre cero	684
16.4 Cuándo utilizar el manejo de excepciones	689
16.5 Volver a lanzar una excepción	690
16.6 Especificaciones de excepciones	691
16.7 Procesamiento de excepciones inesperadas	692
16.8 Limpieza de la pila	692
16.9 Constructores, destructores y manejo de excepciones	694

16.10	Excepciones y herencia	694
16.11	Procesamiento de las fallas de <code>new</code>	694
16.12	La clase <code>auto_ptr</code> y la asignación dinámica de memoria	698
16.13	Jerarquía de excepciones de la Biblioteca estándar	700
16.14	Otras técnicas para manejar errores	701
16.15	Repaso	702

17 Procesamiento de archivos **708**

17.1	Introducción	709
17.2	Jerarquía de datos	709
17.3	Archivos y flujos	711
17.4	Creación de un archivo secuencial	712
17.5	Cómo leer datos de un archivo secuencial	715
17.6	Actualización de archivos secuenciales	720
17.7	Archivos de acceso aleatorio	721
17.8	Creación de un archivo de acceso aleatorio	721
17.9	Cómo escribir datos al azar a un archivo de acceso aleatorio	726
17.10	Cómo leer de un archivo de acceso aleatorio en forma secuencial	728
17.11	Ejemplo práctico: un programa para procesar transacciones	730
17.12	Generalidades acerca de la serialización de objetos	735
17.13	Repaso	736

18 La clase `string` y el procesamiento de flujos de cadena **745**

18.1	Introducción	746
18.2	Asignación y concatenación de objetos <code>string</code>	747
18.3	Comparación de objetos <code>string</code>	749
18.4	Subcadenas	751
18.5	Intercambio de objetos <code>string</code>	752
18.6	Características de los objetos <code>string</code>	752
18.7	Búsqueda de subcadenas y caracteres en un objeto <code>string</code>	754
18.8	Reemplazo de caracteres en un objeto <code>string</code>	756
18.9	Inserción de caracteres en un objeto <code>string</code>	758
18.10	Conversión a cadenas estilo C	758
18.11	Iteradores	760
18.12	Procesamiento de flujos de cadena	761
18.13	Repaso	764

19 Búsqueda y ordenamiento **769**

19.1	Introducción	770
19.2	Algoritmos de búsqueda	770
19.2.1	Eficiencia de la búsqueda lineal	770
19.2.2	Búsqueda binaria	772
19.3	Algoritmos de ordenamiento	776
19.3.1	Eficiencia del ordenamiento por selección	776
19.3.2	Eficiencia del ordenamiento por inserción	777
19.3.3	Ordenamiento por combinación (una implementación recursiva)	777
19.4	Repaso	783

20 Estructuras de datos **788**

20.1	Introducción	789
20.2	Clases autorreferenciadas	790

20.3	Asignación dinámica de memoria y estructuras de datos	790
20.4	Listas enlazadas	791
20.5	Pilas	804
20.6	Colas	807
20.7	Árboles	810
20.8	Repaso	818
	Sección especial: construya su propio compilador	826

21 Bits, caracteres, cadenas estilo C y estructuras**837**

21.1	Introducción	838
21.2	Definiciones de estructuras	838
21.3	Inicialización de estructuras	840
21.4	Uso de estructuras con funciones	840
21.5	<code>typedef</code>	840
21.6	Ejemplo: simulación para barajar y repartir cartas de alto rendimiento	841
21.7	Operadores a nivel de bits	843
21.8	Campos de bits	851
21.9	Biblioteca de manejo de caracteres	854
21.10	Funciones de conversión de cadenas basadas en apuntador	859
21.11	Funciones de búsqueda de la biblioteca de manejo de cadenas basadas en apuntador	863
21.12	Funciones de memoria de la biblioteca de manejo de cadenas basadas en apuntador	867
21.13	Repaso	871

22 Biblioteca de plantillas estándar (STL)**881**

22.1	Introducción a la Biblioteca de plantillas estándar (STL)	882
22.1.1	Introducción a los contenedores	884
22.1.2	Introducción a los iteradores	887
22.1.3	Introducción a los algoritmos	892
22.2	Contenedores de secuencia	893
22.2.1	Contenedor de secuencia <code>vector</code>	894
22.2.2	Contenedor de secuencia <code>list</code>	900
22.2.3	Contenedor de secuencia <code>deque</code>	903
22.3	Contenedores asociativos	904
22.3.1	Contenedor asociativo <code>multiset</code>	904
22.3.2	Contenedor asociativo <code>set</code>	907
22.3.3	Contenedor asociativo <code>multimap</code>	908
22.3.4	Contenedor asociativo <code>map</code>	910
22.4	Adaptadores de contenedores	911
22.4.1	Adaptador <code>stack</code>	911
22.4.2	Adaptador <code>queue</code>	913
22.4.3	Adaptador <code>priority_queue</code>	914
22.5	Algoritmos	915
22.5.1	<code>fill</code> , <code>fill_n</code> , <code>generate</code> y <code>generate_n</code>	916
22.5.2	<code>equal</code> , <code>mismatch</code> y <code>lexicographical_compare</code>	917
22.5.3	<code>remove</code> , <code>remove_if</code> , <code>remove_copy</code> y <code>remove_copy_if</code>	919
22.5.4	<code>replace</code> , <code>replace_if</code> , <code>replace_copy</code> y <code>replace_copy_if</code>	921
22.5.5	Algoritmos matemáticos	923
22.5.6	Algoritmos básicos de búsqueda y ordenamiento	926
22.5.7	<code>swap</code> , <code>iter_swap</code> y <code>swap_ranges</code>	928
22.5.8	<code>copy_backward</code> , <code>merge</code> , <code>unique</code> y <code>reverse</code>	929
22.5.9	<code>inplace_merge</code> , <code>unique_copy</code> y <code>reverse_copy</code>	931
22.5.10	Operaciones <code>set</code>	933
22.5.11	<code>lower_bound</code> , <code>upper_bound</code> y <code>equal_range</code>	935
22.5.12	Ordenamiento de montón (heapsort)	937

22.5.13 <code>min</code> y <code>max</code>	939
22.5.14 Algoritmos de la STL que no se cubren en este capítulo	940
22.6 La clase <code>bitset</code>	941
22.7 Objetos de funciones	944
22.8 Conclusión	947
22.9 Recursos Web de la STL	947
23 Programación de juegos con Ogre	955
23.1 Introducción	956
23.2 Instalación de Ogre, OgreAL y OpenAL	956
23.3 Fundamentos de la programación de juegos	956
23.4 El juego de Pong: recorrido a través del código	959
23.4.1 Inicialización de Ogre	959
23.4.2 Creación de una escena	967
23.4.3 Agregar elementos a la escena	968
23.4.4 Animación y temporizadores	978
23.4.5 Entrada del usuario	979
23.4.6 Detección de colisiones	980
23.4.7 Sonido	984
23.4.8 Recursos	985
23.4.9 Controlador de Pong	985
23.5 Repaso	986
23.6 Recursos Web de Ogre	987
24 Bibliotecas Boost, Reporte técnico 1 y C++0x	995
24.1 Introducción	996
24.2 Centros de recursos de C++ (y relacionados) en línea de Deitel	996
24.3 Bibliotecas Boost	996
24.4 Cómo agregar una nueva biblioteca a Boost	997
24.5 Instalación de las Bibliotecas Boost	997
24.6 Las Bibliotecas Boost en el Reporte técnico 1 (TR1)	997
24.7 Uso de expresiones regulares con la biblioteca <code>Boost.Regex</code>	1000
24.7.1 Ejemplo de una expresión regular	1000
24.7.2 Cómo validar la entrada del usuario mediante expresiones regulares	1002
24.7.3 Cómo reemplazar y dividir cadenas	1005
24.8 Apuntadores inteligentes con <code>Boost.Smart_ptr</code>	1007
24.8.1 Uso de <code>shared_ptr</code> y conteo de referencias	1007
24.8.2 <code>weak_ptr</code> : observador de <code>shared_ptr</code>	1011
24.9 Reporte técnico 1	1016
24.10 C++0x	1017
24.11 Cambios en el lenguaje básico	1017
24.12 Repaso	1021
25 Otros temas	1028
25.1 Introducción	1029
25.2 Operador <code>const_cast</code>	1029
25.3 <code>Espacios de nombres</code>	1030
25.4 Palabras clave de operadores	1034
25.5 Miembros de clases <code>mutable</code>	1036
25.6 Apuntadores a miembros de clases (<code>.*</code> y <code>->*</code>)	1037
25.7 Herencia múltiple	1039
25.8 Herencia múltiple y clases base <code>virtual</code>	1043
25.9 Repaso	1047

A Tabla de precedencia de operadores y asociatividad	1051
A.1 Precedencia de operadores	1051
B Conjunto de caracteres ASCII	1053
C Tipos fundamentales	1054
D Sistemas numéricicos	1056
D.1 Introducción	1057
D.2 Abreviatura de los números binarios como números octales y hexadecimales	1059
D.3 Conversión de números octales y hexadecimales a binarios	1060
D.4 Conversión de un número binario, octal o hexadecimal a decimal	1061
D.5 Conversión de un número decimal a binario, octal o hexadecimal	1061
D.6 Números binarios negativos: notación de complemento a dos	1062
E Temas sobre código heredado de C	1067
E.1 Introducción	1068
E.2 Redirección de la entrada/salida en sistemas UNIX/LINUX/Mac OS X y Windows	1068
E.3 Listas de argumentos de longitud variable	1069
E.4 Uso de argumentos de línea de comandos	1071
E.5 Observaciones acerca de la compilación de programas con varios archivos de código fuente	1072
E.6 Terminación de los programas con <code>exit</code> y <code>atexit</code>	1074
E.7 Calificador de tipo <code>volatile</code>	1075
E.8 Sufijos para constantes enteras y de punto flotante	1075
E.9 Manejo de señales	1076
E.10 Asignación dinámica de memoria con <code>calloc</code> y <code>realloc</code>	1078
E.11 Bifurcación incondicional: <code>goto</code>	1078
E.12 Uniones	1080
E.13 Especificaciones de vinculación	1082
E.14 Repaso	1083
F Preprocesador	1088
F.1 Introducción	1089
F.2 La directiva del procesador <code>#include</code>	1089
F.3 La directiva del preprocesador <code>#define</code> : constantes simbólicas	1090
F.4 La directiva del preprocesador <code>#define</code> : macros	1090
F.5 Compilación condicional	1092
F.6 Las directivas del preprocesador <code>#error</code> y <code>#pragma</code>	1092
F.7 Los operadores <code>#</code> y <code>##</code>	1093
F.8 Constantes simbólicas predefinidas	1093
F.9 Aserciones	1093
F.10 Repaso	1094

G	Código del caso de estudio del ATM	1098
G.1	Implementación del caso de estudio del ATM	1098
G.2	La clase ATM	1099
G.3	La clase Pantalla	1104
G.4	La clase Teclado	1105
G.5	La clase DispensadorEfectivo	1106
G.6	La clase RanuraDeposito	1108
G.7	La clase Cuenta	1109
G.8	La clase BaseDatosBanco	1111
G.9	La clase Transaccion	1114
G.10	La clase SolicitudSaldo	1115
G.11	La clase Retiro	1117
G.12	La clase Deposito	1121
G.13	El programa de prueba EjemploPracticoATM.cpp	1124
G.14	Repaso	1124
H	UML 2: tipos de diagramas adicionales	1125
H.1	Introducción	1125
H.2	Tipos de diagramas adicionales	1125
I	Uso del depurador de Visual Studio	1127
I.1	Introducción	1128
I.2	Los puntos de interrupción y el comando Continuar	1128
I.3	Las ventanas Variables locales e Inspección	1132
I.4	Control de la ejecución mediante los comandos Paso a paso por instrucciones, Paso a paso por procedimientos, Paso a paso para salir y Continuar	1135
I.5	La ventana Automático	1137
I.6	Repaso	1138
J	Uso del depurador de GNU C++	1141
J.1	Introducción	1142
J.2	Los puntos de interrupción y los comandos run, stop, continue y print	1142
J.3	Los comandos print y set	1147
J.4	Control de la ejecución mediante los comandos step, finish y next	1149
J.5	El comando watch	1151
J.6	Repaso	1153
Bibliografía		1157
Índice		1163

Prefacio

“El principal mérito del lenguaje es la claridad...”

—Galen

¡Bienvenido a C++ y *Cómo programar en C++, sexta edición!* En Deitel & Associates escribimos libros de texto sobre lenguajes de programación y libros de nivel profesional para Prentice Hall, impartimos capacitación a empresas en todo el mundo y desarrollamos negocios en Internet con Web 2.0. Este libro refleja los cambios importantes en el lenguaje C++ y en las formas preferidas de impartir y aprender programación. Se han realizado ajustes considerables en todos los capítulos. La sección *Recorrido a través del libro* del prefacio proporciona a los instructores, estudiantes y profesionales una idea del tipo de cobertura que ofrece este texto sobre C++ y la programación orientada a objetos.

Características nuevas y mejoradas

He aquí una lista de las actualizaciones que hemos realizado a la sexta edición de *Cómo programar en C++*:

- **Programación de juegos.** Hemos agregado un nuevo capítulo sobre la programación de juegos. Los ingresos de la industria de juegos de computadora son ya mayores que los de la industria cinematográfica, con lo cual se crean muchas oportunidades para los estudiantes interesados en carreras relacionadas con la programación de juegos. El capítulo 23, Programación de juegos con Ogre, introduce la programación de juegos y los gráficos con el motor de gráficos Ogre 3D, de código fuente abierto. Hablaremos sobre las cuestiones básicas relacionadas con la programación de juegos. Después le mostraremos cómo utilizar Ogre para crear un juego simple con una mecánica de juego similar al clásico videojuego Pong®, desarrollado originalmente por Atari en 1972. Demostraremos cómo crear una escena con gráficos 3D a colores, cómo animar los objetos móviles de manera uniforme, cómo usar los temporizadores para controlar la velocidad de animación, detectar colisiones entre objetos, agregar sonido, aceptar la entrada mediante el teclado y mostrar salida de texto.
- **El futuro de C++.** Hemos agregado el capítulo 24, en el que se considera el futuro de C++; presentamos las Bibliotecas Boost de C++, el Informe técnico 1 (TR1) y C++0x. Las bibliotecas gratuitas Boost de código fuente abierto son creadas por miembros de la comunidad de C++. El Informe técnico 1 describe los cambios propuestos a la Biblioteca estándar de C++, muchos de los cuales están basados en las bibliotecas Boost actuales. El Comité de estándares de C++ está revisando el Estándar de C++. Los principales objetivos para el nuevo estándar son facilitar el aprendizaje de este lenguaje, mejorar las herramientas para construir bibliotecas e incrementar su compatibilidad con el lenguaje C. El último estándar se publicó en 1998. El trabajo sobre el nuevo estándar, que se conoce actualmente como C++0x, empezó en 2003 y es probable que se publique en 2009. Incluirá cambios al lenguaje del núcleo y, muy probablemente, a muchas de las bibliotecas en el TR1. Aquí veremos las generalidades acerca de las bibliotecas del TR1 y proporcionaremos códigos de ejemplo para las de “expresiones regulares” y “apuntadores inteligentes”. Las expresiones regulares se utilizan para relacionar patrones de caracteres específicos en el texto. Pueden usarse para validar los datos, para asegurar que se encuentren en un formato específico, para sustituir partes de una cadena con otra o para dividirla. Muchos de los errores comunes en el código de C y C++ se relacionan con los apuntadores, una poderosa herramienta de programación que estudiaremos en el capítulo 8, Apuntadores y cadenas basadas en apuntadores. Los apuntadores inteligentes nos ayudan a evitar errores al proporcionar una funcionalidad adicional a los apuntadores estándar.
- **Importantes revisiones de contenido.** Todos los capítulos se han actualizado y mejorado de manera considerable. Ajustamos la claridad y precisión de la escritura, así como el uso de la terminología de C++, de conformidad con el documento del estándar ISO/IEC de C++ que define el lenguaje.

- **Introducción temprana a las metodologías de clases y objetos.** Presentamos a los estudiantes los conceptos básicos y la terminología de la tecnología de objetos en el capítulo 1, y empezamos a desarrollar clases reutilizables personalizadas y objetos en el capítulo 3. Este libro presenta la programación orientada a objetos, según sea apropiado, desde el principio y a lo largo de este libro. El análisis temprano sobre los objetos y las clases hace que los estudiantes “piensen en objetos” de inmediato, y que dominen estos conceptos de una manera más completa. La programación orientada a objetos no es trivial de ningún modo, pero es divertido escribir programas orientados a objetos, y los estudiantes pueden ver resultados de inmediato.
- **Ejemplos prácticos integrados.** Proporcionamos varios ejemplos prácticos que abarcan varias secciones y capítulos, que con frecuencia se basan en una clase presentada en una sección anterior del libro, para demostrar los nuevos conceptos de programación que se presentan más adelante en el libro. Estos ejemplos prácticos incluyen el desarrollo de la clase *LibroCalificaciones* en los capítulos 3 a 7, la clase *Tiempo* en varias secciones de los capítulos 9 y 10, la clase *Empleado* en los capítulos 12 y 13, y el ejemplo práctico opcional de DOO/UML del ATM en los capítulos 1 a 7, 9, 13 y en el apéndice G.
- **Ejemplo práctico integrado de LibroCalificaciones.** El ejemplo práctico *LibroCalificaciones* refuerza nuestra primera presentación de las clases. Utiliza clases y objetos en los capítulos 3 a 7 para construir en forma incremental una clase llamada *LibroCalificaciones*, la cual representa el libro de calificaciones de un instructor y realiza varios cálculos con base en los resultados obtenidos por los estudiantes, como calcular la calificación promedio, buscar las calificaciones máxima y mínima e imprimir un gráfico de barras.
- **Lenguaje Unificado de Modelado™ 2 (UML 2).** El Lenguaje Unificado de Modelado (UML) se ha convertido en el lenguaje de modelado gráfico preferido por los diseñadores de sistemas orientados a objetos. Todos los diagramas de UML en el libro cumplen con la especificación de UML 2. Usamos los diagramas de clases de UML para representar en forma visual las clases y sus relaciones de herencia, y utilizamos los diagramas de actividad de UML para demostrar el flujo de control en cada una de las instrucciones de control de C++. En el ejemplo práctico de DOO/UML del ATM, hacemos un uso especialmente intensivo de UML.
- **Ejemplo práctico opcional de DOO/UML del ATM.** El ejemplo práctico opcional de DOO/UML del cajero automático (ATM) en las secciones tituladas Ejemplo práctico de Ingeniería de Software de los capítulos 1 a 7, 9 y 13, es apropiado para un primer y segundo curso de programación. Las nueve secciones del ejemplo práctico presentan una introducción cuidadosamente planeada al diseño orientado a objetos, mediante el uso de UML. Presentamos un subconjunto conciso y simplificado de UML 2, para después guiarlo a través de su primera experiencia de diseño, elaborada para el diseñador/programador orientado a objetos principiante. Nuestra meta aquí es ayudar a los estudiantes a desarrollar un diseño orientado a objetos para complementar los conceptos que sobre este tema empezarán a ver en el capítulo 1, y que implementarán en el capítulo 3. El ejemplo práctico fue revisado por un distinguido equipo de profesores y profesionales de la industria relacionados con el DOO/UML. El ejemplo práctico no es un ejercicio: es una experiencia de aprendizaje de principio a fin, cuidadosamente diseñada, que concluye con un recorrido detallado a través de la implementación del código completo de 877 líneas en C++. Más adelante en este prefacio, veremos una descripción detallada de las nueve secciones de este ejemplo práctico.
- **Proceso de compilación y vinculación para los programas con varios archivos de código fuente.** En el capítulo 3 se incluye un diagrama detallado y un análisis acerca del proceso de compilación y vinculación que produce una aplicación ejecutable.
- **Explicación de la pila de llamadas a funciones.** En el capítulo 6 proporcionamos un análisis detallado (con ilustraciones) acerca de la pila de llamadas a funciones y los registros de activación, para explicar cómo C++ puede llevar el registro acerca de cuál función se está ejecutando en un momento dado, cómo se mantienen en memoria las variables automáticas de las funciones, y cómo sabe una función a dónde regresar una vez que termina de ejecutarse.
- **Las clases string y vector de la Biblioteca estándar de C++.** Las clases *string* y *vector* se utilizan para hacer que los primeros ejemplos sean más orientados a objetos.
- **La clase string.** Utilizamos la clase *string* en vez de las cadenas *char ** basadas en apuntadores al estilo C para la mayoría de las manipulaciones de cadenas a lo largo de este libro. En los capítulos 8, 10, 11 y 21 seguimos incluyendo discusiones acerca de las cadenas *char ** para que los estudiantes puedan practicar con las manipulaciones de apuntadores, para ilustrar la asignación dinámica de memoria con *new* y *delete*, para construir nuestra propia clase *String* y para preparar a los estudiantes con el fin de trabajar con las cadenas *char ** en el código heredado de C y C++.

- **La plantilla de clase vector.** A lo largo del libro, utilizamos la plantilla de clase `vector` en vez de las manipulaciones de arreglos basadas en apuntadores al estilo C. Sin embargo, primero hablamos sobre los arreglos basados en apuntadores estilo C en el capítulo 7, con el fin de preparar a los estudiantes para trabajar con el código heredado de C y C++, y para usarlos como base para construir nuestra propia clase `Arreglo` personalizada en el capítulo 11, Sobrecarga de operadores: objetos `String` y `Array`.
- **Tratamiento optimizado de la herencia y el polimorfismo.** Los capítulos 12 y 13 se han optimizado cuidadosamente mediante el uso de una jerarquía de clases `Empleado`, para que el tratamiento de la herencia y el polimorfismo sea un proceso más claro y accesible para los estudiantes que incursionen por primera vez en la POO.
- **Análisis e ilustración acerca del funcionamiento interno del polimorfismo.** El capítulo 13 contiene un diagrama detallado y una explicación acerca de cómo puede C++ implementar internamente el polimorfismo, las funciones `virtual` y la vinculación dinámica. Esto proporciona a los estudiantes una sólida comprensión acerca de cómo funcionan realmente estas herramientas. Y lo que es más importante, ayuda a los estudiantes a apreciar la sobrecarga impuesta por el polimorfismo, en términos de consumo adicional de memoria y tiempo del procesador. Esto ayuda a los estudiantes a determinar cuándo deben usar el polimorfismo, y cuándo deben evitarlo.
- **Biblioteca de plantillas estándar (STL).** Éste podría ser uno de los temas más importantes del libro, en términos de la apreciación que tenga sobre la reutilización de software. La STL define poderosos componentes reutilizables, basados en plantillas, que implementan muchas estructuras de datos y algoritmos comunes utilizados para procesar esas estructuras de datos. En el capítulo 22 se presenta la STL y se describen sus tres componentes clave: contenedores, iteradores y algoritmos. Aquí mostramos que el uso de los componentes de la STL nos proporciona un enorme poder expresivo, lo cual a menudo reduce muchas líneas de código a una sola instrucción.
- **Conformidad con el estándar ISO/IEC de C++.** Hemos auditado nuestra presentación para compararla con el documento más reciente del estándar ISO/IEC de C++, en cuanto a su grado de exactitud y precisión. [Nota: si necesita detalles técnicos adicionales sobre C++, tal vez sea conveniente que adquiera el documento del estándar de C++. en webstore.ansi.org/ansidocstore/default.asp (número de documento INCITS/ISO/IEC 14882-2003)].
- **Apéndices del depurador.** Incluimos dos apéndices sobre el uso del depurador: apéndice I, Uso del Depurador de Visual Studio, y el apéndice J, Uso del Depurador de GNU C++.
- **Prueba del código en varias plataformas.** Probamos los ejemplos de código en varias plataformas populares para C++. En su mayor parte, todos los ejemplos del libro se pueden portar con facilidad a todos los compiladores populares que cumplen con el estándar.
- **Errores y advertencias mostrados para varias plataformas.** Para los programas que contienen errores intencionales para ilustrar un concepto clave, mostramos los mensajes de error que se producen en varias plataformas populares.

Todo esto ha sido cuidadosamente revisado por distinguidos profesores y desarrolladores de la industria que trabajaron con nosotros en la quinta y sexta edición de *Cómo programar en C++*.

Creemos que este libro y sus materiales de apoyo proporcionarán a los estudiantes y profesionales una experiencia educativa informativa, interesante, retadora y divertida en relación con C++. Este libro incluye una extensa suite de materiales auxiliares que ayudan a los instructores a maximizar la experiencia de aprendizaje de sus estudiantes.

A medida que lea este libro, si tiene preguntas, no dude en enviar un correo electrónico a deitel@deitel.com; le responderemos lo más pronto posible. Para obtener actualizaciones en relación con este libro y con todo el software de soporte para C++, así como para ver las noticias más recientes acerca de todas las publicaciones y servicios Deitel, visite www.deitel.com. Suscríbase en www.deitel.com/newsletter/subscribe.html para recibir el boletín de noticias electrónico *Deitel® Buzz Online*, y dé un vistazo a nuestra creciente lista de Centros de Recursos relacionados en www.deitel.com/ResourceCenters.html. Cada semana anunciaremos nuestros Centros de Recursos más recientes en el boletín de noticias. No dude en comentarnos acerca de otros Centros de Recursos que desee ver incluidos.

Método de enseñanza

Cómo programar en C++, sexta edición, contiene una extensa colección de ejemplos. El libro se concentra en los principios de la buena ingeniería de software, haciendo hincapié en la claridad de los programas. Enseñamos mediante ejemplos.

Somos educadores que presentamos temas de vanguardia en salones de clases de la industria alrededor de todo el mundo. El Dr. Harvey M. Deitel tiene 20 años de experiencia en la enseñanza a nivel universitario y 18 en la industria. Paul Deitel tiene 16 años de experiencia en la enseñanza dentro de la industria. Los Deitel han impartido cursos en todos los niveles, a clientes gubernamentales, industriales, militares y académicos.

Método del código activo. *Cómo programar en C++, sexta edición*, está lleno de ejemplos de “código activo”; esto significa que cada nuevo concepto se presenta en el contexto de una aplicación en C++ completa y funcional, que es seguido inmediatamente por una o más ejecuciones actuales, que muestran la entrada y salida del programa. Este estilo ejemplifica la manera en que enseñamos y escribimos acerca de la programación; a esto le llamamos “método de código activo”.

Resaltado de código. Colocamos rectángulos de color gris alrededor de los segmentos de código clave en cada programa.

Uso de tipos de letra para dar énfasis. Colocamos los términos clave en **negritas** para facilitar su referencia. Enfatizamos los componentes en pantalla con el tipo de letra **Helvética en negritas** (por ejemplo, el menú **Archivo**) y enfatizamos el texto del programa con el tipo de letra **Lucida** (por ejemplo, `int x = 5`).

Acceso Web. Todos los ejemplos de código fuente (en inglés) para *Cómo programar en C++, sexta edición*, están disponibles para su descarga en:

www.deitel.com/books/cpphtp6

El registro en el sitio es un proceso fácil y rápido. Descargue todos los ejemplos a medida que lea los correspondientes análisis en el libro de texto, después ejecute cada programa. Realizar modificaciones a los ejemplos y ver los efectos de esos cambios es una excelente manera de mejorar su experiencia de aprendizaje en C++.

Objetivos. Cada capítulo comienza con una declaración de objetivos. Esto le permite saber qué es lo que debe esperar y le brinda la oportunidad, después de leer el capítulo, de determinar si ha cumplido con ellos.

Frases. Después de los objetivos de aprendizaje aparecen una o más frases. Algunas son graciosas, otras filosóficas; y las demás ofrecen ideas interesantes. Esperamos que disfrute relacionando las frases con el material del capítulo.

Plan general. El plan general de cada capítulo le permite abordar el material de manera ordenada, para poder anticiparse a lo que está por venir y establecer un ritmo cómodo y efectivo de aprendizaje.

Ilustraciones/Figuras. Incluimos una gran cantidad de gráficas, tablas, dibujos lineales, programas y salidas de programa. Modelamos el flujo de control en las instrucciones de control mediante diagramas de actividad en UML. Los diagramas de clases de UML modelan los campos, constructores y métodos de las clases. En el ejemplo práctico opcional del ATM de DOO/UML 2 hacemos uso extensivo de seis tipos principales de diagramas en UML.

Tips de programación. Incluimos tips de programación para ayudarle a enfocarse en los aspectos importantes del desarrollo de programas. Estos tips y prácticas representan lo mejor que hemos podido recabar a lo largo de seis décadas combinadas de experiencia en la programación y la enseñanza. Una de nuestras alumnas, estudiante de matemáticas, recientemente nos comentó que siente que este método es similar al de resaltar axiomas, teoremas y corolarios en los libros de matemáticas, ya que proporciona una base sólida sobre la cual se puede construir buen software.



Buena práctica de programación

Las buenas prácticas de programación llaman la atención hacia técnicas que le ayudarán a producir programas más claros, comprensibles y fáciles de mantener.



Error común de programación

Con frecuencia, los estudiantes tienden a cometer ciertos tipos de errores. Al poner atención en estos errores comunes de programación se reduce la probabilidad de que pueda cometerlos.



Tip para prevenir errores

Estos tips contienen sugerencias para exponer los errores y eliminarlos de sus programas; muchos de ellos describen aspectos de C++ que evitan que los errores entren a los programas.



Tip de rendimiento

A los estudiantes les gusta “turbo cargar” sus programas. Estos tips resaltan las oportunidades para hacer que sus programas se ejecuten más rápido, o para minimizar la cantidad de memoria que ocupan.



Tip de portabilidad

Incluimos tips de portabilidad para ayudarle a escribir código que pueda ejecutarse en una variedad de plataformas, y que expliquen cómo es que C++ logra su alto grado de portabilidad.



Observación de Ingeniería de Software

Las observaciones de ingeniería de software resaltan los asuntos de arquitectura y diseño, lo cual afecta la construcción de los sistemas de software, especialmente los de gran escala.

Sección de repaso. Cada uno de los capítulos termina con una breve sección de “repaso”, que recapitula el contenido del capítulo y ofrece una transición al siguiente capítulo.

Viñetas de resumen. Cada capítulo termina con estrategias pedagógicas adicionales. Presentamos un resumen detallado del capítulo, estilo lista con viñetas, sección por sección.

Terminología. Incluimos una lista alfabetizada de los términos importantes definidos en cada capítulo.

Ejercicios de autoevaluación y respuestas. Se incluyen diversos ejercicios de autoevaluación con sus respuestas, para que los estudiantes practiquen por su cuenta.

Ejercicios. Cada capítulo concluye con un diverso conjunto de ejercicios, incluyendo recordatorios simples de terminología y conceptos importantes; identificar los errores en muestras de código, escribir instrucciones individuales de C++; escribir pequeñas porciones de funciones y clases; escribir funciones, clases y programas completos; y crear proyectos finales importantes. El extenso número de ejercicios permite a los instructores adaptar sus cursos a las necesidades únicas de sus estudiantes, y variar las asignaciones de los cursos cada semestre. Los instructores pueden usar estos ejercicios para formar tareas, exámenes cortos, exámenes regulares y proyectos finales. En nuestro Centro de Recursos de Proyectos de Programación (www.deitel.com/ProgrammingProjects/) podrá consultar muchos ejercicios adicionales y posibilidades de proyectos.

[NOTA: No nos escriba para solicitar acceso al Centro de Recursos para Instructores. El acceso está limitado estrictamente a profesores universitarios que imparten clases con base en el libro. Sólo se puede obtener acceso a través de los representantes de Pearson Educación].

Miles de entradas en el índice. Hemos incluido un extenso índice, que es en especial útil cuando se utiliza el libro como referencia.

“Doble indexado” de ejemplos de código activo de C++. Para cada programa de código fuente en el libro, indexamos la leyenda de la figura en forma alfabética y como subíndice, bajo “Ejemplos”. Esto facilita encontrar los ejemplos usando las características especiales.

Recorrido a través del libro

Ahora vamos a dar un vistazo a las herramientas de C++ que estudiará en *Cómo programar en C++, sexta edición*. La figura 1 ilustra las dependencias entre los capítulos. Le recomendamos estudiar los temas en el orden indicado por las flechas, aunque es posible utilizar otras secuencias. Este libro se utiliza ampliamente en todos los niveles de cursos de programación de C++. Busque en Web la palabra “programa de estudios”, “C++” y “Deitel” para encontrar los programas de estudios que se utilizan con las ediciones recientes de este libro.

Capítulo 1, Introducción a las computadoras, Internet y World Wide Web: habla sobre lo que son las computadoras, cómo trabajan y cómo se programan. Este capítulo presenta una breve historia del desarrollo de los lenguajes de programación, partiendo desde los lenguajes máquina, pasando por los lenguajes ensambladores y terminando con los lenguajes de alto nivel. Se habla también sobre los orígenes del lenguaje de programación C++. El capítulo incluye una introducción a un entorno de programación común en C++. Llevamos a los lectores a través de un “recorrido de prueba” de una aplicación común de C++ en las plataformas Windows y Linux. También se presentan los conceptos básicos y la terminología de la tecnología de objetos, y el Lenguaje Unificado de Modelado.

Capítulo 2, Introducción a la programación en C++: ofrece una breve introducción a la programación de aplicaciones en el lenguaje de programación C++. Este capítulo presenta conceptos y construcciones básicas de programación para no programadores. Los programas de este capítulo ilustran cómo mostrar datos en pantalla y cómo obtener datos del usuario mediante el teclado. El capítulo termina con un análisis detallado de la toma de decisiones y las operaciones aritméticas.

Capítulo 3, Introducción a las clases y los objetos: ofrece una primera introducción amigable a las clases y los objetos. Pone a los estudiantes a trabajar con la orientación a objetos de una manera confortable desde el principio.

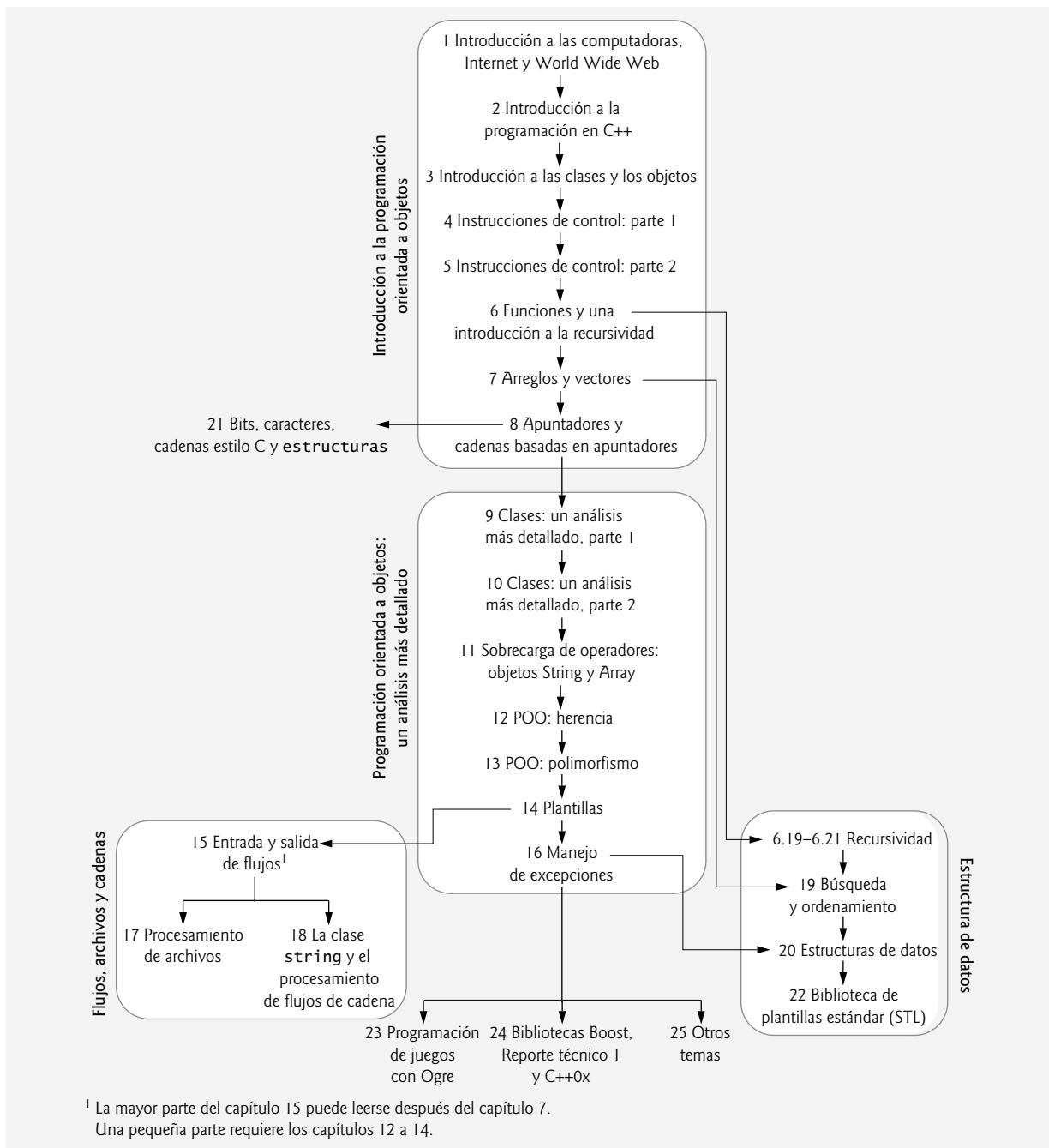


Figura 1 | Gráfico de dependencias de los capítulos de *Cómo programar en C++*, sexta edición.

El capítulo se desarrolló con la guía de un distinguido equipo de revisores académicos y de la industria. Presentamos las clases, los objetos, las funciones miembro, los constructores y los miembros de datos, utilizando una serie de ejemplos simples y reales. Desarrollamos un marco de trabajo bien diseñado para organizar programas orientados a objetos en C++. Primero, motivamos la noción de las clases con un ejemplo simple. Después presentamos una secuencia cuidadosamente elaborada de siete programas funcionales completos, para demostrar cómo puede crear y utilizar sus propias clases. Estos ejemplos comienzan nuestro **ejemplo práctico integrado acerca del desarrollo de una clase de libro de calificaciones**, que los instructores pueden utilizar para administrar las calificaciones de las pruebas de sus estudiantes. Este ejemplo práctico se mejora a lo largo de los siguientes capítulos, culminando con la versión que se presenta en el capítulo 7, Arreglos y vectores. El ejemplo práctico de la clase `LibroCalificaciones` describe cómo definir y utilizar una clase para

crear un objeto. También describe cómo declarar y definir funciones miembro para implementar los comportamientos de la clase, cómo declarar miembros de datos para implementar los atributos de la clase y cómo llamar a las funciones miembro de un objeto para que lleven a cabo sus tareas. Presentamos la clase `string` de la Biblioteca Estándar de C++ y creamos objetos `string` para almacenar el nombre de un curso representado por un objeto `LibroCalificaciones`. El capítulo 3 explica las diferencias entre los miembros de datos de una clase y las variables locales de una función, y cómo utilizar un constructor para asegurar que se inicialicen los datos de un objeto a la hora de crearlo. Le mostramos cómo promover la reutilización de software, separando la definición de una clase del código cliente (por ejemplo, la función `main`) que utiliza a esa clase. También introducimos otro principio fundamental de la buena ingeniería de software: separar la interfaz de la implementación. El capítulo incluye un diagrama detallado y un análisis en el que se explica el proceso de compilación y vinculación, que produce una aplicación ejecutable.

Capítulo 4, Instrucciones de control: parte 1: se enfoca en el proceso de desarrollo de programas implicado en la creación de clases útiles. Este capítulo habla sobre cómo tomar un enunciado del problema y, a partir de él, desarrollar un programa funcional en C++, incluyendo la realización de los pasos intermedios con seudocódigo. También introduce ciertas instrucciones simples de control para la toma de decisiones (`if` e `if...else`) y la repetición (`while`). Analizamos la repetición controlada por contador y la repetición controlada por centinela, usando la clase `LibroCalificaciones` del capítulo 3, y presentamos los operadores de incremento, decremento y asignación de C++. Este capítulo incluye **dos versiones mejoradas de la clase LibroCalificaciones**, cada una de ellas basada en la versión final del capítulo 3. Cada versión incluye una función miembro que utiliza instrucciones de control para calcular el promedio de un conjunto de calificaciones de estudiantes. En la primera versión, la función miembro utiliza la repetición controlada por contador para recibir 10 calificaciones de estudiantes del usuario, y después determina la calificación promedio. En la segunda versión, la función miembro utiliza la repetición controlada por contador para recibir un número arbitrario de calificaciones del usuario, y después calcula el promedio de las calificaciones que se introdujeron. El capítulo utiliza los diagramas de actividad simples de UML para mostrar el flujo de control a través de cada una de las instrucciones de control.

Capítulo 5, Instrucciones de control, parte 2: continúa hablando sobre las instrucciones de control de C++, con ejemplos de la instrucción de repetición `for`, la instrucción de repetición `do...while`, la instrucción de selección `switch`, las instrucciones `break` y `continue`. Creamos una **versión mejorada de la clase LibroCalificaciones** que utiliza una instrucción `switch` para contar el número de calificaciones A, B, C, D y F introducidas por el usuario. Esta versión utiliza la repetición controlada por centinela para introducir las calificaciones. Mientras se leen las calificaciones del usuario, una función miembro modifica los miembros de datos que llevan la cuenta de las calificaciones en cada categoría de letra. Después, otra función miembro de la clase utiliza estos miembros de datos para mostrar un informe de resumen, con base en las calificaciones introducidas. También se incluye una discusión sobre los operadores lógicos.

Capítulo 6, Funciones y una introducción a la recursividad: ofrece un análisis más detallado de los objetos y sus funciones miembro. Aquí hablaremos sobre las funciones de la Biblioteca estándar de C++ y analizaremos con más detalle la forma en que los estudiantes pueden crear sus propias funciones. Las técnicas que se presentan en el capítulo 6 son esenciales para producir programas correctamente estructurados, especialmente los programas más grandes y el software que desarrollan los programadores de sistemas y de aplicaciones en el mundo real. La estrategia “divide y vencerás” se presenta como un medio efectivo para resolver problemas complejos, al dividirlos en componentes interactivos más simples. El primer ejemplo del capítulo continúa el **ejemplo práctico de la clase LibroCalificaciones** con un ejemplo de una función con varios parámetros. Los estudiantes disfrutarán el tratamiento que se da a los números aleatorios y la simulación en este capítulo, así como la discusión sobre el juego de dados llamado “craps”, que hace un uso elegante de las instrucciones de control. También se habla sobre las denominadas “Mejoras que C++ hace a C”, incluyendo las funciones `inline`, los parámetros por referencia, los argumentos predeterminados, el operador unario de resolución de alcance, la sobrecarga de funciones y las plantillas de funciones. Además, presentamos las herramientas de llamada por valor y llamada por referencia de C++. La tabla de archivos de encabezado introduce muchos de los archivos de encabezado que utilizará a través del libro. En esta nueva edición, proporcionamos una discusión detallada (con ilustraciones) sobre la pila de llamadas a funciones y los registros de **activación**, para explicar cómo C++ puede llevar el registro de la función que se esté ejecutando en un momento dado, cómo se mantienen las variables automáticas de funciones en la memoria y cómo sabe una función a dónde debe regresar, una vez que termine su ejecución. Después el capítulo ofrece una concisa introducción a la recursividad, e incluye una tabla que sintetiza los ejemplos y ejercicios de recursividad distribuidos en el resto del libro. Algunos textos dejan la recursividad para uno de los últimos capítulos; nosotros sentimos que este tema se cubre mejor de manera gradual, a lo largo del texto. La extensa colección de ejercicios al final del capítulo incluye varios problemas clásicos de recursividad, incluyendo las Torres de Hanoi.

Capítulo 7, Arreglos y vectores: explica cómo procesar las listas de procesamiento y las tablas de valores. Aquí hablaremos sobre la estructuración de datos en arreglos de elementos de datos del mismo tipo, y demostraremos cómo los arreglos facilitan las tareas que realizan los objetos. En las primeras partes de este capítulo se utilizan los arreglos basa-

dos en apuntadores estilo C que, como veremos en el capítulo 8, se pueden tratar como apuntadores al contenido de los arreglos en la memoria. Luego presentaremos los arreglos como objetos completos, con una introducción a la plantilla de clase `vector` de la Biblioteca estándar de C++: una estructura de datos tipo arreglo robusta. El capítulo presenta numerosos ejemplos, tanto de los arreglos unidimensionales como de los multidimensionales. Los ejemplos en este capítulo investigan varias manipulaciones comunes de los arreglos, la impresión de gráficos de barras, el ordenamiento de datos y el proceso de pasar arreglos a las funciones. También se incluyen las **últimas dos secciones del ejemplo práctico de LibroCalificaciones**, en donde utilizamos arreglos para almacenar las calificaciones de los estudiantes durante la ejecución de un programa. Las versiones anteriores de la clase procesan un conjunto de calificaciones que introduce el usuario, pero no mantienen los valores de cada calificación en los miembros de datos de la clase. En este capítulo utilizamos los arreglos para permitir que un objeto de la clase `LibroCalificaciones` mantenga un conjunto de calificaciones en memoria, con lo cual se elimina la necesidad de introducir repetidas veces el mismo conjunto de calificaciones. La primera versión de la clase almacena las calificaciones en un arreglo unidimensional, y puede producir un informe que contiene el promedio de las calificaciones, las calificaciones mínima y máxima, y un gráfico de barras que representa la distribución de las calificaciones. La segunda versión (es decir, la versión final en el ejemplo práctico) utiliza un arreglo bidimensional para almacenar las calificaciones de varios estudiantes, en diversos exámenes durante un semestre. Esta versión puede calcular el promedio semestral de cada estudiante, así como las calificaciones mínima y máxima de todas las calificaciones recibidas durante el semestre. La clase también produce un gráfico de barras, el cual muestra la distribución total de calificaciones para el semestre. Otra característica clave de este capítulo es la discusión sobre las técnicas elementales de ordenamiento y búsqueda. Los ejercicios de final de capítulo incluyen una variedad de problemas interesantes y retadores, como las técnicas de ordenamiento mejoradas, el diseño de un sistema simple de reservaciones de una aerolínea, una introducción al concepto de los gráficos de tortuga (que se hicieron famosos en el lenguaje LOGO) y los problemas Paseo del caballo y Ocho reinas, que introducen la noción de la programación heurística, que se utiliza ampliamente en el campo de la inteligencia artificial. Los ejercicios concluyen con muchos problemas de recursividad, incluyendo el ordenamiento por selección, los palíndromos, la búsqueda lineal, las Ocho reinas, imprimir un arreglo, imprimir una cadena al revés y buscar el valor mínimo en un arreglo.

Capítulo 8, Apuntadores y cadenas basadas en apuntadores: presenta una de las características más poderosas del lenguaje C++: los apuntadores. El capítulo proporciona explicaciones detalladas de los operadores de apuntadores, la llamada por referencia, las expresiones de apuntadores, la aritmética de apuntadores, la relación entre apuntadores y arreglos, los arreglos de apuntadores y los apuntadores a funciones. Demostramos cómo utilizar `const` con apuntadores para cumplir con el principio del menor privilegio y poder crear software más robusto. Hablamos acerca del uso del operador `sizeof` para determinar el tamaño de un tipo de datos, o de los elementos de datos en bytes durante la compilación de un programa. Hay una estrecha relación entre los apuntadores, los arreglos y las cadenas estilo C en C++, por lo que presentamos los conceptos básicos de manipulación de cadenas estilo C y hablamos sobre algunas de las funciones para manejar cadenas estilo C más populares, como `getline` (recibir una línea de texto), `strcpy` y `strncpy` (copiar una cadena), `strcat` y `strncat` (concatenar dos cadenas), `strcmp` y `strncmp` (comparar dos cadenas), `strtok` (dividir una cadena en sus piezas básicas) y `strlen` (devolver la longitud de una cadena). En esta nueva edición utilizamos con frecuencia objetos `string` (introducidos en el capítulo 3, Introducción a las clases y los objetos) en vez de las cadenas basadas en apuntadores `char *`, estilo C. Sin embargo, incluimos las cadenas `char *` en el capítulo 8 para ayudar al lector a dominar los apuntadores, y prepararlo para el mundo profesional, en el que verá una gran cantidad de código heredado de C, que se ha implementado durante las últimas tres décadas. Por ende, se familiarizará con los dos métodos más prevalecientes de crear y manipular cadenas en C++. Muchas personas descubren que el tema de los apuntadores es, hasta ahora, la parte más difícil de un curso introductorio de programación. En C y en “C++ puro”, los arreglos y las cadenas son apuntadores al contenido de arreglos y cadenas en la memoria (inclusive hasta los nombres de las funciones son apuntadores). Si estudia este capítulo con cuidado, obtendrá como recompensa una detallada comprensión de los apuntadores. Este capítulo está repleto de ejercicios retadores. Los ejercicios del capítulo incluyen una simulación de la clásica carrera entre la tortuga y la liebre, algoritmos para barajar y repartir cartas, el ordenamiento rápido (quicksort) recursivo y recorridos recursivos de laberintos. También se incluye una **sección especial titulada Construya su propia computadora**. Esta sección explica la programación en lenguaje máquina y procede con un proyecto en el que se involucra el diseño y la implementación de un simulador de computadora, el cual lleva al estudiante a escribir y ejecutar programas en lenguaje máquina. Esta característica única del texto será en especial útil para los lectores que deseen comprender el verdadero funcionamiento de las computadoras. Nuestros estudiantes disfrutan este proyecto y a menudo implementan mejoras sustanciales, muchas de las cuales se sugieren en los ejercicios. Una segunda sección especial incluye ejercicios retadores de manipulación de cadenas, relacionados con el análisis de texto, procesamiento de palabras, impresión de fechas en varios formatos, protección de cheques, escribir el equivalente de un monto de un cheque, código Morse y conversiones del sistema métrico al inglés.

Capítulo 9, Clases: un análisis más detallado, parte 1: continúa con nuestra discusión sobre la programación orientada a objetos. Este capítulo utiliza un ejemplo práctico muy completo con la clase `Tiempo`, para ilustrar el acceso a los miembros de una clase, separar la interfaz de la implementación, usar las funciones de acceso y las funciones utilitarias, inicializar objetos mediante constructores, destruir objetos mediante destructores, la asignación mediante la copia a nivel de miembro predeterminada y la reutilización de software. Los estudiantes aprenden el orden en el que se llama a los constructores y destructores durante el tiempo de vida de un objeto. Una modificación del ejemplo práctico de `Tiempo` demuestra los problemas que pueden ocurrir cuando una función miembro devuelve una referencia a un miembro de datos `private`, lo cual quebranta el encapsulamiento de la clase. Los ejercicios del capítulo retan al estudiante a que desarrolle clases para tiempos, fechas, rectángulos y jugar al tres en raya. Por lo general, los estudiantes disfrutan los programas de juegos. Los lectores inclinados hacia las matemáticas disfrutarán los ejercicios relacionados con la creación de la clase `Complejo` (para los números complejos), la clase `Racional` (para los números racionales) y la clase `EnteroEnorme` (para los enteros arbitrariamente grandes).

Capítulo 10, Clases: un análisis más detallado, parte 2: continúa el estudio de las clases y presenta conceptos adicionales de programación orientada a objetos. El capítulo habla sobre cómo declarar y utilizar objetos constantes, funciones miembro constantes, la composición (el proceso de crear clases que tengan objetos de otras clases como miembros), las funciones `friend` y las clases `friend` que tienen derechos de acceso especiales para los miembros `private` y `protected` de las clases, el apuntador `this`, que permite a un objeto conocer su propia dirección, la asignación dinámica de memoria, los miembros de clase `static` para contener y manipular datos a nivel de clase, ejemplos de los tipos de datos abstractos populares (arreglos, cadenas y colas), clases contenedoras e iteradores. En nuestra discusión sobre los objetos `const` mencionamos la palabra clave `mutable`, que se utiliza de una manera sutil para permitir la modificación de la implementación “invisible” en los objetos `const`. Hablaremos sobre la asignación dinámica de memoria mediante el uso de `new` y `delete`. Cuando falla `new`, el programa termina de manera predeterminada, ya que `new` “lanza una excepción” en el lenguaje C++ estándar. Motivamos la discusión de los miembros de clase `static` con un ejemplo basado en un videojuego. Enfatizamos lo importante que es ocultar los detalles de implementación de los clientes de una clase; después hablamos sobre las clases de proxy, que proporcionan un medio para ocultar la implementación (incluyendo los datos `private` en los encabezados de las clases) a los clientes de una clase. Los ejercicios del capítulo incluyen el desarrollo de una clase de cuenta de ahorros y una clase para contener conjuntos de enteros.

Capítulo 11, Sobrecarga de operadores: objetos String y Array: presenta uno de los temas más populares en nuestros cursos de C++. Los estudiantes realmente disfrutan este material. Les parece un complemento perfecto para el análisis detallado de cómo construir clases valiosas en los capítulos 9 y 10. La sobrecarga de operadores nos permite decir al compilador cómo utilizar los operadores existentes con objetos de nuevos tipos. C++ sabe de antemano cómo utilizar estos operadores con objetos de tipos integrados, como enteros, números de punto flotante y caracteres. Pero suponga que creamos una nueva clase `String`; ¿qué significaría el signo de suma al utilizarlo entre objetos `String`? Muchos programadores usan la suma (+) con cadenas para indicar la concatenación. En el capítulo 11 aprenderá a “sobrecargar” el signo de suma, de manera que cuando se escriba entre dos objetos `String` en una expresión, el compilador generará la llamada a una “función operador” que concatenará los dos objetos `String`. El capítulo describe los fundamentos de la sobrecarga de operadores, las restricciones en la sobrecarga de operadores, la comparación entre la sobrecarga con funciones miembro de clase y funciones no miembro, la sobrecarga de operadores unarios y binarios, y la conversión entre tipos. El capítulo 11 contiene una colección de ejemplos prácticos sustanciales, incluyendo una clase de arreglo, una clase `String`, una clase de fecha, una clase de entero enorme y una clase de números complejos (las últimas dos aparecen con el código fuente completo en los ejercicios). Los estudiantes con inclinación a las matemáticas disfrutarán la creación de la clase `polinomio` en los ejercicios. Este material es distinto de la mayoría de los lenguajes y cursos de programación. La sobrecarga de operadores es un tema complejo, pero enriquecedor. El uso inteligente de la sobrecarga de operadores nos ayuda a agregar un “pulido” adicional a nuestras clases. Las discusiones sobre la clase `Array` y la clase `String` son especialmente valiosas para los estudiantes que ya han utilizado la clase `string` de la Biblioteca estándar de C++ y la plantilla de clase `vector`, las cuales proporcionan características similares. Los ejercicios alientan al estudiante para que agregue la sobrecarga de operadores a las clases `Complejo`, `Racional` y `EnteroEnorme`, para permitir una manipulación conveniente de objetos de estas clases con los símbolos de los operadores (al igual que en las matemáticas), en vez de hacerlo con las llamadas a funciones, como hizo el estudiante en los ejercicios del capítulo 10.

Capítulo 12, Programación orientada a objetos: herencia: introduce una de las capacidades más fundamentales de los lenguajes de programación orientados a objetos: la herencia, una forma de reutilización de software en la que las nuevas clases se desarrollan con rapidez y facilidad, absorbiendo las capacidades de las clases existentes y agregando nuevas capacidades apropiadas. En el contexto de un ejemplo práctico con la jerarquía `Empleado`, este capítulo (que se revisó minuciosamente) presenta una secuencia de cinco ejemplos en la que se demuestra el uso de los datos `private`, los datos `protected` y la buena ingeniería de software con la herencia. Para empezar, demostramos una clase con miembros de datos `private` y funciones miembro `public` para manipular esos datos. Después, implementamos una segunda

clase con capacidades adicionales, duplicando de manera intencional y tediosa la mayor parte del código del primer ejemplo. El tercer ejemplo empieza nuestra discusión sobre la herencia y la reutilización de software; utilizamos la clase del primer ejemplo como una clase base y heredamos de una manera rápida y simple sus datos y su funcionalidad, para crear una nueva clase derivada. Este ejemplo introduce el mecanismo de la herencia y demuestra que una clase derivada no puede acceder directamente a los miembros `private` de su clase base. Esto motiva nuestro cuarto ejemplo, en el que introducimos datos `protected` en la clase base y demostramos que la clase derivada puede, sin duda, acceder a los datos `protected` heredados de la clase base. El último ejemplo en la secuencia demuestra la ingeniería de software apropiada, al definir los datos de la clase base como `private` y utilizar las funciones miembro `public` de la clase (que heredó la clase derivada) para manipular los datos `private` de la clase base en la clase derivada. El capítulo habla acerca de las nociones de las clases base y las clases derivadas, los miembros `protected`, la herencia `public`, la herencia `protected`, la herencia `private`, las clases base directas, las clases base indirectas, los constructores y los destructores en las clases base y las clases derivadas, y la ingeniería de software con la herencia. El capítulo también compara la herencia (la relación “*es un*”) con la composición (la relación “*tiene un*”), e introduce las relaciones “*utiliza un*” y “*conoce un*”.

Capítulo 13, Programación orientada a objetos: polimorfismo: trata con otra herramienta fundamental de la programación orientada a objetos: el comportamiento polimórfico. El capítulo 13, que se revisó por completo, se basa en los conceptos de la herencia presentados en el capítulo 12 y se enfoca en las relaciones entre las clases en una jerarquía de clases, y en las poderosas capacidades de procesamiento que habilitan estas relaciones. Cuando muchas clases están relacionadas con una clase base común a través de la herencia, cada objeto de clase derivada puede tratarse como un objeto de la clase base. Esto permite que los programas se escriban de una manera simple y general, independiente de los tipos específicos de los objetos de las clases derivadas. Pueden manejarse nuevos tipos de objetos mediante el mismo programa, con lo cual los sistemas se hacen más extensibles. El polimorfismo permite a los programas eliminar la compleja lógica de la instrucción `switch`, a favor de una lógica de “línea recta” más simple. Por ejemplo, el administrador de la pantalla de un videojuego puede enviar un mensaje `dibujar` a cada objeto en una lista enlazada de objetos a dibujar. Cada objeto sabe cómo dibujarse a sí mismo. Puede agregarse un objeto de una nueva clase al programa sin necesidad de modificarlo (siempre y cuando el nuevo objeto también sepa cómo dibujarse a sí mismo). Este capítulo habla sobre la mecánica de lograr el comportamiento polimórfico a través de las funciones `virtual`. Hace diferencia entre las clases abstractas (de las que no se pueden crear instancias de objetos) y las clases concretas (de las que se pueden crear instancias de objetos). Las clases abstractas son útiles para proporcionar una interfaz heredable a las clases, a lo largo de la jerarquía. Demostramos las clases abstractas y el comportamiento polimórfico con una nueva revisión de la **jerarquía Empleado** del capítulo 12. Presentamos una clase base `Empleado` abstracta, a partir de la cual las clases `EmpleadoPorComision`, `EmpleadoPorHoras` y `EmpleadoAsalariado` heredan directamente, y la clase `EmpleadoBaseMasComision` hereda indirectamente. En el pasado, nuestros clientes profesionales han insistido que proporcionemos una explicación más detallada, que muestre con precisión cómo se implementa el polimorfismo en C++, y por ende, precisamente en cuánto tiempo de ejecución y uso de memoria se incurre al programar con esta poderosa herramienta. Respondimos a esta petición desarrollando una ilustración y una explicación precisa de las *vtables* (tablas de funciones `virtual`), que el compilador de C++ genera en forma automática para dar soporte al polimorfismo. Para concluir, presentamos la información de tipos en tiempo de ejecución (RTTI) y la conversión dinámica, que permiten a un programa determinar el tipo de un objeto en tiempo de ejecución, y después actuar sobre ese objeto de manera acorde. Mediante el uso de RTTI y la conversión dinámica, otorgamos un 10% de incremento en el sueldo de los empleados de un tipo específico, y después calculamos los ingresos para dichos empleados. Para todos los demás tipos de empleados, calculamos los ingresos mediante el polimorfismo.

Capítulo 14, Plantillas: habla acerca de una de las características de reutilización de software más poderosas: a saber, las plantillas. Las plantillas de funciones y las plantillas de clases nos permiten especificar, con un solo segmento de código, un rango completo de funciones sobrecargadas relacionadas (llamadas especializaciones de plantillas de funciones), o todo un rango de clases relacionadas (llamadas especializaciones de plantillas de clases). A esta técnica se le conoce como programación genérica. En el capítulo 6 se introdujeron las plantillas de funciones. Este capítulo presenta discusiones y ejemplos adicionales sobre las plantilla de funciones. Podríamos escribir una sola plantilla de clase para una clase de pila, y después hacer que C++ genere especializaciones de plantilla de clases separadas, como una clase “pila de `int`”, una clase “pila de `float`”, una clase “pila de `string`”, y así en lo sucesivo. El capítulo habla sobre el uso de parámetros con tipo, parámetros sin tipo y tipos predeterminados para las plantillas de clases. También hablamos sobre las relaciones entre las plantillas y otras características de C++, como la sobrecarga, la herencia, los miembros `friend` y `static`. Los ejercicios retan al estudiante a escribir una variedad de plantillas de funciones y plantillas de clases, y a emplear éstas en programas completos. Mejoramos en forma considerable el tratamiento de las plantillas en nuestra discusión sobre los contenedores, iteradores y algoritmos de la Biblioteca de plantillas estándar (STL) en el capítulo 22.

Capítulo 15, Entrada y salida de flujos: contiene un tratamiento detallado sobre las herramientas de entrada/salida de C++. Este capítulo habla sobre un rango de herramientas suficiente como para realizar la mayoría de las operaciones comunes de E/S, y describe las generalidades del resto de las herramientas. Muchas de las características

de E/S están orientadas a objetos. Este estilo de E/S hace uso de otras características de C++, como las referencias, la sobrecarga de funciones y la sobrecarga de operadores. Las diversas herramientas de E/S de C++, incluyen las operaciones de salida con el operador de inserción de flujo, las operaciones de entrada con el operador de extracción de flujo, la E/S con seguridad de tipos, la E/S con formato, y la E/S sin formato (para mejorar el rendimiento). Los usuarios pueden especificar cómo llevar a cabo la E/S para objetos de tipos definidos por el usuario, mediante la sobrecarga del operador de inserción de flujo (`<<`) y el operador de extracción de flujo (`>>`). Esta extensibilidad es una de las características más valiosas de C++. Este lenguaje cuenta con varios manipuladores de flujos que realizan tareas de formato. Este capítulo habla sobre los manipuladores de flujos que proporcionan herramientas tales como mostrar enteros en varias bases, controlar la precisión de los números de punto flotante, establecer las anchuras de los campos, mostrar el punto decimal y los ceros a la derecha, justificar la salida, ajustar y desajustar el estado del formato, establecer el carácter de relleno en los campos. También presentamos un ejemplo que crea manipuladores de flujos de salida definidos por el usuario.

Capítulo 16, Manejo de excepciones: habla acerca de cómo el manejo de excepciones nos permite escribir programas robustos, tolerantes a fallas y apropiados para los entornos de “negocio crítico” o de “misión crítica”. El capítulo habla acerca de cuándo es apropiado el manejo de excepciones; introduce las herramientas básicas de manejo de excepciones con los bloques `try`, las instrucciones `throw` y los manejadores `catch`; indica cómo y cuándo volver a lanzar una excepción; explica cómo escribir la especificación de una excepción y procesar las excepciones inesperadas; y habla acerca de los importantes lazos entre las excepciones y los constructores, los destructores y la herencia. Los ejercicios en este capítulo muestran al estudiante la diversidad y el poder de las herramientas de manejo de excepciones de C++. Hablamos sobre cómo volver a lanzar una excepción, e ilustramos cómo puede fallar `new` cuando se agota la memoria. Muchos compiladores antiguos de C++ devuelven 0 de manera predeterminada cuando falla `new`. Mostramos el nuevo estilo de falla de `new`, mediante el lanzamiento de una excepción `bad_alloc` (asignación incorrecta). Ilustramos cómo utilizar la función `set_new_handler` para especificar que se va a llamar una función personalizada para lidiar con las situaciones en las que se agota la memoria. Hablamos acerca de cómo usar la plantilla de clase `auto_ptr` para eliminar (`delete`) implícitamente la memoria asignada en forma dinámica, con lo cual se evitan las fugas de memoria. Para concluir este capítulo, presentamos la jerarquía de excepciones de la Biblioteca estándar.

Capítulo 17, Procesamiento de archivos: habla sobre las técnicas para crear y procesar tanto archivos secuenciales, como archivos de acceso aleatorio. Este capítulo empieza con una introducción a la jerarquía de datos, de los bits a los bytes, los campos, los registros y los archivos. Después, presentamos la forma en que C++ interpreta los archivos y los flujos. Hablamos sobre los archivos secuenciales y creamos programas que muestran cómo abrir y cerrar archivos, cómo almacenar datos en forma secuencial en un archivo, y cómo leer datos en forma secuencial de un archivo. Después hablamos sobre los archivos de acceso aleatorio y creamos programas que muestren cómo crear un archivo para acceso aleatorio, cómo leer y escribir datos de/a un archivo con acceso aleatorio, y cómo leer datos secuencialmente de un archivo con acceso aleatorio. El ejemplo práctico combina las técnicas de acceso a los archivos, tanto en forma secuencial como aleatoria, en un programa completo para procesar transacciones. Los estudiantes en nuestros seminarios industriales han mencionado que, después de estudiar el material sobre procesamiento de archivos, pudieron producir programas substanciales para procesar archivos, que fueron útiles de inmediato en sus organizaciones. Los ejercicios piden al estudiante que implemente una variedad de programas que generen y procesen tanto archivos secuenciales como archivos de acceso aleatorio.

Capítulo 18, La clase `string` y el procesamiento de flujos de cadena: habla acerca de las herramientas de C++ para recibir datos de entrada de cadenas en la memoria, y enviar datos a cadenas en la memoria; con frecuencia, a estas herramientas se les conoce como E/S en memoria o procesamiento de flujos de cadena. La clase `string` es un componente requerido de la Biblioteca estándar. Preservamos el tratamiento de cadenas basadas en apuntadores estilo C en los capítulos 8 y posteriores, por varias razones. En primer lugar, refuerza la comprensión del lector acerca de los apuntadores. En segundo lugar, durante la siguiente década o más, los programadores de C++ tendrán que ser capaces de leer y modificar la enorme cantidad de código heredado de C, que se ha acumulado durante el último trimestre de un siglo; este código procesa las cadenas como apuntadores, al igual que una gran parte del código de C++ que se ha escrito en la industria, durante los últimos años. En el capítulo 18 hablamos sobre la asignación, concatenación y comparación de objetos `string`. Mostramos cómo determinar varias características de `string`, como el tamaño y la capacidad de un objeto `string`, y si está vacío o no. Hablamos sobre cómo cambiar el tamaño de un objeto `string`. Consideramos las diversas funciones de “búsqueda” que nos permiten encontrar una subcadena dentro de un objeto `string` (podemos buscar ya sea hacia adelante o hacia atrás), y mostramos cómo encontrar la primera ocurrencia o la última ocurrencia de un carácter seleccionado de una cadena de caracteres, y cómo encontrar la primera ocurrencia o la última ocurrencia de un carácter que no se encuentra en una cadena seleccionada de caracteres. Mostramos cómo reemplazar, borrar e insertar caracteres en un objeto `string` y cómo convertir un objeto `string` en una cadena `char *` al estilo C.

Capítulo 19, Búsqueda y ordenamiento: habla sobre dos de las clases más importantes de algoritmos en la ciencia computacional. Consideramos una variedad de algoritmos específicos para cada una de estas clases, y los comparamos

con respecto a su consumo de memoria y de procesador (presentamos la notación Big O, la cual indica qué tan duro tiene que trabajar un algoritmo para resolver un problema). La búsqueda de datos implica determinar si un valor (denominado clave de búsqueda) está presente en los datos y, de ser así, buscar la ubicación de ese valor. En los ejemplos y ejercicios de este capítulo, hablamos acerca de una variedad de algoritmos de búsqueda, incluyendo: la búsqueda binaria y versiones recursivas de la búsqueda lineal y la búsqueda binaria. Mediante ejemplos y ejercicios, el capítulo 19 habla sobre el ordenamiento por combinación recursivo, el ordenamiento de burbuja, el ordenamiento de cubeta y el ordenamiento rápido (quicksort) recursivo.

Capítulo 20, Estructuras de datos: habla sobre las técnicas utilizadas para crear y manipular estructuras de datos dinámicas. Este capítulo empieza con discusiones acerca de las clases autorreferenciadas y la asignación dinámica de memoria, y después continúa con una discusión acerca de cómo crear y mantener varias estructuras de datos dinámicas, incluyendo las listas enlazadas, las colas (o líneas de espera), las pilas y los árboles. Para cada tipo de estructura de datos, presentamos programas funcionales completos y mostramos resultados de ejemplo. El capítulo también ayuda a que el estudiante domine los apuntadores. Incluye abundantes ejemplos que utilizan la indirección y la indirección doble: un concepto especialmente difícil. Uno de los problemas al trabajar con apuntadores es que a los estudiantes se les dificulta visualizar las estructuras de datos y la forma en que sus nodos están enlazados entre sí. Hemos incluido ilustraciones que muestran los enlaces y la secuencia en la que se crean. El ejemplo de árbol binario es un excelente toque final para el estudio de los apuntadores y las estructuras de datos dinámicas. Este ejemplo crea un árbol binario, hace cumplir la eliminación de duplicados y presenta los recorridos de árboles preorden, inorden y postorden recursivos. Los estudiantes tienen un sentido genuino de éxito cuando estudian e implementan este ejemplo. En especial, aprecian ver que el recorrido inorden imprime los valores de los nodos en orden. Incluimos una extensa colección de ejercicios. La sección especial Cree su propio compilador es un punto a resaltar de los ejercicios. Éstos llevan al estudiante a través del desarrollo de un programa de conversión de infijo a postfix, y un programa de evaluación de expresiones postfix. Después modificamos el algoritmo de evaluación postfix para generar código en lenguaje máquina. El compilador coloca este código en un archivo (usando las técnicas del capítulo 17). Después, los estudiantes ejecutan el lenguaje máquina producido por sus compiladores ¡en los simuladores de software que construyeron en los ejercicios del capítulo 8! Los numerosos ejercicios incluyen la búsqueda recursiva en una lista, la impresión recursiva de una lista al revés, la eliminación de nodos de un árbol binario, el recorrido por orden de nivel de un árbol binario, la impresión de árboles, la escritura de una parte de un compilador optimizado, la escritura de un intérprete, la inserción/eliminación en cualquier parte de una lista enlazada, la implementación de listas y colas sin apuntadores a la parte final, el análisis del rendimiento de la búsqueda y ordenamiento de árboles binarios, la implementación de una clase de lista indexada y una simulación de un supermercado que utiliza colas. Después de estudiar el capítulo 20, el lector estará preparado para el tratamiento de los contenedores, iteradores y algoritmos de la STL en el capítulo 22. Los contenedores de la STL son estructuras de datos pre-empaquetadas y colocadas en plantillas, que la mayoría de los programadores considerarán suficientes para la vasta mayoría de aplicaciones que necesitarán implementar. La STL es un salto gigante hacia adelante, en cuanto al cumplimiento de la visión de la reutilización.

Capítulo 21, Bits, caracteres, cadenas estilo C y estructuras: presenta varias características importantes. Este capítulo empieza comparando las estructuras con las clases en C++, para después definir y usar las estructuras estilo C. Mostramos cómo declarar estructuras, inicializarlas y pasárlas a las funciones. El capítulo presenta una simulación para barajar y repartir cartas de alto desempeño. Ésta es una excelente oportunidad para que el instructor enfatice la calidad de los algoritmos. Las poderosas herramientas de manipulación de bits de C++ nos permiten escribir programas que utilicen herramientas de hardware de bajo nivel. Esto ayuda a los programas a procesar cadenas de bits, establecer bits individuales y almacenar información en forma más compacta. Dichas herramientas, que por lo general sólo se encuentran en lenguajes ensambladores de bajo nivel, son apreciadas por los programadores que escriben software de sistema, como sistemas operativos y software de red. Si recuerda, presentamos la manipulación de cadenas `char *` estilo C en el capítulo 8, y las funciones más populares para manipular cadenas. En el capítulo 22 continuaremos con nuestra presentación de los caracteres y las cadenas `char *` estilo C. Presentamos las diversas herramientas de manipulación de caracteres de la biblioteca `<cctype>`, como la habilidad de probar un carácter para determinar si es un dígito, un carácter alfabético o alfanumérico, un dígito hexadecimal, una letra minúscula o mayúscula. Presentamos el resto de las funciones de manipulación de cadenas de las diversas bibliotecas relacionadas con cadenas; como siempre, cada función está presente en el contexto de un programa completo y funcional en C++. Los diversos ejercicios alientan al estudiante a probar la mayoría de las herramientas que se describen en este capítulo. El ejercicio principal lleva al estudiante a través del desarrollo de un programa para corregir ortografía. Este capítulo presenta un tratamiento más detallado de las cadenas `char *n` estilo C, para beneficio de los programadores de C++ que trabajen con código heredado de C.

Capítulo 22, Biblioteca de plantillas estándar (STL): a lo largo de este libro, hablamos sobre la importancia de la reutilización de software. Al reconocer que los programadores de C++ utilizan comúnmente muchas estructuras de datos y algoritmos, el comité de estándares de C++ agregó la Biblioteca de plantillas estándar (STL) a la Biblioteca estándar de C++. La STL define componentes poderosos, basados en plantillas y reutilizables, que implementan muchas estructuras

de datos comunes y algoritmos utilizados para procesar esas estructuras de datos. La STL ofrece la prueba del concepto para la programación genérica con plantillas; que presentamos en el capítulo 14 y demostramos con detalle en el capítulo 20. Este capítulo presenta la STL y habla sobre sus tres componentes clave: contenedores (estructuras de datos populares integradas en plantillas), iteradores y algoritmos. Los contenedores de la STL son estructuras de datos capaces de almacenar objetos de cualquier tipo de datos. Aquí veremos que hay tres categorías de contenedores: contenedores de primera clase, adaptadores y “casi contenedores”. Los iteradores de la STL, que tienen propiedades similares a las de los apuntadores, se utilizan en los programas para manipular los elementos de los contenedores de la STL. De hecho, pueden manipularse arreglos estándar como contenedores de la STL, usando apuntadores estándar como iteradores. Aquí veremos que es conveniente manipular los contenedores con iteradores, y que proporcionan un enorme poder expresivo al combinarse con los algoritmos de la STL; en algunos casos, se reducen muchas líneas de código a una sola instrucción. Los algoritmos de la STL son funciones que realizan manipulaciones comunes de datos, como buscar, ordenar y comparar elementos (o contenedores completos). Hay aproximadamente 70 algoritmos implementados en la STL. La mayoría de ellos utilizan iteradores para acceder a los elementos de los contenedores. Veremos que cada contenedor de primera clase soporta tipos específicos de iteradores, algunos de los cuales son más poderosos que otros. El tipo de iterador que soporta un contenedor determina si éste se puede usar con un algoritmo específico. Los iteradores encapsulan el mecanismo que se utiliza para acceder a los elementos de los contenedores. Este encapsulamiento permite aplicar muchos de los algoritmos de la STL a varios contenedores, sin importar la implementación subyacente del contenedor. Mientras que los iteradores de un contenedor soportan los requerimientos mínimos del algoritmo, entonces éste puede procesar los elementos del contenedor. Esto también nos permite crear algoritmos que puedan procesar los elementos de varios tipos de contenedores distintos. En el capítulo 20 hablamos acerca de cómo implementar estructuras de datos con apuntadores, clases y memoria dinámica. El código basado en apuntadores es complejo, y la más ligera omisión o inadvertencia puede provocar graves violaciones al acceso de la memoria y errores de fuga de memoria sin que el compilador se entere. Para implementar estructuras de datos adicionales, como deques, colas de prioridad, mapas, etcétera, se requiere un trabajo adicional considerable. Además, si muchos programadores en un proyecto extenso implementan contenedores y algoritmos similares para distintas tareas, el código se vuelve difícil de modificar, mantener y depurar. Una ventaja de la STL es que podemos reutilizar los contenedores, iteradores y algoritmos de la STL para implementar representaciones comunes de datos y manipulaciones. Esta reutilización produce ahorros considerables en el tiempo de desarrollo y en los recursos. Éste es un capítulo amigable y accesible, que deberá convencer al lector del valor de la STL, y alentar a que se estudie con más detalle.

Capítulo 23, Programación de juegos con Ogre: introduce la programación de juegos con gráficos en 3D mediante el uso de Ogre (Motor de despliegue de gráficos orientado a objetos). Ogre es uno de los motores de gráficos de código fuente abierto más populares. Se ha utilizado en aplicaciones comerciales, incluyendo varios juegos de computadora. Aquí hablaremos sobre los conceptos básicos de la programación de juegos en 3D, incluyendo gráficos, modelos en 3D, sonido, entrada del usuario, detección de colisiones y control de la velocidad del juego. Ofrecemos un ejemplo completamente funcional de un juego simple, que presenta una mecánica de juego similar al clásico videojuego Pong®, diseñado originalmente por Atari en 1972. El capítulo describe el ejemplo paso a paso, explicando los conceptos y funciones clave a medida que los vamos encontrando. Hablamos sobre los diversos recursos que utiliza Ogre, y cómo crearlos mediante secuencias de comandos. Los estudiantes aprenderán a mover, posicionar y ajustar el tamaño de los objetos en un entorno en 3D, a realizar la detección simple de colisiones, mostrar texto dentro del juego y responder a la entrada del usuario mediante el teclado. También demostraremos cómo utilizar OgreAL, una envoltura para la biblioteca de audio OpenAL, para agregar efectos de sonido al juego.

Capítulo 24, Bibliotecas Boost, Reporte técnico 1 y C++0x: se enfoca en el futuro de C++. Presentamos las Bibliotecas Boost, una colección de bibliotecas de C++ gratuitas, de código fuente abierto. Las bibliotecas Boost están cuidadosamente diseñadas para obtener un buen desempeño con la Biblioteca estándar de C++. Después hablamos sobre el Informe técnico 1 (TR1), una descripción de los cambios y adiciones que se proponen para la Biblioteca estándar. Muchas de las bibliotecas en el TR1 se derivaron de las bibliotecas que se encuentran actualmente en Boost. El capítulo describe brevemente todas las bibliotecas incluidas en el TR1. Proporcionamos ejemplos de código detallados para dos de las bibliotecas más utilizadas, `Boost.Regex` y `Boost.Smart_ptr`. La biblioteca `Boost.Regex` proporciona soporte para las expresiones regulares. Demostramos cómo utilizar la biblioteca para buscar coincidencias de una expresión regular en una cadena, validar datos, sustituir partes de una cadena y dividir una cadena en tokens. La biblioteca `Boost.Smart_ptr` proporciona apuntadores inteligentes para ayudar a administrar la memoria que se asigna en forma dinámica. Hablamos sobre los dos tipos de apuntadores inteligentes que se incluyen en el TR1: `shared_ptr` y `weak_ptr`. Proporcionamos ejemplos para demostrar cómo se pueden utilizar estos apuntadores inteligentes para evitar errores comunes de administración de la memoria. En este capítulo también hablamos acerca de la futura presentación del nuevo estándar para C++, que se conoce actualmente como C++0x. Describimos las metas para el nuevo estándar y vemos las generalidades acerca de las modificaciones básicas al lenguaje, que muy probablemente se estandaricen.

Capítulo 25, Otros temas: es una colección de diversos temas acerca de C++. En este capítulo hablamos acerca de un operador adicional de conversión: `const_cast`. Este operador, junto con `static_cast` (capítulo 5), `dynamic_cast` (capítulo 13) y `reinterpret_cast` (capítulo 17), proporciona un mecanismo más robusto para realizar conversiones entre tipos que los operadores de conversión originales que C++ heredó de C (que ahora son obsoletos). Hablamos sobre los espacios de nombres, una característica especialmente imprescindible para los desarrolladores de software que construyen sistemas considerables, en especial para aquellos quienes crean sistemas a partir de las bibliotecas de clases. Los espacios de nombres evitan los conflictos de nomenclatura, que pueden entorpecer dichos esfuerzos grandes de software. El capítulo habla acerca de las palabras clave de los operadores, que son útiles para los programadores que tienen teclados que no soportan ciertos caracteres que se utilizan en los símbolos de los operadores, como `!`, `&`, `^`, `~` y `|`. Los programadores a los que no les gusten los enigmáticos símbolos de los operadores también pueden usar estas palabras clave. Hablamos sobre la palabra clave `mutable`, que nos permite cambiar un miembro de un objeto `const`. Anteriormente, esto se realizaba mediante el proceso de “convertir para eliminar la característica de `const`”, lo cual se considera una práctica peligrosa. También hablamos sobre los operadores de apuntador a miembro `.*` y `->.*`, la herencia múltiple (incluyendo el problema de la “herencia tipo diamante”) y las clases base `virtual`.

Apéndice A, Tabla de precedencia de operadores y asociatividad: presenta el conjunto completo de símbolos de operadores de C++, en donde cada operador aparece en una línea por sí solo, con su símbolo, su nombre y su asociatividad.

Apéndice B, Conjunto de caracteres ASCII: todos los programas en este libro utilizan el conjunto de caracteres ASCII, que se presenta en este apéndice.

Apéndice C, Tipos fundamentales: lista los tipos fundamentales de C++.

Apéndice D, Sistemas numéricos: habla sobre los sistemas numéricos binario, octal, decimal y hexadecimal. Considera cómo convertir números de una base a otra, y explica las representaciones binarias de complemento a uno y complemento a dos.

Apéndice E, Temas sobre código heredado de C: presenta material adicional, incluyendo varios temas avanzados que por lo general no se cubren en cursos de introducción. Mostramos cómo redirigir la entrada de un programa para que provenga de un archivo, cómo redirigir la salida de un programa para colocarla en un archivo, cómo redirigir la salida de un programa para que sea la entrada de otro programa (canalización) y cómo adjuntar la salida de un programa a un archivo existente. Desarrollamos funciones que utilizan listas de argumentos de longitud variable y mostramos cómo pasar argumentos de línea de comandos a la función `main`, para utilizarlos en un programa. Hablamos sobre cómo compilar programas cuyos componentes se espacien en varios archivos, registrar funciones con `atexit` para ejecutarlas cuando termina un programa, y terminar la ejecución de un programa mediante la función `exit`. También hablamos sobre los calificadores de tipo `const` y `volatile`, cómo especificar el tipo de una constante numérica mediante el uso de los sufijos enteros y de punto flotante, utilizar la biblioteca de manejo de señales para atrapar eventos inesperados, crear y utilizar arreglos dinámicos mediante `calloc` y `realloc`, usar uniones como una técnica para ahorrar espacio y usar especificaciones de enlace cuando se van a enlazar programas de C++ con código heredado de C. Como el título sugiere, este apéndice está diseñado principalmente para los programadores de C++ que van a trabajar con código heredado de C, ya que la mayoría de los programadores de C++ lo hacen en cierto punto de sus carreras.

Apéndice F, Preprocesador: habla sobre las directivas del preprocesador. El apéndice incluye información más completa acerca de la directiva `#include`, lo cual hace que se incluya una copia de un archivo especificado en vez de la directiva antes de compilar el archivo, y la directiva `#define` que crea constantes y macros simbólicas. El apéndice explica la compilación condicional, la cual nos permite controlar la ejecución de las directivas del preprocesador y la compilación del código del programa. Hablamos también sobre el operador `#` que convierte su operando en una cadena, y el operador `##` que concatena dos tokens. Se presentan las diversas constantes simbólicas predefinidas del preprocesador: `_LINE_`, `_FILE_`, `_DATE_`, `_STDC_`, `_TIME_` y `_TIMESTAMP_`. Por último hablamos sobre la macro `assert` del archivo de encabezado `<cassert>`, que es imprescindible para la prueba, depuración, verificación y validación de programas.

Apéndice G, Código del caso de estudio del ATM: contiene la implementación de nuestro ejemplo práctico acerca del diseño orientado a objetos con el UML. Este apéndice se describe en las generalidades del ejemplo práctico (que veremos en breve).

Apéndice H, UML 2: tipos de diagramas adicionales: presenta las generalidades de los tipos de diagramas de UML 2 que no se encuentran en el Ejemplo práctico de DOO/UML.

Apéndice I, Uso del depurador de Visual Studio: demuestra las características clave del Depurador de Visual Studio, el cual permite a un programador supervisar la ejecución de las aplicaciones, para localizar y eliminar los errores lógicos. Este apéndice presenta instrucciones paso a paso, para que los estudiantes aprendan a utilizar el depurador en forma práctica.

Apéndice J, Uso del depurador de GNU C++: demuestra las características clave del Depurador de GNU C++, el cual permite a un programador supervisar la ejecución de las aplicaciones, para localizar y eliminar los errores lógicos.

Este apéndice presenta instrucciones paso a paso, para que los estudiantes aprendan a utilizar el depurador en forma práctica.

Bibliografía: La bibliografía enlista muchos libros y artículos para alentar a los estudiantes a leer más acerca de C++ y POO.

Índice: el extenso índice permite al lector localizar mediante palabras clave cualquier término o concepto en el libro.

Diseño orientado a objetos de un ATM con el UML: un recorrido a través del Ejemplo práctico opcional de ingeniería de software

En esta sección daremos un paseo por el ejemplo práctico opcional de diseño orientado a objetos con UML que se incluye en este libro. Este paseo presenta las generalidades acerca del contenido de las nueve secciones del Ejemplo práctico de Ingeniería de Software (en los capítulos 1 a 7, 9 y 13). Después de completar este ejemplo práctico, el lector estará muy familiarizado con los procesos cuidadosamente revisados de diseño orientado a objetos e implementación, para una aplicación de considerable complejidad en C++.

El diseño que se presenta en el ejemplo práctico del ATM se desarrolló en Deitel & Associates, Inc, y fue revisado detalladamente por un distinguido equipo de profesionales de la industria y profesores. Construimos este diseño para cumplir con los requerimientos de las secuencias de los cursos de introducción. Los sistemas de ATM reales que utilizan los bancos y sus clientes en todo el mundo se basan en diseños más sofisticados, los cuales toman en cuenta muchas cuestiones más de las que hemos considerado aquí. Nuestro principal objetivo durante el proceso de diseño fue crear un diseño simple que fuera claro para los principiantes en la POO y el UML, pero que pudiera demostrar los conceptos clave de la POO y las técnicas de modelado del UML. Trabajamos duro para mantener el tamaño del diseño y del código relativamente reducido, de manera que tuviera un buen desempeño en la secuencia de un curso de introducción.

Sección 1.21, Ejemplo práctico de Ingeniería de Software: introducción a la tecnología de objetos y el UML: presenta el ejemplo práctico de diseño orientado a objetos con el UML. Esta sección introduce los conceptos básicos y la terminología de la tecnología de objetos, incluyendo las clases, los objetos, el encapsulamiento, la herencia y el polimorfismo. Hablamos sobre la historia del UML. Ésta es la única sección obligatoria del ejemplo práctico.

Sección 2.8, (Opcional) Ejemplo práctico de Ingeniería de Software: Cómo examinar la especificación de requerimientos del ATM: habla sobre una *especificación de requerimientos*, que especifica los requerimientos para un sistema que vamos a diseñar e implementar: el software para una máquina de cajero automático (ATM) simple. Investigamos la estructura y el comportamiento de los sistemas orientados a objetos en general. Hablamos acerca de cómo el UML facilitará el proceso de diseño en secciones subsiguientes del Ejemplo práctico de Ingeniería de Software, al proporcionar varios tipos adicionales de diagramas para modelar nuestro sistema. Incluimos una lista de URLs y referencias bibliográficas acerca del diseño orientado a objetos con el UML. Hablamos sobre la interacción entre el sistema ATM especificado por la especificación de requerimientos y el usuario. En especial, investigamos los escenarios que pueden ocurrir entre el usuario y el sistema en sí; a éstos les llamamos *casos-uso*. Modelamos estas interacciones mediante el uso de *diagramas de caso-uso* del UML.

Sección 3.11, (Opcional) Ejemplo práctico de Ingeniería de Software: identificación de las clases en la especificación de requerimientos del ATM: aquí empezamos a diseñar el sistema ATM. Identificamos sus clases, o “bloques de construcción”, extrayendo los sustantivos y frases nominales de la especificación de requerimientos. Ordenamos estas clases en un diagrama de clases de UML que describe la estructura de las clases de nuestra simulación. Este diagrama de clases también describe las relaciones, conocidas como *asociaciones*, entre las clases.

Sección 4.13, (Opcional) Ejemplo práctico de Ingeniería de Software: identificación de los atributos de las clases en el sistema ATM: se enfoca en los atributos de las clases descritas en la sección 3.11. Una clase contiene tanto *atributos* (datos) como *operaciones* (comportamientos). Como veremos en secciones posteriores, los cambios en los atributos de un objeto comúnmente afectan su comportamiento. Para determinar los atributos de las clases en nuestro ejemplo práctico, extraemos los adjetivos que describen a los sustantivos y frases nominales (que definen nuestras clases) de la especificación de requerimientos, y después colocamos estos atributos en el diagrama de clases que creamos en la sección 3.11.

Sección 5.11, (Opcional) Ejemplo práctico de Ingeniería de Software: identificación de los estados y las actividades de los objetos en el sistema ATM: habla acerca de cómo un objeto, en cualquier momento dado, ocupa una condición específica conocida como *estado*. Una *transición de estado* ocurre cuando ese objeto recibe un mensaje para que cambie de estado. UML proporciona el *diagrama de máquinas de estado*, el cual identifica el conjunto de posibles estados que puede ocupar un objeto y modela las transiciones de estado de ese objeto. Un objeto también tiene una *actividad*: el trabajo que desempeña durante su tiempo de vida. UML proporciona el *diagrama de actividad*: un diagrama de flujo que modela la actividad de un objeto. En esta sección utilizamos ambos tipos de diagramas para empezar a modelar los aspectos específicos del comportamiento de nuestro sistema ATM, como la forma en que el ATM lleva a cabo una transacción de retiro, y cómo responde cuando el usuario es autenticado.

Sección 6.22, (Opcional) Ejemplo práctico de Ingeniería de Software: identificación de las operaciones de las clases en el sistema ATM: identifica las operaciones, o servicios, de nuestras clases. Extraemos de la especificación de requerimientos los verbos y las frases nominales que especifican las operaciones para cada clase. Después modificamos el diagrama de clases de la sección 3.11 para incluir cada operación con su clase asociada. En este punto en el ejemplo práctico, hemos recopilado toda la información posible de la especificación de requerimientos. Sin embargo, a medida que los siguientes capítulos introduzcan temas como la herencia, modificaremos nuestras clases y diagramas.

Sección 7.12, (Opcional) Ejemplo práctico de Ingeniería de Software: colaboración entre los objetos en el sistema ATM: proporciona un “bosquejo” del modelo de nuestro sistema ATM. En esta sección vemos cómo funciona. Investigamos el comportamiento de la simulación mediante una discusión acerca de las *colaboraciones*: mensajes que los objetos se envían entre sí para comunicarse. Las operaciones de las clases que descubrimos en la sección 6.22 resultan ser las colaboraciones entre los objetos en nuestro sistema. Determinamos las colaboraciones y después las recolectamos en un *diagrama de comunicaciones*: el diagrama de UML para modelar las colaboraciones. Este diagrama revela cuáles objetos son los que colaboran, y cuándo lo hacen. Presentamos un diagrama de comunicaciones de las colaboraciones entre los objetos, para realizar una consulta de saldo en el ATM. Después presentamos el *diagrama de secuencia* de UML, para modelar las interacciones en un sistema. Este diagrama enfatiza el orden cronológico de los mensajes. Un diagrama de secuencia modela la forma en que los objetos del sistema interactúan para llevar a cabo transacciones de retiro y de depósito.

Sección 9.11, (Opcional) Ejemplo práctico de Ingeniería de Software: inicio de la programación de las clases del sistema ATM: detiene un momento el diseño del comportamiento del sistema. Empezamos el proceso de implementación para hacer énfasis en el material descrito en el capítulo 9. Mediante el uso del diagrama de clases de UML de la sección 3.11 y los atributos y operaciones que se describen en las secciones 4.13 y 6.22, mostramos cómo implementar una clase en C++ a partir de un diseño. No implementamos todas las clases, ya que todavía no completamos el proceso de diseño. Trabajando con base en nuestros diagramas de UML, creamos código para la clase *Retiro*.

Sección 13.10, (Opcional) Ejemplo práctico de Ingeniería de Software: incorporación de la herencia en el sistema ATM: continúa nuestra discusión sobre la programación orientada a objetos. Aquí consideramos la herencia: las clases que comparten características comunes pueden heredar atributos y operaciones de una clase “base”. En esta sección investigamos cómo se puede beneficiar nuestro sistema ATM del uso de la herencia. Documentamos nuestros descubrimientos en un diagrama de clases que modela las relaciones de herencia; UML se refiere a estas relaciones como *generalizaciones*. Modificamos el diagrama de clases de la sección 3.11 mediante el uso de la herencia para agrupar clases con características similares. En esta sección concluye el diseño de la parte correspondiente al modelo de nuestra simulación. En el apéndice G implementamos por completo este modelo, en 877 líneas de código de C++.

Apéndice G, Código del caso de estudio del ATM: la mayor parte del ejemplo práctico se enfoca en diseñar el modelo (es decir, los datos y la lógica) del sistema ATM. En este apéndice, implementamos ese modelo en C++. Utilizando todos los diagramas de UML que creamos, presentamos las clases de C++ necesarias para implementar el modelo. Aplicamos los conceptos del diseño orientado a objetos con el UML y la programación orientada a objetos en C++ que aprendió en los capítulos. Al final de este apéndice, los estudiantes habrán completado el diseño y la implementación de un sistema real, y deberán sentirse con la confianza de lidiar con sistemas más grandes, como los que construyen los ingenieros de software profesionales.

Apéndice H, UML 2: tipos de diagramas adicionales: presenta las generalidades acerca de los tipos de diagramas de UML 2 que no se utilizan en el Ejemplo práctico de POO/UML.

Recursos para los estudiantes

Hay muchas herramientas de desarrollo disponibles para C++. Escribimos este libro basándonos principalmente en el software Visual C++ Express Edition disponible en forma gratuita a través de Microsoft (y que se incluye en el CD que viene con este libro) y en el software gratuito GNU C++, que ya se encuentra instalado en la mayoría de los sistemas Linux, y puede instalarse en sistemas Mac OS X también. Puede aprender más acerca de Visual C++ Express en msdn.microsoft.com/vstudio/express/visualc (para la versión en español, visite el sitio www.microsoft.com/spanish/msdn/vstudio/express/VC/default.mspx). Puede aprender más acerca de GNU C++ en gcc.gnu.org. Apple incluye el software GNU C++ en sus herramientas de desarrollo Xcode, que los usuarios de Mac OS X pueden descargar del sitio developer.apple.com/tools/xcode.

Hay recursos y descargas de software adicionales disponibles en nuestro Centro de recursos de C++:

www.deitel.com/cplusplus/

y en el sitio Web para este libro

www.pearsoneducacion.net/deitel o directamente en
www.deitel.com/books/cpphp6/

Para obtener una lista de los demás compiladores que se pueden descargar en forma gratuita, visite los siguientes sitios:

www.thefreecountry.com/developercity/ccompilers.shtml
www.compilers.net

Advertencias y mensajes de error en compiladores anteriores de C++

Los programas en este libro están diseñados para usarse con compiladores que tengan soporte para el C++ estándar. Sin embargo, existen variaciones entre los compiladores que pueden producir advertencias o errores ocasionales. Además, aunque el estándar especifica varias situaciones que requieren la generación de errores, no especifica los mensajes que deben emitir los compiladores. Los mensajes de advertencia y error pueden variar de un compilador a otro; esto es normal.

Algunos compiladores anteriores de C++, como Microsoft Visual C++ 6, Borland C++ 5.5 y varias versiones anteriores de GNU C++ generan mensajes de error o de advertencia en lugares en los que los compiladores nuevos no lo hacen. Aunque la mayoría de los ejemplos de este libro funcionarán con estos compiladores antiguos, algunos ejemplos requieren pequeñas modificaciones para poder funcionar con esos compiladores.

Observaciones acerca de las declaraciones using y las funciones de la Biblioteca estándar de C

La Biblioteca estándar de C++ incluye las funciones de la Biblioteca estándar de C. De acuerdo con el documento del estándar de C++, el contenido de los archivos de encabezado que provienen de la Biblioteca estándar de C forman parte del espacio de nombres “`std`”. Algunos compiladores (antiguos y nuevos) generan mensajes de error al encontrar declaraciones `using` para las funciones de C.

C++ Multimedia Cyber Classroom, 6/e

Este libro incluye contenido multimedia interactivo gratuito, basado en Web y con gran cantidad de material de audio, como complemento para el libro (*C++ Multimedia Cyber Classroom, 6/e*), disponible en los libros nuevos. Nuestro *Cyber Classroom* basado en Web incluye recorridos de audio de los ejemplos de código de los capítulos 1 a 14, soluciones a casi la mitad de los ejercicios del libro y otras cosas más. Todo en inglés. Para obtener más información acerca del *Cyber Classroom* basado en Web, visite el sitio

www.prenhall.com/deitel/cyberclassroom/

A los estudiantes que utilizan nuestros Cyber Classrooms les gusta sus capacidades de interactividad y referencia. Los profesores nos comentan que los estudiantes disfrutan usar el Cyber Classroom y, en consecuencia, invierten más tiempo en los cursos, con lo cual dominan más cantidad de material que en los cursos que se basan sólo en el libro de texto.

Recursos para los instructores

Este libro tiene muchos recursos para los instructores. El *Centro de recursos para instructores* de Prentice Hall contiene el *Manual de soluciones* con soluciones para la gran mayoría de los ejercicios de fin de capítulo, un *Archivo de elementos de prueba* de preguntas de opción múltiple (aproximadamente dos por cada sección del libro) y diapositivas de PowerPoint® que contienen todo el código y las figuras del libro(en inglés), además de los elementos en viñetas que sintetizan los puntos clave del libro. Los instructores pueden personalizar las diapositivas.

Boletín de correo electrónico gratuito Deitel® Buzz Online

Cada semana, el boletín de correo electrónico *Deitel® Buzz Online* anuncia nuestro(s) Centro(s) de Recursos más reciente(s) e incluye comentarios acerca de las tendencias y desarrollos en la industria, vínculos a artículos y recursos gratuitos de nuestros libros publicados y de las próximas publicaciones, itinerarios de lanzamiento de productos, fe de erratas, retos, anécdotas, información sobre nuestros cursos de capacitación corporativa impartidos por instructores y mucho más. Es también una buena forma de que usted se mantenga actualizado acerca de todo lo relacionado con *Cómo programar en C++, 6/e*. Para suscribirse, visite la página Web:

www.deitel.com/newsletter/subscribe.html

Centros de Recursos en línea de Deitel

Nuestro sitio Web www.deitel.com proporciona Centros de Recursos acerca de varios temas, incluyendo: lenguajes de programación, software, Web 2.0, negocios en Internet y proyectos de código fuente abierto (figura 2). Los Centros de Recursos surgieron de la investigación que hemos realizado para nuestros libros y asuntos de negocios. Hemos encontrado muchos recursos excepcionales, incluyendo: tutoriales, documentación, descargas de software, artículos, blogs, videos, ejemplos de código, libros, libros electrónicos y mucho más. La mayoría de ellos son gratuitos. Como tributo a Web 2.0, compartimos estos recursos con la comunidad mundial. Los Centros de Recursos de Deitel son un punto de

Centros de Recursos de Deitel		
Programación	Microsoft	Google Analytics
.NET	.NET	Google Services
.NET 3.0	.NET 3.0	Publicidad por Internet
Ajax	ASP.NET	Iniciativa de negocios en Internet
Apex, Lenguaje de programación bajo demanda	C#	Relaciones públicas en Internet
ASP.NET	DotNetNuke (DNN)	Creación de vínculos
C	Internet Explorer 7	Podcasting
C#	Silverlight	Optimización de motores de búsqueda
C++	Visual Basic	Mapas de sitios
Bibliotecas Boost de C++	Visual C++	Analítica de Web
Programación de juegos en C++	Windows Vista	Monetización de sitios Web
Motores de búsqueda de código y sitios Web de código		
Programación de juegos de computadora		
CSS 2.1	Java	Código fuente abierto
Flash 9	Java	Apache
Flex	Pruebas de certificación y valoración de Java	DotNetNuke (DNN)
Java	Patrones de diseño en Java	Eclipse
Pruebas de certificación y valoración de Java	Java EE 5	Firefox
Patrones de diseño en Java	Java SE 6	Linux
Java EE 5	JavaFX	MySQL
Java SE 6	JavaScript	Código fuente abierto
JavaFX		Perl
JavaScript	Web 2.0	PHP
OpenGL	Servicios de alerta	Python
Perl	Economía de atención	Ruby
PHP	Blogging	
Proyectos de programación	Creación de comunidades Web	
Python	Contenido generado por la comunidad	
Ruby	Google Base	Otros temas
Silverlight	Google Video	Juegos de computadora
Visual Basic	Google Web Toolkit	Trabajos relacionados con la computación
Visual C++	Video en Internet	Gadgets y Gizmos
Servicios Web	Joost	Sudoku
Tecnologías Web 3D	Mashups	
XML	Microformatos	
	Ning	
Software	Sistemas de recomendación	
Apache	RSS	
DotNetNuke (DNN)	Skype	
Eclipse	Medios sociales	
Firefox	Redes sociales	
Internet Explorer 7	Software como servicio (SaaS)	
Linux	Mundos Virtuales	
MySQL	Web 2.0	
Código fuente abierto	Web 3.0	
Motores de búsqueda	Widgets	
Wikis	Wikis	
Windows Vista		
	Negocios en Internet	
	Programas de afiliados	
	Google Adsense	

Figura 2 | Centros de Recursos de Deitel.

inicio para que el lector investigue por su cuenta. Le ayudamos a involucrarse en la enorme cantidad de contenido en Internet, al proporcionarle vínculos a los recursos más valiosos. Cada semana anunciamos nuestros Centros de Recursos más recientes en nuestro boletín de correo electrónico gratuito, *Deitel® Buzz Online* (www.deitel.com/newsletter/subscribe.html). Los siguientes Centros de Recursos pueden serle de interés a la hora de estudiar *Cómo programar en C++, 6/e*:

- C++
- Bibliotecas Boost de C++
- Programación de juegos en C++
- Motores de búsqueda de código y sitios Web de código
- Programación de juegos de computadora
- Trabajos relacionados con la computación
- Código fuente abierto
- Proyectos de programación
- Eclipse
- Linux
- .NET
- Windows Vista

Reconocimientos

Es un gran placer para nosotros reconocer el esfuerzo de mucha gente, cuyos nombres quizás no aparezcan en la portada, pero cuyo arduo trabajo, cooperación, amistad y comprensión fue crucial para la elaboración de este libro. Mucha gente en Deitel & Associates, Inc. dedicó largas horas a este proyecto; queremos agradecer en especial a Abbey Deitel y Barbara Deitel.

También agradecemos a uno de los participantes de nuestro programa de Pasantía con Honores, que contribuyó a esta publicación: Greg Ayer, con especialidad en ciencias computacionales en la Universidad Northeastern.

Somos afortunados al haber trabajado en este proyecto con un talentoso y dedicado equipo de editores profesionales en Prentice Hall. Apreciamos el extraordinario esfuerzo de Marcia Horton, Directora Editorial de la División de Ingeniería y Ciencias Computacionales de Prentice Hall. Carole Snyder y Dolores Mars hicieron un excelente trabajo al reclutar el equipo de revisión del libro y administrar el proceso de revisión. Francesco Santalucia (un artista independiente) y Kristine Carney de Prentice Hall hicieron un maravilloso trabajo al diseñar la portada del libro; nosotros proporcionamos el concepto y ellos lo hicieron realidad. Vince O'Brien, Scout Disanno, Bob Engelhardt, y Marta Samsel hicieron un extraordinario trabajo al administrar la producción del libro.

Deseamos reconocer el esfuerzo de nuestros revisores. Al adherirse a un estrecho itinerario, scrutinizaron el texto y los programas, proporcionando innumerables sugerencias para mejorar la precisión e integridad de la presentación.

Apreciamos con sinceridad los esfuerzos de nuestros revisores de post-publicación de la quinta edición, y nuestros revisores de la sexta edición:

Revisores de Cómo programar en C++, 6/e

Revisores académicos y de la industria: Dr. Richard Albright (Goldey-Beacom College), William B. Higdon (Universidad de Indianápolis), Howard Hinnant (Apple), Anne B. Horton (Lockheed Martin), Terrell Hull (Logicalis Integration Solutions), Rex Jaeschke (Consultor independiente), Maria Jump (Universidad de Texas en Austin), Geoffrey S. Knauth (GNU), Don Kostuch (Consultor independiente), Colin Laplace (Consultor de software independiente), Stephan T. Lavavej (Microsoft), Amar Raheja (Universidad Politécnica Estatal de California, Pomona), G. Anthony Reina (Universidad de Maryland University College, Europa), Daveed Vandevoorde (Comité de estándares de C++), Jeffrey Wiener (DEKA Research & Development Corporation, New Hampshire Community Technical College) y Chad Willwerth (Universidad de Washington, Tacoma). **Revisores de Ogre:** Casey Borders (Sensis Corp.), Gregory Junker (Autor de *Pro OGRE3D Programming*, Apress Books), Mark Pope (THQ, Inc.) y Steve Streeting (Torus Know Software, Ltd.). **Revisores de Boost/C++0X:** Edgard Brey (Kohler Co.), Jeff Garland (Boost.org), Douglas Gregor (Universidad de Indiana) y Björn Karlsson (Autor de *Beyond the C++ Standard Library: An Introduction to Boost*, Addison-Wesley/Readsoft, Inc.).

Revisores de Cómo programar en Java, 5/e

Revisores académicos: Richard Albright (Colegio Goldey-Beacom), Karen Arlien (Colegio Estatal de Bismarck), David Branigan (Universidad DeVry, Illinois), Jimmy Chen (Colegio Comunitario de Salt Lake), Martin Dulberg (Universidad Estatal de Carolina del Norte), Ric Herishman (Colegio Comunitario de Virginia del Norte), Richard Holladay (San Diego Mesa College), William ONG (Universidad de Loyola), Earl LaBatt (OPNET Technologies, Inc./Universidad de New Hampshire), Brian Larson (Modesto Junior College), Robert Myers (Universidad Estatal de Florida), Gavin Osborne (Saskatchewan Institute of Applied Science and Technology), Wolfgang Pelz (La Universidad de Akron) y Donna Reese (Universidad Estatal de Mississippi). **Revisores de la industria:** Curtis Green (Boeing Integrated Defense Systems), Mahesh Hariharan (Microsoft), James Huddleston (Consultor independiente), Ed James-Beckham (Borland Software Corporation), Don Kostuch (Consultor independiente), Meng Lee (Hewlett-Packard), Kriang Lerdsuwanakij (Siemens Limited), William Mike Millar (Edison Design Group, Inc.), Mark Schimmel (Borland Internacional), Vicki Scout (Metrowerks), James Snell (Boeing Integrated Defense Sistemas) y Raymond Stephenson (Microsoft). **Revisores del ejemplo práctico opcional de Ingeniería de Software de POO/UML:** Sinan Si Alhir (Consultor independiente), Karen Arlien (Colegio Estatal de Bismarck), David Branigan (Universidad DeVry, Illinois), Martin Dulberg (Universidad Estatal de Carolina del Norte), Ric Eximan (Colegio Comunitario de Virginia del Norte), Richard Holladay (San Diego Mesa College), Earl LaBatt (OPNET Technologies, Inc./Universidad de New Hampshire), Brian Larson (Modesto Junior College), Gavin Osborne (Saskatchewan Institute of Applied Science and Technology), Praveen Sadhu (Infodat Internacional, Inc.), Cameron Skinner (Embarcadero Technologies, Inc./OMG) y Steve Tockey (Construx Software).

Estos profesores revisaron con sumo cuidado cada aspecto del libro y realizaron innumerables sugerencias para mejorar la precisión e integridad de la presentación.

Bueno ¡ahí lo tiene! Le damos la bienvenida al excitante mundo de C++ y la programación orientada a objetos. Esperamos que disfrute este análisis de la programación de computadoras contemporánea. ¡Buena suerte! A medida que lea el libro, apreciaremos con sinceridad sus comentarios, críticas, correcciones y sugerencias para mejorar el texto. Dirija toda su correspondencia a:

deitel@deitel.com

Le responderemos oportunamente y publicaremos las correcciones y aclaraciones en nuestro sitio Web,

www.deitel.com/books/cpphtp6/

¡Esperamos que disfrute aprendiendo con este libro tanto como nosotros disfrutamos el escribirlo!

*Paul J. Deitel
Dr. Harvey M. Deitel
Maynard, Massachusetts
Julio de 2007*

Acerca de los autores

Paul J. Deitel, CEO y Director Técnico de Deitel & Associates, Inc., es egresado del Sloan School of Management del MIT, en donde estudió Tecnología de la Información. Posee las certificaciones Programador Certificado en Java (Java Certified Programmer) y Desarrollador Certificado en Java (Java Certified Developer), y ha sido designado por Sun Microsystems como Java Champion. A través de Deitel & Associates, Inc., ha impartido cursos en Java, C, C++, C# y Visual Basic a clientes de la industria, incluyendo: IBM, Sun Microsystems, Dell, Lucent Technologies, Fidelity, NASA en el Centro Espacial Kennedy, el National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, Stratus, Cambridge Technology Partners, Open Environment Corporation, One Wave, Hyperion Software, Adra Systems, Entergy, CableData Systems, Nortel Networks, Puma, iRobot, Invensys y muchos más. También ha ofrecido conferencias de Java y C++ para la Boston Chapter de la Association for Computing Machinery. Él y su padre, el Dr. Harvey M. Deitel, son autores de los libros de programación con más ventas en el mundo.

El **Dr. Harvey M. Deitel**, Presidente y Consejero de Estrategia de Deitel & Associates, Inc., tiene 45 años de experiencia en el campo de la computación, esto incluye un amplio trabajo académico y en la industria. El Dr. Deitel tiene una licenciatura y una maestría por el MIT y un doctorado de la Universidad de Boston. Tiene 20 años de experiencia como profesor universitario, la cual incluye un puesto vitalicio y el haber sido presidente del departamento de Ciencias de la computación en el Boston College antes de fundar, con su hijo Paul J. Deitel, Deitel & Associates, Inc. Él y Paul son coautores de varias docenas de libros y paquetes multimedia, y piensan escribir muchos más. Los textos de los Deitel se han ganado el reconocimiento internacional y han sido traducidos al japonés, alemán, ruso, español, chino tradicional, chino simplificado, coreano, francés, polaco, italiano, portugués, griego, urdú y turco. El Dr. Deitel ha impartido cientos

de seminarios profesionales para grandes empresas, instituciones académicas, organizaciones gubernamentales y diversos sectores del ejército.

Acerca de Deitel & Associates, Inc.

Deitel & Associates, Inc. es una empresa reconocida a nivel mundial, dedicada al entrenamiento corporativo y la creación de contenido, con especialización en lenguajes de programación, a tecnología de software para Internet/World Wide Web, educación de tecnología de objetos y desarrollo de negocios por Internet a través de su Iniciativa de Negocios en Internet. La empresa proporciona cursos impartidos por instructores sobre la mayoría de los lenguajes y plataformas de programación, como C++, Java, Java Avanzado, C, C#, Visual C++, Visual Basic, XML, Perl, Python, tecnología de objetos y programación en Internet y World Wide Web. Los fundadores de Deitel & Associates, Inc. son el Dr. Harvey M. Deitel y Paul J. Deitel. Sus clientes incluyen muchas de las empresas más grandes del mundo, agencias gubernamentales, sectores del ejército e instituciones académicas. A lo largo de su sociedad editorial de 30 años con Prentice Hall, Deitel & Associates Inc. ha publicado libros de texto de vanguardia sobre programación, libros profesionales, multimedia interactiva en CD como los *Cyber Classrooms*, *Cursos Completos de Capacitación*, cursos de capacitación basados en Web y contenido electrónico para los populares sistemas de administración de cursos WebCT, Blackboard y CourseCompass de Pearson. Deitel & Associates, Inc. y los autores pueden ser contactados mediante correo electrónico en:

deitel@deitel.com

Para conocer más acerca de Deitel & Associates, Inc., sus publicaciones y su currículum mundial de la Serie de Capacitación Corporativa *Dive Into*®, visite:

www.deitel.com

y suscríbase al boletín gratuito de correo electrónico, *Deitel® Buzz Online*, en:

www.deitel.com/newsletter/subscribe.html

Puede verificar la lista creciente de Centros de Recursos Deitel en:

www.deitel.com/resourcecenters.html

Los individuos que deseen comprar publicaciones Deitel pueden hacerlo en:

www.deitel.com/books/index.html

Las empresas, el gobierno, las instituciones militares y académicas que deseen realizar pedidos en masa deben hacerlo directamente con Prentice Hall. Para obtener más información, visite:

www.prenhall.com/mischtm/support.html#order

Antes de empezar

Por favor siga las instrucciones en esta sección para descargar los ejemplos del libro, antes de que empiece a utilizarlo.

Descargue el código de ejemplo de *Cómo programar en C++, 6/e*

Los ejemplos para *Cómo programar en C++, sexta edición* (en inglés) se pueden descargar como un archivo ZIP de www.deitel.com/books/cpphtp6/. Después de registrarse y de iniciar sesión, haga clic en el vínculo para los ejemplos en **Download Code Examples and Other Premium Content for Registered Users**. Guarde el archivo ZIP en una ubicación que no olvide. Extraiga los archivos de ejemplo en su disco duro, utilizando un programa extractor de archivos ZIP, como WinZip (www.winzip.com). [Nota: si trabaja en un laboratorio de computadoras, consulte con su instructor para determinar en dónde puede guardar el código de ejemplo]. También puede bajar el código de todos los ejercicios, en español, desde la página de este libro.

Instale/elija un compilador

El CD que se incluye con este libro contiene la versión del entorno de desarrollo integrado (IDE) Microsoft® Visual C++® Express que estaba disponible al momento de imprimir esta edición. Para obtener la versión más reciente de este IDE, visite msdn.microsoft.com/vstudio/express/visualc/default.aspx, o www.microsoft.com/spanish/msdn/vstudio/express/VC/default.mspx, para la versión en español.

Si tiene una computadora con Windows XP o Windows Vista, puede instalar Visual C++ Express para editar, compilar, ejecutar, depurar y modificar sus programas. Cuando inserte el CD en su computadora, el programa de instalación de Visual C++ Express iniciará en forma automática; siga las instrucciones en pantalla para instalarlo. Le recomendamos que utilice las opciones predeterminadas y que seleccione la opción para instalar la documentación.

Si su computadora tiene Linux o Mac OS X, tal vez ya tenga instalado el compilador de línea de comandos GNU C++. Hay muchos otros compiladores e IDEs para C++. En nuestro Centro de Recursos en

www.deitel.com/cplusplus/

proporcionamos vínculos a diversas herramientas de desarrollo de C++ para las plataformas Windows, LINUX y Mac OS X.

Este Centro de Recursos también contiene vínculos a tutoriales en línea, que le ayudarán a empezar con varias herramientas de desarrollo de C++.

Ahora está listo para empezar sus estudios de C++ con este libro. ¡Esperamos que lo disfrute! Puede contactarnos fácilmente en deitel@deitel.com. Si tiene alguna pregunta, envíenos un correo electrónico. Le responderemos a la brevedad.



La principal cualidad del lenguaje es la claridad.

—Galen

Nuestra vida es malgastada por el detalle... simplificar, simplificar.

—Henry David Thoreau

Tenía un maravilloso talento para empacar estrechamente el pensamiento, haciéndolo portable.

—Thomas B. Macaulay

El hombre sigue siendo la computadora más extraordinaria de todas.

—John F. Kennedy



Introducción a las computadoras, Internet y World Wide Web

OBJETIVOS

En este capítulo aprenderá a:

- Comprender los conceptos básicos de hardware y software.
- Conocer los conceptos básicos de la tecnología de objetos, como clases, objetos, atributos, comportamientos, encapsulamiento, herencia y polimorfismo.
- Familiarizarse con los distintos tipos de lenguajes de programación.
- Comprender un típico entorno de desarrollo de programas en C++.
- Conocer la historia del UML: el lenguaje de diseño orientado a objetos estándar en la industria.
- Conocer la historia de Internet y World Wide Web, junto con el fenómeno Web 2.0.
- Probar aplicaciones en C++, en dos entornos populares: GNU C++ ejecutándose en Linux, y Microsoft Visual C++® en Windows® XP.
- Conocer qué es el código fuente abierto, y dos bibliotecas de código fuente abierto de C++ populares: Ogre para programación de gráficos y juegos, y Boost para mejorar ampliamente las capacidades de la Biblioteca estándar de C++.

- I.1** Introducción
- I.2** ¿Qué es una computadora?
- I.3** Organización de una computadora
- I.4** Los primeros sistemas operativos
- I.5** Computación personal, distribuida y cliente/servidor
- I.6** Internet y World Wide Web
- I.7** Web 2.0
- I.8** Lenguajes máquina, lenguajes ensambladores y lenguajes de alto nivel
- I.9** Historia de C y C++
- I.10** Biblioteca estándar de C++
- I.11** Historia de Java
- I.12** FORTRAN, COBOL, Pascal y Ada
- I.13** BASIC, Visual Basic, Visual C++, C# y .NET
- I.14** Tendencia clave de software: la tecnología de los objetos
- I.15** Entorno de desarrollo típico en C++
- I.16** Generalidades acerca de C++ y este libro
- I.17** Prueba de una aplicación en C++
- I.18** Tecnologías de software
- I.19** Programación de juegos con las bibliotecas Ogre
- I.20** Futuro de C++: Bibliotecas Boost de código fuente abierto, TR1 y C++0x
- I.21** Ejemplo práctico de Ingeniería de Software: introducción a la tecnología de objetos y el UML
- I.22** Repaso
- I.23** Recursos Web

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)

1.1 Introducción

¡Bienvenido a C++! Hemos trabajado duro para crear lo que creemos será una experiencia de aprendizaje informativa, divertida y retadora para usted. C++ es un poderoso lenguaje de programación de computadoras, apropiado para las personas con orientación técnica con poca (o ninguna) experiencia de programación, y para los programadores experimentados que desarrollan sistemas de información de tamaño considerable. *Cómo programar en C++, sexta edición* es una herramienta efectiva de aprendizaje para cada una de estas audiencias. Hemos agregado un nuevo capítulo de programación de juegos, para que los estudiantes, instructores y profesionales lo disfruten. ¡Nos gustaría ver los juegos que desarrolle!

La parte central del libro se enfoca en la claridad de los programas, a través de las técnicas comprobadas de la programación orientada a objetos. En este libro las clases y los objetos se presentan desde los primeros capítulos; los principiantes aprenderán programación de la manera correcta, desde el principio. La presentación es clara, simple y tiene muchas ilustraciones. Enseñamos las características de C++ en un contexto de programas completos y funcionales en C++, y mostramos los resultados que se producen cuando estos programas se ejecutan en una computadora; a esto le llamamos el **método de código activo** (LIVE-CODE™). Puede descargar los programas de ejemplo del sitio Web www.deitel.com/books/cpphtp6/.

Los primeros capítulos presentan los fundamentos de las computadoras, la programación de éstas y el lenguaje de programación C++, con lo cual se provee una base sólida para un análisis más detallado de C++ en los capítulos posteriores. Los programadores experimentados tienden a leer los primeros capítulos rápidamente, y descubren que el análisis de C++ en los capítulos posteriores es riguroso y retador.

La mayoría de las personas están familiarizadas con las emocionantes tareas que realizan las computadoras. Por medio de este libro de texto, usted aprenderá a programar las computadoras para que realicen esas tareas. A menudo las computadoras (conocidas comúnmente como **hardware**) se controlan mediante el **software** (las instrucciones que usted escribe para indicar a la computadora que realice **acciones** y tome **decisiones**). C++ es uno de los lenguajes para desarrollo de software más populares en la actualidad. Este libro ofrece una introducción a la programación en la versión de C++

estandarizado en Estados Unidos, a través del **Instituto nacional estadounidense de estándares (ANSI)** y en todo el mundo a través de los esfuerzos de la **Organización internacional para la estandarización (ISO)**.

El uso de las computadoras se está incrementando en casi cualquier campo de trabajo. Los costos de las computadoras se han reducido en forma dramática, debido al rápido desarrollo en la tecnología de hardware y software. Las computadoras que ocupaban grandes habitaciones y que costaban millones de dólares, hace algunas décadas, ahora pueden colocarse en las superficies de chips de silicio más pequeños que una uña, y con un costo de quizás unos cuantos dólares cada uno. (A esas enormes computadoras se les llamaba **mainframes**, y hoy en día se utilizan ampliamente versiones actualizadas en los negocios, el gobierno y la industria.) Por fortuna, el silicio es uno de los materiales más abundantes en el planeta (es uno de los ingredientes de la tierra común). La tecnología de los chips de silicio ha vuelto tan económica a la tecnología de la computación que cientos de millones de computadoras de uso general se encuentran actualmente ayudando a la gente de todo el mundo en empresas, en la industria, en el gobierno y en sus vidas.

A través de los años, muchos programadores aprendieron la metodología de programación conocida como programación estructurada. Usted aprenderá tanto la programación estructurada como la novedosa y excitante metodología de la programación orientada a objetos. ¿Por qué enseñamos ambas? La orientación a objetos es la metodología clave de programación utilizada hoy en día por los programadores. Usted creará y trabajará con muchos objetos de software en este libro. Sin embargo, descubrirá que la estructura interna de estos objetos se construye, a menudo, utilizando técnicas de programación estructurada. Además, la lógica requerida para manipular objetos se expresa algunas veces mediante la programación estructurada.

Para mantenerse al tanto de los desarrollos con C++ en Deitel & Associates, regístrate para recibir nuestro boletín de correo electrónico, *Deitel® Buzz Online* en

www.deitel.com/newsletter/subscribe.html

Puede revisar nuestra creciente lista de centros de recursos sobre C++ en

www.deitel.com/ResourceCenters.html

Algunos de los Centros de Recursos que serán de importancia para usted a medida que lea este libro son C++, Programación de juegos en C++, Bibliotecas Boost de C++, Motores de búsqueda de código y Sitios de código, Programación de juegos de computadora, Proyectos de programación, Eclipse, Linux, Código fuente abierto y Windows Vista. Cada semana anunciamos nuestros Centros de Recursos más recientes en el boletín de correo electrónico. La fe de erratas y las actualizaciones para este libro se publican en

www.deitel.com/books/cpphtp6/

Está a punto de comenzar una ruta de desafíos y recompensas. A medida que avance, si tiene dudas o preguntas, envíenos un correo a

deitel@deitel.com

Le responderemos con prontitud. Esperamos que disfrute aprender con *Cómo programar en C++, sexta edición*.

1.2 ¿Qué es una computadora?

Una **computadora** es un dispositivo capaz de realizar cálculos y tomar decisiones lógicas a velocidades de miles de millones de veces más rápidas que los humanos. Por ejemplo, muchas de las computadoras personales actuales pueden realizar varios miles de millones de sumas en un segundo. Una persona con una calculadora podría requerir toda una vida para realizar cálculos, ¡y aun así no podría completar tantos cálculos como los que puede realizar una computadora personal en un segundo! (Puntos a considerar: ¿cómo sabría que la persona sumó los números de manera correcta?, ¿cómo sabría que la computadora sumó los números de manera correcta?) ¡Las **supercomputadoras** actuales más rápidas pueden realizar *miles de millones* de sumas por segundo!

Las computadoras procesan los **datos** bajo el control de conjuntos de instrucciones llamadas **programas de computo**. Estos programas guían a la computadora a través de conjuntos ordenados de acciones especificadas por gente llamada **programadores de computadoras**.

Una computadora está compuesta por varios dispositivos (como teclado, monitor, ratón, discos, memoria, DVDs y unidades de procesamiento) conocidos como **hardware**. A los programas que se ejecutan en una computadora se les denomina **software**. Los costos de las piezas de hardware han disminuido de manera espectacular en años recientes, al punto en el que las computadoras personales se han convertido en artículos domésticos. En este libro aprenderá métodos comprobados que están reduciendo los costos de desarrollo del software: programación orientada a objetos y (en nuestro Ejemplo práctico opcional de Ingeniería de Software, acerca de la construcción de un cajero automático, en los capítulos 2 a 7, 9 y 13) diseño orientado a objetos.

1.3 Organización de una computadora

Independientemente de las diferencias en su apariencia física, casi todas las computadoras pueden representarse mediante seis **unidades lógicas** o secciones:

1. **Unidad de entrada.** Es la sección “receptora” de la computadora. Obtiene información (datos y programas de cómputo) desde varios **dispositivos de entrada** y pone esta información a disposición de las otras unidades para que pueda procesarse. La mayor parte de la información se introduce a través de los teclados y ratones. La información también puede introducirse de muchas otras formas, como hablar con su computadora, digitalizar imágenes, enviar fotos y videos, y mediante la recepción de información desde una red, como Internet.
2. **Unidad de salida.** Es la sección de “embarque” de la computadora. Toma información que ya ha sido procesada por ésta y la coloca en los diferentes **dispositivos de salida**, para que esté disponible fuera de la computadora. Hoy en día, la mayor parte de la información de salida de las computadoras se despliega en el monitor, se imprime en papel o se utiliza para controlar otros dispositivos. Las computadoras también pueden dar salida a información a través de redes como Internet.
3. **Unidad de memoria.** Es la sección de “almacén” de acceso rápido, pero con relativa baja capacidad. Almacena los programas de la computadora mientras se ejecutan. Retiene la información que se introduce a través de la unidad de entrada, para que la información pueda estar disponible de manera inmediata para procesarla cuando sea necesario. La unidad de memoria también retiene la información procesada hasta que ésta pueda ser colocada en los dispositivos de salida por la unidad de salida. Por lo general, la información en la unidad de memoria se pierde cuando se apaga la computadora. Con frecuencia, a esta unidad de memoria se le llama **memoria** o **memoria primaria**. [A través de la historia, a esta unidad se le ha llamado “memoria básica o magnética”, pero ese término está dejando de utilizarse en la actualidad].
4. **Unidad aritmética y lógica (ALU).** Es la sección de “manufactura” de la computadora. Es la responsable de realizar cálculos como suma, resta, multiplicación y división. Contiene los mecanismos de decisión que permiten a la computadora hacer operaciones como, por ejemplo, comparar dos elementos de la unidad de memoria para determinar si son iguales o no.
5. **Unidad central de procesamiento (CPU).** Es la sección “administrativa” de la computadora. Coordina y supervisa la operación de las demás secciones. La CPU indica a la unidad de entrada cuándo debe grabarse la información dentro de la unidad de memoria, a la ALU cuándo debe utilizarse la información de la unidad de memoria para los cálculos, y a la unidad de salida cuándo enviar la información desde la unidad de memoria hasta ciertos dispositivos de salida. Muchas de las computadoras actuales contienen múltiples CPUs y, por lo tanto, pueden realizar muchas operaciones de manera simultánea (a estas computadoras se les conoce como **multiprocesadores**).
6. **Unidad de almacenamiento secundario.** Es la sección de “almacén” de alta capacidad y de larga duración de la computadora. Los programas o datos que no se encuentran en ejecución por las otras unidades, normalmente se colocan en dispositivos de almacenamiento secundario, como el disco duro, hasta que son requeridos de nuevo, posiblemente horas, días, meses o incluso años después. El tiempo para acceder a la información en almacenamiento secundario es mucho mayor que el necesario para acceder a la de la memoria principal, pero el costo por unidad de memoria secundaria es mucho menor que el correspondiente a la unidad de memoria primaria. Los CDs y DVDs son ejemplos de dispositivos de almacenamiento secundario, los cuales pueden contener hasta cientos de millones de caracteres y miles de millones de caracteres, respectivamente.

1.4 Los primeros sistemas operativos

Las primeras computadoras eran capaces de realizar solamente una **tarea** o **trabajo** a la vez. A esta forma de operación de la computadora a menudo se le conoce como **procesamiento por lotes** (batch) de un solo usuario. La computadora ejecuta un solo programa a la vez, mientras procesa los datos en grupos o **lotes**. En estos primeros sistemas, los usuarios generalmente asignaban sus trabajos a un centro de cómputo que lo introducía en paquetes de tarjetas perforadas, y a menudo tenían que esperar horas, o incluso días, antes de que sus resultados impresos regresaran a sus escritorios.

Los **sistemas operativos** se desarrollaron para facilitar el uso de la computadora. Los primeros sistemas operativos incrementaron la transición entre trabajos, aumentando el procesamiento del monto de trabajo, o **rendimiento** que las computadoras podían procesar.

Conforme las computadoras se volvieron más poderosas, se hizo evidente que un proceso por lotes para un solo usuario era ineficiente, debido al tiempo que se malgastaba esperando a que los lentos dispositivos de entrada/salida

completaran sus tareas. Se pensó que era posible realizar muchos trabajos o tareas que podrían *compartir* los recursos de la computadora y lograr un uso más eficiente de ésta. A esto se le conoce como **multiprogramación**. La multiprogramación significa la operación simultánea de muchas tareas que compiten para compartir los recursos de la computadora. Con los primeros sistemas operativos con multiprogramación, los usuarios seguían enviando sus tareas en paquetes de tarjetas perforadas y esperaban horas, incluso hasta días, por los resultados.

En la década de los sesenta, varios grupos en la industria y en las universidades marcaron la pauta de los sistemas operativos de **tiempo compartido**. El tiempo compartido es un caso especial de la multiprogramación, en el cual los usuarios acceden a la computadora a través de terminales, que por lo general son dispositivos compuestos por un teclado y un monitor. Puede haber docenas o incluso cientos de usuarios compartiendo la computadora al mismo tiempo. La computadora en realidad no ejecuta los procesos de todos los usuarios a la vez. Lo que hace es ejecutar una pequeña porción del trabajo de un usuario y después procede a dar servicio al siguiente usuario, con la posibilidad de proporcionar el servicio a cada usuario varias veces por segundo. Así, los programas de los usuarios *aparentemente* se ejecutan de manera simultánea. Una ventaja del tiempo compartido es que el usuario recibe respuestas casi inmediatas a las peticiones.

1.5 Computación personal, distribuida y cliente/servidor

En 1977, Apple Computer popularizó el fenómeno de la **computación personal**. Las computadoras se hicieron lo suficientemente económicas para que la gente las pudiera adquirir para su uso personal o para negocios. En 1981, IBM, el vendedor de computadoras más grande del mundo, introdujo la Computadora Personal (PC) de IBM. Con esto se legitimó rápidamente la computación en las empresas, en la industria y en las organizaciones gubernamentales, donde se utilizaban ampliamente las mainframes de IBM.

Estas computadoras eran unidades “independientes” (la gente transportaba sus discos de un lado a otro para compartir información; a esto se le conoce comúnmente como “sneakernet”). Aunque las primeras computadoras personales no eran lo suficientemente poderosas para compartir el tiempo entre varios usuarios, estas máquinas podían interconectarse mediante redes computacionales, algunas veces a través de líneas telefónicas y otras mediante **redes de área local (LANs)** dentro de una empresa. Esto derivó en el fenómeno denominado **computación distribuida**, donde la computación de una empresa, en lugar de realizarse estrictamente dentro de un centro de cómputo, se distribuye mediante redes a los sitios donde se realiza el trabajo de la empresa. Las computadoras personales eran lo suficientemente poderosas para manejar los requerimientos de cómputo de usuarios individuales, y para manejar las tareas básicas de comunicación que involucraban la transferencia de información entre una computadora y otra, de manera electrónica.

Las computadoras personales actuales son tan poderosas como las máquinas de un millón de dólares de hace apenas unas décadas. Las máquinas de escritorio más poderosas (denominadas **estaciones de trabajo**) proporcionan a cada usuario enormes capacidades. La información se comparte fácilmente a través de redes de computadoras, donde algunas computadoras denominadas **servidores** (servidores de archivos, servidores de bases de datos, servidores Web, etc.) ofrecen un almacén común de datos que pueden ser utilizados por computadoras **cliente** distribuidas en toda la red, de ahí el término de **computación cliente/servidor**. C++ se está utilizando ampliamente para escribir software para redes de computadoras y para aplicaciones cliente/servidor distribuidas. Los sistemas operativos actuales más populares como UNIX, Linux, Mac OS X y Microsoft Windows proporcionan el tipo de capacidades que explicamos en esta sección.

1.6 Internet y World Wide Web

Internet (una red global de computadoras) se inició hace casi cuatro décadas; su patrocinio estuvo a cargo del Departamento de Defensa de Estados Unidos. Diseñada originalmente para conectar los sistemas de cómputo principales de aproximadamente una docena de universidades y organizaciones de investigación, Internet es actualmente utilizada por computadoras en todo el mundo.

Con la introducción de **World Wide Web** (que permite a los usuarios de computadora localizar y ver documentos basados en multimedia, sobre casi cualquier tema, a través de Internet), Internet se ha convertido explosivamente en uno de los principales mecanismos de comunicación en todo el mundo.

Internet y World Wide Web se encuentran, sin duda, entre las creaciones más importantes y profundas de la humanidad. En el pasado, la mayoría de las aplicaciones de computadora se ejecutaban en equipos que no estaban conectados entre sí. Las aplicaciones de la actualidad pueden diseñarse para intercomunicarse entre computadoras en todo el mundo. Internet mezcla las tecnologías de la computación y las comunicaciones. Facilita nuestro trabajo. Hace que la información esté accesible en forma instantánea y conveniente para todo el mundo. Hace posible que los individuos y negocios pequeños locales obtengan una exposición mundial. Está cambiando la forma en que se hacen los negocios. La gente puede buscar los mejores precios para casi cualquier producto o servicio. Los miembros de las comunidades con intereses especiales pueden mantenerse en contacto unos con otros. Los investigadores pueden estar inmediatamente al tanto de los últimos descubrimientos.

1.7 Web 2.0¹

En 2006, la persona del año era usted, según la revista *TIME* en su artículo “Person of the Year”.² En este artículo, Web 2.0 y el fenómeno social asociado se reconocieron como un desplazamiento de unos cuantos poderosos a muchos con poderes. Web 2.0 no tiene una sola definición, pero puede explicarse a través de una serie de tendencias en Internet, una de las cuales es el otorgamiento de poderes al usuario. Compañías como eBay se basan casi por completo en el **contenido generado por la comunidad**. Web 2.0 aprovecha la **inteligencia colectiva**, la idea de que la colaboración producirá ideas inteligentes. Por ejemplo, los **wikis** como la enciclopedia Wikipedia, permiten a los usuarios editar el contenido. El **marcado**, o etiquetado de contenido, es otra parte clave del tema de colaboración de Web 2.0, que puede verse en sitios como Flickr, un sitio Web para compartir fotos, y en del.icio.us, un sitio con enlaces a sitios favoritos sociales.

Los sitios de **redes sociales**, que llevan el registro de las relaciones interpersonales de los usuarios, han experimentando un extraordinario crecimiento como parte de Web 2.0. Los sitios como MySpace, Facebook y LinkedIn dependen en gran parte de los **efectos de red**, y atraen a los usuarios sólo si sus amigos o colegas son miembros también. De manera similar, los sitios de medios sociales como YouTube (un sitio de videos en línea) y Last.fm (una plataforma musical social), han obtenido una inmensa popularidad, en parte debido al aumento en la disponibilidad de **Internet de banda ancha**, que a menudo se le conoce como Internet de alta velocidad.

Los **blogs** (sitios Web que se caracterizan por mensajes cortos en orden cronológico inverso) se han convertido en un fenómeno social importante dentro de Web 2.0. Muchos bloggers se reconocen como parte de los medios, y las compañías están alcanzando la **blogósfera**, o comunidad de blogging, para rastrear las opiniones de los consumidores.

El aumento en la popularidad del software de **código fuente abierto** (un estilo para desarrollar software, en el cual los individuos y las compañías desarrollan, mantienen y evolucionan software a cambio del derecho de usar ese software para sus propios fines) ha contribuido a que sea más fácil y sencillo empezar compañías Web 2.0. Los **servicios Web** (componentes de software accesibles para aplicaciones [u otros componentes de software] a través de Internet) van en aumento, donde se da preferencia al “**webtop**” (escritorio web) en lugar del “**desktop**” (escritorio) en la mayor parte del nuevo desarrollo. Los **mashups** combinan dos o más aplicaciones Web existentes para servir un nuevo propósito, y dependen del acceso abierto a los servicios Web. Por ejemplo, **housingmaps.com** es un mashup de Google Maps y listados de bienes raíces en Craigslist. En nuestro libro *Internet & World Wide Web How to Program*, 4/e, describimos las tecnologías clave de Web 2.0, incluyendo: XML, RSS, Ajax, RIA, Podcasting, video en Internet y otras. También mencionamos muchas de las compañías clave de Web 2.0.

Muchas compañías de Web 2.0 utilizan la publicidad como fuente principal de monetización. Los programas de publicidad por Internet como **Google AdSense** relacionan a los publicistas con los propietarios de sitios Web. Otro modelo de monetización es el **contenido Premium**, el cual proporciona servicios o información adicionales por una cuota.

Web 3.0 se refiere al siguiente movimiento en el desarrollo Web; uno que realiza el potencial completo de Web. En su estado actual, Internet es una conglomeración gigante de sitios Web individuales con conexiones pobres. Web 3.0 resolverá esto al avanzar hacia la **Web semántica** (o “Web de significado”), en la cual Web se convierte en una base de datos gigante, en la que las computadoras pueden buscar información con significado.

Para obtener más información, consulte nuestro Centro de Recursos Web 2.0 en www.deitel.com/web2.0/.

1.8 Lenguajes máquina, lenguajes ensambladores y lenguajes de alto nivel

Los programadores escriben instrucciones en diversos lenguajes de programación, algunos de los cuales los comprende directamente la computadora, mientras que otros requieren pasos intermedios de **traducción**. En la actualidad se utilizan cientos de lenguajes de computación. Éstos se dividen en tres tipos generales:

1. Lenguajes de máquina
2. Lenguajes ensambladores
3. Lenguajes de alto nivel

1. O'Reilly, T. “What is Web 2.0: Design Patterns and Business Models for the Next Generation of Software.” Septiembre 2005 <<http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html?page=1>>.

2. Grossman, L. “Time's Person of the Year: You”. *Time*, Diciembre 2006 <<http://www.time.com/time/magazine/article/0,9171,1569514,00.html>>.

Cualquier computadora puede entender de manera directa sólo su propio **lenguaje de máquina**. El lenguaje de máquina es el “lenguaje natural” de una computadora, y como tal, está definido por el diseño del hardware de dicha computadora. [Nota: al lenguaje de máquina se le conoce comúnmente como **código objeto**. Este término es anterior al de “programación orientada a objetos”. Estos dos usos de “objeto” no están relacionados.] Por lo general, los lenguajes máquina consisten en cadenas de números (que finalmente se reducen a 1s y 0s) que instruyen a las computadoras para realizar sus operaciones más elementales, una a la vez. Los lenguajes máquina son **dependientes de la máquina** (es decir, un lenguaje de máquina en particular puede usarse solamente en un tipo de computadora). Dichos lenguajes son difíciles de comprender para los humanos, como se ilustra mediante la siguiente sección de uno de los primeros programas en lenguaje de máquina, el cual suma el pago de las horas extra al sueldo base y almacena el resultado en el sueldo bruto:

```
+1300042774
+1400593419
+1200274027
```

En esencia, la programación en lenguaje de máquina era demasiado lenta, tediosa y propensa a errores para la mayoría de los programadores. En lugar de utilizar las cadenas de números que las computadoras podían entender directamente, los programadores empezaron a utilizar abreviaturas del inglés para representar las operaciones elementales. Estas abreviaturas formaron la base de los **lenguajes ensambladores**. Los **programas traductores** conocidos como **ensambladores** se desarrollaron para convertir los primeros programas en lenguaje ensamblador a lenguaje de máquina, a la velocidad de la computadora. La siguiente sección de un programa en lenguaje ensamblador también suma el pago de las horas extras al sueldo base y almacena el resultado en el sueldo bruto:

```
load    sueldobase
add     sueldoextra
store   sueldobruto
```

Aunque este código es más claro para los humanos, las computadoras no lo pueden entender sino hasta que se traduce en lenguaje de máquina.

El uso de las computadoras se incrementó rápidamente con la llegada de los lenguajes ensambladores, pero los programadores aún requerían de muchas instrucciones para llevar a cabo incluso hasta las tareas más simples. Para agilizar el proceso de programación se desarrollaron los **lenguajes de alto nivel**, donde podían escribirse instrucciones individuales para realizar tareas importantes. Los **programas traductores**, denominados **compiladores**, convierten programas en lenguaje de alto nivel a lenguaje de máquina. Los lenguajes de alto nivel permiten a los programadores escribir instrucciones que son muy similares al inglés común, y contienen la notación matemática común. Un programa de nómina escrito en un lenguaje de alto nivel podría contener una instrucción como la siguiente:

```
sueldoBruto = sueldoBase + sueldoExtra;
```

Obviamente, los lenguajes de alto nivel son mucho más recomendables, desde el punto de vista del programador, que los lenguajes de máquina o ensamblador. C, C++, los lenguajes .NET de Microsoft (por ejemplo, Visual Basic Visual C++ y Visual C#) y Java son algunos de los lenguajes de programación de alto nivel más ampliamente utilizados.

El proceso de compilación de un programa escrito en lenguaje de alto nivel a un lenguaje de máquina puede tardar un tiempo considerable en la computadora. Los **programas intérpretes** se desarrollaron para ejecutar programas en lenguaje de alto nivel directamente, aunque con más lentitud.

1.9 Historia de C y C++

C++ evolucionó de C, que a su vez evolucionó de dos lenguajes de programación anteriores, BCPL y B. En 1967, Martin Richards desarrolló BCPL como un lenguaje para escribir software de sistemas operativos y compiladores para sistemas operativos. Ken Thompson modeló muchas características en su lenguaje B a partir del trabajo de sus contrapartes en BCPL, y utilizó a B para crear las primeras versiones del sistema operativo UNIX, en los laboratorios Bell en 1970.

El lenguaje C evolucionó a partir de B, gracias al trabajo de Dennis Ritchie en los laboratorios Bell. C utiliza muchos conceptos importantes de BCPL y B. Inicialmente, se hizo muy popular como lenguaje de desarrollo para el sistema operativo UNIX. En la actualidad, la mayor parte de los sistemas operativos se escriben en C y/o C++. C está disponible para la mayoría de las computadoras y es independiente del hardware. Con un diseño cuidadoso, es posible escribir programas en C que sean **portables** para la mayoría de las computadoras.

Por desgracia, el amplio uso de C con diversos tipos de computadoras (a las que algunas veces se les denomina **plataformas de hardware**) produjo muchas variaciones. Éste fue un grave problema para los desarrolladores de programas, quienes necesitaban escribir programas portables que pudieran ejecutarse en varias plataformas. Era necesaria una versión estándar de C. El Instituto nacional estadounidense de estándares (ANSI) cooperó con la Organización internacional

para la estandarización (ISO) para estandarizar C a nivel mundial; el documento estándar colectivo se publicó en 1990, y se conoce como *ANSI/ISO 9899: 1990*.

C99 es el estándar ANSI más reciente para el lenguaje de programación C. Se desarrolló para evolucionar el lenguaje C, de manera que pudiera estar al corriente con el poderoso hardware de la actualidad, y con los requerimientos cada vez más exigentes de los usuarios. El Estándar C99 es más capaz (que los primeros Estándares de C) de competir con los lenguajes como Fortran para las aplicaciones matemáticas. Las capacidades de C99 incluyen el tipo `long long` para las máquinas de 64 bits, los números complejos para las aplicaciones de ingeniería y un mayor soporte para la aritmética de punto flotante. C99 también hace a C más consistente con C++, al permitir el polimorfismo a través de funciones matemáticas de tipo genérico, y por medio de la creación de un tipo booleano definido. Para obtener más información acerca de C y C99, consulte nuestro libro *C How to Program, quinta edición* y nuestro Centro de Recursos de C (ubicado en www.deitel.com/C).



Tip de portabilidad 1.1

Como C es un lenguaje estandarizado, independiente del hardware y ampliamente disponible, las aplicaciones escritas en C comúnmente pueden ejecutarse con pocas modificaciones (o ninguna) en un amplio rango de sistemas computacionales.

A principios de la década de los ochenta, Bjarne Stroustrup desarrolló una extensión de C en los laboratorios Bell: C++. Este lenguaje proporciona un conjunto de características que “pulen” al lenguaje C, pero lo más importante es que proporciona la capacidad de una **programación orientada a objetos**.

Una revolución se está gestando en la comunidad del software. Escribir software rápida, correcta y económica mente es aún una meta escurridiza, en una época en que la demanda de nuevo y más poderoso software se encuentra a la alza. Los **objetos** son en esencia **componentes** reutilizables de software, que modelan elementos del mundo real. Los desarrolladores de software están descubriendo que el uso de una metodología de diseño e implementación modular y orientada a objetos puede hacerlos más productivos que mediante las populares técnicas de programación anteriores. Los programas orientados a objetos son más fáciles de entender, corregir y modificar.

Por esta razón, cambiamos a una pedagogía en la que se presentan las clases y los objetos en los primeros capítulos. En la sección 1.21 recibirá una introducción a los conceptos básicos y la terminología de la tecnología de objetos. Empezará a desarrollar clases personalizadas y reutilizables en el capítulo 3, Introducción a las clases y los objetos. Esta nueva edición del libro está orientada a objetos, donde sea apropiado, desde el principio y a lo largo del texto. Al mover la discusión sobre las clases y los objetos a los primeros capítulos, usted empieza a “pensar acerca de los objetos” de inmediato, y puede dominar estos conceptos con más detalle. De ningún modo la programación orientada a objetos es algo trivial, pero es divertido escribir programas orientados a objetos, y los estudiantes pueden obtener resultados inmediatos.

También proporcionamos un ejemplo práctico opcional sobre un cajero automático (ATM) en las secciones tituladas Ejemplo práctico de Ingeniería de Software de los capítulos 1 a 7, 9 y 13, y en el apéndice G, el cual contiene una implementación completa en C++. Las nueve secciones del ejemplo práctico presentan una introducción cuidadosamente pautada al diseño orientado a objetos mediante el UML: un lenguaje de modelación gráfico estándar en la industria, para desarrollar sistemas orientados a objetos. Lo guiamos a través de su primera experiencia de diseño, enfocada hacia el programador/diseñador orientado a objetos principiante. Nuestro objetivo es ayudarlo a desarrollar un diseño orientado a objetos para complementar los conceptos de programación orientada a objetos que aprenderá en este capítulo, y que empezará a implementar en el capítulo 3.

1.10 Biblioteca estándar de C++

Los programas en C++ consisten de piezas llamadas **clases y funciones**. Usted puede programar cada pieza que pueda necesitar para formar un programa en C++. Sin embargo, la mayoría de los programadores de C++ aprovechan las extensas colecciones de clases y funciones existentes en la **Biblioteca estándar de C++**. Por ende, en realidad hay dos partes que debemos conocer en el “mundo” de C++. La primera es aprender acerca del lenguaje C++ en sí; la segunda es aprender a utilizar las clases y funciones en la Biblioteca estándar de C++. A lo largo del libro, hablaremos sobre muchas de estas clases y funciones. El libro de P. J. Plauger, titulado *The Standard C Library* (Upper Saddle River, NJ: Prentice Hall PTR, 1992) es una lectura obligatoria para los programadores que requieren una comprensión detallada de las funciones de biblioteca de ANSI C que se incluyen en C++, cómo implementarlas y cómo usarlas para escribir código portable. Por lo general, las bibliotecas de clases estándar las proporcionan los distribuidores de los compiladores. Hay muchas bibliotecas de clases de propósito especial que proporcionan los distribuidores de software independientes.



Observación de Ingeniería de Software I.1

Utilice un método de “construcción en bloques” para crear programas. Evite reinventar la rueda. Use piezas existentes siempre que sea posible. Esta reutilización de software es un beneficio clave de la programación orientada a objetos.



Observación de Ingeniería de Software I.2

Cuando programe en C++, generalmente utilizará los siguientes bloques de construcción: clases y funciones de la Biblioteca estándar de C++, clases y funciones creadas por usted mismo y sus colegas, y clases y funciones de varias bibliotecas populares desarrolladas por terceros.

Incluimos muchas **Observaciones de Ingeniería de Software** a lo largo de este texto para explicar los conceptos que afectan y mejoran la arquitectura y calidad de los sistemas de software. También resaltamos otras clases de tips, incluyendo las **Buenas prácticas de programación** (que le ayudarán a escribir programas más claros, comprensibles, de fácil mantenimiento, y fáciles de probar y depurar; es decir, eliminar errores de programación), los **Errores comunes de programación** (problemas de los que tenemos que cuidarnos y evitar), **Tips de rendimiento** (técnicas para escribir programas que se ejecuten más rápido y ocupen menos memoria), **Tips de portabilidad** (técnicas que le ayudarán a escribir programas que se ejecuten, con poca o ninguna modificación, en una variedad de computadoras; estos tips también incluyen observaciones generales acerca de cómo logra C++ su alto grado de portabilidad) y **Tips para prevenir errores** (técnicas que le ayudarán a eliminar errores de sus programas; programas libres de errores desde el principio). Muchas de estas técnicas y prácticas son sólo guías. Usted deberá, sin duda, desarrollar su propio estilo de programación.

La ventaja de crear sus propias funciones y clases es que sabe exactamente cómo funcionan. La desventaja es el tiempo que consumen y el esfuerzo complejo que se requiere para diseñar, desarrollar y dar mantenimiento a las nuevas funciones y clases que sean correctas y operen con eficiencia.



Tip de rendimiento I.1

Utilizar las funciones y clases de la Biblioteca estándar de C++ en lugar de escribir sus propias versiones puede mejorar el rendimiento de sus programas, ya que estas clases y métodos están escritos cuidadosamente para funcionar de manera eficiente. Esta técnica también reduce el tiempo de desarrollo de los programas.



Tip de portabilidad I.2

Utilizar las funciones y clases de la Biblioteca estándar de C++ en lugar de escribir sus propias versiones mejora la portabilidad de sus programas, ya que estas funciones y clases se incluyen en todas las implementaciones de C++.



I.11 Historia de Java

Los microprocesadores están teniendo un profundo impacto en los dispositivos electrónicos inteligentes para uso doméstico. Al reconocer esto, Sun Microsystems patrocinó en 1991 un proyecto interno de investigación corporativa denominado Green, el cual desembocó en el desarrollo de un lenguaje basado en C++ al que su creador, James Gosling, llamó Oak debido a un roble que tenía a la vista desde su ventana en las oficinas de Sun. Posteriormente se descubrió que ya existía un lenguaje de computadora con el mismo nombre. Cuando un grupo de gente de Sun visitó una cafetería local, sugirieron el nombre Java y así se quedó.

El proyecto Green tuvo algunas dificultades. El mercado para los dispositivos electrónicos inteligentes de uso doméstico no se desarrollaba tan rápido a principios de los noventa como Sun había anticipado. El proyecto corría el riesgo de cancelarse. Pero para su buena fortuna, la popularidad de World Wide Web explotó en 1993 y la gente de Sun se dio cuenta inmediatamente del potencial de Java para agregar **contenido dinámico** (como interactividad y animaciones), a las páginas Web. Esto trajo nueva vida al proyecto.

Sun anunció formalmente a Java en una conferencia importante que tuvo lugar en mayo de 1995. Java generó la atención de la comunidad de negocios debido al fenomenal interés en World Wide Web. En la actualidad, Java se utiliza para desarrollar aplicaciones empresariales a gran escala, para mejorar la funcionalidad de los servidores Web (las computadoras que proporcionan el contenido que vemos en nuestros exploradores Web), para proporcionar aplicaciones para los dispositivos domésticos (como teléfonos celulares, radiolocalizadores y asistentes digitales personales) y para muchos otros propósitos.



I.12 FORTRÁN, COBOL, Pascal y Ada

Se han desarrollado cientos de lenguajes de alto nivel, pero sólo unos cuantos han logrado una amplia aceptación. FORTRAN (FORmula TRANslator, Traductor de fórmulas) fue desarrollado por IBM Corporation a mediados de

la década de los cincuenta para utilizarse en aplicaciones científicas y de ingeniería que requerían cálculos matemáticos complejos. Fortran se utiliza ampliamente todavía en aplicaciones de ingeniería.

COBOL (COmmon Business Oriented Language, Lenguaje común orientado a negocios) fue desarrollado a finales de la década de los cincuenta por fabricantes de computadoras, el gobierno estadounidense y usuarios de computadoras de la industria. COBOL se utiliza en aplicaciones comerciales que requieren de una manipulación precisa y eficiente de grandes volúmenes de datos. Gran parte del software de negocios aún se programa en COBOL.

Durante la década de los sesenta, muchos de los grandes esfuerzos para el desarrollo de software encontraron severas dificultades. Los itinerarios de software generalmente se retrasaban, los costos rebasaban en gran medida a los presupuestos y los productos terminados no eran confiables. La gente comenzó a darse cuenta de que el desarrollo de software era una actividad mucho más compleja de lo que habían imaginado. Las actividades de investigación en la década de los sesenta dieron como resultado la evolución de la **programación estructurada** (un método disciplinado para escribir programas que sean más claros, fáciles de probar y depurar, y más fáciles de modificar que los programas extensos producidos con técnicas anteriores).

Uno de los resultados más tangibles de esta investigación fue el desarrollo del lenguaje de programación **Pascal** por el profesor Niklaus Wirth, en 1971. Pascal, cuyo nombre se debe al matemático y filósofo Blaise Pascal del siglo diecisiete, se diseñó para la enseñanza de la programación estructurada en ambientes académicos, y de inmediato se convirtió en el lenguaje de programación preferido en la mayoría de las universidades. Pascal carece de muchas de las características necesarias para poder utilizarse en aplicaciones comerciales, industriales y gubernamentales, por lo que no ha sido muy aceptado en estos entornos.

El lenguaje de programación **Ada** se desarrolló bajo el patrocinio del Departamento de Defensa de Estados Unidos (DoD) durante la década de los setenta y los primeros años de la década de los ochenta. Cientos de lenguajes independientes se utilizaron para producir los sistemas de software masivos de comando y control del departamento de defensa. Éste quería un solo lenguaje que pudiera satisfacer la mayoría de sus necesidades. El nombre del lenguaje es en honor de Lady Ada Lovelace, hija del poeta Lord Byron. A Lady Lovelace se le atribuye el haber escrito el primer programa para computadoras en el mundo, a principios de la década de 1800 (para la Máquina Analítica, un dispositivo de cómputo mecánico diseñado por Charles Babbage). Una de las características importantes de Ada se conoce como **multitarea**, la cual permite a los programadores especificar que muchas actividades ocurrirán en paralelo. Java, a través de una técnica que se conoce como **subprocesamiento múltiple (multihilos)**, también permite a los programadores escribir programas con actividades paralelas. Aunque el subprocesamiento múltiple no forma parte del lenguaje C++ estándar, está disponible a través de varias bibliotecas de clases complementarias, como Boost (www.boost.org).

1.13 BASIC, Visual Basic, Visual C++, C# y .NET

El lenguaje de programación **BASIC** (Beginner's All-purpose Symbolic Instruction Code, Código de instrucciones simbólicas de uso general para principiantes) fue desarrollado a mediados de la década de los sesenta en el Dartmouth College, como un medio para escribir programas simples. El propósito principal de BASIC era que los principiantes se familiarizaran con las técnicas de programación. El lenguaje Visual Basic de Microsoft se introdujo a principios de la década de los noventa para simplificar el desarrollo de aplicaciones para Microsoft Windows, y es uno de los lenguajes de programación más populares en el mundo.

Las herramientas de desarrollo más recientes de Microsoft forman parte de su estrategia a nivel corporativo para integrar Internet y Web en las aplicaciones de computadora. Esta estrategia se implementa en la **plataforma .NET** de Microsoft, la cual proporciona a los desarrolladores las herramientas que necesitan para crear y ejecutar aplicaciones de computadora que se puedan ejecutar en computadoras distribuidas a través de Internet. Los tres principales lenguajes de programación de Microsoft son **Visual Basic** (basado en el lenguaje BASIC original), **Visual C++** (basado en C++) y **Visual C#** (un nuevo lenguaje basado en C++ y Java, y desarrollado expresamente para la plataforma .NET). Los desarrolladores que utilizan .NET pueden escribir componentes de software en el lenguaje con el que estén más familiarizados y formar aplicaciones al combinar esos componentes con los componentes escritos en cualquier lenguaje .NET.

1.14 Tendencia clave de software: la tecnología de los objetos

Uno de los autores, Harvey Deitel, recuerda la gran frustración que sentían en la década de los sesenta las organizaciones de desarrollo de software, en especial las que trabajaban en proyectos a gran escala. Durante sus años de estudiante, tuvo el privilegio de trabajar durante los veranos con un distribuidor de cómputo líder, en los equipos de desarrollo de los sistemas operativos de tiempo compartido y memoria virtual. Ésta fue una gran experiencia para un estudiante universitario. Pero, en el verano de 1967, la realidad se estableció cuando la compañía se “descomprometió” de producir como

producto comercial el sistema específico en el que cientos de personas habían trabajado durante muchos años. Era difícil crear bien este software; el software es “algo complejo”.

Sin embargo, surgieron mejoras en la tecnología del software, donde se hicieron realidad los beneficios de la programación estructurada (y las disciplinas relacionadas del **análisis y diseño de sistemas estructurados**) en la década de los setenta. No fue sino hasta que la tecnología de la programación orientada a objetos se empezó a utilizar ampliamente en la década de los noventa, que los desarrolladores de software sintieron que tenían las herramientas necesarias para avanzar con grandes pasos en el proceso de desarrollo de software.

En realidad, la tecnología de objetos se remonta hasta a mediados de la década de los sesenta. El lenguaje de programación C++, desarrollado en AT&T por Bjarne Stroustrup a principios de la década de los ochenta, se basa en dos lenguajes (C, que se desarrolló inicialmente en AT&T para implementar el sistema operativo UNIX a principios de la década de 1970, y Simula 67, un lenguaje de programación de simulación, desarrollado en Europa y puesto en circulación en 1967). C++ absorbió las características de C y agregó las herramientas de Simula para crear y manipular objetos. Ninguno de los lenguajes C o C++ estaba diseñado originalmente para usarse ampliamente, más allá de los laboratorios de investigación de AT&T. Pero el soporte básico se desarrolló rápidamente para cada uno de ellos.

¿Qué son los objetos y por qué son especiales? En realidad, la tecnología de objetos es un esquema de empaquetamiento que nos ayuda a crear unidades de software con significado. Éstas pueden ser extensas, y se enfocan principalmente en áreas de aplicación específicas. Existen objetos fecha, objetos tiempo, objetos salario, objetos factura, objetos audio, objetos video, objetos archivo, objetos registro y así, en lo sucesivo. De hecho, casi cualquier sustantivo se puede representar de manera razonable como un objeto.

Vivimos en un mundo de objetos. Sólo necesita mirar a su alrededor. Hay autos, aviones, personas, animales, edificios, semáforos, elevadores, etcétera. Antes de que aparecieran los lenguajes orientados a objetos, los lenguajes de programación por procedimientos (como Fortran, COBOL, Pascal, BASIC y C) se enfocaban en acciones (verbos), en lugar de cosas u objetos (sustantivos). Los programadores que vivían en un mundo de objetos, programaban principalmente mediante el uso de verbos. Esto dificultaba el proceso de escribir programas. Ahora, con la disponibilidad de lenguajes orientados a objetos populares, como C++ y Java, los programadores siguen viviendo en un mundo orientado a objetos, y pueden programar de una manera orientada a objetos. Este es un proceso más natural que la programación por procedimientos, y ha producido aumentos considerables en la productividad.

Un problema clave con la programación por procedimientos es que las unidades de los programas no reflejan con efectividad a las entidades del mundo real, por lo que estas unidades no son especialmente reutilizables. Es común para los programadores “empezar como nuevos” en cada nuevo proyecto y tener que escribir software similar “desde cero”. Esto desperdicia tiempo y dinero, a medida que las personas “reinventan la rueda” en forma repetida. Con la tecnología de los objetos, las entidades de software creadas (llamadas **clases**), si se diseñan en forma apropiada, tienden a ser reutilizables en proyectos futuros. El uso de bibliotecas de componentes reutilizables puede reducir en forma considerable el esfuerzo requerido para implementar ciertos tipos de sistemas (en comparación con el esfuerzo que se requeriría para reinventar estas herramientas en nuevos proyectos).



Observación de Ingeniería de Software I.3

Hay una gran cantidad de bibliotecas de componentes de software reutilizables en Internet. Muchas de estas bibliotecas son gratuitas.

Algunas empresas reportan que el beneficio clave que les proporciona la programación orientada a objetos no es la reutilización de software, sino que el software que producen es más comprensible, está mejor organizado y es más fácil de mantener, modificar y depurar. Esto puede ser importante, ya que tal vez hasta 80 por ciento de los costos de software están asociados no con los esfuerzos originales por desarrollar el software, sino con la evolución continua y el mantenimiento de ese software a lo largo de su vida.

Cualesquiera que sean los beneficios percibidos, está claro que la programación orientada a objetos será la metodología de programación clave durante las siguientes décadas.

I.15 Entorno de desarrollo típico en C++

Vamos a considerar los pasos para crear y ejecutar una aplicación de C++, usando un entorno de desarrollo de C++ (el cual se ilustra en la figura 1.1). Por lo general, los sistemas de C++ consisten en tres partes: un entorno de desarrollo de programas, el lenguaje y la Biblioteca estándar de C++. Comúnmente, los programas en C++ pasan a través de seis fases: **edición, preprocessamiento, compilación, enlace, carga y ejecución**. A continuación explicaremos un típico entorno de desarrollo de programas en C++:

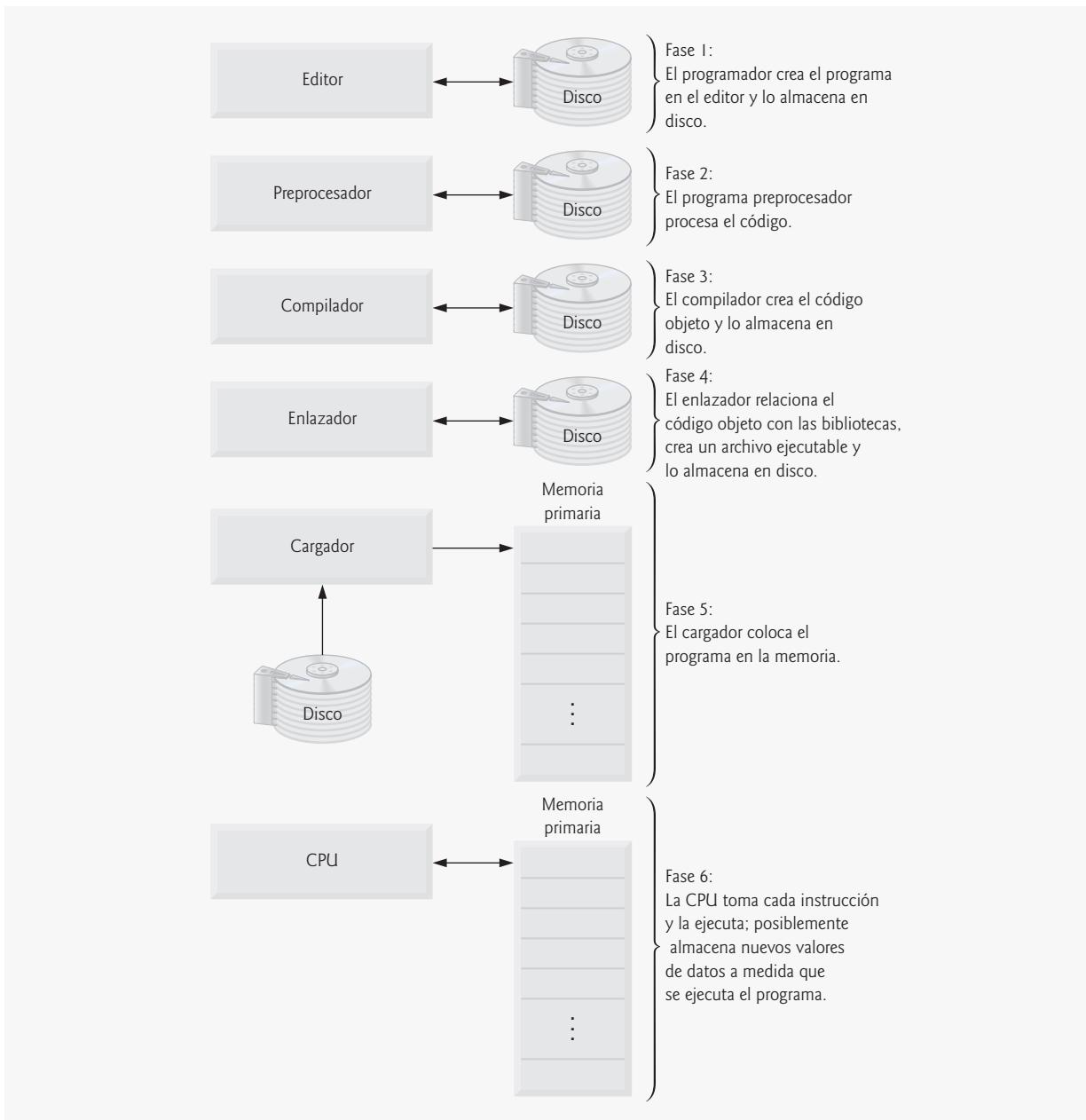


Figura 1.1 | Entorno típico de C++.

Fase 1: Creación de un programa

La fase 1 consiste en editar un archivo con un **programa de edición** (conocido comúnmente como **editor**). Usted escribe un programa en C++ (conocido por lo general como **código fuente**) utilizando el editor, realiza las correcciones necesarias y guarda el programa en un dispositivo de almacenamiento secundario, tal como su disco duro. A menudo, los nombres de archivos de código fuente en C++ terminan con las extensiones .cpp, .cxx, .cc o .C (observe que C está en mayúsculas), para indicar que un archivo contiene código fuente en C++. Consulte la documentación para su compilador de C++ si desea obtener más información acerca de las extensiones de archivo.

Dos de los editores que se utilizan ampliamente en sistemas UNIX son **vi** y **emacs**. Los paquetes de software de C++ para Microsoft Windows tales como Microsoft Visual C++ (msdn.microsoft.com/vstudio/express/visualc/default.aspx en inglés, y www.microsoft.com/spanish/msdn/vstudio/express/VC/default.mspx en español) y Code Gear C++ Builder (www.codegear.com) tienen editores integrados en el entorno de programación. También puede

utilizar un editor de texto simple, como el Bloc de notas en Windows, para escribir su código de C++. Vamos a suponer que usted sabe cómo editar un archivo.

Fases 2 y 3: Preprocesamiento y compilación de un programa en C++

En la fase 2, se introduce el comando para **compilar** el programa. En un sistema de C++, un programa **preprocesador** se ejecuta de manera automática antes de que empiece la fase de traducción del compilador (por lo que a la fase 2 la llamamos preprocesamiento, y a la fase 3 la llamamos compilación). El preprocesador de C++ obedece a comandos denominados **directivas del preprocesador**, las cuales indican que deben realizarse ciertas manipulaciones en el programa, antes de compilarlo. Por lo general, estas manipulaciones incluyen otros archivos de texto a ser compilados y realizan varios reemplazos de texto. Las directivas más comunes del preprocesador se describen en los primeros capítulos; en el apéndice F, Preprocesador, aparece una discusión detallada acerca de las características del preprocesador. En la fase 3, el compilador traduce el programa de C++ en código de lenguaje de máquina (también conocido como código objeto).

Fase 4: Enlace

A la fase 4 se le llama **enlace**. Por lo general, los programas en C++ contienen referencias a funciones y datos definidos en otra parte, como en las bibliotecas estándar o en las bibliotecas privadas de grupos de programadores que trabajan sobre un proyecto específico. El código objeto producido por el compilador de C++ comúnmente contiene “huecos”, debido a estas partes faltantes. Un **enlazador** relaciona el código objeto con el código para las funciones faltantes, de manera que se produzca una **imagen ejecutable** (sin piezas faltantes). Si el programa se compila y se enlaza de manera correcta, se produce una imagen ejecutable.

Fase 5: Carga

A la fase 5 se le conoce como **carga**. Antes de poder ejecutar un programa, primero se debe colocar en la memoria. Esto se hace mediante el **cargador**, que toma la imagen ejecutable del disco y la transfiere a la memoria. También se cargan los componentes adicionales de bibliotecas compartidas que dan soporte al programa.

Fase 6: Ejecución

Por último, la computadora, bajo el control de su CPU, **ejecuta** el programa una instrucción a la vez.

Problemas que pueden ocurrir en tiempo de ejecución

Es probable que los programas no funcionen la primera vez. Cada una de las fases anteriores puede fallar, debido a diversos errores que describiremos en este texto. Por ejemplo, un programa en ejecución podría intentar una división entre cero (una operación ilegal para la aritmética con números enteros en C++). Esto haría que el programa de C++ imprimiera un mensaje de error. Si esto ocurre, tendría que regresar a la fase de edición, hacer las correcciones necesarias y proseguir con las fases restantes de nuevo, para determinar que las correcciones resolvieron el (los) problema(s).

La mayoría de los programas en C++ reciben y/o producen datos. Ciertas funciones de C++ toman su entrada de `cin` (el **flujo estándar de entrada**; se pronuncia “c-in”), que por lo general es el teclado, pero `cin` puede conectarse a otro dispositivo. Por lo general, los datos se envían a `cout` (el **flujo estándar de salida**), que por lo general es la pantalla de la computadora, pero `cout` puede conectarse a otro dispositivo. Cuando decimos que un programa imprime un resultado, por lo general nos referimos a que el resultado se despliega en una pantalla. Los datos pueden enviarse a otros dispositivos, como los discos y las impresoras. También hay un **flujo estándar de error**, conocido como `cerr`. El flujo `cerr` (que por lo general se conecta a la pantalla) se utiliza para mostrar mensajes de error. Es común para los usuarios asignar `cout` a un dispositivo distinto a la pantalla, mientras mantienen `cerr` asignado a la pantalla, para que los resultados normales se separen de los errores.



Error común de programación I.I

Los errores, como la división entre cero, ocurren a medida que se ejecuta un programa, de manera que a estos errores se les llama errores en tiempo de ejecución. Los errores fatales en tiempo de ejecución hacen que los programas terminen inmediatamente, sin haber realizado correctamente su trabajo. Los errores no fatales en tiempo de ejecución permiten a los programas ejecutarse hasta terminar su trabajo, lo que a menudo produce resultados incorrectos. [Nota: en algunos sistemas, la división entre cero no es un error fatal. Consulte la documentación de su sistema.]

1.16 Generalidades acerca de C++ y este libro

Con frecuencia, los programadores experimentados en C++ están orgullosos de poder hacer un uso excéntrico, deformando y complejo del lenguaje. Ésta es una práctica de programación pobre. Hace que los programas sean más difíciles de

leer, que se comporten de manera extraña, más difíciles de probar y depurar, y más difíciles de adaptarse a los requerimientos cambiantes. Este libro está orientado hacia los programadores principiantes, de manera que buscamos la *claridad* en los programas. La siguiente es nuestra primera “buena práctica de programación”.



Buena práctica de programación 1.1

Escriba sus programas de C++ en una manera simple y directa. A esto se le conoce algunas veces como KIS (“Keep It Simple”, simplifíquelo) No “estire” el lenguaje experimentando con usos excéntricos.

Seguramente habrá escuchado que C y C++ son lenguajes portables, y que los programas escritos en estos lenguajes pueden ejecutarse en muchas computadoras distintas. *La portabilidad es una meta elusiva*. El documento del estándar ANSI C contiene una extensa lista de cuestiones relacionadas con la portabilidad, y se han escrito libros completos acerca de este tema.



Tip de portabilidad 1.3

Aunque es posible escribir programas portables, hay muchos problemas entre los distintos compiladores de C y C++, y las distintas computadoras, los cuales pueden hacer que la portabilidad sea difícil de lograr. No basta con escribir programas en C y C++ para garantizar su portabilidad. A menudo, será necesario tratar directamente con las variaciones entre el compilador y la computadora. Como grupo, a estas variaciones se les conoce algunas veces como variaciones de la plataforma.

Hemos hecho una revisión cuidadosa del documento estándar para C++ ISO/IEC y comparado nuestra presentación contra éste, para que sea completa y acertada. Sin embargo, C++ es un lenguaje rico, y existen algunas características que no hemos cubierto. Si necesita detalles técnicos adicionales sobre C++, tal vez sea conveniente que dé un vistazo al documento del estándar para C++, que puede ordenar de ANSI en el siguiente sitio Web:

webstore.ansi.org/ansidocstore/default.asp

El título del documento es “Information Technology – Programming Languages – C++” y su número de documento es INCITS/ISO/IEC 14882-2003.

Hemos incluido una extensa bibliografía de libros y documentos sobre C++ y la programación orientada a objetos. También enlistamos muchos sitios Web relacionados con C++ y la programación orientada a objetos en nuestro Centro de Recursos de C++ en www.deitel.com/cplusplus/. En la sección 1.23 enlistamos varios sitios Web, incluyendo vínculos a compiladores de C++ gratuitos, sitios de recursos, algunos juegos divertidos en C++ y tutoriales de programación de juegos.



Buena práctica de programación 1.2

Lea la documentación para la versión de C++ que esté utilizando. Consulte con frecuencia esta documentación, para asegurarse de que conoce la extensa colección de características de C++, y que las esté utilizando en forma correcta.



Buena práctica de programación 1.3

Su computadora y su compilador son buenos maestros. Si después de leer su documentación del lenguaje C++, no está seguro de cómo funciona alguna característica de C++, experimente con un pequeño programa de prueba y vea lo que sucede. Establezca las opciones de su compilador en el “máximo número de advertencias”. Estudie cada mensaje que el compilador genere y corrija el programa para eliminar los mensajes.

1.17 Prueba de una aplicación en C++

En esta sección, ejecutará su primera aplicación en C++ e interactuará con ella. Empezará ejecutando un divertido juego de “adivinar el número”, el cual elige un número del 1 al 1000 y le pide que lo adivine. Si su elección es la correcta, el juego termina. Si no es correcta, la aplicación le indica si su elección es mayor o menor que el número correcto. No hay límite en cuanto al número de elecciones que puede realizar. [Nota: sólo para esta prueba hemos modificado esta aplicación del ejercicio que le pediremos que cree en el capítulo 6, Funciones y una introducción a la recursividad. Por lo general, esta aplicación selecciona al azar la respuesta correcta cuando usted ejecuta el programa. La aplicación modificada utiliza la misma respuesta correcta cada vez que el programa se ejecuta (aunque esto podría variar, según el compilador), por lo que puede utilizar las mismas elecciones que utilizamos en esta sección, y verá los mismos resultados a medida que interactúe con su primera aplicación en C++].

Demostraremos cómo ejecutar una aplicación de C++ de dos maneras: mediante el **Símbolo del sistema** de Windows XP y mediante un shell en Linux (similar a un **Símbolo del sistema** de Windows). La aplicación se ejecuta de manera similar en ambas plataformas. Muchos entornos de desarrollo están disponibles, en los cuales los lectores pueden compilar, generar y ejecutar aplicaciones de C++, como C++ Builder de Code Gear, GNU C++, Microsoft Visual C++, etc. Consulte con su instructor para obtener información acerca de su entorno de desarrollo específico.

En los siguientes pasos, ejecutará la aplicación y escribirá varios números para adivinar el número correcto. Los elementos y la funcionalidad que puede ver en esta aplicación son típicos de los que aprenderá a programar en este libro. A lo largo del mismo, utilizamos distintos tipos de letra para diferenciar entre las características que se pueden ver en la pantalla (el **Símbolo del sistema**) y los elementos que no están directamente relacionados con la pantalla. Nuestra convención es enfatizar las características de la pantalla, como los títulos y los menús (como el menú **Archivo**) en un tipo de letra **Helvetica sans-serif** en semi-negritas, y enfatizar los nombres de archivo, el texto desplegado por una aplicación y los valores que debe introducir en una aplicación (como **AdivinarNumero** o 500) en un tipo de letra **Lucida sans-serif**. Como tal vez ya se haya dado cuenta, la **ocurrencia de definición** de cada término se establece en gris. Si desea modificar los colores del **Símbolo del sistema** en su computadora, abra una ventana **Símbolo del sistema**, después haga clic con el botón derecho del ratón en la barra de título y seleccione **Propiedades**. En el cuadro de diálogo **Propiedades de "Símbolo del sistema"** que aparezca, haga clic en la ficha **Colores** y seleccione sus colores de texto y fondo preferidos.

Ejecución de una aplicación de C++ desde el Símbolo del sistema de Windows XP

1. **Revise su configuración.** Lea la sección *Antes de empezar* del libro para confirmar que haya copiado correctamente los ejemplos del libro en su disco duro.
2. **Localice la aplicación completa.** Abra una ventana **Símbolo del sistema**. Si utiliza Windows 95, 98 o 2000, seleccione **Inicio > Programas > Accesorios > Símbolo del sistema**. Si utiliza Windows XP, seleccione **Inicio > Todos los programas > Accesorios > Símbolo del sistema**. Para cambiar al directorio de la aplicación completa **AdivinarNumero**, escriba **cd C:\ejemplos\cap01\AdivinarNumero\Windows**, y después oprima **Intro** (figura 1.2). El comando **cd** se utiliza para cambiar de directorio.



Figura 1.2 | Abrir una ventana **Símbolo del sistema** y cambiar de directorio.

3. **Ejecute la aplicación **AdivinarNumero**.** Ahora que se encuentra en el directorio que contiene la aplicación **AdivinarNumero**, escriba el comando **AdivinarNumero** (figura 1.3) y oprima **Intro**. [Nota: **AdivinarNumero.exe** es el nombre real de la aplicación; sin embargo, Windows asume la extensión **.exe** de manera predeterminada.]



Figura 1.3 | Ejecución de la aplicación **AdivinarNumero**.

4. **Escriba su primera elección.** La aplicación muestra el mensaje "**Escriba su primera elección.**", y después muestra un signo de interrogación (?) como un indicador en la siguiente línea (figura 1.3). En el indicador, escriba **500** (figura 1.4).
5. **Escriba otra elección.** La aplicación muestra "**Demasiado alto. Intento de nuevo.**", lo cual significa que el valor que escribió es mayor que el número que eligió la aplicación como la respuesta correcta. Por lo tanto, debe escribir un número menor como su siguiente elección. En el indicador, escriba **250** (figura 1.5). La apli-

cación muestra de nuevo el mensaje "Demasiado alto. Intento de nuevo.", ya que el valor que escribió sigue siendo mayor que el número que eligió la aplicación como la respuesta correcta.

6. **Escriba más elecciones.** Continúe el juego, escribiendo valores hasta que adivine el número correcto. La aplicación mostrará el mensaje "Excelente! Adivino el numero!" (figura 1.6).
7. **Juegue de nuevo o salga de la aplicación.** Una vez que adivine la respuesta correcta, la aplicación le preguntará si desea jugar otra vez (figura 1.6). En el indicador "Le gustaría jugar de nuevo (s o n)?", al escribir el carácter **s** la aplicación elegirá un nuevo número y mostrará el mensaje "Escriba su primera elección.", seguido de un signo de interrogación como indicador (figura 1.7), de manera que pueda escribir su primera elección en el nuevo juego. Al escribir el carácter **n** la aplicación termina y el sistema nos regresa al directorio de la aplicación en el Símbolo del sistema (figura 1.8). Cada vez que ejecute esta aplicación desde el principio (es decir, desde el *paso 3*), elegirá los mismos números para adivinar .
8. *Cierre la ventana Símbolo del sistema.*



Figura 1.4 | Escriba su primera elección.

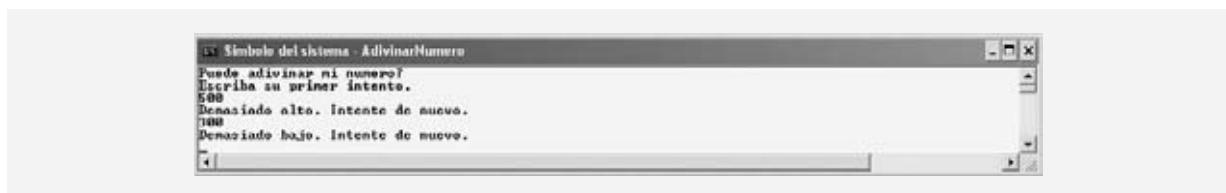


Figura 1.5 | Escriba su segunda elección y reciba retroalimentación.



Figura 1.6 | Escriba más elecciones y adivine el número correcto.



Figura 1.7 | Juegue de nuevo.



Figura 1.8 | Salga del juego.

Ejecución de una aplicación de C++ mediante GNU C++ con Linux

Para esta prueba, vamos a suponer que usted sabe cómo copiar los ejemplos en su directorio de inicio. Consulte con su instructor si tiene dudas acerca de cómo copiar los archivos en su sistema Linux. Además, para las figuras en esta sección utilizamos texto resaltado en negritas para indicar la entrada del usuario requerida en cada paso. El indicador en el shell en nuestro sistema utiliza el carácter tilde (~) para representar el directorio de inicio, y cada indicador termina con el carácter de signo de dólar (\$). El indicador puede variar de un sistema Linux a otro.

1. **Localice la aplicación completa.** Desde un shell en Linux, cambie al directorio de la aplicación **AdivinarNúmero** completa (figura 1.9); para ello escriba

```
cd Ejemplos/cap01/AdivinarNumero/GNU_Linux
```

y oprima *Intro*. El comando cd se utiliza para cambiar de directorio.

2. **Compile la aplicación AdivinarNúmero.** Para ejecutar una aplicación en el compilador GNU C++, primero debe compilarla; para ello escriba

```
g++ AdivinarNumero.cpp -o AdivinarNumero
```

como en la figura 1.10. Este comando compila la aplicación y produce un archivo ejecutable, llamado **AdivinarNúmero**.

3. **Ejecute la aplicación AdivinarNúmero.** Para ejecutar el archivo **AdivinarNúmero**, escriba **./AdivinarNúmero** en el siguiente indicador, y luego oprima *Intro* (figura 1.11).

4. **Escriba su primera elección.** La aplicación muestra el mensaje "Escriba su primera elección.", y después muestra un signo de interrogación (?) como un indicador en la siguiente línea (figura 1.11). En el indicador, escriba **500** (figura 1.12). [Nota: ésta es la misma aplicación que modificamos y probamos para Windows, pero los resultados podrían variar, dependiendo del compilador que se utilice.]

5. **Escriba otra elección.** La aplicación muestra "Demasiado alto. Intenté de nuevo.", lo cual significa que el valor que escribió es mayor que el número que eligió la aplicación como la respuesta correcta (figura 1.12). En el siguiente indicador, escriba **250** (figura 1.13). Esta vez la aplicación muestra el mensaje "Demasiado bajo. Intenté de nuevo.", ya que el valor que escribió es menor que el número que la respuesta correcta.

```
~$ cd ejemplos/cap01/AdivinarNumero/GNU_Linux
~/ejemplos/cap01/AdivinarNumero/GNU_Linux$
```

Figura 1.9 | Cambie al directorio de la aplicación **AdivinarNúmero** después de iniciar sesión con su cuenta de Linux.

```
~/ejemplos/cap01/AdivinarNumero/GNU_Linux$ g++ AdivinarNumero.cpp -o AdivinarNumero
~/ejemplos/cap01/AdivinarNumero/GNU_Linux$
```

Figura 1.10 | Compile la aplicación **AdivinarNúmero** usando el comando g++.

```
~/ejemplos/cap01/AdivinarNumero/GNU_Linux$ ./AdivinarNumero
Tengo un numero entre 1 y 1000.
Puede adivinar mi numero?
Escriba su primera elección.
?
```

Figura 1.11 | Ejecución de la aplicación **AdivinarNúmero**.

```
~/ejemplos/cap01/AdivinarNumero/GNU_Linux$ ./AdivinarNumero
Tengo un numero entre 1 y 1000.
Puede adivinar mi numero?
Escriba su primera eleccion.
? 500
Demasiado alto. Intente de nuevo.
?
```

Figura 1.12 | Escriba su elección inicial.

```
~/ejemplos/cap01/AdivinarNumero/GNU_Linux$ ./AdivinarNumero
Tengo un numero entre 1 y 1000.
Puede adivinar mi numero?
Escriba su primera eleccion.
? 500
Demasiado alto. Intente de nuevo.
? 250
Demasiado bajo. Intente de nuevo.
?
```

Figura 1.13 | Escriba su segunda elección y reciba retroalimentación.

6. **Escriba más elecciones.** Continúe el juego (figura 1.14), escribiendo valores hasta que adivine el número correcto. Cuando adivine la respuesta correcta, la aplicación mostrará el mensaje "Excelente! Adivino el numero." (figura 1.14).

```
Demasiado bajo. Intente de nuevo.
? 375
Demasiado bajo. Intente de nuevo.
? 437
Demasiado alto. Intente de nuevo.
? 406
Demasiado alto. Intente de nuevo.
? 391
Demasiado alto. Intente de nuevo.
? 383
Demasiado bajo. Intente de nuevo.
? 387
Demasiado alto. Intente de nuevo.
? 385
Demasiado alto. Intente de nuevo.
? 384

Excelente! Adivino el numero!
```

Le gustaria jugar de nuevo (s o n)?

Figura 1.14 | Escriba más elecciones y adivine el número correcto.

7. **Juegue de nuevo o salga de la aplicación.** Una vez que adivine la respuesta correcta, la aplicación le preguntará si desea jugar otra vez. En el indicador "Le gustaria jugar de nuevo (s o n)?", al escribir el carácter **s** la aplicación elegirá un nuevo número y mostrará el mensaje "Escriba su primera eleccion.", seguido de un signo de interrogación como indicador (figura 1.15), de manera que pueda escribir su primera elección en el nuevo juego. Al escribir el carácter **n** la aplicación termina y el sistema nos regresa al directorio de la aplicación en el shell (figura 1.16). Cada vez que ejecute esta aplicación desde el principio (es decir, desde el *paso 3*), elegirá los mismos números para que usted adivine.

```

Excelente! Adivino el numero!
Le gustaria jugar de nuevo (s o n)? s

Tengo un numero entre 1 y 1000.
Puede adivinar mi numero?
Escriba su primera eleccion.
?
```

Figura I.15 | Juegue de nuevo el juego.

```

Excelente! Adivino el numero!
Le gustaria jugar de nuevo (s o n)? n

~/ejemplos/cap01/AdivinarNumero/GNU_Linux$
```

Figura I.16 | Salga del juego.

I.18 Tecnologías de software

En esta sección hablaremos sobre varias “palabras de moda” que escuchará en la comunidad de desarrollo de software. Creamos Centros de Recursos sobre la mayor parte de estos temas, y hay muchos por venir.

Agile Software Development (Desarrollo Ágil de Software) es un conjunto de metodologías que tratan de implementar software rápidamente, con menos recursos que las metodologías anteriores. Visite los sitios de Agile Alliance (www.agilealliance.org) y Agile Manifesto (www.agilemanifesto.org). También puede visitar el sitio en español www.agile-spain.com.

Refactoring (Refabricación) implica la reformulación del código para hacerlo más claro y fácil de mantener, al tiempo que se preserva su funcionalidad. Se emplea ampliamente con las metodologías de desarrollo ágil. Hay muchas herramientas de refabricación disponibles para realizar las porciones principales de la reformulación de manera automática.

Los patrones de diseño son arquitecturas probadas para construir software orientado a objetos flexible y que pueda mantenerse. El campo de los patrones de diseño trata de enumerar a los patrones recurrentes, y de alentar a los diseñadores de software para que los reutilicen y puedan desarrollar un software de mejor calidad con menos tiempo, dinero y esfuerzo.

Programación de juegos. El negocio de los juegos de computadora es más grande que el negocio de las películas de estreno. Ahora hay cursos universitarios, e incluso maestrías, dedicados a las técnicas sofisticadas de software que se utilizan en la programación de juegos. Vea nuestros Centros de Recursos sobre Programación de juegos y Proyectos de programación.

El software de código fuente abierto es un estilo de desarrollo de software que contrasta con el desarrollo propietario, que dominó los primeros años del software. Con el desarrollo de código fuente abierto, individuos y compañías contribuyen sus esfuerzos en el desarrollo, mantenimiento y evolución del software, a cambio del derecho de usar ese software para sus propios fines, comúnmente sin costo. Por lo general, el código fuente abierto se examina a detalle por una audiencia más grande que el software propietario, por lo cual los errores se eliminan con más rapidez. El código fuente abierto también promueve más innovación. Sun anunció recientemente que piensa abrir el código fuente de Java. Algunas de las organizaciones de las que se habla mucho en la comunidad de código fuente abierto son Eclipse Foundation (el IDE Eclipse es popular para el desarrollo de software en C++ y Java), Mozilla Foundation (creadores del explorador Web Firefox), Apache Software Foundation (creadores del servidor Web Apache) y SourceForge (que proporciona las herramientas para administrar proyectos de código fuente abierto y en la actualidad cuenta con más de 150,000 proyectos en desarrollo).

Linux es un sistema operativo de código fuente abierto, y uno de los más grandes éxitos de la iniciativa de código fuente abierto. **MySQL** es un sistema de administración de bases de datos con código fuente abierto. **PHP** es el lenguaje de secuencias de comandos del lado servidor para Internet de código fuente abierto más popular, para el desarrollo de aplicaciones basadas en Internet. **LAMP** es un acrónimo para el conjunto de tecnologías de código fuente abierto que utilizaron muchos desarrolladores para crear aplicaciones Web: representa a Linux, Apache, MySQL y PHP (o Perl, o Python; otros dos lenguajes que se utilizan para propósitos similares).

Ruby on Rails combina el lenguaje de secuencias de comandos Ruby con el marco de trabajo para aplicaciones Web Rails, desarrollado por la compañía 37Signals. Su libro, *Getting Real*, es una lectura obligatoria para los desarrolladores de aplicaciones Web de la actualidad; puede leerlo sin costo en gettingreal.37signals.com/toc.php. Muchos

desarrolladores de Ruby on Rails han reportado un considerable aumento en la productividad, en comparación con otros lenguajes al desarrollar aplicaciones Web con uso intensivo de bases de datos.

Por lo general, el software siempre se ha visto como un producto; la mayoría del software aún se ofrece de esta manera. Si desea ejecutar una aplicación, compre un paquete de software de un distribuidor. Después instale ese software en su computadora y lo ejecuta según sea necesario. Al aparecer nuevas versiones del software, usted lo actualiza, a menudo con un costo considerable. Este proceso puede volverse incómodo para empresas con decenas de miles de sistemas que deben mantenerse en una extensa colección de equipo de cómputo. Con **Software as a Service (SaaS)**, el software se ejecuta en servidores ubicados en cualquier parte de Internet. Cuando se actualiza ese servidor, todos los clientes a nivel mundial ven las nuevas características; no se necesita instalación local. Usted accede al servidor a través de un explorador Web; éstos son bastante portables, por lo que puede ejecutar las mismas aplicaciones en distintos tipos de computadoras, desde cualquier parte del mundo. Salesforce.com, Google, Microsoft Office Live y Windows Live ofrecen SaaS.

1.19 Programación de juegos con las bibliotecas Ogre

Ogre (**Motor de despliegue de gráficos orientado a objetos**), uno de los motores de gráficos más importantes, se ha utilizado en muchos productos comerciales, incluyendo videojuegos. Proporciona una interfaz orientada a objetos para la programación de gráficos en 3D, y se ejecuta en las plataformas Windows, Linux y Mac. Ogre es un proyecto de código fuente abierto, cuyo mantenimiento está a cargo del equipo Ogre en www.ogre3d.org. **OgreAL** es una envoltura alrededor de la **biblioteca de audio OpenAL**. La envoltura nos permite integrar la funcionalidad de sonido en el código de Ogre.

El capítulo 23, Programación de juegos con Ogre, introduce la programación de juegos y los gráficos con el motor Ogre para gráficos en 3D. Primero hablamos sobre las cuestiones básicas implicadas en la programación de juegos. Después le mostramos cómo utilizar Ogre para crear un juego simple, que presenta una mecánica de juego similar al clásico videojuego Pong®, desarrollado originalmente por Atari en 1972. Le demostramos cómo crear una escena con gráficos en 3D a colores, animar objetos de manera uniforme, usar temporizadores para controlar la velocidad de la animación, detectar colisiones entre los objetos, agregar sonido, aceptar entrada mediante el teclado y mostrar salida de texto.

1.20 Futuro de C++: Bibliotecas Boost de código fuente abierto, TR1 y C++0x

Bjarne Stroustrup, el creador de C++, ha expresado su visión para el futuro de C++. Las principales metas para el nuevo estándar son facilitar el aprendizaje de C++, mejorar las herramientas para generar bibliotecas e incrementar la compatibilidad con el lenguaje de programación C.

El capítulo 24, Bibliotecas Boost, Reporte técnico 1 y C++0x considera el futuro de C++: presentamos las Bibliotecas Boost de C++, el Reporte técnico 1 (TR1) y C++0x. Las **Bibliotecas Boost de C++** son bibliotecas gratuitas de código fuente abierto, creadas por miembros de la comunidad de C++. El tamaño de Boost ha crecido hasta 70 bibliotecas, y se agregan más con regularidad. Hoy en día hay miles de programadores con bibliotecas útiles y bien diseñadas, que funcionan bien con la Biblioteca estándar de C++ existente. Las bibliotecas Boost las pueden utilizar los programadores de C++ que trabajan en una amplia variedad de plataformas, con muchos compiladores distintos. En este capítulo vemos las generalidades sobre las bibliotecas incluidas en el TR1 y proporcionamos ejemplos de código para las bibliotecas de “expresiones regulares” y “apuntadores inteligentes”.

Las **expresiones regulares** se utilizan para relacionar patrones de caracteres específicos en el texto. Pueden usarse para validar datos y asegurar que se encuentren en un formato específico, para reemplazar partes de una cadena con otra, o para dividir una cadena.

Muchos errores comunes en el código de C y C++ están relacionados con los apuntadores, una poderosa herramienta de programación que estudiaremos en el capítulo 8, Apuntadores y cadenas basadas en apuntadores. Los **apuntadores inteligentes** nos ayudan a evitar errores, al proporcionar una funcionalidad adicional a los apuntadores estándar. Esta funcionalidad generalmente refuerza el proceso de asignación y desasignación de memoria.

El **Informe técnico 1** describe los cambios propuestos a la Biblioteca estándar de C++, muchos de los cuales se basan en las bibliotecas Boost actuales. Estas bibliotecas añaden una funcionalidad útil a C++. El Comité de estándares de C++ está revisando actualmente el Estándar de C++. El último estándar se publicó en 1998. El trabajo sobre el nuevo estándar, que se conoce actualmente como **C++0x**, empezó en 2003. Es muy probable que el nuevo estándar se publique en 2009. Este estándar incluirá modificaciones al lenguaje básico y, probablemente, muchas de las bibliotecas en el TR1.

1.21 Ejemplo práctico de Ingeniería de Software: introducción a la tecnología de objetos y el UML

Ahora empezaremos nuestra primera introducción al tema de la orientación a objetos, una manera natural de pensar acerca del mundo real y de escribir programas de cómputo. Los capítulos 1 a 7, 9 y 13 terminan con una sección breve titulada Ejemplo práctico de Ingeniería de Software, en la cual presentamos una introducción cuidadosamente guiada al tema de la orientación a objetos. Nuestro objetivo aquí es ayudarle a desarrollar una forma de pensar orientada a objetos, y de presentarle el **Lenguaje Unificado de Modelado™ (UML™)**, un lenguaje gráfico que permite a las personas que diseñan sistemas de software orientados a objetos utilizar una notación estándar en la industria para representarlos.

En esta sección requerida, presentamos los conceptos y la terminología de la orientación a objetos. Las secciones opcionales en los capítulos 2 a 7, 9 y 13 presentan un diseño y la implementación orientados a objetos del software para un sistema de cajero automático (ATM) simple. Las secciones tituladas Ejemplo práctico de Ingeniería de Software al final de los capítulos 2 al 7:

- analizan una especificación de requerimientos típica que describe un sistema de software (el ATM) que se va a construir
- determinan los objetos requeridos para implementar ese sistema
- determinan los atributos que deben tener estos objetos
- determinan los comportamientos que exhibirán estos objetos
- especifican la forma en que los objetos deben interactuar entre sí para cumplir con los requerimientos del sistema

Las secciones tituladas Ejemplo práctico de Ingeniería de Software al final de los capítulos 9 y 13 modifican y mejoran el diseño presentado en los capítulos 2 al 7. El apéndice G contiene una implementación completa y funcional en C++ del sistema ATM orientado a objetos.

Aunque nuestro ejemplo práctico es una versión reducida de un problema a nivel industrial, de todas formas cubrimos muchas prácticas comunes en la industria. Usted experimentará una sólida introducción al diseño orientado a objetos con UML. Además, afinará sus habilidades para leer código al ver paso a paso la implementación en C++ del ATM, cuidadosamente escrita y bien documentada.

Conceptos básicos de la tecnología de objetos

Comenzaremos nuestra introducción al tema de la orientación a objetos con cierta terminología clave. En cualquier parte del mundo real puede ver **objetos**: gente, animales, plantas, autos, aviones, edificios, computadoras, monitores, etc. Los humanos pensamos en términos de objetos. Los teléfonos, casas, semáforos, hornos de microondas y enfriadores de agua son sólo unos cuantos objetos más que vemos a nuestro alrededor todos los días.

Algunas veces dividimos a los objetos en dos categorías: animados e inanimados. Los objetos animados están “vivos” en cierto sentido; se mueven a su alrededor y hacen cosas. Los objetos inanimados no se mueven por su propia cuenta. Sin embargo, los objetos de ambos tipos tienen ciertas cosas en común. Todos ellos tienen **atributos** (como tamaño, forma, color y peso), y todos exhiben **comportamientos** (por ejemplo, una pelota rueda, rebota, se infla y desinfla; un bebé llora, duerme, gatea, camina y parpadea; un auto acelera, frena y da vuelta; una toalla absorbe agua). Estudiaremos los tipos de atributos y comportamientos que tienen los objetos de software.

Los humanos aprenden acerca de los objetos existentes estudiando sus atributos y observando sus comportamientos. Distintos objetos pueden tener atributos similares y pueden exhibir comportamientos similares. Por ejemplo, pueden hacerse comparaciones entre los bebés y los adultos, y entre los humanos y los chimpancés.

El **diseño orientado a objetos (OO)** modela el software en términos similares a los que utilizan las personas para describir objetos del mundo real. Este diseño aprovecha las relaciones entre las clases, donde los objetos de cierta clase (como una clase de vehículos) tienen las mismas características; los autos, camiones, pequeños vagones rojos y patines tienen mucho en común. El DOO también aprovecha las relaciones de **herencia**, donde las nuevas clases de objetos se derivan absorbiendo las características de las clases existentes y agregando sus propias características únicas. Un objeto de la clase “convertible” ciertamente tiene las características de la clase más general “automóvil” pero, de manera más específica, el techo de un convertible puede ponerse y quitarse.

El diseño orientado a objetos proporciona una manera natural e intuitiva de ver el proceso de diseño de software: a saber, modelando los objetos por sus atributos, comportamientos e interrelaciones, de igual forma que como describimos los objetos del mundo real. El DOO también modela la comunicación entre los objetos. Así como las personas se envían mensajes unas a otras (por ejemplo, un sargento ordenando a un soldado que permanezca firme), los objetos también

se comunican mediante mensajes. Un objeto cuenta de banco puede recibir un mensaje para reducir su saldo por cierta cantidad, debido a que el cliente ha retirado esa cantidad de dinero.

El DOO **encapsula** (es decir, envuelve) los atributos y las **operaciones** (comportamientos) en los objetos; los atributos y las operaciones de un objeto se enlazan íntimamente entre sí. Los objetos tienen la propiedad de **ocultamiento de información**. Esto significa que los objetos pueden saber cómo comunicarse entre sí a través de **interfaces** bien definidas, pero por lo general no se les permite saber cómo se implementan otros objetos; los detalles de la implementación se ocultan dentro de los mismos objetos. Por ejemplo, podemos conducir un auto con efectividad, sin necesidad de saber los detalles acerca de cómo funcionan internamente los motores, las transmisiones y los sistemas de escape; siempre y cuando sepamos cómo usar el pedal del acelerador, el pedal del freno, el volante, etc. Más adelante veremos por qué el ocultamiento de información es tan imprescindible para la buena ingeniería de software.

Los lenguajes como C++ son **orientados a objetos**. La programación en dichos lenguajes se llama **programación orientada a objetos (POO)**, y permite a los programadores de computadoras implementar un diseño orientado a objetos en forma de sistemas de software funcionales. Por otra parte, los lenguajes como C son **por procedimientos**, de manera que la programación tiende a ser **orientada a la acción**. En C, la unidad de programación es la **función**. En C++, la unidad de programación es la **clase** a partir de la cual se *instancian* (un término de POO para “crean”) los objetos en un momento dado. Las clases en C++ contienen funciones que implementan operaciones y datos que implementan atributos.

Los programadores de C se concentran en escribir funciones. Los programadores agrupan acciones que realizan ciertas tareas comunes en funciones, y agrupan las funciones para formar programas. Evidentemente, los datos son importantes en C, pero la visión es que los datos existen principalmente para apoyar a las acciones que realizan las funciones. Los **verbos** en una especificación de sistema ayudan al programador de C a determinar el conjunto de funciones que trabajarán juntas para implementar el sistema.

Clases, miembros de datos y funciones miembro

Los programadores de C++ se concentran en crear sus propios **tipos definidos por el usuario**, denominados **clases**. Cada clase contiene datos, además del conjunto de funciones que manipulan esos campos y proporcionan servicios a **clientes** (es decir, otras clases o funciones que utilizan esa clase). Los componentes de datos de una clase se llaman **miembros de datos**. Por ejemplo, una clase cuenta de banco podría incluir un número de cuenta y un saldo. Los componentes de las funciones de una clase se llaman **funciones miembro** (por lo general se les llama **métodos** en otros lenguajes de programación orientada a objetos, como Java). Por ejemplo, una clase cuenta de banco podría incluir funciones miembro para realizar un depósito (aumentar el saldo), realizar un retiro (reducir el saldo) y consultar cuál es el saldo actual. Utilizamos los tipos integrados (y otros tipos definidos por el usuario) como “bloques de construcción” para construir nuevos tipos definidos por el usuario (clases). Los **sustantivos** en una especificación de sistema ayudan al programador de C++ a determinar el conjunto de clases a partir de las que se crean los objetos que funcionan en conjunto para implementar el sistema.

Las clases son para los objetos lo que los planos de construcción son para las casas; una clase es un “plano” para construir un objeto de esa clase. Así como podemos construir muchas casas a partir de un plano de construcción, podemos instanciar (crear) muchos objetos a partir de una clase. No puede cocinar alimentos en la cocina de un plano de construcción; puede cocinar alimentos en la cocina de una casa. No puede dormir en la recámara de un plano de construcción; puede dormir en la recámara de una casa.

Las clases pueden tener relaciones con otras clases. Por ejemplo, en un diseño orientado a objetos de un banco, la clase “cajero” necesita relacionarse con otras clases, como la clase “cliente”, la clase “cajón de efectivo”, la clase “bóveda”, y así en lo sucesivo. A estas relaciones se les llama **asociaciones**.

Al empaquetar el software en forma de clases, los sistemas de software posteriores pueden **reutilizar** esas clases. Los grupos de clases relacionadas se empaquetan comúnmente como **componentes** reutilizables. Así como los corredores de bienes raíces dicen a menudo que los tres factores más importantes que afectan el precio de los bienes raíces son “la ubicación, la ubicación y la ubicación”, algunas personas en la comunidad de desarrollo de software dicen a menudo que los tres factores más importantes que afectan el futuro del desarrollo de software son “la reutilización, la reutilización y la reutilización”.



Observación de Ingeniería de Software 1.4

Reutilizar las clases existentes cuando se crean nuevas clases y programas es un proceso que ahorra tiempo, dinero y esfuerzo.

La reutilización también ayuda a los programadores a crear sistemas más confiables y efectivos, ya que las clases y componentes existentes a menudo han pasado por un proceso extenso de prueba, depuración y optimización del rendimiento.

Evidentemente, con la tecnología de objetos podemos crear la mayor parte del software que necesitaremos mediante la combinación de clases existentes, así como los fabricantes de automóviles combinan las piezas intercambiables. Cada nueva clase que usted cree tendrá el potencial de convertirse en una valiosa pieza de software, que usted y otros programadores podrán usar para agilizar y mejorar la calidad de los futuros esfuerzos de desarrollo de software.

Introducción al análisis y diseño orientados a objetos (A/DOO)

Pronto estará escribiendo programas en C++. ¿Cómo creará el código para sus programas? Tal vez, como muchos programadores principiantes, simplemente encenderá su computadora y empezará a teclear. Esta metodología puede funcionar para programas pequeños (como los que presentamos en los primeros capítulos del libro) pero ¿qué haría usted si se le pidiera crear un sistema de software para controlar miles de máquinas de cajero automático para un importante banco? O suponga que le pidieron trabajar en un equipo de 1,000 desarrolladores de software para construir el nuevo sistema de control de tráfico aéreo de Estados Unidos. Para proyectos tan grandes y complejos, no podría simplemente sentarse y empezar a escribir programas.

Para crear las mejores soluciones, debe seguir un proceso detallado para **analizar los requerimientos** de su proyecto (es decir, determinar *qué* es lo que se supone debe hacer el sistema) y desarrollar un **diseño** que cumpla con esos requerimientos (es decir, decidir *cómo* debe hacerlo el sistema). Idealmente usted pasaría por este proceso y revisaría cuidadosamente el diseño (o haría que otros profesionales de software lo revisaran) antes de escribir cualquier código. Si este proceso implica analizar y diseñar su sistema desde un punto de vista orientado a objetos, lo llamamos un **proceso de análisis y diseño orientado a objetos (A/DOO)**. Los programadores experimentados saben que el análisis y el diseño pueden ahorrar innumerables horas, ya que les ayudan a evitar un método de desarrollo de un sistema mal planeado, que tiene que abandonarse en plena implementación, con la posibilidad de desperdiciar una cantidad considerable de tiempo, dinero y esfuerzo.

A/DOO es el término genérico para el proceso de analizar un problema y desarrollar un método para resolverlo. Los pequeños problemas como los que se describen en los primeros capítulos de este libro no requieren de un proceso exhaustivo de A/DOO. Podría ser suficiente con escribir **pseudocódigo** antes de empezar a escribir el código en C++. El pseudocódigo es un medio informal de expresar la lógica de un programa. En realidad no es un lenguaje de programación, pero podemos usarlo como un tipo de bosquejo para guiarnos a medida que escribimos nuestro código. En el capítulo 4, Instrucciones de control: parte 1, presentamos el pseudocódigo.

A medida que los problemas y los grupos de personas que los resuelven aumentan en tamaño, los métodos de A/DOO se vuelven más apropiados que el pseudocódigo. Idealmente, los miembros de un grupo deberían acordar un proceso estrictamente definido para resolver su problema, y acordar también una manera uniforme para que los miembros del grupo se comuniquen los resultados de ese proceso entre sí. Aunque existen muchos procesos de A/DOO distintos, hay un solo lenguaje gráfico para comunicar los resultados de *cualquier* proceso A/DOO que se ha vuelto muy popular. Este lenguaje, conocido como Lenguaje Unificado de Modelado (UML), se desarrolló a mediados de la década de los noventa, bajo la dirección inicial de tres metodologistas de software: Grady Booch, James Rumbaugh e Ivar Jacobson.

Historia de UML

En la década de los ochenta, un creciente número de empresas comenzó a utilizar la POO para crear sus aplicaciones, lo cual generó la necesidad de un proceso estándar de A/DOO. Muchos metodologistas (incluyendo a Booch, Rumbaugh y Jacobson) produjeron y promocionaron, por su cuenta, procesos separados para satisfacer esta necesidad. Cada uno de estos procesos tenía su propia notación, o “lenguaje” (en forma de diagramas gráficos), para transmitir los resultados del análisis y el diseño.

A principios de la década de los noventa, diversas compañías (e inclusive diferentes divisiones dentro de la misma compañía) utilizaban sus propios procesos y notaciones únicos. Al mismo tiempo, estas compañías querían utilizar herramientas de software que tuvieran soporte para sus procesos particulares. Con tantos procesos, se les dificultó a los distribuidores de software proporcionar dichas herramientas. Evidentemente era necesario contar con una notación y procesos estándar.

En 1994, James Rumbaugh se unió con Grady Booch en Rational Software Corporation (ahora una división de IBM), y los dos comenzaron a trabajar para unificar sus populares procesos. Pronto se unió a ellos Ivar Jacobson. En 1996, el grupo liberó las primeras versiones de UML para la comunidad de ingeniería de software, y solicitaron retroalimentación. Casi al mismo tiempo, una organización conocida como **Object Management Group™ (OMG™, Grupo de administración de objetos)** hizo una invitación para participar en la creación de un lenguaje común de modelado. El OMG (www.omg.org) es una organización sin fines de lucro que promueve la estandarización de las tecnologías orientadas a objetos, emitiendo lineamientos y especificaciones como UML. Varias empresas (entre ellas HP, IBM, Microsoft, Oracle y Rational Software) habían reconocido ya la necesidad de un lenguaje común de modelado. Estas

compañías formaron el consorcio **UML Partners (Socios de UML)** en respuesta a la solicitud de proposiciones por parte del OMG (el consorcio que desarrolló la versión 1.1 de UML y la envió al OMG). El OMG aceptó la proposición y, en 1997, asumió la responsabilidad del mantenimiento y revisión de UML en forma continua. La versión 2 de UML que está ahora disponible marca la primera modificación mayor al UML desde el estándar de la versión 1.1 de 1997. A lo largo de este libro, presentaremos la terminología y notación de UML 2.

¿Qué es el UML?

El UML es ahora el esquema de representación gráfica más utilizado para modelar sistemas orientados a objetos. Evidentemente ha unificado los diversos esquemas de notación populares. Aquellos quienes diseñan sistemas utilizan el lenguaje (en forma de diagramas) para modelar sus sistemas.

Una característica atractiva de UML es su flexibilidad. UML es **extensible** (es decir, capaz de mejorarse con nuevas características) e independiente de cualquier proceso de A/DOO específico. Los modeladores de UML tienen la libertad de diseñar sistemas utilizando varios procesos, pero todos los desarrolladores pueden ahora expresar esos diseños con un conjunto de notaciones gráficas estándar.

UML es un lenguaje gráfico complejo, con muchas características. En nuestras secciones del Ejemplo práctico de Ingeniería de Software, presentamos un subconjunto conciso y simplificado de estas características. Luego utilizamos este subconjunto para guiar al lector a través de la experiencia de su primer diseño con UML, la cual está dirigida a los programadores principiantes orientados a objetos en cursos de programación de primer o segundo semestre.

Recursos Web de UML

Para obtener más información acerca de UML, consulte los sitios Web que se enlistan a continuación. Para ver sitios Web adicionales sobre UML, consulte los recursos Web que se enlistan al final de la sección 2.8.

www.uml.org

Esta página de recursos de UML del Grupo de Administración de Objetos (OMG) proporciona documentos de la especificación para UML y otras tecnologías orientadas a objetos.

www.ibm.com/software/rational/uml

Ésta es la página de recursos de UML para IBM Rational, sucesor de Rational Software Corporation (la compañía que creó a UML).

Lecturas recomendadas

Los siguientes libros proporcionan información acerca del diseño orientado a objetos con UML:

Ambler, S. *The Object Primer: Agile Model-Driven Development with UML 2.0, Third Edition*. Nueva York: Cambridge University Press, 2005.

Arlow, J. e I. Neustadt. *UML and the Unified Process: Practical Object-Oriented Analysis and Design, Second Edition*. Boston: Addison-Wesley Professional, 2006.

Fowler, M. *UML Distilled, Third Edition: A Brief Guide to the Standard Object Modeling Language*. Boston: Addison-Wesley Professional, 2004.

Rumbaugh, J., I. Jacobson y G. Booch. *The Unified Modeling Language User Guide, Second Edition*. Boston: Addison-Wesley Professional, 2006.

Ejercicios de autorrepaso de la sección 1.21

1.1 Enliste tres ejemplos de objetos reales que no mencionamos. Para cada objeto, incluya varios atributos y comportamientos.

1.2 El pseudocódigo es _____.

- otro término para el A/DOO.
- un lenguaje de programación utilizado para visualizar diagramas de UML.
- un medio informal de expresar la lógica de un programa.
- un esquema de representación gráfica para modelar sistemas orientados a objetos.

1.3 El UML se utiliza principalmente para _____.

- probar sistemas orientados a objetos.
- diseñar sistemas orientados a objetos.
- implementar sistemas orientados a objetos.
- a y b.

Respuestas a los ejercicios de autorrepaso de la sección 1.21

1.1 [Nota: las respuestas pueden variar.] a) Los atributos de un televisor incluyen el tamaño de la pantalla, el número de colores que puede mostrar, su canal actual y su volumen actual. Un televisor se enciende y se apaga, cambia de canales, muestra video y reproduce sonidos. b) Los atributos de una cafetera incluyen el volumen máximo de agua que puede contener, el tiempo requerido para preparar una jarra de café y la temperatura del plato calentador bajo la jarra de café. Una cafetera se enciende y se apaga, prepara café y lo calienta. c) Los atributos de una tortuga incluyen su edad, el tamaño de su caparazón y su peso. Una tortuga camina, se mete en su caparazón, emerge del mismo y come vegetación.

1.2 c.

1.3 b.

I.22 Repaso

Este capítulo presentó los conceptos básicos de hardware y software. Aquí estudió la historia de Internet y World Wide Web, y aprendió acerca del fenómeno Web 2.0. Hablamos sobre los diferentes tipos de lenguajes de programación, su historia y cuáles son los más utilizados. También hablamos sobre la Biblioteca estándar de C++, que contiene clases y funciones reutilizables que ayudan a los programadores de C++ a crear programas en C++ portables.

Presentamos los conceptos básicos sobre la tecnología de objetos, incluyendo las clases, los objetos, los atributos, los comportamientos, el encapsulamiento y la herencia. También aprendió sobre la historia y el propósito de UML: el lenguaje gráfico estándar en la industria para modelar sistemas de software.

Conoció los pasos típicos para crear y ejecutar una aplicación de C++. Realizó pruebas de una aplicación de C++ de ejemplo, similar a los tipos de aplicaciones que aprenderá a programar en este libro.

Hablamos sobre varias tecnologías y conceptos clave de software, incluyendo el código fuente abierto, y dimos un vistazo al futuro de C++. En capítulos posteriores estudiará dos bibliotecas de código fuente abierto: Ogre para programación de gráficos y juegos, y Boost para mejorar ampliamente las herramientas de la Biblioteca estándar de C++.

En el siguiente capítulo creará sus primeras aplicaciones en C++. Verá varios ejemplos que muestran cómo los programas imprimen mensajes en pantalla y obtienen información del usuario mediante el teclado para procesarla. Analizaremos y explicaremos cada ejemplo, para facilitarle el proceso de aprender a programar en C++.

I.23 Recursos Web

Esta sección proporciona muchos recursos que le serán de utilidad a medida que aprenda C++. Los sitios incluyen recursos de C++, herramientas de desarrollo de C++ para estudiantes y profesionales, y algunos vínculos a divertidos juegos creados con C++. En esta sección también enlistamos nuestros propios sitios Web, donde podrá encontrar descargas y recursos asociados con este libro.

Sitios Web de Deitel & Associates

www.deitel.com/books/cpphtp6/

El sitio Web de *Cómo programar en C++, 6/e* de Deitel & Associates. Aquí encontrará vínculos a los ejemplos del libro y otros recursos, como nuestras guías *Dive Into™* que le ayudarán a empezar a trabajar con varios entornos de desarrollo integrado (IDEs) de C++.

www.deitel.com/cplusplus/

www.deitel.com/cplusplusgameprogramming/

www.deitel.com/cplusplusboostlibraries/

www.deitel.com/codesearchengines

www.deitel.com/programmingprojects

Los Centros de Recursos de Deitel C++ y centros relacionados actualmente disponibles en www.deitel.com. Empiece aquí su búsqueda de recursos, descargas, tutoriales, documentación, libros, libros electrónicos, diarios, artículos, blogs, canales (feeds) RSS y demás, que le ayudarán a desarrollar aplicaciones de C++.

www.deitel.com

Revise el sitio Web de Deitel & Associates para actualizaciones, correcciones y recursos adicionales para todas las publicaciones Deitel.

www.deitel.com/newsletter/subscribe.html

Suscríbase al boletín de correo electrónico gratuito *Deitel® Buzz Online*, para seguir el programa de publicaciones de Deitel & Associates, incluyendo actualizaciones y fe de erratas para este libro.

Compiladores y herramientas de desarrollo

www.thefreecountry.com/developercity/ccompilers.shtml

Este sitio enlista compiladores gratuitos de C y C++ para una variedad de sistemas operativos.

msdn.microsoft.com/vstudio/express/visualc/default.aspx

El sitio *Microsoft Visual C++ Express* ofrece una descarga gratuita de *Visual C++ Express Edition*, información sobre el producto, generalidades y materiales supplementarios para Visual C++.

www.microsoft.com/spanish/msdn/vstudio/express/VC/default.mspx

Sitio de *Microsoft Visual C++ Express Edition* en español.

www.codegear.com/products/cppbuilder

Éste es un vínculo al sitio Web de *Code Gear C++ Builder*.

www.compilers.net

Compilers.net está diseñado para ayudar a los usuarios a localizar compiladores.

developer.intel.com/software/products/compilers/cwin/index.htm

Hay una descarga de evaluación del *compilador Intel C++* disponible en este sitio Web.

Recursos

www.ha19k.com/cug

El sitio *C/C++ Users Group (CUG)* contiene recursos y revistas sobre C++, shareware y freeware.

www.devx.com

DevX es un recurso extenso para programadores, que proporciona las noticias, herramientas y técnicas más recientes para diversos lenguajes de programación. *C++ Zone* ofrece tips, foros de discusión, ayuda técnica y boletines de noticias en línea.

www.acm.org/crossroads/xrds3-2/ovp32.html

El sitio de la asociación para maquinaria de cómputo (*Association for Computing Machinery, ACM*) ofrece un extenso listado de recursos sobre C++, incluyendo textos, diarios y revistas recomendados, estándares publicados, boletines de noticias, FAQs y grupos de noticias.

www.accu.informika.ru/resources/public/terse/cpp.htm

El sitio de la asociación de usuarios de C y C++ (*Association of C & C++ Users, ACCU*) contiene vínculos a tutoriales, artículos, información de desarrolladores, discusiones y reseñas de libros sobre C++.

www.cuj.com

El diario de usuario de C/C++ (*C/C++ User's Journal*) es una revista en línea que contiene artículos, tutoriales y descargas. El sitio presenta noticias acerca de C++, foros y vínculos a información acerca de las herramientas de desarrollo.

www.research.att.com/~bs/homepage.html

Éste es el sitio de Bjarne Stroustrup, diseñador del lenguaje de programación C++. Este sitio contiene una lista de recursos sobre C++, FAQs y demás información de utilidad.

Juegos y programación de juegos

www.codearchive.com/list.php?go=0708

Este sitio tiene varios juegos de C++ disponibles para descargar.

www.mathtools.net/C_C__/Games/

Este sitio incluye vínculos a muchos juegos creados con C++. El código fuente para la mayoría de los juegos está disponible para descargar.

www.gametutorials.com/gtstore/c-3-c-tutorials.aspx

Este sitio Web tiene tutoriales acerca de la programación de juegos en C++. Cada tutorial incluye una descripción del juego y una lista de las funciones y métodos utilizados en el tutorial.

Resumen

Sección 1.1 Introducción

- Las computadoras (comúnmente conocidas como hardware) se controlan mediante software (es decir, las instrucciones que escribimos para ordenar a la computadora que realice acciones y tome decisiones).
- C++ es uno de los lenguajes de desarrollo de software más populares en la actualidad.
- Los costos de las computadoras se han reducido considerablemente, debido a los rápidos desarrollos en las tecnologías de hardware y software.
- La tecnología de los chips de silicio ha hecho que la computación sea tan económica, que hay aproximadamente mil millones de computadoras de propósito general en uso en todo el mundo.

- La orientación a objetos es la metodología de programación clave que utilizan los programadores hoy en día.

Sección 1.2 ¿Qué es una computadora?

- Una computadora es capaz de realizar cálculos y tomar decisiones lógicas a velocidades de miles de millones de veces más rápidas que los humanos.
- Las computadoras procesan los datos bajo el control de conjuntos de instrucciones llamadas programas de cómputo, que guían a las computadoras a través de conjuntos ordenados de acciones especificadas por gente llamada programadores de computadoras.
- Los diversos dispositivos que componen a un sistema de cómputo se denominan hardware.
- A los programas que se ejecutan en una computadora se les denomina software.

Sección 1.3 Organización de una computadora

- La unidad de entrada es la sección “receptora” de la computadora. Obtiene información desde los dispositivos de entrada y pone esta información a disposición de las otras unidades para que pueda procesarse.
- La unidad de salida es la sección de “embarque” de la computadora. Toma información que ya ha sido procesada por la computadora y la coloca en los diferentes dispositivos de salida, para que esté disponible fuera de la computadora.
- La unidad de memoria es la sección de “almacén” de acceso rápido, pero con relativa baja capacidad, de la computadora. Retiene la información que se introduce a través de la unidad de entrada, para que la información pueda estar disponible de manera inmediata para procesarla cuando sea necesario, y retiene la información procesada hasta que ésta pueda ser colocada en los dispositivos de salida por la unidad de salida.
- La unidad aritmética y lógica (ALU) es la sección de “manufactura” de la computadora. Es responsable de realizar cálculos y tomar decisiones.
- La unidad central de procesamiento (CPU) es la sección “administrativa” de la computadora. Coordina y supervisa la operación de las demás secciones.
- La unidad de almacenamiento secundario es la sección de “almacén” de alta capacidad y de larga duración de la computadora. Los programas o datos que no se encuentran en ejecución por las otras unidades, normalmente se colocan en dispositivos de almacenamiento secundario (como discos) hasta que son requeridos de nuevo, posiblemente horas, días, meses o incluso años después.

Sección 1.4 Los primeros sistemas operativos

- Las primeras computadoras eran capaces de realizar solamente una tarea o trabajo a la vez. A esto se le conoce comúnmente como procesamiento por lotes para un solo usuario.
- Los sistemas operativos se desarrollaron para facilitar el uso de la computadora.
- La multiprogramación implica compartir los recursos de una computadora entre los trabajos que compiten por su atención, de manera que éstos parezcan ejecutarse en forma simultánea.
- El tiempo compartido es un caso especial de multiprogramación, en el cual los usuarios utilizan la computadora a través de terminales, que por lo general son dispositivos con teclados y pantallas. Docenas, e incluso hasta cientos de usuarios, comparten la computadora a la vez. En realidad, la computadora no ejecuta todas las tareas de los usuarios al mismo tiempo. En lugar de ello, ejecuta una pequeña porción del trabajo de un usuario y después procede a dar servicio al siguiente usuario, con la posibilidad de proporcionar el servicio a cada usuario varias veces por segundo. Así, los programas de los usuarios parecen ejecutarse en forma simultánea.

Sección 1.5 Computación personal, distribuida y cliente/servidor

- Apple Computer popularizó la computación personal.
- La Computadora Personal de IBM legitimó rápidamente la computación personal en las empresas, en la industria y en las organizaciones gubernamentales, en donde se utilizaban mucho las mainframes de IBM.
- Aunque las primeras computadoras personales no eran lo suficientemente poderosas como para compartir su tiempo entre varios usuarios, estas máquinas podían enlazarse entre sí en redes computacionales, algunas veces a través de líneas telefónicas, y otras mediante redes de área local (LANs) dentro de una empresa. Esto condujo al fenómeno de la computación distribuida.
- Las computadoras de hoy en día son tan poderosas como las máquinas de un millón de dólares de hace apenas unas cuantas décadas, y la información se comparte fácilmente a través de redes computacionales.
- C++ se está utilizando ampliamente para escribir software para sistemas operativos, redes de computadoras y para aplicaciones cliente/servidor distribuidas.

Sección 1.6 Internet y World Wide Web

- Internet (una red global de computadoras) se inició hace casi cuatro décadas, con fondos suministrados por el Departamento de Defensa de Estados Unidos.

- Con la introducción de World Wide Web (que permite a los usuarios de computadora localizar y ver documentos basados en multimedia, sobre casi cualquier tema a través de Internet), Internet se ha convertido explosivamente en uno de los principales mecanismos de comunicación en todo el mundo.

Sección 1.7 Web 2.0

- Web 2.0 no tiene una sola definición, pero puede explicarse a través de una serie de tendencias en Internet, una de las cuales es el otorgamiento de poder al usuario. Empresas como eBay se basan casi completamente en el contenido generado por la comunidad.
- Web 2.0 aprovecha la inteligencia colectiva, la idea de que la colaboración producirá ideas inteligentes.
- El marcado, o etiquetado de contenido, es otra parte clave del tema de colaboración de Web 2.0.
- Los sitios de redes sociales, que llevan la cuenta de las relaciones interpersonales de los usuarios, han experimentado un extraordinario crecimiento como parte de Web 2.0.
- Los sitios de medios sociales también han obtenido una inmensa popularidad, debido al aumento en la disponibilidad y el uso de Internet de banda ancha, a la cual se le conoce comúnmente como Internet de alta velocidad.
- Los blogs, sitios Web caracterizados por la publicación de mensajes breves en orden cronológico inverso, se han convertido en un importante fenómeno social dentro de Web 2.0. Muchos bloggers se reconocen como parte de los medios, y las empresas se están volviendo a la blogósfera para rastrear las opiniones de los consumidores.
- El aumento en la popularidad del software de código fuente abierto ha contribuido a que sea más económico y sencillo empezar compañías Web 2.0. Los servicios Web están a la alza, en los que se da preferencia al “webtop” en lugar del escritorio convencional (desktop) en la mayor parte del nuevo desarrollo.
- Los mashups combinan dos o más aplicaciones Web existentes para servir un nuevo propósito, y dependen de pequeñas piezas modulares y de un acceso abierto a las APIs de los servicios Web, las cuales permiten a los desarrolladores integrar otros servicios Web en sus aplicaciones.
- Muchas compañías Web 2.0 utilizan la publicidad como su fuente principal de monetización.

Sección 1.8 Lenguajes máquina, lenguajes ensambladores y lenguajes de alto nivel

- Cualquier computadora puede entender de manera directa sólo su propio lenguaje de máquina, que por lo general consiste en cadenas de números que instruyen a las computadoras para que realicen sus operaciones más elementales.
- Las abreviaciones en inglés cotidiano forman la base de los lenguajes ensambladores. Los programas traductores llamados ensambladores convierten programas de lenguaje ensamblador a lenguaje de máquina.
- Los compiladores traducen los programas en lenguaje de alto nivel a programas en lenguaje de máquina. Los lenguajes de alto nivel (como C++) contienen palabras en inglés y notaciones matemáticas convencionales.
- Los programas intérpretes ejecutan los programas en lenguajes de alto nivel directamente, con lo cual se elimina la necesidad de compilarlos en lenguaje de máquina.

Sección 1.9 Historia de C y C++

- C++ evolucionó de C, que a su vez evolucionó de dos lenguajes de programación anteriores, BCPL y B.
- C++ es una extensión de C, desarrollado por Bjarne Stroustrup a principios de la década de los ochenta en los laboratorios Bell. Este lenguaje mejora el lenguaje C y proporciona capacidades para la programación orientada a objetos.
- Los objetos son componentes de software reutilizables que modelan elementos en el mundo real. El uso de una metodología de diseño e implementación modular, orientada a objetos, puede hacer que los grupos de desarrollo de software sean más productivos que con las técnicas de programación anteriores.

Sección 1.10 Biblioteca estándar de C++

- Los programas de C++ consisten en piezas llamadas clases y funciones. Usted puede programar cada pieza que pueda necesitar para formar un programa de C++. Sin embargo, la mayoría de los programadores de C++ aprovechan las extensas colecciones de clases y funciones existentes en la Biblioteca estándar de C++.

Sección 1.11 Historia de Java

- Java se utiliza para crear contenido dinámico e interactivo para páginas Web, desarrollar aplicaciones empresariales, mejorar la funcionalidad de los servidores Web, proporcionar aplicaciones para los dispositivos domésticos y para muchos otros propósitos.

Sección 1.12 FORTRAN, COBOL, Pascal y Ada

- FORTRAN (FORmula TRANslator, Traductor de fórmulas) fue desarrollado por IBM Corporation a mediados de la década de los cincuenta para utilizarse en aplicaciones científicas y de ingeniería que requerían cálculos matemáticos complejos.
- COBOL (COmmon Business Oriented Language, Lenguaje común orientado a negocios) fue desarrollado a finales de la década de 1950, por un grupo de fabricantes de computadoras y usuarios de agencias gubernamentales y de la

industria. Se utiliza principalmente en aplicaciones comerciales que requieren de una manipulación precisa y eficiente de datos.

- Ada se desarrolló bajo el patrocinio del Departamento de Defensa de Estados Unidos (DoD) durante la década de los setenta y a principios de los ochenta. Ada cuenta con la multitarea, que permite a los programadores especificar que muchas actividades ocurrirán en paralelo.

Sección 1.13 BASIC, Visual Basic, Visual C++, C# y .NET

- BASIC (Código de instrucciones simbólicas de propósito general para principiantes) fue desarrollado a mediados de la década de los sesenta en Dartmouth College, como un lenguaje para escribir programas simples. El principal objetivo de BASIC era familiarizar a los principiantes con las técnicas de programación.
- El lenguaje Visual Basic de Microsoft se introdujo a principios de la década de 1990 para simplificar el proceso de desarrollar aplicaciones para Microsoft Windows.
- Microsoft cuenta con una estrategia a nivel corporativo para integrar Internet y Web en las aplicaciones de computadora. Esta estrategia se implementa en la plataforma .NET de Microsoft.
- Los tres principales lenguajes de programación de la plataforma .NET son Visual Basic (basado en el BASIC original), Visual C++ (basado en C++) y Visual C# (un nuevo lenguaje basado en C++ y Java, desarrollado explícitamente para la plataforma .NET).
- Los desarrolladores de .NET pueden escribir componentes de software en su lenguaje preferido y después formar aplicaciones, al combinar esos componentes con componentes escritos en cualquier lenguaje .NET.

Sección 1.14 Tendencia clave de software: tecnología de los objetos

- No fue sino hasta que la tecnología de la programación orientada a objetos se empezó a utilizar ampliamente en la década de los noventa, que los desarrolladores de software sintieron que tenían las herramientas necesarias para avanzar con grandes pasos en el proceso de desarrollo de software.
- La tecnología de objetos se remonta hasta a mediados de la década de 1960. El lenguaje de programación C++, desarrollado en AT&T por Bjarne Stroustrup a principios de la década de 1980, se basa en dos lenguajes (C, que se desarrolló inicialmente en AT&T para implementar el sistema operativo UNIX a principios de la década de 1970, y Simula 67, un lenguaje de programación de simulación, desarrollado en Europa y puesto en circulación en 1967). C++ absorbió las características de C y agregó las herramientas de Simula para crear y manipular objetos.
- Ninguno de los lenguajes C o C++ estaba diseñado originalmente para usarse ampliamente, más allá de los laboratorios de investigación de AT&T, pero el soporte básico se desarrolló rápidamente para cada uno de ellos.
- La tecnología de objetos es un esquema de empaquetamiento que nos ayuda a crear unidades de software con significado.
- Un problema clave con la programación por procedimientos es que las unidades de los programas no reflejan con efectividad a las entidades del mundo real, por lo que estas unidades no son especialmente reutilizables.
- Con la tecnología de los objetos, las entidades de software creadas (llamadas clases), si se diseñan en forma apropiada, tienden a ser reutilizables en proyectos futuros. El uso de bibliotecas de componentes reutilizables puede reducir en forma considerable el esfuerzo requerido para implementar ciertos tipos de sistemas.
- Algunas empresas reportan que el beneficio clave que les proporciona la programación orientada a objetos es la producción de software más comprensible, mejor organizado y más fácil de mantener, modificar y depurar. Se ha estimado que tal vez hasta 80 por ciento de los costos de software están asociados no con los esfuerzos originales por desarrollar el software, sino con la evolución continua y el mantenimiento de ese software a lo largo de su tiempo de vida.

Sección 1.15 Entorno de desarrollo típico en C++

- Por lo general, los sistemas de C++ consisten en tres partes: un entorno de desarrollo de programas, el lenguaje y la Biblioteca estándar de C++.
- Por lo general, los programas de C++ pasan por seis fases: edición, preprocessamiento, compilación, enlace, carga y ejecución.
- Los nombres de los archivos de código fuente de C++ terminan comúnmente con las extensiones .cpp, .cxx, .cc o .C.
- Un programa preprocessador se ejecuta de manera automática antes de que empiece la fase de traducción del compilador. El preprocessador de C++ obedece comandos llamados directivas del preprocessador, las cuales indican que deben realizarse ciertas manipulaciones en el programa, antes de compilarlo.
- El código objeto producido por el compilador de C++ contiene comúnmente "huecos" debido a las referencias a las funciones y datos definidos en cualquier otra parte. Un enlazador relaciona el código objeto con el código de las funciones faltantes para producir una imagen ejecutable (sin piezas faltantes).
- El cargador toma la imagen ejecutable del disco y la transfiere a la memoria para ejecutarla.
- La mayoría de los programas en C++ reciben y/o envían datos. A menudo los datos se reciben de `cin` (el flujo estándar de entrada), que por lo general es el teclado, pero `cin` puede conectarse a otro dispositivo. A menudo los datos se envían a `cout` (el flujo estándar de salida), que por lo general es la pantalla de la computadora, pero `cout` puede conectarse a otro dispositivo. El flujo `cerr` se utiliza para mostrar los mensajes de error.

Sección 1.18 Tecnologías de software

- Agile Software Development es un conjunto de metodologías que tratan de implementar software rápidamente, con menos recursos que las metodologías anteriores.
- La refabricación implica la reformulación del código para hacerlo más claro y fácil de mantener, al tiempo que se preserva su funcionalidad.
- Los patrones de diseño son arquitecturas probadas para construir software orientado a objetos flexible y que pueda mantenerse.
- El negocio de los juegos de computadora es más grande que el negocio de las películas de estreno. Ahora hay cursos universitarios, e incluso maestrías, dedicados a las técnicas sofisticadas de software que se utilizan en la programación de juegos.
- El software de código fuente abierto es un estilo de desarrollo de software que contrasta con el desarrollo propietario, que dominó los primeros años del software. Con el desarrollo de código fuente abierto, individuos y compañías contribuyen sus esfuerzos en el desarrollo, mantenimiento y evolución del software, a cambio del derecho de usar ese software para sus propios fines, comúnmente sin costo. Por lo general, el código fuente abierto se examina a detalle por una audiencia más grande que el software propietario, por lo cual los errores se eliminan con más rapidez. El código fuente abierto también promueve más innovación.
- Linux es un sistema operativo de código fuente abierto.
- MySQL es un sistema de administración de bases de datos con código fuente abierto.
- PHP es el lenguaje de secuencias de comandos del lado servidor de código fuente abierto más popular, para el desarrollo de aplicaciones basadas en Internet.
- LAMP es un acrónimo para el conjunto de tecnologías de código fuente abierto que utilizaron muchos desarrolladores para crear aplicaciones Web: representa a Linux, Apache, MySQL y PHP (o Perl, o Python; otros dos lenguajes que se utilizan para propósitos similares).
- Ruby on Rails combina el lenguaje de secuencias de comandos Ruby con el marco de trabajo para aplicaciones Web Rails, desarrollado por la compañía 37Signals.
- Con Software as a Service (SaaS), el software se ejecuta en servidores. Cuando se actualizan esos servidores, todos los clientes a nivel mundial ven las nuevas características; no se necesita instalación local. Usted accede al servidor a través de un explorador Web.

Sección 1.19 Programación de juegos con las bibliotecas Ogre

- Ogre es un proyecto de código fuente abierto, cuyo mantenimiento está a cargo del equipo Ogre en www.ogre3d.org.
- Ogre (Motor de despliegue de gráficos orientado a objetos), uno de los motores de gráficos más importantes, se ha utilizado en muchos productos comerciales, incluyendo videojuegos. Proporciona una interfaz orientada a objetos para la programación de gráficos en 3D, y se ejecuta en las plataformas Windows, Linux y Mac.
- OgreAL es una envoltura alrededor de la biblioteca de audio OpenAL. La envoltura nos permite integrar la funcionalidad de sonido en el código de Ogre.

Sección 1.20 Futuro de C++: Bibliotecas Boost de código fuente abierto, TR1 y C++0x

- Bjarne Stroustrup, el creador de C++, ha expresado su visión para el futuro de C++. Las principales metas para el nuevo estándar son facilitar el aprendizaje de C++, mejorar las herramientas para generar bibliotecas e incrementar la compatibilidad con el lenguaje de programación C.
- Las Bibliotecas Boost de C++ son bibliotecas gratuitas de código fuente abierto, creadas por miembros de la comunidad de C++. Boost proporciona a los programadores de C++ bibliotecas útiles y bien diseñadas, que funcionan bien con la Biblioteca estándar de C++ existente. Las bibliotecas Boost las pueden utilizar los programadores de C++ que trabajan en una amplia variedad de plataformas, con muchos compiladores distintos.
- El Informe técnico 1 describe los cambios propuestos a la Biblioteca estándar de C++, muchos de los cuales se basan en las bibliotecas Boost actuales. Estas bibliotecas añaden una funcionalidad útil a C++.
- C++0x es el nombre funcional para la siguiente versión del Estándar de C++. Incluye algunas modificaciones al lenguaje básico y muchas de las adiciones de bibliotecas descritas en el TR1.
- Boost ha aumentado su tamaño a más de 70 bibliotecas, y se están agregando más con regularidad. Hoy en día, hay miles de programadores en la comunidad de Boost.

Sección 1.21 Ejemplo práctico de Ingeniería de Software: introducción a la tecnología de objetos y el UML

- El Lenguaje unificado de modelado (UML) es un lenguaje gráfico que permite a las personas que crean sistemas representar sus diseños orientados a objetos en una notación común.
- El diseño orientado a objetos (OO) modela los componentes de software en términos de objetos del mundo real. Aprovecha las relaciones de herencia, en donde las nuevas clases de objetos se derivan absorbiendo las características de las clases existentes y agregando sus propias características únicas. El OO encapsula los datos (atributos) y funciones (comportamiento) en objetos: los datos y las funciones de un objeto están íntimamente entrelazados.

- Los objetos tienen la propiedad de ocultamiento de información; por lo general no se les permite saber cómo se implementan otros objetos.
- La programación orientada a objetos (POO) permite a los programadores implementar diseños orientados a objetos en forma de sistemas funcionales.
- Los programadores de C++ se crean sus propios tipos definidos por el usuario, denominados clases. Cada clase contiene datos (conocidos como miembros de datos), además del conjunto de funciones (conocidas como funciones miembro) que manipulan esos campos y proporcionan servicios a clientes.
- Las clases pueden tener relaciones con otras clases; a estas relaciones se les llama asociaciones.
- El proceso de empaquetar software en forma de clases hace posible que los sistemas de software posteriores reutilicen esas clases. Los grupos de clases relacionadas comúnmente se empaquetan como componentes reutilizables.
- A una instancia de una clase se le llama objeto.
- Con la tecnología de objetos, los programadores pueden crear la mayor parte del software que necesitan mediante la combinación de piezas estandarizadas e intercambiables, llamadas clases.
- El proceso de analizar y diseñar un sistema desde un punto de vista orientado a objetos se llama análisis y diseño orientados a objetos (A/DOO).

Terminología

acción	depuración
Ada	directivas del preprocesador
Agile Software Development	diseño
análisis	diseño orientado a objetos (DOO)
análisis y diseño de sistemas estructurados	dispositivo de entrada
análisis y diseño orientados a objetos (A/DOO)	dispositivo de salida
apuntadores inteligentes	editor
asociación	efectos de las redes sociales
atributo de un objeto	encapsular
BASIC (Código de instrucciones de propósito general para principiantes)	enlazador
Biblioteca estándar de C++	ensamblador
Bibliotecas Boost de C++	entrada/salida (E/S)
blog	errores en tiempo de ejecución
blogósfera	especificación de requerimientos
Booch, Grady	estación de trabajo
C	estándar ANSI/ISO de C
C#	Estándar ISO/IEC de C++
C++	extensible
C++0x	fase de carga
cargador	fase de compilación
clase	fase de edición
cliente	fase de ejecución
COBOL (Lenguaje común orientado a negocios)	fase de enlace
código fuente	fase de preprocessamiento
código fuente abierto	flujo de datos
código objeto	Fortran (Traductor de fórmulas)
compilador	función
componente	función miembro
comportamiento de un objeto	Google AdSense
computación cliente/servidor	Grupo de administración de objetos (OMG)
computación distribuida	hardware
computación personal	herencia
computadora	imagen ejecutable
contenido dinámico	independiente de la máquina
contenido generado por la comunidad	Informe técnico 1 (TR1)
contenido Premium	instanciar
datos	Instituto nacional estadounidense de estándares (ANSI)
decisión	inteligencia colectiva
dependiente de la máquina	interfaz
	Internet

Internet de banda ancha	programación de juegos
Intérprete	programación estructurada
Jacobson, Ivar	programación orientada a objetos (POO)
Java	programación por procedimientos
LAMP	programador de computadora
lenguaje de alto nivel	racional Software Corporation
lenguaje de máquina	redes de área local (LANs)
lenguaje ensamblador	redes sociales
Lenguaje Unificado de Modelado (UML)	refabricación
Linux	reutilización de software
marcado	Ruby on Rails
mashups	Rumbaugh, James
memoria	servicios Web
memoria básica	servidor de archivos
memoria principal	seudocódigo
método	sistema operativo
método de código activo (live-code)	Software as a Service (SaaS), software
miembro de datos	software de código fuente abierto
multiprocesador	subprocesamiento múltiple
multiprogramación	supercomputadora
multitareas	tarea
MySQL	tiempo compartido
.NET, plataforma	tipo definido por el usuario
objeto	traducción
ocultamiento de información	unidad aritmético-lógica (ALU)
Ogre (Motor de despliegue de gráficos orientado a objetos)	unidad central de proceso (CPU)
OgreAL	unidad de almacenamiento secundario
OpenAL, biblioteca de audio	unidad de entrada
operación	unidad de memoria
Organización internacional para la estandarización (ISO)	unidad de salida
patrones de diseño	unidad lógica
PHP	Visual Basic
Plataforma	Visual C++
plataforma de hardware	Web 3.0
Portable	Web Semántica
procesamiento por lotes	wiki
programa de computadora	World Wide Web
programa traductor	

Ejercicios de autoevaluación

1.1 Complete las siguientes oraciones:

- La compañía que popularizó la computación personal fue _____.
- La computadora que legitimó la computación personal en los negocios y la industria fue _____.
- Las computadoras procesan los datos bajo el control de conjuntos de instrucciones llamadas _____.
- Las seis unidades lógicas clave de la computadora son _____, _____, _____, _____, _____, _____.
- Los tres tipos de lenguajes descritos en este capítulo son _____, _____ y _____.
- Los programas que traducen programas en lenguaje de alto nivel a lenguaje de máquina se denominan _____.
- C se conoce ampliamente como el lenguaje de desarrollo del sistema operativo _____.
- El lenguaje _____ fue desarrollado por Wirth para enseñar programación estructurada.
- El Departamento de Defensa desarrolló el lenguaje Ada con una herramienta llamada _____, la cual permite a los programadores especificar que muchas actividades pueden procesarse en paralelo.
- El _____, o etiquetado de contenido, es otra parte clave del tema de colaboración de Web 2.0.
- Con las aplicaciones de Internet, el escritorio evoluciona para convertirse en _____.

- l) La _____ implica reformular el código para hacerlo más legible y fácil de mantener, al tiempo que se preserva su funcionalidad.
- m) Con el desarrollo del _____, los individuos y compañías contribuyen sus esfuerzos para desarrollar, dar mantenimiento y evolucionar software, a cambio del derecho de utilizar ese software para sus propios fines, por lo general sin ningún cargo.
- n) Las _____ se utilizan para relacionar patrones de caracteres específicos en el texto. Pueden usarse para validar datos y asegurar que se encuentren en un formato específico, para reemplazar partes de una cadena con otra, o para dividir una cadena.
- I.2** Complete cada una de las siguientes oraciones relacionadas con el entorno de C++:
- Por lo general, los programas de C++ se escriben en una computadora mediante el uso de un programa _____.
 - En un sistema de C++, un programa _____ se ejecuta antes de que empiece la fase de traducción del compilador.
 - El programa _____ combina la salida del compilador con varias funciones de biblioteca para producir una imagen ejecutable.
 - El programa _____ transfiere la imagen ejecutable de un programa de C++, del disco a la memoria.
- I.3** Complete cada una de las siguientes oraciones (basándose en la sección 1.21):
- Los objetos tienen una propiedad que se conoce como _____; aunque éstos pueden saber cómo comunicarse con los demás objetos a través de interfaces bien definidas, generalmente no se les permite saber cómo están implementados los otros objetos.
 - Los programadores de C++ se concentran en crear _____, que contienen miembros de datos y las funciones miembro que manipulan a esos miembros de datos y proporcionan servicios a los clientes.
 - Las clases pueden tener relaciones con otras clases; a éstas relaciones se les llama _____.
 - Al proceso de analizar y diseñar un sistema desde un punto de vista orientado a objetos se le conoce como _____.
 - El DOO también aprovecha las relaciones de _____, donde se derivan nuevas clases de objetos absorbiendo las características de las clases existentes, y después se agregan sus propias características únicas.
 - _____ es un lenguaje gráfico, que permite a las personas que diseñan sistemas de software utilizar una notación estándar en la industria para representarlos.
 - El tamaño, forma, color y peso de un objeto se consideran _____ del objeto.

Respuestas a los ejercicios de autoevaluación

I.1 a) Apple. b) Computadora Personal de IBM. c) programas. d) unidad de entrada, unidad de salida, unidad de memoria, unidad aritmética y lógica, unidad central de procesamiento, unidad de almacenamiento secundario. e) lenguajes máquina, lenguajes ensambladores y lenguajes de alto nivel. f) compiladores. g) UNIX. h) Pascal. i) multitareas. j) Marcado. k) web-top. l) Refabricación. m) código fuente abierto. n) Expresiones regulares.

I.2 a) editor. b) preprocesador. c) enlazador. d) cargador.

I.3 a) ocultamiento de información. b) clases. c) asociaciones. d) análisis y diseño orientados a objetos (A/DOO). e) herencia. f) El Lenguaje Unificado de Modelado (UML). g) atributos.

Ejercicios

I.4 Clasifique cada uno de los siguientes elementos como hardware o software:

- CPU.
- compilador de C++.
- ALU.
- preprocesador de C++.
- unidad de entrada.
- un programa editor.

I.5 ¿Por qué sería conveniente escribir un programa en un lenguaje independiente de la máquina, en lugar de hacerlo en un lenguaje dependiente de la máquina? ¿Por qué sería más apropiado un lenguaje dependiente de la máquina para escribir ciertos tipos de programas?

1.6

Complete cada una de las siguientes oraciones:

- a) ¿Qué unidad lógica de la computadora recibe información desde el exterior de la computadora para que ésta la utilice? _____.
- b) El proceso de indicar a la computadora cómo resolver problemas específicos se llama _____.
- c) ¿Qué tipo de lenguaje computacional utiliza abreviaturas del inglés para las instrucciones de lenguaje de máquina? _____.
- d) ¿Qué unidad lógica de la computadora envía información, que ya ha sido procesada por la computadora, a varios dispositivos, de manera que la información pueda utilizarse fuera de la computadora? _____.
- e) ¿Qué unidades lógicas de la computadora retienen información?
- f) ¿Qué unidad lógica de la computadora realiza cálculos?
- g) ¿Qué unidad lógica de la computadora toma decisiones lógicas?
- h) El nivel de lenguaje de computadora más conveniente para que el programador pueda escribir programas rápida y fácilmente es _____.
- i) Al único lenguaje que una computadora puede entender directamente se le conoce como el _____ de esa computadora.
- j) ¿Qué unidad lógica de la computadora coordina las actividades de todas las demás unidades lógicas? _____.

1.7 ¿Por qué se enfoca tanta atención hoy en día a la programación orientada a objetos en general, y a C++ en especial?

1.8 Indique la diferencia entre los términos error fatal y error no fatal. ¿Por qué sería preferible experimentar un error fatal, en lugar de un error no fatal?

1.9 Dé una respuesta breve a cada una de las siguientes preguntas:

- a) ¿Por qué este libro habla sobre la programación estructurada, además de la programación orientada a objetos?
- b) ¿Cuáles son los pasos típicos (mencionados en el texto) de un proceso de diseño orientado a objetos?
- c) ¿Qué tipos de mensajes se envían las personas entre sí?
- d) Los objetos se envían mensajes entre sí a través de interfaces bien definidas. ¿Qué interfaces presenta la radio (objeto) de un auto a su usuario (un objeto persona)?

1.10 Probablemente usted lleve en su muñeca uno de los tipos de objetos más comunes en el mundo: un reloj. Hable acerca de cómo se aplica cada uno de los siguientes términos y conceptos a la noción de un reloj: objeto, atributos, comportamientos, clase, herencia (considere, por ejemplo, un reloj con alarma), abstracción, modelado, mensajes, encapsulamiento, interfaz, ocultamiento de información, miembros de datos y funciones miembro.

1.11 Complete cada una de las siguientes oraciones (con base en la sección 1.18, Tecnologías de software):

- a) El sistema de administración de bases de datos con código fuente abierto que se utiliza en el desarrollo con LAMP es _____.
- b) Una ventaja clave del Software como servicio (SaaS) es _____.
- c) _____ son arquitecturas probadas para construir software orientado a objetos flexible y al cual se le puede dar mantenimiento.
- d) _____ es el lenguaje de secuencias de comandos del lado servidor de código fuente abierto más popular para desarrollar aplicaciones basadas en Internet.

2



*¿Qué hay en un nombre?
A eso a lo que llamamos rosa,
si le diéramos otro nombre
conservaría su misma
fragancia dulce.*

—William Shakespeare

*Al hacer frente a una
decisión, siempre me
pregunto, “¿Cuál será la
solución más divertida?”*

—Peggy Walker

*“Toma un poco más de
té”, dijo el conejo blanco a
Alicia, con gran seriedad.
“No he tomado nada
todavía.” Alicia contestó en
tono ofendido, “Entonces
no puedo tomar más”.
“Querrás decir que no
puedes tomar menos”, dijo
el sombrerero loco, “es muy
fácil tomar más que nada.”*

—Lewis Carroll

*Los pensamientos elevados
deben tener un lenguaje
elevado.*

—Aristófanes

Introducción a la programación en C++

OBJETIVOS

En este capítulo aprenderá a:

- Escribir programas de cómputo simples en C++.
- Escribir instrucciones simples de entrada y salida.
- Aprender a utilizar los tipos fundamentales.
- Conocer los conceptos básicos de la memoria de la computadora.
- Aprender a utilizar los operadores aritméticos.
- Conocer la precedencia de los operadores aritméticos.
- Escribir instrucciones simples para tomar decisiones.

- 2.1** Introducción
- 2.2** Su primer programa en C++: imprimir una línea de texto
- 2.3** Modificación de nuestro primer programa en C++
- 2.4** Otro programa en C++: suma de enteros
- 2.5** Conceptos acerca de la memoria
- 2.6** Aritmética
- 2.7** Toma de decisiones: operadores de igualdad y relacionales
- 2.8** (Opcional) Ejemplo práctico de Ingeniería de Software: cómo examinar la especificación de requerimientos del ATM
- 2.9** Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)

2.1 Introducción

Ahora presentaremos la programación en C++, que facilita una metodología disciplinada para el diseño de programas. La mayoría de los programas en C++ que estudiará en este libro procesan información y muestran resultados. En este capítulo le presentaremos cinco ejemplos que demuestran cómo sus programas pueden mostrar mensajes y cómo pueden obtener información del usuario para procesarla. Los primeros tres ejemplos simplemente muestran mensajes en la pantalla. El siguiente ejemplo obtiene dos números de un usuario, calcula su suma y muestra el resultado. La discusión que acompaña a este ejemplo le muestra cómo realizar varios cálculos aritméticos y guardar sus resultados para utilizarlos posteriormente. El quinto ejemplo demuestra los fundamentos de toma de decisiones, al mostrarle cómo comparar dos números y después mostrar mensajes con base en los resultados de la comparación. Analizaremos cada ejemplo, una línea a la vez, para ayudarle a aprender a programar en C++. Para ayudarle a aplicar las habilidades que aprenderá aquí, proporcionamos muchos problemas de programación en los ejercicios de este capítulo.

2.2 Su primer programa en C++: imprimir una línea de texto

C++ utiliza notaciones que pueden parecer extrañas a los no programadores. Ahora consideraremos un programa simple que imprime una línea de texto (figura 2.1). Este programa ilustra varias características importantes del lenguaje C++. Consideraremos cada línea en forma detallada.

Las líneas 1 y 2

```
// Fig. 2.1: fig02_01.cpp
// Programa para imprimir texto.
```

```

1 // Fig. 2.1: fig02_01.cpp
2 // Programa para imprimir texto.
3 #include <iostream> // permite al programa imprimir datos en la pantalla
4
5 // la función main comienza la ejecución del programa
6 int main()
7 {
8     std::cout << "Bienvenido a C++!\n"; // muestra un mensaje
9
10    return 0; // indica que el programa terminó con éxito
11
12 } // fin de la función main

```

Bienvenido a C++!

Figura 2.1 | Programa para imprimir texto.

comienzan con `//`, lo cual indica que el resto de la línea es un **comentario**. Los comentarios sólo sirven para documentar nuestros programas y ayudan a que otras personas, al leerlos, comprendan lo que queremos hacer con el código. Cuando la computadora “ve” un comentario, lo ignora y no realiza ninguna acción con él cuando se ejecuta el programa; el compilador de C++ los ignora, por lo que no genera código objeto alguno en lenguaje máquina. El comentario **Programa para imprimir texto** describe el propósito del programa. A un comentario que empieza con `//` se le llama **comentario de una sola línea**, ya que termina al final de la línea actual. [Nota: también puede usar el estilo de C, en el cual un comentario (que posiblemente contenga muchas líneas) empieza con `/*` y termina con `*/`.]



Buena práctica de programación 2.1

Todo programa debe comenzar con un comentario que describa su propósito, autor, fecha y hora. (No mostramos el autor, fecha y hora en los programas de este libro, debido a que esta información sería redundante.)

La línea 3

```
#include <iostream> // permite al programa imprimir datos en la pantalla
```

es una **directiva del preprocesador**, la cual es un mensaje para el preprocesador de C++ (presentado en la sección 1.15). Las líneas que empiezan con `#` son procesadas por el preprocesador antes de que se compile el programa. Esta línea indica al preprocesador que debe incluir en el programa el contenido del **archivo de encabezado de flujos de entrada/salida** `<iostream>`. Este archivo debe incluirse para cualquier programa que muestre datos en la pantalla, o que reciba datos del teclado, mediante el uso de la entrada/salida de flujos al estilo C++. El programa de la figura 2.1 muestra datos en la pantalla, como pronto veremos. En el capítulo 6 hablaremos con más detalles sobre los archivos de encabezado y explicaremos el contenido de `<iostream>` en el capítulo 15.



Error común de programación 2.1

Olvidar incluir el archivo de encabezado <iostream> en un programa que reciba datos del teclado, o que envíe datos a la pantalla, hace que el compilador genere un mensaje de error, ya que no puede reconocer las referencias a los componentes de los flujos (por ejemplo, cout).

La línea 4 es simplemente una línea en blanco. Los programadores usan líneas en blanco, caracteres de espacio y caracteres de tabulación (es decir, “tabuladores”) para facilitar la lectura de los programas. En conjunto, estos caracteres se conocen como **espacio en blanco**. Por lo general, el compilador ignora los caracteres de espacio en blanco. En éste y en los siguientes capítulos, hablaremos sobre las convenciones para utilizar caracteres de espacio en blanco para mejorar la legibilidad de los programas.



Buena práctica de programación 2.2

Utilice líneas en blanco, caracteres de espacio y tabuladores para mejorar la legibilidad del programa.

La línea 5

```
// la función main comienza la ejecución del programa
```

es otro comentario de una sola línea, el cual indica que la ejecución del programa empieza en la siguiente línea.

La línea 6

```
int main()
```

forma parte de todo programa en C++. Los paréntesis después de `main` indican que éste es un bloque de construcción denominado **función**. Los programas en C++ comúnmente consisten en una o más funciones y clases (como aprenderá en el capítulo 3). Sólo debe haber una función `main` en cada programa. La figura 2.1 contiene sólo una función. Los programas en C++ empiezan a ejecutarse en la función `main`, aun si `main` no es la primera función en el programa. La palabra clave `int` a la izquierda de `main` indica que “devuelve” un valor entero. Una **palabra clave** es una palabra en código reservada para C++, para un uso específico. En la figura 4.3 encontrará la lista completa de palabras clave de C++. Explicaremos lo que significa que una función “devuelva un valor” cuando le demostremos cómo crear sus propias funciones en la sección 3.5, y cuando estudiemos las funciones con mayor detalle en el capítulo 6. Por ahora, simplemente incluya la palabra clave `int` a la izquierda de `main` en todos sus programas.

La **llave izquierda**, {, (línea 7) debe comenzar el **cuerpo** de toda función. Su correspondiente **llave derecha**, }, (línea 12) debe terminar el cuerpo de cada función. La línea 8

```
std::cout << "Bienvenido a C++!\n"; // muestra un mensaje
```

indica a la computadora que debe **realizar una acción**; a saber, imprimir la cadena de caracteres contenida entre las comillas dobles. A una cadena se le conoce algunas veces como **cadena de caracteres**, **mensaje** o **literal de cadena**. A los caracteres entre comillas dobles los denominamos simplemente **cadenas**. El compilador no ignora los caracteres de espacio en blanco en las cadenas.

A la línea 8 completa, incluyendo std::cout, el **operador** <<, la cadena "Bienvenido a C++!\n" y el **punto y coma** (;), se le conoce como **instrucción**. Cada instrucción en C++ debe terminar con un punto y coma (también conocido como **terminador de instrucciones**). Las directivas del preprocesador (como #include) no terminan con un punto y coma. En C++, las operaciones de entrada y salida se realizan mediante **flujos** de caracteres. Por ende, cuando se ejecuta la instrucción anterior, envía el flujo de caracteres Bienvenido a C++!\n al **objeto flujo estándar de salida** (std::cout), el cual por lo general está “conectado” a la pantalla. En el capítulo 15, Entrada y salida de flujos, hablaremos con detalle sobre las diversas características de std::cout.

Observe que colocamos std:: antes de cout. Esto se requiere cuando utilizamos nombres que hemos traído al programa por la directiva del preprocesador #include <iostream>. La notación std::cout especifica que estamos usando un nombre (en este caso cout) que pertenece al “espacio de nombres” std. Los nombres cin (el flujo de entrada estándar) y cerr (el flujo de error estándar), que presentamos en el capítulo 1, también pertenecen al espacio de nombres std. Los espacios de nombres son una característica avanzada de C++, que veremos con detalle en el capítulo 25, Otros temas. Por ahora, simplemente debe recordar incluir std:: antes de cada mención de cout, cin y cerr en un programa. Esto puede ser incómodo; en la figura 2.13 introduciremos la declaración using, la cual nos permitirá omitir std:: antes de cada uso de un nombre en el espacio de nombres std.

El operador << se conoce como el **operador de inserción de flujo**. Cuando este programa se ejecuta, el valor a la derecha del operador (el **operando** derecho) se inserta en el flujo de salida. Observe que el operador apunta en la dirección hacia la que van los datos. Por lo general, los caracteres del operando derecho se imprimen exactamente como aparecen entre las comillas dobles. Sin embargo, los caracteres \n no se imprimen en la pantalla (figura 2.1). A la barra diagonal inversa (\) se le llama **carácter de escape**. Este carácter indica que se va a imprimir en pantalla un carácter “especial”. Cuando se encuentra una barra diagonal inversa en una cadena de caracteres, el siguiente carácter se combina con la barra diagonal inversa para formar una **secuencia de escape**. La secuencia de escape \n representa una **nueva línea**. Hace que el **cursor** (es decir, el indicador de la posición actual en la pantalla) se desplace al principio de la siguiente línea en la pantalla. Algunas secuencias de escape comunes se enlistan en la figura 2.2.

Error común de programación 2.2



Omitir el punto y coma al final de una instrucción de C++ es un error de sintaxis. (De nuevo, las directivas del preprocesador no terminan en un punto y coma.) La **sintaxis** de un lenguaje de programación especifica las reglas para crear programas apropiados en ese lenguaje. Un **error de sintaxis** ocurre cuando el compilador encuentra código que viola las reglas del lenguaje C++ (es decir, su sintaxis). Por lo general, el compilador genera un mensaje de error para ayudarnos a localizar y corregir el código incorrecto. A los errores de sintaxis también se les llama **errores del compilador**, **errores en tiempo de compilación** o **errores de compilación**, ya que el compilador los detecta durante la fase de compilación. No podrá ejecutar su programa sino hasta que corrija todos los errores de sintaxis que contenga. Como veremos más adelante, algunos errores de compilación no son errores de sintaxis.

La línea 10

```
return 0; // indica que el programa terminó con éxito
```

es uno de varios medios que utilizaremos para **salir de una función**. Cuando se utiliza la instrucción return al final de main, como se muestra aquí, el valor 0 indica que el programa ha terminado correctamente. En el capítulo 6 veremos las funciones con detalle, y comprenderá las razones para incluir esta instrucción. Por ahora, simplemente incluya esta instrucción en cada programa, o de lo contrario el compilador puede producir una advertencia en algunos sistemas. La llave derecha, }, (línea 12) indica el fin de la función main.

Buena práctica de programación 2.3



Muchos programadores hacen que el último carácter que debe imprimir una función sea una nueva línea (\n). Esto asegura que la función deje el cursor de la pantalla colocado al inicio de una nueva línea. Las convenciones de esta naturaleza fomentan la reutilización de software; un objetivo clave en el desarrollo de software.

Secuencia de escape	Descripción
\n	Nueva línea. Coloca el cursor de la pantalla al inicio de la siguiente línea.
\t	Tabulador horizontal. Desplaza el cursor de la pantalla hasta la siguiente posición de tabulación.
\r	Retorno de carro. Coloca el cursor de la pantalla al inicio de la línea actual; no avanza a la siguiente línea.
\a	Alerta. Suena la campana del sistema.
\\\	Barra diagonal inversa. Se usa para imprimir un carácter de barra diagonal inversa.
\'	Comilla sencilla. Se usa para imprimir un carácter de comilla sencilla.
\"	Doble comilla. Se usa para imprimir un carácter de doble comilla.

Figura 2.2 | Secuencias de escape.



Buena práctica de programación 2.4

Aplique sangría a todo el cuerpo de cada función, usando un nivel de sangría dentro de las llaves que delimitan el cuerpo de la función. Esto hace que resalte la estructura funcional de un programa, y facilita su lectura.



Buena práctica de programación 2.5

Establezca una convención para el tamaño de sangría que prefiera, y luego aplíquela de manera uniforme. Puede utilizar la tecla de tabulación para crear sangrías, pero las posiciones de los tabuladores pueden variar. Le recomendamos utilizar posiciones de 1/4 de pulgada o (de preferencia) tres espacios para formar un nivel de sangría.

2.3 Modificación de nuestro primer programa en C++

Esta sección continúa con nuestra introducción a la programación en C++, con dos ejemplos que muestran cómo modificar el ejemplo de la figura 2.1 para imprimir texto en una línea utilizando varias instrucciones, y para imprimir texto en varias líneas utilizando una sola instrucción.

Cómo mostrar una sola línea de texto con varias instrucciones

Bienvenido a la programación en C++! Puede mostrarse en varias formas. La figura 2.3 realiza la inserción de flujos en varias instrucciones (líneas 8 y 9), y produce el mismo resultado que el programa de la figura 2.1. [Nota: de aquí en adelante, utilizaremos un contraste con el fondo de la tabla de código (si el fondo es gris lo pondremos en blanco, si el fondo es blanco lo pondremos en gris) para destacar las características clave que se introduzcan en cada programa.] Cada inserción de flujo reanuda la impresión donde se detuvo la anterior. La primera inserción de flujo (línea 8) imprime la palabra Bienvenido seguida de un espacio, y la segunda inserción de flujo (línea 9) empieza a imprimir en la misma línea, justo después del espacio. En general, C++ nos permite expresar las instrucciones en varias formas.

Cómo mostrar varias líneas de texto con una sola instrucción

Una sola instrucción puede mostrar varias líneas, utilizando caracteres de nueva línea, como en la línea 8 de la figura 2.4. Cada vez que se encuentra la secuencia de escape \n (nueva línea) en el flujo de salida, el cursor de la pantalla se coloca al inicio de la siguiente línea. Para obtener una línea en blanco en sus resultados, coloque dos caracteres de nueva línea, uno después del otro, como en la línea 8.

```

1 // Fig. 2.3: fig02_03.cpp
2 // Imprimir una línea de texto con varias instrucciones.
3 #include <iostream> // permite que el programa envíe datos a la pantalla
4
5 // La función main comienza la ejecución del programa

```

Figura 2.3 | Impresión de una línea de texto con varias instrucciones. (Parte 1 de 2).

```

6 int main()
7 {
8     std::cout << "Bienvenido ";
9     std::cout << "a C++!\n";
10
11     return 0; // indica que el programa terminó correctamente
12
13 } // fin de la función main

```

Bienvenido a C++!

Figura 2.3 | Impresión de una línea de texto con varias instrucciones. (Parte 2 de 2).

```

1 // Fig. 2.4: fig02_04.cpp
2 // Impresión de varias líneas de texto con una sola instrucción.
3 #include <iostream> // permite al programa imprimir datos en la pantalla
4
5 // la función main empieza la ejecución del programa
6 int main()
7 {
8     std::cout << "Bienvenido\na\nnC++!\n";
9
10    return 0; // indica que el programa terminó correctamente
11
12 } // fin de la función main

```

Bienvenido

a

C++!

Figura 2.4 | Impresión de varias líneas de texto con una sola instrucción.

2.4 Otro programa en C++: suma de enteros

Nuestro siguiente programa utiliza el objeto flujo de entrada `std::cin` y el operador de extracción de flujo, `>>`, para obtener dos enteros escritos por el usuario mediante el teclado, calcula la suma de esos valores e imprime el resultado mediante el uso de `std::cout`. En la figura 2.5 se muestra el programa, junto con los datos de entrada y salida de ejemplo.

Los comentarios en las líneas 1 y 2

```

// Fig. 2.5: fig02_05.cpp
// Programa de suma que muestra la suma de dos enteros.

```

indican el nombre del archivo y el propósito del programa. La directiva del preprocesador de C++

```
#include <iostream> // permite al programa realizar operaciones de entrada y salida
```

en la línea 3 incluye el contenido del archivo de encabezado `<iostream>` en el programa.

El programa empieza a ejecutarse con la función `main` (línea 6). La llave izquierda (línea 7) marca el inicio del cuerpo de `main` y la correspondiente llave derecha (línea 25) marca el fin de `main`.

Las líneas 9 a 11

```

int numero1; // primer entero a sumar
int numero2; // segundo entero a sumar
int suma; // suma de numero1 y numero2

```

son **declaraciones**. Los identificadores `numero1`, `numero2` y `suma` son nombres de **variables**. Una variable es una ubicación en la memoria de la computadora, en la que puede almacenarse un valor para usarlo mediante un programa. Estas declaraciones especifican que las variables `numero1`, `numero2` y `suma` son datos de tipo `int`, lo cual significa que estas variables contienen valores **enteros**; es decir, números enteros como 7, -11, 0 y 31914. Todas las variables se deben declarar con un nombre y un tipo de datos antes de poder usarlas en un programa. Pueden declararse diversas variables

```

1 // Fig. 2.5: fig02_05.cpp
2 // Programa de suma que muestra la suma de dos enteros.
3 #include <iostream> // permite al programa realizar operaciones de entrada y salida
4
5 // la función main empieza la ejecución del programa
6 int main()
7 {
8     // declaraciones de variables
9     int numero1; // primer entero a sumar
10    int numero2; // segundo entero a sumar
11    int suma; // suma de numero1 y numero2
12
13    std::cout << "Escriba el primer entero: "; // pide los datos al usuario
14    std::cin >> numero1; // lee el primer entero del usuario y lo coloca en numero1
15
16    std::cout << "Escriba el segundo entero: "; // pide los datos al usuario
17    std::cin >> numero2; // lee el segundo entero del usuario y lo coloca en numero2
18
19    suma = numero1 + numero2; // suma los números; almacena el resultado en suma
20
21    std::cout << "La suma es " << suma << std::endl; // muestra la suma; fin de línea
22
23    return 0; // indica que el programa terminó correctamente
24
25 } // fin de la función main

```

```

Escriba el primer entero: 45
Escriba el segundo entero: 72
La suma es 117

```

Figura 2.5 | Programa de suma que muestra la suma de dos enteros introducidos mediante el teclado.

del mismo tipo en una declaración, o en varias declaraciones. Podríamos haber declarado las tres variables en una declaración, como se muestra a continuación:

```
int numero1, numero2, suma;
```

Esto reduce la legibilidad del programa y evita que podamos proporcionar comentarios que describan el propósito de cada variable. Si se declara más de un nombre en una declaración (como se muestra aquí), los nombres van separados por comas (,); a esto se le conoce como **lista separada por comas**.



Buena práctica de programación 2.6

Coloque un espacio después de cada coma (,) para aumentar la legibilidad de los programas.



Buena práctica de programación 2.7

Algunos programadores prefieren declarar cada variable en una línea separada. Este formato permite insertar fácilmente un comentario descriptivo a un lado de cada declaración.

Pronto hablaremos sobre el tipo de datos **double** para especificar números reales, y sobre el tipo de datos **char** para especificar datos tipo carácter. Los números reales son números con puntos decimales, como 3.4, 0.0 y -11.19. Una variable **char** puede guardar sólo una letra minúscula, una letra mayúscula, un dígito o un carácter especial (como \$ o *). Los tipos como **int**, **double** y **char** se conocen comúnmente como **tipos fundamentales**, **tipos primitivos** o **tipos integrados**. Los nombres de los tipos fundamentales son palabras clave y, por lo tanto, deben aparecer todos en minúsculas. El apéndice C contiene la lista completa de tipos fundamentales.

El nombre de una variable (como **numero1**) es cualquier **identificador** válido que no sea una palabra clave. Un identificador es una serie de caracteres que consisten en letras, dígitos y símbolos de guión bajo (_) que no empieza con un dígito. C++ es **sensible a mayúsculas y minúsculas**: las letras mayúsculas y minúsculas son distintas, por lo que **a1** y **A1** se consideran identificadores distintos.



Tip de portabilidad 2.1

C++ permite usar identificadores de cualquier longitud, pero tal vez la implementación de C++ que usted utilice imponga algunas restricciones en cuanto a la longitud de los identificadores. Use identificadores de 31 caracteres o menos, para asegurar la portabilidad.



Buena práctica de programación 2.8

Seleccionar nombres de variables significativos ayuda a que un programa se autodocumente (es decir, que sea más fácil entender el programa con sólo leerlo, en lugar de leer manuales o ver un número excesivo de comentarios).



Buena práctica de programación 2.9

Evite usar abreviaciones en los identificadores. Esto fomenta la legibilidad del programa.



Buena práctica de programación 2.10

Evite el uso de identificadores que empiecen con guiones bajos y dobles guiones bajos, ya que los compiladores de C++ pueden utilizar nombres como esos para sus propios fines, de manera interna. Esto evitara que los nombres que usted elija se confundan con los nombres que elijan los compiladores.



Tip para prevenir errores 2.1

Los lenguajes como C++ son “objetivos móviles”. A medida que evolucionan, podrían agregarse más palabras clave al lenguaje. Evite usar palabras “cargadas”, tales como “object”, como identificadores. Aun y cuando “object” no es actualmente una palabra clave en C++, podría convertirse en una; por lo tanto, la compilación a futuro con nuevos compiladores podría quebrantar el código existente.

Las declaraciones de las variables se pueden colocar casi en cualquier parte dentro de un programa, pero deben aparecer antes de que sus correspondientes variables se utilicen en el programa. Por ejemplo, en el programa de la figura 2.5, la declaración en la línea 9

```
int numero1; // primer entero a sumar
```

se podría haber colocado justo antes de la línea 14

```
std::cin >> numero1; // lee el primer entero del usuario y lo coloca en numero1
```

la declaración en la línea 10

```
int numero2; // segundo entero a sumar
```

se podría haber colocado justo antes de la línea 17

```
std::cin >> numero2; // lee el segundo entero del usuario y lo coloca en numero2
```

y la declaración en la línea 11

```
int suma; // suma de numero1 y numero2
```

se podría haber colocado justo antes de la línea 19

```
suma = numero1 + numero2; // suma los números; almacena el resultado en suma
```



Buena práctica de programación 2.11

Coloque siempre una línea en blanco entre una declaración y las instrucciones ejecutables adyacentes. Esto hace que la declaración sobresalga en el programa, y contribuye a mejorar su claridad.



Buena práctica de programación 2.12

Si prefiere colocar declaraciones al principio de una función, sepárelas de las instrucciones ejecutables en esa función mediante una línea en blanco, para resaltar dónde terminan las declaraciones y dónde empiezan las instrucciones ejecutables.

La línea 13

```
std::cout << "Escriba el primer entero: "; // pide los datos al usuario
```

imprime la cadena **Escriba el primer entero:** en la pantalla. A este mensaje se le conoce como **indicador**, debido a que indica al usuario que debe realizar una acción específica. Nos gusta pronunciar la anterior instrucción como “`std::cout` obtiene la cadena de caracteres **Escriba el primer entero: .**” La línea 14

```
std::cin >> numero1; // lee el primer entero del usuario y lo coloca en numero1
```

utiliza el **objeto flujo de entrada** `cin` (del espacio de nombres `std`) y el **operador de extracción de flujo**, `>>`, para obtener un valor del teclado. Al usar el operador de extracción de flujo con `std::cin` se toma la entrada de caracteres del flujo de entrada estándar, que por lo general es el teclado. Nos gusta pronunciar la instrucción anterior como “`std::cin` proporciona un valor a `numero1`”, o simplemente como “`std::cin` proporciona `numero1`”.



Tip para prevenir errores 2.2

Los programas deben validar que todos los valores de entrada sean correctos, para evitar que información errónea afecte los cálculos de un programa.

Cuando la computadora ejecuta la instrucción anterior, espera a que el usuario introduzca un valor para la variable `numero1`. El usuario responde escribiendo un entero (en forma de caracteres), y después oprime la tecla *Intro* (a la que algunas veces se le conoce como tecla *Return*) para enviar los caracteres a la computadora. Ésta a su vez convierte la representación de caracteres del número en un entero, y asigna (copia) este número (o **valor**) a la variable `numero1`. Cualquier referencia posterior a `numero1` en este programa utilizará este mismo valor.

Los objetos flujo `std::cout` y `std::cin` facilitan la interacción entre el usuario y la computadora. Debido a que esta interacción se asemeja a un diálogo, a menudo se le conoce como **computación conversacional** o **computación interactiva**.

La línea 16

```
std::cout << "Escriba el segundo entero: "; // pide los datos al usuario
```

imprime el mensaje **Escriba el segundo entero:** en la pantalla, pidiendo al usuario que realice una acción. En la línea 17

```
std::cin >> numero2; // lee el segundo entero del usuario y lo coloca en numero2
```

se obtiene un valor para la variable `numero2` de parte del usuario.

La instrucción de asignación en la línea 19

```
suma = numero1 + numero2; // suma los números; almacena el resultado en suma
```

calcula la suma de las variables `numero1` y `numero2`, y asigna el resultado a la variable `suma`, mediante el uso del **operador de asignación** `=`. La instrucción se lee como “`suma` obtiene el valor de `numero1 + numero2`”. La mayoría de los cálculos se realizan en instrucciones de asignación. Los operadores `=` y `+` se conocen como **operadores binarios**, ya que cada uno de ellos tiene dos operandos. En el caso del operador `+`, los dos operandos son `numero1` y `numero2`. En el caso del operador `=` anterior, los dos operandos son `suma` y el valor de la expresión `numero1 + numero2`.



Buena práctica de programación 2.13

Coloque espacios en cualquier lado de un operador binario. Esto hace que el operador resalte y mejora la legibilidad del programa.

La línea 21

```
std::cout << "La suma es " << suma << std::endl; // muestra la suma; fin de línea
```

muestra a la cadena de caracteres **La suma es**, seguida del valor numérico de la variable `suma`, seguido de `std::endl`; a este último se le conoce como **manipulador de flujos**. El nombre `endl` es una abreviación de “fin de línea” (“end line”, en inglés) y pertenece al espacio de nombres `std`. El manipulador de flujos `std::endl` imprime una nueva línea y después “vacía el búfer de salida”. Esto simplemente significa que, en algunos sistemas donde los datos de salida se acumulan en el equipo hasta que haya suficientes como para que “valga la pena” mostrarlos en pantalla, `std::endl` obliga a que todos los datos de salida acumulados se muestren en ese momento. Esto puede ser importante cuando los datos de salida piden al usuario una acción, como introducir datos.

Observe que la instrucción anterior imprime múltiples valores de distintos tipos. El operador de inserción de flujo “sabe” cómo imprimir cada tipo de datos. Al uso de varios operadores de inserción de flujo (`<<`) en una sola instrucción se le conoce como **operaciones de inserción de flujo en cascada, concatenamiento o encadenamiento**. No es necesario tener varias instrucciones para imprimir varias piezas de datos.

Los cálculos también se pueden realizar en instrucciones de salida. Podríamos haber combinado las instrucciones en las líneas 19 y 21 en la instrucción

```
std::cout << "La suma es " << numero1 + numero2 << std::endl;
```

con lo cual se elimina la necesidad de usar la variable `suma`.

Una poderosa característica de C++ es que los usuarios pueden crear sus propios tipos de datos conocidos como clases (presentaremos esta herramienta en el capítulo 3 y la exploraremos con detalle en los capítulos 9 y 10). Los usuarios pueden entonces “enseñar” a C++ cómo debe recibir y mostrar valores de estos nuevos tipos de datos, usando los operadores `>>` y `<<` (a esto se le conoce como **sobre carga de operadores**; un tema que exploraremos en el capítulo 11).

2.5 Conceptos acerca de la memoria

Los nombres de variables como `numero1`, `numero2` y `suma` en realidad corresponden a las **ubicaciones** en la memoria de la computadora. Cada variable tiene un nombre, un tipo, un tamaño y un valor.

En el programa de suma de la figura 2.5, cuando se ejecuta la instrucción

```
std::cin >> numero1; // lee el primer entero del usuario y lo coloca en numero1
```

en la línea 14, los caracteres que escribe el usuario se convierten en un entero, el cual se coloca en una ubicación de memoria a la que el compilador de C++ haya asignado el nombre `numero1`. Suponga que el usuario introduce el número 45 como el valor para `numero1`. La computadora colocará el 45 en la ubicación `numero1`, como se muestra en la figura 2.6.

Cada vez que se coloca un valor en una ubicación en memoria, ese valor sobrescribe al valor anterior en esa ubicación; por ende, se dice que la acción de colocar un nuevo valor en una ubicación en memoria es un proceso **destructivo**.

Regresando a nuestro programa de suma, cuando se ejecuta la instrucción

```
std::cin >> numero2; // lee el segundo entero del usuario y lo coloca en numero2
```

en la línea 17, suponga que el usuario introduce el valor 72. Este valor se coloca en la ubicación `numero2`, y la memoria aparece como se muestra en la figura 2.7. Observe que estas ubicaciones no necesariamente están adyacentes en la memoria.

Una vez que el programa obtiene valores para `numero1` y `numero2`, los suma y coloca el resultado de esta suma en la variable `suma`. La instrucción

```
suma = numero1 + numero2; // suma los números; almacena el resultado en suma
```

que realiza la suma también reemplaza el valor que estaba almacenado en `suma`. Esto ocurre cuando se coloca la suma calculada de `numero1` y `numero2` en la ubicación `suma` (sin importar qué valor haya tenido antes `suma`; ese valor se pierde). Después de calcular `suma`, la memoria aparece como se muestra en la figura 2.8. Observe que los valores de `numero1` y `numero2` aparecen exactamente como cuando se utilizaron en el cálculo de `suma`. Se utilizaron estos valores, pero no se destruyeron, en el momento en el que la computadora realizó el cálculo. Por ende, cuando se lee un valor de una ubicación de memoria, el proceso es **no destructivo**.



Figura 2.6 | Ubicación de memoria que muestra el nombre y el valor de la variable `numero1`.

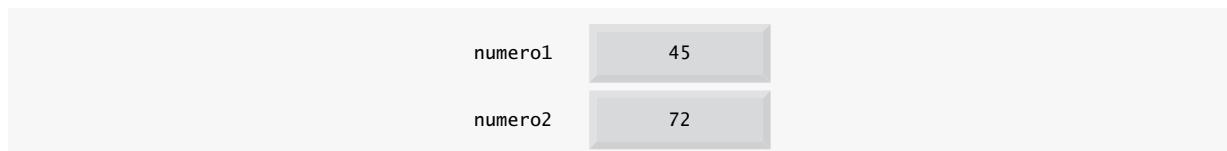


Figura 2.7 | Ubicaciones de memoria, después de almacenar valores para `numero1` y `numero2`.

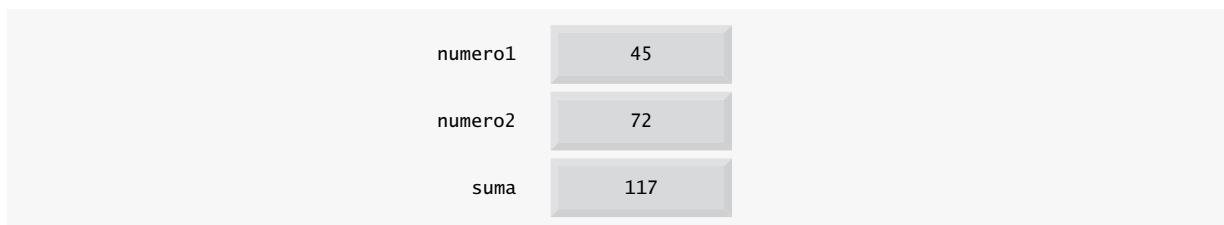


Figura 2.8 | Ubicaciones de memoria, después de calcular y almacenar la `suma` de `numero1` y `numero2`.

2.6 Aritmética

La mayoría de los programas realizan cálculos aritméticos. Los **operadores aritméticos** se sintetizan en la figura 2.9. Observe el uso de varios símbolos especiales que no se utilizan en álgebra. El asterisco (*) indica la multiplicación, y el **signo de porcentaje (%)** es el operador **módulo**, el cual describiremos en breve. Los operadores aritméticos en la figura 2.9 son operadores binarios, ya que funcionan con dos operandos. Por ejemplo, la expresión `numero1 + numero2` contiene el operador binario + y los dos operandos `numero1` y `numero2`.

La **división de enteros** (es decir, donde tanto el numerador como el denominador son enteros) produce un cociente entero; por ejemplo, la expresión `7 / 4` da como resultado 1, y la expresión `17 / 5` da como resultado 3. Observe que cualquier parte fraccionaria en una división de enteros se descarta (es decir, se **trunca**); no ocurre un redondeo.

C++ proporciona el **operador módulo**, %, el cual produce el residuo después de la división entera. El operador módulo sólo se puede utilizar con operandos enteros. La expresión `x % y` produce el residuo después de que `x` se divide entre `y`. Por lo tanto, `7 % 4` produce 3, y `17 % 5` produce 2. En capítulos posteriores consideraremos muchas aplicaciones interesantes del operador módulo, como determinar si un número es múltiplo de otro (un caso especial de esta aplicación es determinar si un número es par o impar).

Error común de programación 2.3



Tratar de utilizar el operador módulo (%) con operandos no enteros es un error de compilación.

Expresiones aritméticas en formato de línea recta

En la computadora, las expresiones aritméticas en C++ deben escribirse en **formato de línea recta**. Por lo tanto, las expresiones como “`a dividida entre b`” deben escribirse como `a / b`, de manera que todas las constantes, variables y operadores aparezcan en una línea recta. La siguiente notación algebraica:

$$\frac{a}{b}$$

no es generalmente aceptable para los compiladores, aunque ciertos paquetes de software de propósito especial soportan una notación más natural para las expresiones matemáticas complejas.

Paréntesis para agrupar subexpresiones

Los paréntesis se utilizan en las expresiones en C++ de la misma manera que en las expresiones algebraicas. Por ejemplo, para multiplicar `a` por la cantidad `b + c`, escribimos `a * (b + c)`.

Operación en C++	Operador aritmético de C++	Expresión algebraica	Expresión en C++
Suma	+	$f + 7$	<code>f + 7</code>
Resta	-	$p - c$	<code>p - c</code>
Multiplicación	*	bm o $b \cdot m$	<code>b * m</code>
División	/	x/y o $\frac{x}{y}$ o $x \div y$	<code>x / y</code>
Residuo	%	$r \bmod s$	<code>r % s</code>

Figura 2.9 | Operadores aritméticos.

Reglas de precedencia de operadores

C++ aplica los operadores en expresiones aritméticas en una secuencia precisa, determinada por las siguientes **reglas de precedencia de operadores**, que generalmente son las mismas que las que se utilizan en álgebra:

1. Los operadores en las expresiones contenidas dentro de pares de paréntesis se evalúan primero. Se dice que los paréntesis tienen el “nivel más alto de precedencia”. En casos de **paréntesis anidados o incrustados**, como:

$$((a + b) + c)$$

los operadores en el par más interno de paréntesis se aplican primero.

2. Las operaciones de multiplicación, división y módulo se aplican a continuación. Si una expresión contiene varias de esas operaciones, los operadores se aplican de izquierda a derecha. Se dice que los operadores de multiplicación, división y residuo tienen el mismo nivel de precedencia.
3. Las operaciones de suma y resta se aplican al último. Si una expresión contiene varias de esas operaciones, los operadores se aplican de izquierda a derecha. Los operadores de suma y resta tienen el mismo nivel de precedencia.

El conjunto de reglas de precedencia de operadores define el orden en el que C++ aplica los operadores. Cuando decimos que ciertos operadores se aplican de izquierda a derecha, nos referimos a su **asociatividad**. Por ejemplo, en la expresión

$$a + b + c$$

los operadores de suma (+) se asocian de izquierda a derecha, por lo que $a + b$ se calcula primero, y después se agrega c a esa suma para determinar el valor de toda la expresión. Más adelante veremos que algunos operadores se asocian de derecha a izquierda. En la figura 2.10 se sintetizan estas reglas de precedencia de operadores. Esta tabla se expandirá, a medida que se introduzcan operadores adicionales de C++. En el apéndice A se incluye una tabla de precedencia completa.

Ejemplos de expresiones algebraicas y de C++

Ahora, consideremos varias expresiones en vista de las reglas de precedencia de operadores. Cada ejemplo enlista una expresión algebraica y su equivalente en C++. El siguiente es un ejemplo de una media (promedio) aritmética de cinco términos:

$$\text{Álgebra: } m = \frac{a + b + c + d + e}{5}$$

$$C++: \quad m = (a + b + c + d + e) / 5;$$

Los paréntesis son obligatorios, ya que la división tiene una mayor precedencia que la suma. La cantidad completa ($a + b + c + d + e$) va a dividirse entre 5. Si los paréntesis se omiten por error, obtenemos $a + b + c + d + e / 5$, lo cual da como resultado

$$a + b + c + d + \frac{e}{5}$$

Operador(es)	Operación(es)	Orden de evaluación (precedencia)
()	Paréntesis	Se evalúa primero. Si los paréntesis son anidados, la expresión en el par más interno se evalúa primero. Si hay varios pares de paréntesis “en el mismo nivel” (es decir, no anidados), se evalúan de izquierda a derecha.
*, /, %	Multiplicación, División, Módulo	Se evalúan en segundo lugar. Si hay varios operadores de este tipo, se evalúan de izquierda a derecha.
+	Suma	Se evalúan al último. Si hay varios operadores de este tipo, se evalúan de izquierda a derecha.
-	Resta	

Figura 2.10 | Precedencia de los operadores aritméticos.

El siguiente es un ejemplo de la ecuación de una línea recta:

Álgebra: $y = mx + b$

C++: $y = m * x + b;$

No se requieren paréntesis. El operador de multiplicación se aplica primero, ya que la multiplicación tiene una mayor precedencia sobre la suma.

El siguiente ejemplo contiene las operaciones módulo (%), multiplicación, división, suma y resta:

Álgebra: $z = pr \% q + w / x - y$

C++: $z = p * r \% q + w / x - y;$

6 1 2 4 3 5

Los números dentro de los círculos bajo la instrucción indican el orden en el que C++ aplica los operadores. Las operaciones de multiplicación, residuo y división se evalúan primero, en orden de izquierda a derecha (es decir, se asocian de izquierda a derecha), ya que tienen mayor precedencia que la suma y la resta. Las operaciones de suma y resta se evalúan a continuación. Estas operaciones también se aplican de izquierda a derecha. Después se aplica el operador de asignación.

Evaluación de un polinomio de segundo grado

Para desarrollar una mejor comprensión de las reglas de precedencia de operadores, considere la evaluación de un polinomio de segundo grado ($y = ax^2 + bx + c$):

$y = a * x * x + b * x + c;$

6 1 2 4 3 5

Los números dentro de los círculos debajo de la instrucción indican el orden en el que C++ aplica los operadores. En C++ no hay operador aritmético para la exponenciación, por lo que hemos representado a x^2 como $x * x$. Pronto hablaremos sobre la función `pow` (“power” o potencia) de la biblioteca estándar que realiza la exponenciación. Debido a ciertas cuestiones sutiles en relación con los tipos de datos requeridos por `pow`, deferiremos una explicación detallada de esta función hasta el capítulo 6.



Error común de programación 2.4

Algunos lenguajes de programación usan los operadores `**` o `^` para representar la exponenciación. C++ no soporta estos operadores de exponenciación; si se utilizan para este fin, se producen errores.

Suponga que las variables `a`, `b`, `c` y `x` en el polinomio de segundo grado anterior se inicializan como sigue: `a = 2`, `b = 3`, `c = 7` y `x = 5`. La figura 2.11 muestra el orden en el que se aplican los operadores.

Al igual que en álgebra, es aceptable colocar paréntesis innecesarios en una expresión para hacer que ésta sea más clara. A dichos paréntesis innecesarios se les llama **paréntesis redundantes**. Por ejemplo, la instrucción de asignación anterior podría colocarse entre paréntesis, de la siguiente manera:

$y = (a * x * x) + (b * x) + c;$



Buena práctica de programación 2.14

El uso de paréntesis redundantes en expresiones aritméticas complejas puede hacer que éstas sean más fáciles de leer.

2.7 Toma de decisiones: operadores de igualdad y relacionales

Esta sección presenta una versión simple de la instrucción `if` de C++, la cual permite que un programa tome una acción alternativa, con base en la verdad o falsedad de cierta **condición**. Si se cumple la condición (es decir, si es verdadera), se ejecuta la instrucción que está en el cuerpo de la instrucción `if`. Si la condición no se cumple (es falsa), el cuerpo no se ejecuta. Veremos un ejemplo en breve.

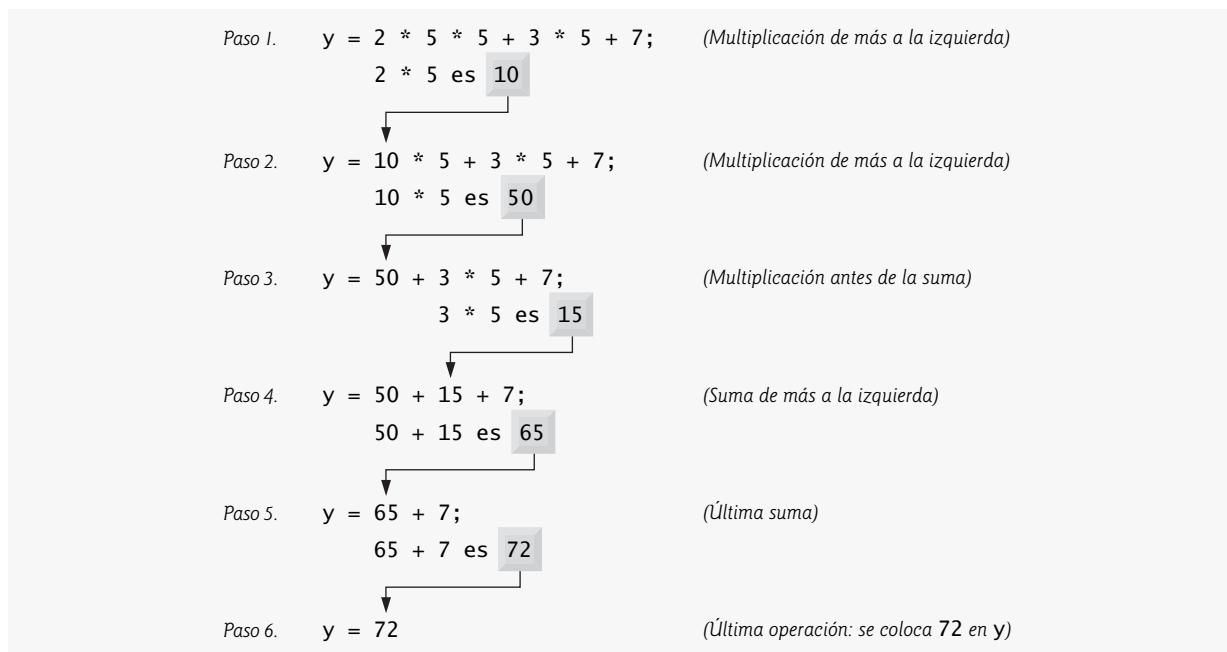


Figura 2.11 | Orden en el cual se evalúa un polinomio de segundo grado.

Las condiciones en las instrucciones `if` pueden formarse utilizando los **operadores de igualdad** y los **operadores relacionales**, que se sintetizan en la figura 2.12. Los operadores relacionales tienen todos el mismo nivel de precedencia y se asocian de izquierda a derecha. Los dos operadores de igualdad tienen el mismo nivel de precedencia, que es menor que la precedencia de los operadores relacionales, y se asocian de izquierda a derecha.



Error común de programación 2.5

Se producirá un error de sintaxis si cualquiera de los operadores ==, !=, >= y <= aparece con espacios entre su par de símbolos.



Error común de programación 2.6

Invertir el orden del par de símbolos en cualquiera de los operadores !=, >= y <= (al escribirlos como !=, => y =<, respectivamente) es comúnmente un error de sintaxis. En algunos casos, escribir != como != no será un error de sintaxis, sino casi con certeza será un error lógico, el cual tiene un efecto en tiempo de ejecución. En el capítulo 5 comprenderá esto, cuando aprenda acerca de los operadores lógicos. Un error lógico fatal hace que un programa falle y termine antes de tiempo. Un error lógico no fatal permite que un programa continúe ejecutándose, pero por lo general produce resultados incorrectos.



Error común de programación 2.7

Confundir el operador de igualdad == con el operador de asignación = produce errores lógicos. El operador de igualdad se debe leer como "es igual a", y el operador de asignación se debe leer como "obtiene" u "obtiene el valor de", o "se le asigna el valor de". Algunas personas prefieren leer el operador de igualdad como "doble igual". Como veremos en la sección 5.9, confundir estos operadores no necesariamente puede provocar un error de sintaxis fácil de reconocer, pero puede producir errores lógicos extremadamente no obvios.

El siguiente ejemplo utiliza seis instrucciones `if` para comparar dos números introducidos por el usuario. Si la condición en cualquiera de estas instrucciones `if` es verdadera, se ejecuta la instrucción de salida asociada con esa instrucción `if`. En la figura 2.13 se muestra el programa y los diálogos de entrada/salida de tres ejecuciones del ejemplo.

Las líneas 6 a 8

```
using std::cout; // el programa usa cout
using std::cin; // el programa usa cin
using std::endl; // el programa usa endl
```

Operador estándar algebraico de igualdad o relacional	Operador de igualdad o relacional de C++	Ejemplo de condición en C++	Significado de la condición en C++
<i>Operadores relacionales</i>			
>	>	x > y	x es mayor que y
<	<	x < y	x es menor que y
\$	>=	x >= y	x es mayor o igual que y
#	<=	x <= y	x es menor o igual que y
<i>Operadores de igualdad</i>			
=	==	x == y	x es igual a y
≠	!=	x != y	x no es igual a y

Figura 2.12 | Operadores de igualdad y relacionales.

son declaraciones `using`, las cuales eliminan la necesidad de repetir el prefijo `std::`, como hicimos en programas anteriores. Una vez que insertamos estas declaraciones `using`, podemos escribir `cout` en lugar de `std::cout`, `cin` en vez de `std::cin` y `endl` en lugar de `std::endl`, respectivamente, en el resto del programa. [Nota: de aquí en adelante en el libro, cada ejemplo contiene una o más declaraciones `using`.]

```

1 // Fig. 2.13: fig02_13.cpp
2 // Comparación de enteros mediante instrucciones if, operadores
3 // relacionales y operadores de igualdad.
4 #include <iostream> // permite al programa realizar operaciones de entrada y salida
5
6 using std::cout; // el programa usa cout
7 using std::cin; // el programa usa cin
8 using std::endl; // el programa usa endl
9
10 // La función main empieza la ejecución del programa
11 int main()
12 {
13     int numero1; // primer entero a comparar
14     int numero2; // segundo entero a comparar
15
16     cout << "Escriba dos enteros a comparar: "; // pide los datos al usuario
17     cin >> numero1 >> numero2; // lee dos enteros del usuario
18
19     if ( numero1 == numero2 )
20         cout << numero1 << " == " << numero2 << endl;
21
22     if ( numero1 != numero2 )
23         cout << numero1 << " != " << numero2 << endl;
24
25     if ( numero1 < numero2 )
26         cout << numero1 << " < " << numero2 << endl;
27
28     if ( numero1 > numero2 )
29         cout << numero1 << " > " << numero2 << endl;
30
31     if ( numero1 <= numero2 )
32         cout << numero1 << " <= " << numero2 << endl;
33

```

Figura 2.13 | Comparación de enteros mediante instrucciones if, operadores relacionales y operadores de igualdad. (Parte I de 2).

```

34     if ( numero1 >= numero2 )
35         cout << numero1 << " >= " << numero2 << endl;
36
37     return 0; // indica que el programa terminó correctamente
38
39 } // fin de la función main

```

Escriba dos enteros a comparar: 3 7

3 != 7
3 < 7
3 <= 7

Escriba dos enteros a comparar: 22 12

22 != 12
22 > 12
22 >= 12

Escriba dos enteros a comparar: 7 7

7 == 7
7 < 7
7 >= 7

Figura 2.13 | Comparación de enteros mediante instrucciones `if`, operadores relacionales y operadores de igualdad. (Parte 2 de 2).



Buena práctica de programación 2.15

Coloque las declaraciones `using` justo después de la instrucción `#include` a la que hagan referencia.

Las líneas 13 y 14

```

int numero1; // primer entero a comparar
int numero2; // segundo entero a comparar

```

declaran las variables que se utilizan en el programa. Recuerde que las variables pueden declararse en una sola declaración, o en declaraciones separadas.

El programa utiliza operaciones de extracción de flujos en cascada (línea 17) para recibir dos enteros como entrada. Recuerde que podemos escribir `cin` (en lugar de `std::cin`) debido a la línea 7. Primero se lee un valor y se coloca en la variable `numero1`, y después se lee otro valor y se coloca en la variable `numero2`.

La instrucción `if` en las líneas 19 y 20

```

if ( numero1 == numero2 )
    cout << numero1 << " == " << numero2 << endl;

```

compara los valores de las variables `numero1` y `numero2`, para probar su igualdad. Si los valores son iguales, la instrucción en la línea 20 muestra una línea de texto, la cual indica que los números son iguales. Si las condiciones son `true` en una o más de las instrucciones `if` que empiezan en las líneas 22, 25, 28, 31 y 34, la correspondiente instrucción del cuerpo muestra una línea de texto apropiada.

Observe que cada instrucción `if` en la figura 2.13 tiene una sola instrucción en su cuerpo y que cada instrucción del cuerpo tiene sangría. En el capítulo 4 le mostraremos cómo especificar instrucciones `if` con cuerpos con varias instrucciones (encerrando las instrucciones del cuerpo en un par de llaves, `{ }`), creando lo que se conoce como una **instrucción compuesta** o **un bloque**.



Buena práctica de programación 2.16

Aplique sangría a la(s) instrucción(es) en el cuerpo de una instrucción `if` para mejorar la legibilidad.



Buena práctica de programación 2.17

Para aumentar la legibilidad, no debe haber más de una instrucción por cada línea en el programa.



Error común de programación 2.8

Colocar un punto y coma inmediatamente después del paréntesis derecho de la condición en una instrucción `if` es, generalmente, un error lógico (aunque no es un error de sintaxis). El punto y coma hace que el cuerpo de la instrucción `if` esté vacío, por lo que esta instrucción no realiza ninguna acción, sin importar que la condición sea verdadera o no. Peor aún, la instrucción del cuerpo original de la instrucción `if` ahora se convierte en una instrucción en secuencia con la instrucción `if` y siempre se ejecuta, lo cual a menudo ocasiona que el programa produzca resultados incorrectos.

Observe el uso del espacio en blanco en la figura 2.13. Recuerde que los caracteres de espacio en blanco, como tabuladores, nuevas líneas y espacios, generalmente son ignorados por el compilador. Por lo tanto, las instrucciones pueden dividirse en varias líneas y pueden espaciarse de acuerdo con las preferencias del programador. Es un error de sintaxis dividir identificadores, cadenas (como "holá") y constantes (como el número 1000) a través de varias líneas.



Error común de programación 2.9

Es un error de sintaxis dividir un identificador al insertar caracteres de espacio en blanco (por ejemplo, escribir `main` como `ma in`).



Buena práctica de programación 2.18

Una instrucción larga puede esparcirse en varias líneas. Si una sola instrucción debe dividirse entre varias líneas, los puntos que elija para hacer la división deben tener sentido, como después de una coma en una lista separada por comas, o después de un operador en una expresión larga. Si una instrucción se divide entre dos o más líneas, aplique sangría a todas las líneas subsecuentes y alinee a la izquierda el grupo de líneas con sangría.

La figura 2.14 muestra la precedencia y asociatividad de los operadores que se presentan en este capítulo. Los operadores se muestran de arriba a abajo, en orden descendente de precedencia. Observe que todos estos operadores, con la excepción del operador de asignación, `=`, se asocian de izquierda a derecha. La suma es asociativa a la izquierda, por lo que una expresión como `x + y + z` se evalúa como si se hubiera escrito así: `(x + y) + z`. El operador de asignación, `=`, asocia de derecha a izquierda, por lo que una expresión como `x = y = 0` se evalúa como si se hubiera escrito así: `x = (y = 0)`, donde, como pronto veremos, primero se asigna el valor 0 a la variable `y`, y después se asigna el resultado de esa asignación, 0, a `x`.



Buena práctica de programación 2.19

Consulte la tabla de precedencia y asociatividad de operadores cuando escriba expresiones que contengan muchos operadores. Confirme que los operadores en la expresión se ejecuten en el orden que usted espera. Si no está seguro en cuanto al orden de evaluación en una expresión compleja, divida la expresión en instrucciones más pequeñas o use paréntesis para forzar el orden de evaluación, de la misma forma que como lo haría en una expresión algebraica. Asegúrese de observar que ciertos operadores, como la asignación (`=`), se asocian de derecha a izquierda, en lugar de hacerlo de izquierda a derecha.

Operadores	Asociatividad	Tipo
<code>()</code>	izquierda a derecha	paréntesis
<code>*</code> <code>/</code> <code>%</code>	izquierda a derecha	multiplicativa
<code>+</code> <code>-</code>	izquierda a derecha	suma
<code><<</code> <code>>></code>	izquierda a derecha	inserción/extracción de flujo
<code><</code> <code><=</code> <code>></code> <code>>=</code>	izquierda a derecha	relacional
<code>==</code> <code>!=</code>	izquierda a derecha	igualdad
<code>=</code>	derecha a izquierda	asignación

Figura 2.14 | Precedencia y asociatividad de los operadores descritos hasta ahora.

2.8 (Opcional) Ejemplo práctico de Ingeniería de Software: cómo examinar la especificación de requerimientos del ATM

Ahora empezaremos nuestro ejemplo práctico opcional de diseño e implementación orientados a objetos. Las secciones del Ejemplo práctico de Ingeniería de Software al final de éste y los siguientes capítulos le ayudarán a incursionar en la orientación a objetos. Desarrollaremos software para un sistema de cajero automático (ATM) simple, con lo cual le brindaremos una experiencia de diseño e implementación sustancial, cuidadosamente pautada y completa. En los capítulos 3 al 7, 9 y 13, llevaremos a cabo los diversos pasos de un proceso de diseño orientado a objetos (DOO) utilizando UML, mientras relacionamos estos pasos con los conceptos orientados a objetos que se describen en los capítulos. El apéndice G implementa el ATM utilizando las técnicas de la programación orientada a objetos (POO) en C++. Presentamos la solución completa al ejemplo práctico. Éste no es un ejercicio, sino una experiencia de aprendizaje de extremo a extremo, que concluye con un análisis detallado del código en C++ que implementamos, con base en nuestro diseño. Este ejemplo práctico le ayudará a acostumbrarse a los tipos de problemas sustanciales que se encuentran en la industria, y sus soluciones potenciales.

Empezaremos nuestro proceso de diseño con la presentación de una **especificación de requerimientos**, la cual especifica el propósito general del sistema ATM y *qué* debe hacer. A lo largo del ejemplo práctico, nos referiremos a la especificación de requerimientos para determinar con precisión la funcionalidad que debe incluir el sistema.

Especificación de requerimientos

Un banco local pretende instalar una nueva máquina de cajero automático (ATM), para permitir a los usuarios (es decir, los clientes del banco) realizar transacciones financieras básicas (figura 2.15). Cada usuario sólo puede tener una cuenta en el banco. Los usuarios del ATM deben poder ver el saldo de su cuenta, retirar efectivo (es decir, sacar dinero de una cuenta) y depositar fondos (es decir, meter dinero en una cuenta).

La interfaz de usuario del cajero automático contiene los siguientes componentes de hardware:

- una pantalla que muestra mensajes al usuario
- un teclado que recibe datos numéricos de entrada del usuario
- un dispensador de efectivo que dispensa efectivo al usuario, y
- una ranura de depósito que recibe sobres para depósitos del usuario.

El dispensador de efectivo comienza cada día cargado con 500 billetes de \$20. [Nota: debido al alcance limitado de este ejemplo práctico, ciertos elementos del ATM que se describen aquí no imitan exactamente a los de un ATM real. Por ejemplo, generalmente un ATM contiene un dispositivo que lee el número de cuenta del usuario de una tarjeta para ATM, mientras que este ATM pide al usuario que escriba su número de cuenta en el teclado. Un ATM verdadero también imprime por lo general un recibo al final de una sesión, pero toda la salida de este ATM aparece en la pantalla.]

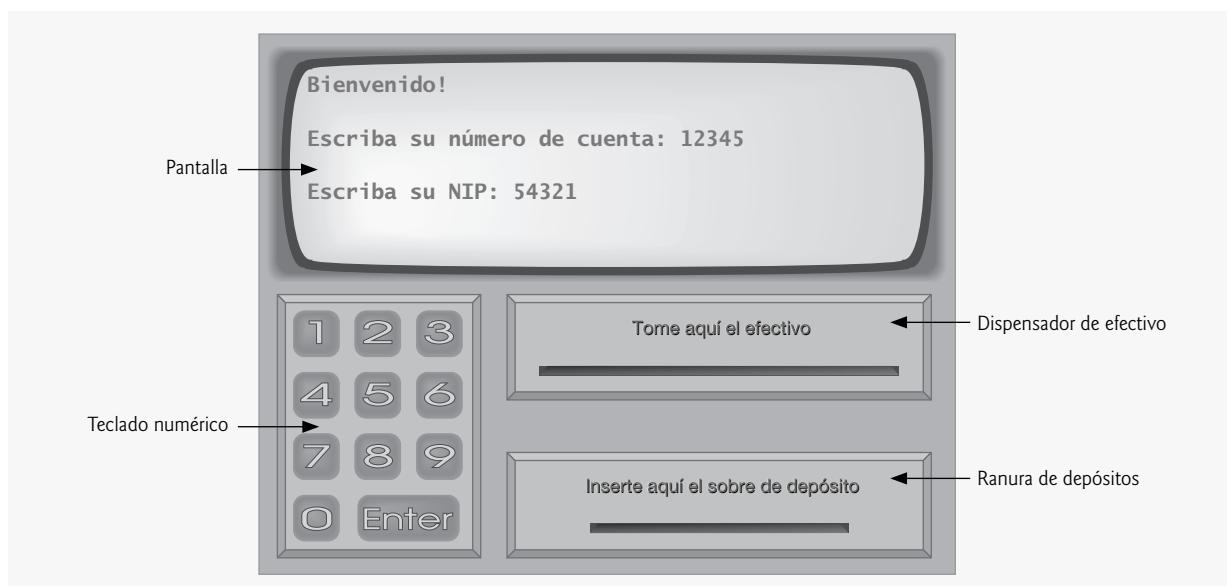


Figura 2.15 | Interfaz de usuario del cajero automático.

El banco desea que usted desarrolle software para realizar las transacciones financieras que inicien los clientes del banco a través del ATM. El banco integrará posteriormente el software con el hardware del ATM. El software debe encapsular la funcionalidad de los dispositivos de hardware (por ejemplo: dispensador de efectivo, ranura para depósito) dentro de los componentes de software, pero no necesita preocuparse por la forma en que estos dispositivos deben realizar sus tareas. El hardware del ATM no se ha desarrollado aún, por lo que en lugar de que usted escriba su software para ejecutarse en el ATM, deberá desarrollar una primera versión del software para que se ejecute en una computadora personal. Esta versión debe utilizar el monitor de la computadora para simular la pantalla del ATM y el teclado de la computadora para simular el teclado numérico del ATM.

Una sesión con el ATM consiste en la autenticación de un usuario (es decir, proporcionar la identidad del usuario) con base en un número de cuenta y un número de identificación personal (NIP), seguida de la creación y la ejecución de transacciones financieras. Para autenticar un usuario y realizar transacciones, el ATM debe interactuar con la base de datos de información sobre las cuentas del banco. [Nota: una base de datos es una colección organizada de datos almacenados en una computadora.] Para cada cuenta de banco, la base de datos almacena un número de cuenta, un NIP y un saldo que indica la cantidad de dinero en la cuenta. [Nota: para simplificar, vamos a asumir que el banco planea construir sólo un ATM, por lo que no necesitamos preocuparnos por que varios ATM accedan a esta base de datos al mismo tiempo. Lo que es más, vamos a suponer que el banco no va a realizar modificaciones en la información que hay en la base de datos mientras un usuario accede al ATM. Además, cualquier sistema comercial como un ATM se topa con cuestiones de seguridad con una complejidad razonable, las cuales van más allá del alcance de un curso de programación de primer o segundo semestre. No obstante, para simplificar nuestro ejemplo vamos a suponer que el banco confía en el ATM para que acceda a la información en la base de datos y la manipule sin necesidad de medidas de seguridad considerables.]

Al acercarse al ATM, el usuario deberá experimentar la siguiente secuencia de eventos (vea la figura 2.15):

1. La pantalla muestra un mensaje de bienvenida y pide al usuario que introduzca un número de cuenta.
2. El usuario introduce un número de cuenta de cinco dígitos, mediante el uso del teclado.
3. En la pantalla aparece un mensaje, en el que se pide al usuario que introduzca su NIP (número de identificación personal) asociado con el número de cuenta especificado.
4. El usuario introduce un NIP de cinco dígitos mediante el teclado numérico.
5. Si el usuario introduce un número de cuenta válido y el NIP correcto para esa cuenta, la pantalla muestra el menú principal (figura 2.16). Si el usuario introduce un número de cuenta inválido o un NIP incorrecto, la pantalla muestra un mensaje apropiado y después el ATM regresa al *paso 1* para reiniciar el proceso de autenticación.

Una vez que el ATM autentica al usuario, el menú principal (figura 2.16) debe contener una opción numerada para cada uno de los tres tipos de transacciones: solicitud de saldo (opción 1), retiro (opción 2) y depósito (opción 3). El menú

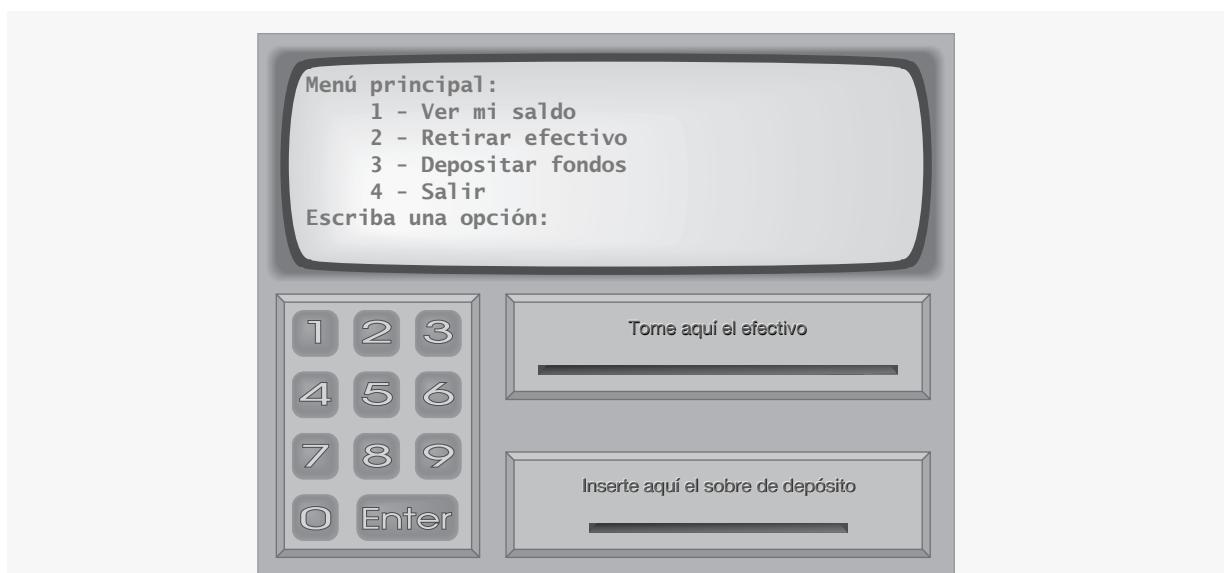


Figura 2.16 | Menú principal del ATM.

principal también muestra una opción para que el usuario pueda salir del sistema (opción 4). Después el usuario elegirá si desea realizar una transacción (oprime 1, 2 o 3) o salir del sistema (oprime 4). Si el usuario introduce una opción inválida, la pantalla muestra un mensaje de error y vuelve a mostrar el menú principal.

Si el usuario oprime 1 para solicitar su saldo, la pantalla mostrará el saldo de su cuenta bancaria. Para ello, el ATM deberá obtener el saldo de la base de datos del banco.

Las siguientes acciones ocurren cuando el usuario elige la opción 2 para hacer un retiro:

1. La pantalla muestra un menú (vea la figura 2.17) que contiene montos de retiro estándar: \$20 (opción 1), \$40 (opción 2), \$60 (opción 3), \$100 (opción 4) y \$200 (opción 5). El menú también contiene una opción que permite al usuario cancelar la transacción (opción 6).
2. El usuario introduce la selección del menú (1 a 6) mediante el teclado numérico.
3. Si el monto a retirar elegido es mayor que el saldo de la cuenta del usuario, la pantalla muestra un mensaje indicando esta situación y pide al usuario que seleccione un monto más pequeño. Entonces el ATM regresa al *paso 1*. Si el monto a retirar elegido es menor o igual que el saldo de la cuenta del usuario (es decir, un monto de retiro aceptable), el ATM procede al *paso 4*. Si el usuario opta por cancelar la transacción (opción 6), el ATM muestra el menú principal (figura 2.16) y espera la entrada del usuario.
4. Si el dispensador contiene suficiente efectivo para satisfacer la solicitud, el ATM procede al *paso 5*. En caso contrario, la pantalla muestra un mensaje indicando el problema y pide al usuario que seleccione un monto de retiro más pequeño. Despues el ATM regresa al *paso 1*.
5. El ATM carga (resta) el monto de retiro al saldo de la cuenta del usuario en la base de datos del banco.
6. El dispensador de efectivo entrega el monto deseado de dinero al usuario.
7. La pantalla muestra un mensaje para recordar al usuario que tome el dinero.

Las siguientes acciones ocurren cuando el usuario elige la opción 3 (mientras se muestre el menú principal) para hacer un depósito:

1. La pantalla muestra un mensaje que pide al usuario que introduzca un monto de depósito o que escriba 0 (cero) para cancelar la transacción.
2. El usuario introduce un monto de depósito o 0 mediante el teclado numérico. [Nota: el teclado no contiene un punto decimal o signo de dólares, por lo que el usuario no puede escribir una cantidad real en dólares (por ejemplo, \$1.25), sino que debe escribir un monto de depósito en forma de número de centavos (por ejemplo, 125). Despues, el ATM divide este número entre 100 para obtener un número que represente un monto en dólares (por ejemplo, $125 \div 100 = 1.25$).]

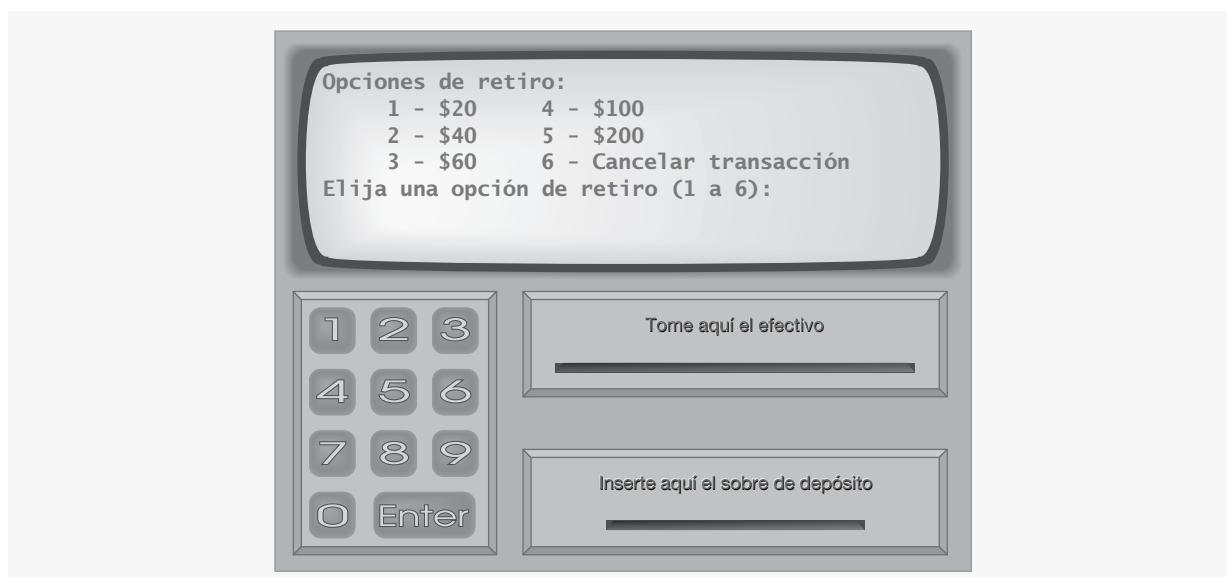


Figura 2.17 | Menú de retiro del ATM.

3. Si el usuario especifica un monto a depositar, el ATM procede al *paso 4*. Si elige cancelar la transacción (escribiendo 0), el ATM muestra el menú principal (figura 2.16) y espera la entrada del usuario.
4. La pantalla muestra un mensaje indicando al usuario que introduzca un sobre de depósito en la ranura para depósitos.
5. Si la ranura de depósitos recibe un sobre dentro de un plazo de tiempo no mayor a 2 minutos, el ATM abona (suma) el monto del depósito al saldo de la cuenta del usuario en la base de datos del banco. [Nota: este dinero no está disponible de inmediato para retirarse. El banco debe primero verificar físicamente el monto de efectivo en el sobre de depósito, y cualquier cheque que éste contenga debe validarse (es decir, el dinero debe transferirse de la cuenta del emisor del cheque a la cuenta del beneficiario). Cuando ocurra uno de estos eventos, el banco actualizará de manera apropiada el saldo del usuario que está almacenado en su base de datos. Esto ocurre de manera independiente al sistema ATM.] Si la ranura de depósito no recibe un sobre dentro de un plazo de tiempo no mayor a dos minutos, la pantalla muestra un mensaje indicando que el sistema canceló la transacción debido a la inactividad. Después el ATM muestra el menú principal y espera la entrada del usuario.

Una vez que el sistema ejecuta una transacción en forma exitosa, debe volver a mostrar el menú principal (figura 2.16) para que el usuario pueda realizar transacciones adicionales. Si el usuario elige salir del sistema, la pantalla debe mostrar un mensaje de agradecimiento y después el mensaje de bienvenida para el siguiente usuario.

Análisis del sistema de ATM

En la declaración anterior se presentó un ejemplo simplificado de una especificación de requerimientos. Por lo general, dicho documento es el resultado de un proceso detallado de **recopilación de requerimientos**, el cual podría incluir entrevistas con usuarios potenciales del sistema y especialistas en campos relacionados con el mismo. Por ejemplo, un analista de sistemas que se contrate para preparar una especificación de requerimientos para software bancario (por ejemplo, el sistema ATM que describimos aquí) podría entrevistar expertos financieros para obtener una mejor comprensión de *qué* es lo que debe hacer el software. El analista utilizaría la información recopilada para compilar una lista de **requerimientos del sistema**, para guiar a los diseñadores de sistemas en el proceso de diseño del mismo.

El proceso de recopilación de requerimientos es una tarea clave de la primera etapa del ciclo de vida del software. El **ciclo de vida del software** especifica las etapas a través de las cuales el software evoluciona, desde el tiempo en que fue concebido hasta el tiempo en que se retira de su uso. Por lo general, estas etapas incluyen: análisis, diseño, implementación, prueba y depuración, despliegue, mantenimiento y retiro. Existen varios modelos de ciclo de vida del software, cada uno con sus propias preferencias y especificaciones respecto a cuándo y qué tan a menudo deben llevar a cabo los ingenieros de software las diversas etapas. Los **modelos de cascada** realizan cada etapa una vez en sucesión, mientras que los **modelos iterativos** pueden repetir una o más etapas varias veces a lo largo del ciclo de vida de un producto.

La etapa de análisis del ciclo de vida del software se enfoca en definir el problema a resolver. Al diseñar cualquier sistema, uno debe *resolver el problema de la manera correcta*, pero de igual manera uno debe *resolver el problema correcto*. Los analistas de sistemas recolectan los requerimientos que indican el problema específico a resolver. Nuestra especificación de requerimientos describe nuestro sistema ATM con el suficiente detalle como para que usted no necesite pasar por una etapa de análisis exhaustiva; ya lo hicimos por usted.

Para capturar lo que debe hacer un sistema propuesto, los desarrolladores emplean a menudo una técnica conocida como **modelado de caso-uso**. Este proceso identifica los **casos de uso** del sistema, cada uno de los cuales representa una capacidad distinta que el sistema provee a sus clientes. Por ejemplo, es común que los ATM tengan varios casos de uso, como "Ver saldo de cuenta", "Retirar efectivo", "Depositar fondos", "Transferir fondos entre cuentas" y "Comprar estampas postales". El sistema ATM simplificado que construiremos en este ejemplo práctico requiere sólo los tres primeros casos de uso (figura 2.18).

Cada uno de los casos de uso describe un escenario común en el cual el usuario utiliza el sistema. Usted ya leyó las descripciones de los casos de uso del sistema ATM en la especificación de requerimientos; las listas de pasos requeridos para realizar cada tipo de transacción (como solicitud de saldo, retiro y depósito) describen en realidad los tres casos de uso de nuestro ATM: "Ver saldo de cuenta", "Retirar efectivo" y "Depositar fondos".

Diagramas de caso-uso

Ahora presentaremos el primero de varios diagramas de UML en nuestro ejemplo práctico del ATM. Vamos a crear un **diagrama de caso-uso** para modelar las interacciones entre los clientes de un sistema (en este ejemplo práctico, los clientes del banco) y el sistema. El objetivo es mostrar los tipos de interacciones que tienen los usuarios con un sistema sin proveer los detalles; éstos se mostrarán en otros diagramas de UML (los cuales presentaremos a lo largo del ejemplo práctico). A menudo, los diagramas de caso-uso se acompañan de texto informal que describe los casos de uso con más

detalle, como el texto que aparece en la especificación de requerimientos. Los diagramas de caso-uso se producen durante la etapa de análisis del ciclo de vida del software. En sistemas más grandes, los diagramas de caso-uso son herramientas simples pero indispensables, que ayudan a los diseñadores de sistemas a enfocarse en satisfacer las necesidades de los usuarios.

La figura 2.18 muestra el diagrama de caso-uso para nuestro sistema ATM. La figura humana representa a un **actor**, el cual define los roles que desempeña una entidad externa (como una persona u otro sistema) cuando interactúa con el sistema. Para nuestro cajero automático, el actor es un Usuario que puede ver el saldo de una cuenta, retirar efectivo y depositar fondos del ATM. El Usuario no es una persona real, sino que constituye los roles que puede desempeñar una persona real (al desempeñar el papel de un Usuario) mientras interactúa con el ATM. Hay que tener en cuenta que un diagrama de caso-uso puede incluir varios actores. Por ejemplo, el diagrama de caso-uso para un sistema ATM de un banco real podría incluir también un actor llamado Administrador, que rellene el dispensador de efectivo a diario.

Para identificar al actor en nuestro sistema, debemos examinar la especificación de requerimientos, la cual dice: “los usuarios del ATM deben poder ver el saldo de su cuenta, retirar efectivo y depositar fondos”. Por lo tanto, el actor en cada uno de estos tres casos de uso es el Usuario que interactúa con el ATM. Una entidad externa (una persona real) desempeña el papel del Usuario para realizar transacciones financieras. La figura 2.18 muestra un actor, cuyo nombre (Usuario) aparece debajo del actor en el diagrama. UML modela cada caso de uso como un óvalo conectado a un actor con una línea sólida.

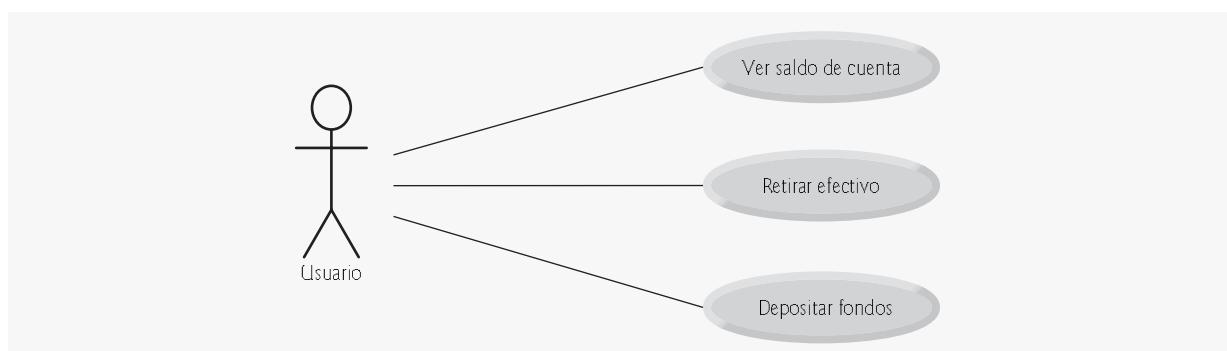


Figura 2.18 | Diagrama de caso-uso para el sistema ATM, desde la perspectiva del Usuario.

Los ingenieros de software (más específicamente, los diseñadores de sistemas) deben analizar la especificación de requerimientos o un conjunto de casos de uso, y diseñar el sistema antes de que los programadores lo implementen. Durante la etapa de análisis, los analistas de sistemas se enfocan en comprender la especificación de requerimientos para producir una especificación de alto nivel que describa *qué* es lo que el sistema debe hacer. El resultado de la etapa de diseño (una **especificación de diseño**) debe especificar claramente *cómo* debe construirse el sistema para satisfacer estos requerimientos. En las siguientes secciones del Ejemplo práctico de Ingeniería de Software, llevaremos a cabo los pasos de un proceso simple de diseño orientado a objetos (DOO) con el sistema ATM, para producir una especificación de diseño que contenga una colección de diagramas de UML y texto de apoyo. Recuerde que UML está diseñado para utilizarse con cualquier proceso de DOO. Existen muchos de esos procesos, de los cuales el más conocido es Rational Unified Process™ (RUP), desarrollado por Rational Software Corporation (ahora una división de IBM). RUP es un proceso robusto para diseñar aplicaciones a “nivel industrial”. Para este ejemplo práctico, presentaremos nuestro propio proceso de diseño simplificado.

Diseño del sistema ATM

Ahora comenzaremos la etapa de diseño de nuestro sistema ATM. Un **sistema** es un conjunto de componentes que interactúan para resolver un problema. Por ejemplo, para realizar sus tareas designadas, nuestro sistema ATM tiene una interfaz de usuario (figura 2.15), contiene software para ejecutar transacciones financieras e interactúa con una base de datos de información de cuentas bancarias. La **estructura del sistema** describe los objetos del sistema y sus interrelaciones. El **comportamiento del sistema** describe la manera en que cambia el sistema a medida que sus objetos interactúan entre sí. Todo sistema tiene tanto estructura como comportamiento; los diseñadores deben especificar ambos. Existen varios tipos distintos de estructuras y comportamientos de un sistema. Por ejemplo, las interacciones entre los objetos en el sistema son distintas a las interacciones entre el usuario y el sistema, pero aun así ambas constituyen una porción del comportamiento del sistema.

El estándar UML 2 especifica 13 tipos de diagramas para documentar los modelos de sistemas. Cada tipo de diagrama modela una característica distinta de la estructura o del comportamiento de un sistema: seis diagramas se relacionan con la estructura del sistema; los siete restantes se relacionan con su comportamiento. Aquí enlistaremos sólo los seis tipos de diagramas que utilizaremos en nuestro ejemplo práctico, uno de los cuales (el diagrama de clases) modela la estructura del sistema, mientras que los otros cinco modelan el comportamiento. En el apéndice H, UML 2: Tipos de diagramas adicionales, veremos las generalidades sobre los siete tipos restantes de diagramas de UML.

1. Los **diagramas de caso-uso**, como el de la figura 2.18, modelan las interacciones entre un sistema y sus entidades externas (actores) en términos de casos de uso (capacidades del sistema, como “Ver saldo de cuenta”, “Retirar efectivo” y “Depositar fondos”).
2. Los **diagramas de clases**, que estudiará en la sección 3.11, modelan las clases o “bloques de construcción” que se utilizan en un sistema. Cada sustantivo u “objeto” que se describe en la especificación de requerimientos es candidato para ser una clase en el sistema (por ejemplo, “cuenta”, “teclado”). Los diagramas de clases nos ayudan a especificar las relaciones estructurales entre las partes del sistema. Por ejemplo, el diagrama de clases del sistema ATM especificará que el ATM está compuesto físicamente de una pantalla, un teclado, un dispensador de efectivo y una ranura para depósitos.
3. Los **diagramas de máquina de estado**, que estudiará en la sección 3.11, modelan las formas en que un objeto cambia de estado. El **estado** de un objeto se indica mediante los valores de todos los atributos del objeto, en un momento dado. Cuando un objeto cambia de estado, puede comportarse de manera distinta en el sistema. Por ejemplo, después de validar el NIP de un usuario, el ATM cambia del estado “usuario no autenticado” al estado “usuario autenticado”, punto en el cual el ATM permite al usuario realizar transacciones financieras (por ejemplo, ver el saldo de su cuenta, retirar efectivo, depositar fondos).
4. Los **diagramas de actividad**, que también estudiará en la sección 5.11, modelan la **actividad** de un objeto: el flujo de trabajo (secuencia de eventos) del objeto durante la ejecución del programa. Un diagrama de actividad modela las acciones que realiza el objeto y especifica el orden en el cual desempeña estas acciones. Por ejemplo, un diagrama de actividad muestra que el ATM debe obtener el saldo de la cuenta del usuario (de la base de datos de información de las cuentas del banco) antes de que la pantalla pueda mostrar el saldo al usuario.
5. Los **diagramas de comunicación** (llamados **diagramas de colaboración** en versiones anteriores de UML) modelan las interacciones entre los objetos en un sistema, con un énfasis acerca de *qué* interacciones ocurren. En la sección 7.12 aprenderá que estos diagramas muestran cuáles objetos deben interactuar para realizar una transacción en el ATM. Por ejemplo, el ATM debe comunicarse con la base de datos de información de las cuentas del banco para obtener el saldo de una cuenta.
6. Los **diagramas de secuencia** modelan también las interacciones entre los objetos en un sistema, pero a diferencia de los diagramas de comunicación, enfatizan *cuándo* ocurren las interacciones. En la sección 7.12 aprenderá que estos diagramas ayudan a mostrar el orden en el que ocurren las interacciones al ejecutar una transacción financiera. Por ejemplo, la pantalla pide al usuario que escriba un monto de retiro antes de dispensar el efectivo.

En la sección 3.11 seguiremos diseñando nuestro sistema ATM, al identificar las clases de la especificación de requerimientos. Para lograr esto, vamos a extraer sustantivos clave y frases nominales de la especificación de requerimientos. Mediante el uso de estas clases, desarrollaremos nuestro primer borrador del diagrama de clases que modelará la estructura de nuestro sistema ATM.

Recursos en Internet y Web

Los siguientes URLs proporcionan información sobre el diseño orientado a objetos con el UML.

www-306.ibm.com/software/rational/uml/

Lista preguntas frecuentes (FAQs) acerca del UML, proporcionado por IBM Rational.

www.douglass.co.uk/documents/softdocwiz.com.UML.htm

Sitio anfitrión del Diccionario del Lenguaje unificado de modelado, el cual lista y define todos los términos utilizados en el UML.

www-306.ibm.com/software/rational/offering/design.html

Proporciona información acerca del software IBM Racional, disponible para el diseño de sistemas. Ofrece descargas de versiones de prueba de 30 días de varios productos, como IBM Rational Rose® XDE Developer.

www.borland.com/us/products/together/index.html

Proporciona una licencia gratuita de 30 días para descargar una versión de prueba de Borland® Together® Control Center™, una herramienta de desarrollo de software que soporta el UML.

<http://modelingcommunity.telelogic.com/developer-trial.aspx>

Proporciona una licencia gratuita de 30 días para descargar una versión de prueba de I-Logix Rhapsody®: un entorno de desarrollo controlado por modelos y basado en UML 2.

argouml.tigris.org

Contiene información y descargas para ArgoUML, una herramienta gratuita de código fuente abierto de UML, escrita en Java.

www.objectsbydesign.com/books/booklist.html

Provee una lista de libros acerca de UML y el diseño orientado a objetos.

www.objectsbydesign.com/tools/umltools_byCompany.html

Provee una lista de herramientas de software que utilizan UML, como IBM Rational Rose, Embarcadero Describe, Sparx Systems Enterprise Architect, I-Logix Rhapsody y Gentleware Poseidon para UML.

www.ootips.org/ood-principles.html

Proporciona respuestas a la pregunta “¿Qué se requiere para tener un buen diseño orientado a objetos?”

parlezuml.com/tutorials/umlforjava.htm

Ofrece un tutorial de UML para desarrolladores de Java, el cual presenta los diagramas de UML y los compara detalladamente con el código que los implementa.

www.cetus-links.org/oo_uml.html

Introduce el UML y proporciona vínculos a numerosos recursos sobre UML.

www.agilemodeling.com/essays/umlDiagrams.htm

Proporciona descripciones detalladas y tutoriales acerca de cada uno de los 13 tipos de diagramas de UML 2.

Lecturas recomendadas

Los siguientes libros proporcionan información acerca del diseño orientado a objetos con el UML.

Booch, G. *Object-Oriented Analysis and Design with Applications*, Tercera edición. Boston: Addison-Wesley, 2004.

Eriksson, H., et al. *UML 2 Toolkit*. Nueva York: John Wiley, 2003.

Fowler, M. *UML Distilled*, Tercera edición. Boston: Addison-Wesley Professional, 2004.

Kruchten, P. *The Rational Unified Process: An Introduction*. Boston: Addison-Wesley, 2004.

Larman, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, Segunda edición. Upper Saddle River, NJ: Prentice Hall, 2002.

Roques, P. *UML in Practice: The Art of Modeling Software Systems Demonstrated Through Worked Examples and Solutions*. Nueva York: John Wiley, 2004.

Rosenberg, D. y K. Scott. *Applying Use Case Driven Object Modeling with UML: An Annotated e-Commerce Example*. Reading, MA: Addison-Wesley, 2001.

Rumbaugh, J., I. Jacobson y G. Booch. *The Complete UML Training Course*. Upper Saddle River, NJ: Prentice Hall, 2000.

Rumbaugh, J., I. Jacobson y G. Booch. *The Unified Modeling Language Reference Manual*. Reading, MA: Addison-Wesley, 1999.

Rumbaugh, J., I. Jacobson y G. Booch. *The Unified Software Development Process*. Reading, MA: Addison-Wesley, 1999.

Schneider, G. y J. Winters. *Applying Use Cases: A Practical Guide*. Segunda edición. Boston: Addison-Wesley Professional, 2002.

Ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

2.1 Suponga que habilitamos a un usuario de nuestro sistema ATM para transferir dinero entre dos cuentas bancarias. Modifique el diagrama de caso-uso de la figura 2.18 para reflejar este cambio.

2.2 Los _____ modelan las interacciones entre los objetos en un sistema, con énfasis acerca de *cuándo* ocurren estas interacciones.

- a) Diagramas de clases.
- b) Diagramas de secuencia.
- c) Diagramas de comunicación.
- d) Diagramas de actividad.

2.3 ¿Cuál de las siguientes opciones enumera las etapas de un típico ciclo de vida de software, en orden secuencial?

- a) diseño, análisis, implementación, prueba.
- b) diseño, análisis, prueba, implementación.
- c) análisis, diseño, prueba, implementación.
- d) análisis, diseño, implementación, prueba.

Respuestas a los ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

2.1 La figura 2.19 contiene un diagrama de caso-uso para una versión modificada de nuestro sistema ATM, que también permite a los usuarios transferir dinero entre cuentas.

2.2 b.

2.3 d.

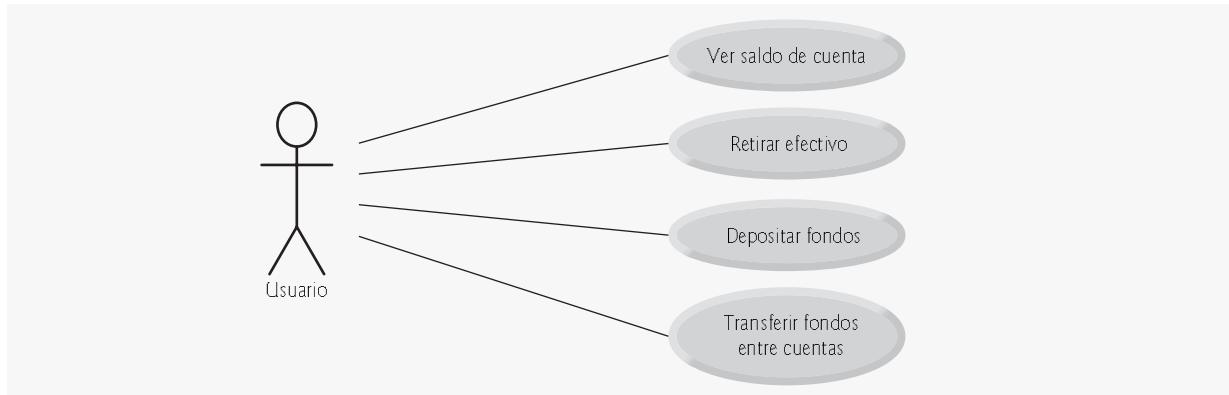


Figura 2.19 | Diagrama de caso-uso para una versión modificada de nuestro sistema ATM, que también permite a los usuarios transferir dinero entre varias cuentas.

2.9 Repaso

En este capítulo aprendió muchas características importantes de C++, incluyendo cómo mostrar datos en la pantalla, recibir datos del teclado y declarar variables de tipos fundamentales. En especial, aprendió a usar el objeto flujo de salida `cout` y el objeto flujo de entrada `cin` para crear programas simples interactivos. Explicamos cómo se almacenan las variables en memoria, y cómo se obtienen de ésta. También aprendió a usar los operadores aritméticos para realizar cálculos. Vimos el orden en el que C++ aplica los operadores (es decir, las reglas de precedencia de los operadores), así como la asociatividad de los mismos. También aprendió cómo es que la instrucción `if` de C++ permite a un programa tomar decisiones. Por último, presentamos los operadores de igualdad y relacionales, los cuales se utilizan para formar condiciones en las instrucciones `if`.

Las aplicaciones no orientadas a objetos que presentamos en este capítulo lo introdujeron a los conceptos de programación básicos. Como veremos en el capítulo 3, por lo general las aplicaciones de C++ contienen sólo unas cuantas líneas de código en la función `main`; estas instrucciones comúnmente crean los objetos que realizan el trabajo de la aplicación, y después los objetos “se encargan del resto”. En el capítulo 3 aprenderá a implementar sus propias clases y a utilizar objetos de esas clases en las aplicaciones.

Resumen

Sección 2.2 Su primer programa en C++: imprimir una línea de texto

- Los comentarios de una sola línea empiezan con `//`. Los programadores insertan comentarios en sus programas para documentarlos y mejorar su legibilidad.
- La computadora no realiza acción alguna con los comentarios cuando se ejecuta el programa; el compilador de C++ los ignora y no genera ningún código objeto en lenguaje máquina.
- Una directiva del preprocesador empieza con `#` y es un mensaje para el preprocesador de C++. El preprocesador procesa estas directivas antes de compilar el programa, y a diferencia de las instrucciones de C++, no terminan con punto y coma.
- La línea `#include <iostream>` indica al preprocesador de C++ que debe incluir el contenido del archivo de encabezado de flujos de entrada/salida en el programa. Este archivo contiene la información necesaria para compilar programas que utilizan `std::cin` y `std::cout`, y los operadores `<>`.
- El espacio en blanco (es decir, las líneas en blanco, los espacios y los tabuladores) hace que los programas sean más fáciles de leer. El compilador ignora los caracteres de espacio en blanco fuera de las literales.
- Los programas en C++ empiezan su ejecución en `main`, aun si `main` no aparece primero en el programa.

- La palabra clave `int` a la izquierda de `main` indica que `main` “devuelve” un valor entero.
- Una llave izquierda (`{`) debe empezar el cuerpo de toda función. Su correspondiente llave derecha (`}`) debe terminar el cuerpo de cada función.
- A una cadena encerrada entre comillas dobles se le conoce comúnmente como cadena de caracteres, mensaje o literal de cadena. El compilador no ignora los caracteres de espacio en blanco dentro de las cadenas.
- Toda instrucción debe terminar con un punto y coma (también conocido como terminador de instrucciones).
- Las operaciones de entrada y salida en C++ se realizan mediante flujos de caracteres.
- El objeto flujo de salida `std::cout` (que por lo general está conectado a la pantalla) se utiliza para enviar datos de salida. Pueden imprimirse varios elementos de datos mediante la concatenación de operadores de inserción de flujo (`<<`).
- El objeto flujo de entrada `std::cin` (que por lo general está conectado al teclado) se utiliza para recibir datos de entrada. Pueden recibirse como entrada varios elementos de datos, mediante la concatenación de operadores de extracción de flujo (`>>`).
- Los objetos flujo `std::cin` y `std::cout` facilitan la interacción entre el usuario y la computadora. Como esta interacción se asemeja a un diálogo, a menudo se le denomina computación conversacional o interactiva.
- La notación `std::cout` especifica que estamos usando un nombre (en este caso, `cout`) que pertenece al “espacio de nombres” `std`.
- Al encontrar una barra diagonal inversa (es decir, un carácter de escape) en una cadena de caracteres, el siguiente carácter se combina con la barra diagonal inversa para formar una secuencia de escape.
- La secuencia de escape `\n` representa una nueva línea. Hace que el cursor (es decir, el indicador de la posición actual en la pantalla) se desplace al principio de la siguiente línea en la pantalla.
- Un mensaje que pide al usuario que realice una acción específica se conoce como indicador (`prompt`).
- La palabra clave `return` de C++ es uno de varios medios para salir de una función.

Sección 2.4 Otro programa en C++: suma de enteros

- Todas las variables en un programa en C++ deben declararse antes de poder utilizarlas.
- El nombre de una variable en C++ es cualquier identificador válido que no sea una palabra clave. Un identificador es una serie de caracteres compuesta por letras, dígitos y símbolos de guión bajo (`_`). Los identificadores no pueden empezar con un dígito. Los identificadores de C++ pueden tener cualquier longitud; sin embargo, algunos sistemas y/o implementaciones de C++ pueden imponer ciertas restricciones en cuanto a la longitud de los identificadores.
- C++ es sensible a mayúsculas y minúsculas.
- La mayoría de los cálculos se realizan en instrucciones de asignación.
- Una variable es una ubicación en la memoria de la computadora donde se puede almacenar un valor, para que lo utilice un programa.
- Las variables de tipo `int` contienen valores enteros; es decir, números como `7, -11, 0, 31914`.

Sección 2.5 Conceptos acerca de la memoria

- Toda variable almacenada en la memoria de la computadora tiene un nombre, un valor, un tipo y un tamaño.
- Cada vez que se coloca un nuevo valor en una ubicación de memoria, el proceso es destructivo; es decir, el nuevo valor sustituye al valor anterior en esa ubicación. El valor anterior se pierde.
- Cuando se lee un valor de la memoria, el proceso es no destructivo; es decir, se lee una copia del valor, dejando el valor original sin modificación en la ubicación de memoria.
- El manipulador de flujos `std::endl` imprime una nueva línea y después “vacía el búfer de salida”.

Sección 2.6 Aritmética

- C++ evalúa las expresiones aritméticas en una secuencia precisa, determinada con base en las reglas de precedencia y asociatividad de los operadores.
- Los paréntesis pueden utilizarse para agrupar expresiones.
- La división entera (es decir, tanto el numerador como el denominador son enteros) produce un cociente entero. Cualquier parte fraccionaria en la división entera se trunca; no hay redondeo.
- El operador módulo (%) produce el residuo después de la división entera. Este operador sólo se puede usar con operandos enteros.

Sección 2.7 Toma de decisiones: operadores de igualdad y relacionales

- La instrucción `if` permite que un programa tome una acción alternativa, con base en la verdad o falsedad de cierta condición. El formato de una instrucción `if` es

```
if ( condición )
    instrucción;
```

Si la condición es verdadera, se ejecuta la instrucción en el cuerpo del `if`. Si la condición no se cumple (es falsa), se omite la instrucción del cuerpo.

- Las condiciones en las instrucciones `if` se forman comúnmente mediante el uso de operadores de igualdad y relacionales. El resultado de usar estos operadores siempre es el valor verdadero o falso.

- La declaración

```
using std::cout;
```

es una declaración `using` que informa al compilador dónde puede encontrar `cout` (espacio de nombres `std`), y elimina la necesidad de repetir el prefijo `std::`. Una vez que incluimos esta declaración `using` podemos escribir, por ejemplo, `cout` en lugar de `std::cout` en el resto de un programa.

Terminología

<code>/* ... */</code> , comentario (comentario estilo C)	literal
<code>//</code> , comentario	literal de cadena
asociatividad de izquierda a derecha	<code>main</code> , función
asociatividad de los operadores	manipulador de flujos
bloque	memoria
cadena	mensaje
cadena de caracteres	módulo, operador (%)
carácter de escape (\)	multiplicación, operador (*)
<code>cin</code> , objeto	nueva línea, carácter (\n)
comentario (//)	objeto flujo de entrada estándar (<code>cin</code>)
concatenación de operaciones de inserción de flujo	objeto flujo de salida estándar (<code>cout</code>)
condición	operaciones de inserción de flujo en cascada
<code>cout</code> , objeto	operador
cuerpo de una función	operador aritmético
cursor	operador binario
decisión	operador de asignación (=)
declaración	operador de extracción de flujo (>>)
declaración <code>using</code>	operador de inserción de flujo (<<)
directiva del preprocesador	operadores de igualdad
división entera	<code>==</code> , “es igual a”
encadenamiento de las operaciones de inserción de flujo	<code>!=</code> , “no es igual a”
entero (<code>int</code>)	operadores relacionales
error de compilación	<code><</code> , “es menor que”
error de sintaxis	<code><=</code> , “es menor o igual que”
error del compilador	<code>></code> , “es mayor que”
error en tiempo de compilación	<code>>=</code> , “es mayor o igual que”
error fatal	operando
error lógico	paréntesis ()
error lógico no fatal	paréntesis anidados
escritura destructiva	paréntesis redundantes
espacio en blanco	precedencia
flujo	programa autodocumentado
función	punto y coma (;), terminador de instrucciones
identificador	realizar una acción
indicador	reglas de precedencia de operadores
instrucción	<code>return</code> , instrucción
instrucción compuesta	salir de una función
instrucción <code>if</code>	secuencia de escape
<code>int</code> , tipo de datos	sensible a mayúsculas/minúsculas
<code><iostream></code> , archivo de encabezado de flujos de entrada/salida	terminador de instrucciones (;
lectura no destructiva	tipo de datos
lista separada por comas	ubicación de memoria
	variable

Ejercicios de autoevaluación

- 2.1 Complete las siguientes oraciones:

- Todo programa en C++ empieza su ejecución en la función _____.
- Un(a) _____ empieza el cuerpo de toda función, y un(a) _____ termina el cuerpo.
- Toda instrucción de C++ termina con un(a) _____.

62 Capítulo 2 Introducción a la programación en C++

- d) La secuencia de escape `\n` representa el carácter _____, el cual hace que el cursor se posicione al principio de la siguiente línea en la pantalla.
- e) La instrucción _____ se utiliza para tomar decisiones.
- 2.2** Indique si cada una de las siguientes instrucciones es *verdadera* o *falsa*. Si es *falsa*, explique por qué. Asuma que se usa la instrucción `using std::cout;`
- Los comentarios hacen que la computadora imprima el texto que va después de los caracteres `//` en la pantalla, al ejecutarse el programa.
 - Cuando la secuencia de escape `\n` se imprime con `cout` y el operador de inserción de flujo, el cursor se posiciona al principio de la siguiente línea en la pantalla.
 - Todas las variables deben declararse antes de utilizarlas.
 - Todas las variables deben recibir un tipo al declararlas.
 - C++ considera que las variables `numero` y `NuMeRo` son idénticas.
 - Las declaraciones pueden aparecer casi en cualquier parte del cuerpo de una función de C++.
 - El operador módulo (`%`) se puede utilizar sólo con operandos enteros.
 - Los operadores aritméticos `*`, `/`, `%`, `+ y -` tienen todos el mismo nivel de precedencia.
 - Un programa en C++ que imprime tres líneas de salida debe contener tres instrucciones en las que se utilicen `cout` y el operador de inserción de flujo.
- 2.3** Escriba una sola instrucción en C++ para realizar cada una de las siguientes tareas (suponga que no se han utilizado declaraciones `using`):
- Declarar las variables `c`, `estaEsUnaVariable`, `q76354` y `numero` como de tipo `int`.
 - Pedir al usuario que introduzca un entero. Termine el mensaje del indicador con un signo de dos puntos (`:`) seguido de un espacio, y deje el cursor posicionado después del espacio.
 - Recibir un entero como entrada del usuario mediante el teclado, y almacenarlo en la variable entera `edad`.
 - Si la variable `numero` no es igual a 7, mostrar "La variable numero no es igual a 7".
 - Imprimir el mensaje "Este es un programa en C++" en una línea.
 - Imprimir el mensaje "Este es un programa en C++" en dos líneas. La primera línea debe terminar con `es un`.
 - Imprimir el mensaje "Este es un programa en C++"; cada palabra se debe escribir en una línea separada.
 - Imprimir el mensaje "Este es un programa en C++". Separe una palabra de otra mediante un tabulador.
- 2.4** Escriba una declaración (o comentario) para realizar cada una de las siguientes tareas (suponga que se han utilizado declaraciones `using` para `cin`, `cout` y `endl`):
- Indicar que un programa calculará el producto de tres enteros.
 - Declarar las variables `x`, `y`, `z` y `resultado` de tipo `int` (en instrucciones separadas).
 - Pedir al usuario que escriba tres enteros.
 - Leer tres enteros del usuario y almacenarlos en las variables `x`, `y` y `z`.
 - Calcular el producto de los tres enteros contenidos en las variables `x`, `y` y `z`, y asignar el resultado a la variable `resultado`.
 - Imprimir "El producto es ", seguido del valor de la variable `resultado`.
 - Devolver un valor de `main`, indicando que el programa terminó correctamente.
- 2.5** Utilizando las instrucciones que escribió en el ejercicio 2.4, escriba un programa completo que calcule e imprima el producto de tres enteros. Agregue comentarios al código donde sea apropiado. [Nota: necesitará escribir las declaraciones `using` necesarias.]
- 2.6** Identifique y corrija los errores en cada una de las siguientes instrucciones (suponga que se utiliza la instrucción `using std::cout`):
- `if (c < 7);
cout << "c es menor que 7\n";`
 - `if (c => 7)
cout << "c es igual o mayor que 7\n";`

Respuestas a los ejercicios de autoevaluación

- 2.1** a) `main`. b) llave izquierda `{}`, llave derecha `}`. c) punto y coma `(;)`. d) nueva línea. e) `if`.
- 2.2** a) Falso. Los comentarios no producen ninguna acción cuando el programa se ejecuta. Se utilizan para documentar programas y mejorar su legibilidad.
b) Verdadero.
c) Verdadero.

- d) Verdadero.
e) Falso. C++ es sensible a mayúsculas y minúsculas, por lo que estas variables son distintas.
f) Verdadero.
g) Verdadero.
h) Falso. Los operadores *, / y % se encuentran en el mismo nivel de precedencia, y los operadores + y - se encuentran en un nivel menor de precedencia.
i) Falso. Una instrucción con cout y varias secuencias de escape \n puede imprimir varias líneas.
- 2.3**
- a) int c, estaEsUnaVariable, q76354, numero;
 - b) std::cout << "Escriba un entero: ";
 - c) std::cin >> edad;
 - d) if (numero != 7)
 - std::cout << "La variable numero no es igual a 7\n";
 - e) std::cout << "Este es un programa en C++\n";
 - f) std::cout << "Este es un\n programa en C++\n";
 - g) std::cout << "Este\nes\nun\nprograma\nen\nC++\n";
 - h) std::cout << "Este\tes\tun\tprograma\tten\tC++\n";
- 2.4**
- a) // Calcula el producto de tres enteros
 - b) int x;
int y;
int z;
int resultado;
 - c) cout << "Escriba tres enteros: ";
 - d) cin >> x >> y >> z;
 - e) resultado = x * y * z;
 - f) cout << "El producto es " << resultado << endl;
 - g) return 0;
- 2.5** (Vea el siguiente programa).

```

1 // Calcula el producto de tres enteros
2 #include <iostream> // permite al programa realizar operaciones de entrada y salida
3
4 using std::cout; // el programa usa cout
5 using std::cin; // el programa usa cin
6 using std::endl; // el programa usa endl
7
8 // la función main empieza la ejecución del programa
9 int main()
10 {
11     int x; // primer entero a multiplicar
12     int y; // segundo entero a multiplicar
13     int z; // tercer entero a multiplicar
14     int resultado; // el producto de los tres enteros
15
16     cout << "Escriba tres enteros: "; // pide los datos al usuario
17     cin >> x >> y >> z; // lee tres enteros del usuario
18     resultado = x * y * z; // multiplica los tres enteros; almacena el resultado
19     cout << "El producto es " << resultado << endl; // imprime el resultado; fin de línea
20
21     return 0; // indica que el programa se ejecutó correctamente
22 } // fin de la función main

```

- 2.6**
- a) *Error:* punto y coma después del paréntesis derecho de la condición en la instrucción if.
Corrección: elimine el punto y coma después del paréntesis derecho. [Nota: el resultado de este error es que la instrucción de salida se ejecutará sin importar que la condición en la instrucción if sea verdadera o no.] El punto y coma después del paréntesis derecho es una instrucción nula (o vacía): una instrucción que no hace nada. En el siguiente capítulo aprenderemos más acerca de la instrucción nula.
 - b) *Error:* el operador relacional =>.
Corrección: cambie => a >=, y tal vez quiera cambiar “igual o mayor que” a “mayor o igual que”, también.

Ejercicios

2.7 Hable sobre el significado de cada uno de los siguientes objetos:

- a) `std::cin`
- b) `std::cout`

2.8 Complete las siguientes oraciones:

- a) _____ se utilizan para documentar un programa y mejorar su legibilidad.
- b) El objeto que se utiliza para imprimir información en la pantalla es _____.
- c) Una instrucción de C++ que toma una decisión es _____.
- d) La mayoría de los cálculos se realizan normalmente mediante instrucciones _____.
- e) El objeto _____ recibe valores de entrada del teclado.

2.9 Escriba una sola instrucción o línea en C++ que realice cada una de las siguientes tareas:

- a) Imprimir el mensaje "Escriba dos números".
- b) Asignar el producto de las variables `b` y `c` a la variable `a`.
- c) Indicar que un programa va a realizar un cálculo de nómina (es decir, usar texto que ayude a documentar un programa).
- d) Recibir tres valores de entrada del teclado y colocarlos en las variables enteras `a`, `b` y `c`.

2.10 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

- a) Los operadores en C++ se evalúan de izquierda a derecha.
- b) Los siguientes nombres de variables son todos válidos: `_barra_inferior_`, `m928134`, `t5`, `j7`, `sus_ventas`, `su_cuenta_total`, `a`, `b`, `c`, `z`, `z2`.
- c) La instrucción `cout << "a = 5;"`; es un ejemplo típico de una instrucción de asignación.
- d) Una expresión aritmética válida en C++ sin paréntesis se evalúa de izquierda a derecha.
- e) Los siguientes nombres de variables son todos inválidos: `3g`, `87`, `67h2`, `h22`, `2h`.

2.11 Complete cada una de las siguientes oraciones:

- a) ¿Qué operaciones aritméticas se encuentran en el mismo nivel de precedencia que la multiplicación? _____.
- b) Cuando los paréntesis están anidados, ¿cuál conjunto de paréntesis se evalúa primero en una expresión aritmética? _____.
- c) Una ubicación en la memoria de la computadora que puede contener distintos valores en distintos momentos durante la ejecución de un programa se llama _____.

2.12 ¿Qué se imprime (si acaso) cuando se ejecuta cada una de las siguientes instrucciones de C++? Si no se imprime nada, entonces responda "nada". Suponga que $x = 2$ y $y = 3$.

- a) `cout << x;`
- b) `cout << x + x;`
- c) `cout << "x=" ;`
- d) `cout << "x = " << x;`
- e) `cout << x + y << " = " << y + x;`
- f) `z = x + y;`
- g) `cin >> x >> y;`
- h) // `cout << "x + y = " << x + y;`
- i) `cout << "\n";`

2.13 ¿Cuáles de las siguientes instrucciones de C++ contienen variables, cuyos valores se modifican?

- a) `cin >> b >> c >> d >> e >> f;`
- b) `p = i + j + k + 7;`
- c) `cout << "variables cuyos valores se destruyen";`
- d) `cout << "a = 5";`

2.14 Dada la ecuación algebraica $y = ax^3 + 7$, ¿cuáles de las siguientes instrucciones (si acaso) en C++ son correctas para esta ecuación?

- a) `y = a * x * x * x + 7;`
- b) `y = a * x * x * (x + 7);`
- c) `y = (a * x) * x * (x + 7);`
- d) `y = (a * x) * x * x + 7;`
- e) `y = a * (x * x * x) + 7;`
- f) `y = a * x * (x * x + 7);`

2.15 Indique el orden de evaluación de los operadores en cada una de las siguientes instrucciones en C++, y muestre el valor de x después de ejecutar cada una de ellas:

- a) $x = 7 + 3 * 6 / 2 - 1;$
 b) $x = 2 \% 2 + 2 * 2 - 2 / 2;$
 c) $x = (3 * 9 * (3 + (9 * 3 / (3))));$

2.16 Escriba un programa que pida al usuario que escriba dos números, que obtenga los números del usuario e imprima la suma, producto, diferencia y cociente de los números.

2.17 Escriba un programa que imprima los números del 1 al 4 en la misma línea, con cada par de números adyacentes separado por un espacio. Haga esto de varias formas:

- a) Utilizando una instrucción con un operador de inserción de flujo.
 - b) Utilizando una instrucción con cuatro operadores de inserción de flujo.
 - c) Utilizando cuatro instrucciones.

2.18 Escriba un programa que pida al usuario que escriba dos enteros, que obtenga los números del usuario e imprima el número más grande, seguido de las palabras "es más grande". Si los números son iguales, imprima el mensaje "Estos números son iguales."

2.19 Escriba un programa que reciba tres enteros del teclado e imprima la suma, promedio, producto, menor y mayor de esos números. El diálogo de la pantalla debe aparecer de la siguiente manera:

```
Introduzca tres enteros distintos: 13 27 14
La suma es 54
El promedio es 18
El producto es 4914
El menor es 13
El mayor es 27
```

2.20 Escriba un programa que lea el radio de un círculo como un número entero y que imprima su diámetro, circunferencia y área. Use el valor constante 3.14159 para π . Realice todos los cálculos en instrucciones de salida. [Nota: en este capítulo sólo hemos visto constantes enteras y variables. En el capítulo 4 hablaremos sobre los números de punto flotante; es decir, valores que pueden tener puntos decimales.]

2.21 Escriba un programa que imprima un cuadro, un óvalo, una flecha y un diamante como se muestra a continuación:

2.22 ¿Qué imprime el siguiente código?

```
cout << "*\n**\n***\n****\n*****" << endl;
```

2.23 Escriba un programa que lea cinco enteros y que determine e imprima los enteros mayor y menor en el grupo. Use solamente las técnicas de programación que aprendió en este capítulo.

2.24 Escriba un programa que lea un entero y que determine e imprima si es impar o par. [Sugerencia: use el operador módulo. Un número par es un múltiplo de dos. Cualquier múltiplo de dos deja un residuo de cero cuando se divide entre dos.]

2.25 Escriba un programa que lea dos enteros, determine si el primero es un múltiplo del segundo e imprima el resultado. [Sugerencia: use el operador de módulo.]

2.26 Escriba una aplicación que muestre un patrón de tablero de damas con ocho instrucciones de salida, y después muestre el mismo patrón utilizando el menor número de instrucciones posible.

* * * * *

A 5x5 grid of black asterisks (*). The grid consists of five rows and five columns, with each cell containing a single asterisk.

2.27 He aquí un adelanto. En este capítulo, aprendió sobre los enteros y el tipo `int`. C++ también puede representar letras mayúsculas, minúsculas y una considerable variedad de símbolos especiales. C++ utiliza enteros pequeños de manera interna para representar cada uno de los distintos caracteres. Al conjunto de caracteres que utiliza una computadora, y las correspondientes representaciones enteras para esos caracteres, se le conoce como el **conjunto de caracteres** de esa computadora. Podemos imprimir un carácter encerrándolo entre comillas sencillas, como se muestra a continuación:

```
cout << 'A'; // imprimir una letra A mayúscula
```

Podemos imprimir el equivalente entero de un carácter mediante el uso de `static_cast`, como se muestra a continuación:

```
cout << static_cast< int >( 'A' ); // imprime 'A' como un entero
```

A esto se le conoce como operación de **conversión** (en el capítulo 4 presentaremos formalmente las conversiones). Cuando se ejecuta la instrucción anterior, imprime el valor 65 (en sistemas que utilizan el **conjunto de caracteres ASCII**). Escriba un programa que imprima el equivalente entero de un carácter escrito en el teclado. Almacene la entrada en una variable de tipo `char`. Pruebe su programa varias veces, usando letras mayúsculas, minúsculas, dígitos y caracteres especiales (como \$).

2.28 Escriba un programa que reciba como entrada un número entero de cinco dígitos, que separe ese número en sus dígitos individuales y los imprima, cada uno separado de los demás por tres espacios. [Sugerencia: use los operadores de división entera y módulo.] Por ejemplo, si el usuario escribe el número 42339, el programa debe imprimir:

4		2		3		3		9
---	--	---	--	---	--	---	--	---

2.29 Utilizando sólo las técnicas que aprendió en este capítulo, escriba un programa que calcule los cuadrados y cubos de los números enteros del 0 al 10, y que imprima los valores resultantes en formato de tabla, como se muestra a continuación:

numero	cuadrado	cubo
0	0	0
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

3



*Nada puede tener valor
sin ser un objeto de utilidad.*

—Karl Marx

*Sus sirvientes públicos
le sirven bien.*

—Adlai E. Stevenson

*Saber cómo responder a
alguien que habla,
Contestar a alguien que
envía un mensaje.*

—Amenemope

*Usted verá algo nuevo.
Dos cosas. Y las llamo
Cosa Uno y Cosa Dos.*

—Dr. Theodor Seuss Geisel

Introducción a las clases y los objetos

OBJETIVOS

En este capítulo aprenderá a:

- Comprender qué son las clases, los objetos, las funciones miembro y los miembros de datos.
- Cómo definir una clase y utilizarla para crear un objeto.
- Cómo definir las funciones miembro en una clase para implementar los comportamientos de ésta.
- Cómo declarar miembros de datos en una clase para implementar los atributos de ésta.
- Saber cómo llamar a una función miembro un objeto para hacer que realice su tarea.
- Conocer las diferencias entre los miembros de datos de una clase y las variables locales de una función.
- Cómo utilizar un constructor para asegurar que los datos de un objeto se inicialicen cuando se cree el objeto.
- Saber cómo maquinar una clase para separar su interfaz de su implementación, y fomentar la reutilización.

- 3.1 Introducción
- 3.2 Clases, objetos, funciones miembro y miembros de datos
- 3.3 Generalidades acerca de los ejemplos del capítulo
- 3.4 Definición de una clase con una función miembro
- 3.5 Definición de una función miembro con un parámetro
- 3.6 Miembros de datos, funciones *establecer* y funciones *obtener*
- 3.7 Inicialización de objetos mediante constructores
- 3.8 Colocar una clase en un archivo separado para fines de reutilización
- 3.9 Separar la interfaz de la implementación
- 3.10 Validación de datos mediante funciones *establecer*
- 3.11 (Opcional) Ejemplo práctico de Ingeniería de Software: identificación de las clases en la especificación de requerimientos del ATM
- 3.12 Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)

3.1 Introducción

En el capítulo 2 creó programas simples que mostraban mensajes al usuario, obtenían información de él, realizaban cálculos y tomaban decisiones. En este capítulo empezará a escribir programas que emplean los conceptos básicos de la programación orientada a objetos que presentamos en la sección 1.21. Una característica común de todos los programas del capítulo 2 fue que todas las instrucciones que ejecutaban tareas se encontraban en la función `main`. Por lo general, los programas que desarrollará en este libro consistirán de la función `main` y una o más clases, cada una de las cuales puede contener miembros de datos y funciones miembro. Si usted se integra a un equipo de desarrollo en la industria, podría trabajar en sistemas de software que contengan cientos, o incluso miles de clases. En este capítulo desarrollaremos un marco de trabajo simple y bien maquinado para organizar las aplicaciones orientadas a objetos en C++.

Primeramente explicaremos la noción de las clases mediante el uso de un ejemplo real. Despues presentaremos una secuencia cuidadosamente pausada de siete programas funcionales completos para demostrarle cómo crear y utilizar sus propias clases. Estos ejemplos empiezan nuestro ejemplo práctico acerca de cómo desarrollar una clase tipo libro de calificaciones, que los instructores pueden utilizar para mantener las calificaciones de las pruebas de sus estudiantes. Durante los siguientes capítulos ampliaremos este ejemplo práctico y culminaremos con la versión que se presenta en el capítulo 7, Arreglos y vectores. También presentaremos la clase `string` de la biblioteca estándar de C++ en este capítulo.

3.2 Clases, objetos, funciones miembro y miembros de datos

Vamos a empezar con una analogía simple, para ayudarle a comprender el concepto de las clases y su contenido, que vimos en la sección 1.21. Suponga que desea conducir un auto y, para hacer que aumente su velocidad, debe presionar el pedal del acelerador. ¿Qué debe ocurrir antes de que pueda hacer esto? Bueno, antes de poder conducir un auto, alguien tiene que diseñarlo y construirlo. Por lo general, un auto empieza en forma de dibujos de ingeniería, similares a los planos de construcción que se utilizan para diseñar una casa. Estos dibujos de ingeniería incluyen el diseño del pedal del acelerador, que el conductor utiliza para aumentar su velocidad. En cierto sentido, el pedal “oculta” los complejos mecanismos que se encargan de que el auto aumente su velocidad, de igual forma que el pedal del freno “oculta” los mecanismos que disminuyen la velocidad del auto y el volante “oculta” los mecanismos que hacen que el auto dé vuelta, entre otros. Esto permite que las personas con poco o nada de conocimiento acerca de cómo funcionan los motores puedan conducir un auto con facilidad, con sólo utilizar el pedal del acelerador, el pedal del freno, el volante, el mecanismo de cambio de velocidad de la transmisión y otras “interfaces” simples y amigables para el usuario para los mecanismos internos complejos del auto.

Desafortunadamente, no puede conducir los dibujos de ingeniería de un auto; antes de poder conducir un auto, éste debe construirse a partir de los dibujos de ingeniería que lo describen. Un auto completo tendrá un pedal acelerador verdadero para hacer que aumente su velocidad. Pero aun así no es suficiente; el auto no acelerará por su propia cuenta, así que el conductor debe oprimir el pedal del acelerador.

Ahora vamos a utilizar nuestro ejemplo del auto para introducir los conceptos clave de programación de esta sección. Para realizar una tarea en una aplicación se requiere una función (como `main`, que describimos en el capítulo 2).

La función describe los mecanismos que se encargan de realizar sus tareas. La función oculta al usuario las tareas complejas que realiza, de la misma forma que el pedal del acelerador de un auto oculta al conductor los complejos mecanismos para hacer que el auto vaya más rápido. En C++, empezamos por crear una unidad de aplicación llamada clase para alojar a una función, así como los dibujos de ingeniería de un auto alojan el diseño del pedal del acelerador. En la sección 1.21 vimos que una función que pertenece a una clase se llama función miembro. En una clase se proporcionan una o más funciones miembro, las cuales están diseñadas para realizar las tareas de esa clase. Por ejemplo, una clase que representa a una cuenta bancaria podría contener una función miembro para depositar dinero en una cuenta, otra para retirar dinero de una cuenta y una tercera para solicitar el saldo actual de la cuenta.

Así como no podemos conducir un dibujo de ingeniería de un auto, tampoco podemos “conducir” una clase. De la misma forma que alguien tiene que construir un auto a partir de sus dibujos de ingeniería para poder conducirlo, también debemos construir un objeto de una clase para poder hacer que un programa realice las tareas que la clase le describe cómo realizar. Ésta es una de las razones por las cuales C++ se conoce como un lenguaje de programación orientado a objetos. Observe además que, así como se pueden construir *muchos* autos a partir del mismo dibujo de ingeniería, se pueden construir *muchos* objetos a partir de la misma clase.

Cuando usted conduce un auto, si oprime el pedal del acelerador se envía un mensaje al auto para que realice una tarea: hacer que el auto vaya más rápido. De manera similar, se envían **mensajes** a un objeto; cada mensaje se conoce como la **llamada a una función miembro**, e indica a una función miembro del objeto que realice su tarea. A esto se le conoce comúnmente como **solicitar un servicio de un objeto**.

Hasta ahora, hemos utilizado la analogía del auto para introducir las clases, los objetos y las funciones miembro. Además de las capacidades con las que cuenta un auto, también tiene muchos atributos como su color, el número de puertas, la cantidad de gasolina en su tanque, su velocidad actual y el total de kilómetros recorridos (es decir, la lectura de su odómetro). Al igual que las capacidades del auto, estos atributos se representan como parte del diseño del auto en sus diagramas de ingeniería. Cuando usted conduce un auto, estos atributos siempre están asociados a él. Cada auto mantiene sus propios atributos. Por ejemplo, cada auto sabe cuánta gasolina tiene en su propio tanque, pero no cuánta hay en los tanques de otros autos. De manera similar, un objeto tiene atributos que lleva consigo cuando se utiliza en un programa. Estos atributos se especifican como parte de la clase del objeto. Por ejemplo, un objeto tipo cuenta bancaria tiene un atributo llamado saldo, el cual representa la cantidad de dinero en la cuenta. Cada objeto tipo cuenta bancaria conoce el saldo de la cuenta que representa, pero no los saldos de las otras cuentas en el banco. Los atributos se especifican mediante los miembros de datos de la clase.

3.3 Generalidades acerca de los ejemplos del capítulo

El resto de este capítulo presenta siete ejemplos simples que demuestran los conceptos que presentamos aquí, dentro del contexto de la analogía del auto. Estos ejemplos, que se sintetizan a continuación, se encargan de construir en forma incremental una clase llamada **LibroCalificaciones** para demostrar estos conceptos:

1. El primer ejemplo presenta una clase llamada **LibroCalificaciones**, con una función miembro que simplemente muestra un mensaje de bienvenida cuando se le llama. Después le mostraremos cómo crear un objeto de esa clase y cómo llamarlo, para que muestre el mensaje de bienvenida.
2. El segundo ejemplo modifica el primero, al permitir que la función miembro reciba el nombre de un curso como argumento. Después, la función miembro muestra ese nombre como parte del mensaje de bienvenida.
3. El tercer ejemplo muestra cómo almacenar el nombre del curso en un objeto tipo **LibroCalificaciones**. Para esta versión de la clase, también le mostraremos cómo utilizar las funciones miembro para establecer el nombre del curso y obtener este nombre del objeto.
4. El cuarto ejemplo demuestra cómo pueden inicializarse los datos en un objeto tipo **LibroCalificaciones**, a la hora de crear el objeto; la inicialización se lleva a cabo mediante una función miembro especial, conocida como el constructor de la clase. Este ejemplo también demuestra que cada objeto **LibroCalificaciones** mantiene su propio miembro de datos que representa el nombre del curso.
5. El quinto ejemplo modifica el cuarto, al demostrar cómo colocar la clase **LibroCalificaciones** en un archivo separado para habilitar la reutilización de software.
6. El sexto ejemplo modifica el quinto, al demostrar el principio de la buena ingeniería de software, en el cual la interfaz de la clase se separa de su implementación. Esto hace que la clase sea más fácil de modificar, sin afectar a los **clientes de los objetos de la clase**; es decir, cualquier clase o función que llame a las funciones miembro de los objetos de la clase desde el exterior de los objetos.

7. El último ejemplo mejora la clase `LibroCalificaciones` al introducir la validación de datos, la cual asegura que los datos en un objeto se adhieran a un formato particular, o se encuentren en un rango de valores adecuado. Por ejemplo, un objeto `Fecha` requeriría un valor para el mes en el rango de 1 a 12. En este ejemplo de `LibroCalificaciones`, la función miembro que establece el nombre del curso para un objeto `LibroCalificaciones` asegura que sólo se utilicen los primeros 25 caracteres del nombre del curso y se muestra un mensaje de advertencia.

Observe que los ejemplos sobre `LibroCalificaciones` en este capítulo en realidad no procesan o almacenan calificaciones. Empezaremos a procesar las calificaciones con la clase `LibroCalificaciones` en el capítulo 4, y almacenaremos calificaciones en un objeto `LibroCalificaciones` en el capítulo 7, Arreglos y vectores.

3.4 Definición de una clase con una función miembro

Vamos a empezar con un ejemplo (figura 3.1) que consiste en la clase `LibroCalificaciones` (líneas 9 a 17), la cual representa un libro de calificaciones que un instructor puede utilizar para mantener las calificaciones de los exámenes de sus estudiantes, y una función `main` (líneas 20 a 25) que crea un objeto `LibroCalificaciones`. La función `main` utiliza este objeto y su función miembro para mostrar un mensaje en la pantalla, para dar la bienvenida al instructor al programa del libro de calificaciones.

Primero vamos a describir cómo definir una clase y una función miembro. Después explicaremos cómo se crea un objeto, y cómo llamar a una función miembro de éste. Los primeros ejemplos contienen la función `main` y la clase `LibroCalificaciones` que utiliza en el mismo archivo. Más adelante en el capítulo, presentaremos maneras más sofisticadas de estructurar nuestros programas para lograr una mejor ingeniería de software.

```

1 // Fig. 3.1: fig03_01.cpp
2 // Define la clase LibroCalificaciones con una función miembro llamada mostrarMensaje;
3 // Crea un objeto LibroCalificaciones y llama a su función mostrarMensaje.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 // Definición de la clase LibroCalificaciones
9 class LibroCalificaciones
10 {
11 public:
12     // función que muestra un mensaje de bienvenida para el usuario de LibroCalificaciones
13     void mostrarMensaje()
14     {
15         cout << "Bienvenido al Libro de calificaciones!" << endl;
16     } // fin de la función mostrarMensaje
17 }; // fin de la clase LibroCalificaciones
18
19 // la función main empieza la ejecución del programa
20 int main()
21 {
22     LibroCalificaciones miLibroCalificaciones; // crea un objeto LibroCalificaciones
23     miLibroCalificaciones.mostrarMensaje(); // llama a la función mostrarMensaje del objeto
24     return 0; // indica que terminó correctamente
25 } // fin de main

```

Bienvenido al Libro de calificaciones!

Figura 3.1 | Define la clase `LibroCalificaciones` con una función miembro llamada `mostrarMensaje`, crea un objeto `LibroCalificaciones` y llama a su función `mostrarMensaje`.

La clase `LibroCalificaciones`

Antes de que la función `main` (líneas 20 a 25) pueda crear un objeto de la clase `LibroCalificaciones`, debemos indicar al compilador qué funciones miembro y cuáles miembros de datos pertenecen a la clase. A esto se le conoce como **definir una clase**. La **definición de la clase `LibroCalificaciones`** (líneas 9 a 17) contiene una función miembro llamada `mostrarMensaje` (líneas 13 a 16), la cual muestra un mensaje en la pantalla (línea 15). Recuerde que una clase es como

un plano de construcción; por lo tanto, necesitamos crear un objeto de la clase `LibroCalificaciones` (línea 22) y llamar a su función miembro `mostrarMensaje` (línea 23) para hacer que se ejecute la línea 15 y se muestre el mensaje de bienvenida. Pronto explicaremos las líneas 22 y 23 con detalle.

La definición de la clase empieza en la línea 9 con la palabra clave `class`, seguida del nombre de la clase `LibroCalificaciones`. Por convención, el nombre de una clase definida por el usuario empieza con una letra mayúscula, y por legibilidad, cada palabra subsiguiente en el nombre de la clase empieza con una letra mayúscula. Este estilo de capitalización se conoce comúnmente como **nomenclatura de camello**, debido a que el patrón de letras mayúsculas y minúsculas se asemeja a la silueta de un camello.

El **cuerpo** de cada clase va encerrado entre un par de llaves izquierda y derecha (`{` y `}`), como en las líneas 10 y 17. La definición de la clase termina con un punto y coma (línea 17).



Error común de programación 3.1

Olvidar el punto y coma al final de la definición de una clase es un error de sintaxis.

Recuerde que la función `main` siempre se llama de manera automática cuando ejecutamos un programa. La mayoría de las funciones no se llaman de manera automática. Como pronto veremos, debemos llamar a la función miembro `mostrarMensaje` de manera explícita para indicarle que debe realizar su tarea.

La línea 11 contiene la etiqueta del **especificador de acceso `public`**: La palabra clave `public` es un **especificador de acceso**. En las líneas 13 a 16 se define la función miembro `mostrarMensaje`. Esta función miembro aparece después del especificador de acceso `public`: para indicar que la función está “disponible para el público”; es decir, otras funciones en el programa (como `main`) la pueden llamar, y también las funciones miembro de otras clases (si las hay). Los especificadores de acceso siempre van seguidos de un signo de dos puntos (`:`). En el resto del libro, cuando hagamos referencia al especificador de acceso `public`, omitiremos el punto y coma como en esta oración. En la sección 3.6 presentaremos un segundo especificador de acceso, `private`.

Cada función en un programa realiza una tarea y puede devolver un valor cuando complete su tarea; por ejemplo, una función podría realizar un cálculo y después devolver el resultado del mismo. Al definir una función, debemos especificar un **tipo de valor de retorno** para indicar el tipo de valor que devuelve la función cuando completa su tarea. En la línea 13, la palabra clave `void` a la izquierda del nombre de la función `mostrarMensaje` es el tipo de valor de retorno de ésta. El tipo de valor de retorno `void` indica que `mostrarMensaje` no devolverá (regresará) datos a la **función que la llamó** (en este ejemplo, `main`, como veremos en unos momentos) cuando complete su tarea. En la figura 3.5 veremos un ejemplo de una función que devuelve un valor.

El nombre de la función miembro, `mostrarMensaje`, va después del tipo de valor de retorno. Por convención, los nombres de las funciones empiezan con la primera letra en minúscula, y todas las palabras subsiguientes en el nombre empiezan con letra mayúscula. Los paréntesis después del nombre de la función miembro indican que ésta es una función. Un conjunto vacío de paréntesis, como se muestra en la línea 13, indica que esta función miembro no requiere datos adicionales para realizar su tarea. En la sección 3.5 veremos un ejemplo de una función miembro que requiere datos adicionales. La línea 13 se conoce comúnmente como el **encabezado de la función**. El cuerpo de todas las funciones está delimitado por las llaves izquierda y derecha (`{` y `}`), como en las líneas 14 y 16.

El cuerpo de una función contiene instrucciones que realizan la tarea de la función. En este caso, la función miembro `mostrarMensaje` contiene una instrucción (línea 15) que muestra el mensaje “`Bienvenido al Libro de calificaciones!`”. Una vez que se ejecuta esta instrucción, la función ha completado su trabajo.



Error común de programación 3.2

Devolver un valor de una función cuyo tipo de valor de retorno se ha declarado como `void` es un error de compilación.



Error común de programación 3.3

Definir una función dentro de otra función es un error de sintaxis.

Prueba de la clase `LibroCalificaciones`

A continuación nos gustaría utilizar la clase `LibroCalificaciones` en un programa. Como aprendió en el capítulo 2, la función `main` (líneas 20 a 25) empieza la ejecución de todos los programas.

En este programa queremos llamar a la función miembro `mostrarMensaje` de la clase `LibroCalificaciones` para mostrar el mensaje de bienvenida. Por lo general, no podemos llamar a una función miembro de una clase, sino hasta

crear un objeto de esa clase. (Como veremos en la sección 10.7, las funciones miembro `static` son una excepción.) En la línea 22 se crea un objeto de la clase `LibroCalificaciones`, llamado `miLibroCalificaciones`. Observe que el tipo de la variable es `LibroCalificaciones`; la clase que definimos en las líneas 9 a 17. Cuando declaramos variables de tipo `int`, como en el capítulo 2, el compilador sabe lo que es `int`: un tipo fundamental. Sin embargo, en la línea 22 el compilador no sabe automáticamente qué tipo corresponde a `LibroCalificaciones`: es un **tipo definido por el usuario**. Para indicar al compilador qué es `LibroCalificaciones`, incluimos la definición de la clase (líneas 9 a 17). Si omitiéramos estas líneas, el compilador generaría un mensaje de error (como “`'LibroCalificaciones': undeclared identifier`” en Microsoft Visual C++, o “`'LibroCalificaciones': undeclared`” en GNU C++). Cada nueva clase que creamos se convierte en un nuevo tipo, que puede usarse para crear objetos. Los programadores pueden definir nuevos tipos de clases según lo necesiten; ésta es una razón por la cual C++ se conoce como un **lenguaje extensible**.

En la línea 23 se hace una llamada a la función miembro `mostrarMensaje` (definida en las líneas 13 a 16), usando la variable `miLibroCalificaciones` seguida del **operador punto** (`.`), el nombre de la función `mostrarMensaje` y un conjunto vacío de paréntesis. Esta llamada hace que la función `mostrarMensaje` realice su tarea. Al principio de la línea 23, “`miLibroCalificaciones.`” indica que `main` debe usar el objeto `LibroCalificaciones` que se creó en la línea 22. Los paréntesis vacíos en la línea 13 indican que la función miembro `mostrarMensaje` no requiere datos adicionales para realizar su tarea. (En la sección 3.5 veremos cómo pasar datos a una función.) Cuando `mostrarMensaje` completa su tarea, la función `main` continúa su ejecución en la línea 24, la cual indica que `main` realizó sus tareas con éxito. Éste es el fin de `main`, por lo que el programa termina.

Diagrama de clases de UML para la clase LibroCalificaciones

En la sección 1.21 vimos que UML es un lenguaje gráfico estandarizado, utilizado por los programadores para representar sistemas orientados a objetos. En UML, cada clase se modela en un **diagrama de clases de UML** en forma de un rectángulo con tres compartimientos. La figura 3.2 presenta un diagrama de clases para la clase `LibroCalificaciones` (figura 3.1). El compartimiento superior contiene el nombre de la clase, centrado en forma horizontal y en negrita. El compartimiento de en medio contiene los atributos de la clase, que en C++ corresponden a los miembros de datos. En estos momentos el compartimiento está vacío, ya que la clase `LibroCalificaciones` no tiene atributos. (En la sección 3.6 presentaremos una versión de la clase `LibroCalificaciones` con un atributo.) El compartimiento inferior contiene las operaciones de la clase, que en C++ corresponden a las funciones miembro. Para modelar las operaciones, UML lista el nombre de la operación seguido de un conjunto de paréntesis. La clase `LibroCalificaciones` tiene una sola función miembro llamada `mostrarMensaje`, por lo que el compartimiento inferior de la figura 3.2 lista una operación con este nombre. La función miembro `mostrarMensaje` no requiere información adicional para realizar sus tareas, por lo que los paréntesis que van después de `mostrarMensaje` en el diagrama de clases están vacíos, de igual forma que como aparecieron en el encabezado de la función miembro, en la línea 13 de la figura 3.1. El signo más (+) que va antes del nombre de la operación indica que `mostrarMensaje` es una operación `public` en UML (es decir, una función miembro `public` en C++).



Figura 3.2 | Diagrama de clases de UML, el cual indica que la clase `LibroCalificaciones` tiene una operación `mostrarMensaje` pública.

3.5 Definición de una función miembro con un parámetro

En nuestra analogía del auto de la sección 3.2, hablamos sobre el hecho de que al oprimir el pedal del acelerador se envía un mensaje al auto para que realice una tarea: hacer que vaya más rápido. Pero ¿qué tan rápido debería acelerar el auto? Como sabe, entre más oprima el pedal, mayor será la aceleración del auto. Por lo tanto, el mensaje para el auto en realidad incluye tanto la tarea a realizar como información adicional que ayuda al auto a realizar su tarea. A la información adicional se le conoce como **parámetro**; el valor del parámetro ayuda al auto a determinar qué tan rápido debe acelerar. De manera similar, una función miembro puede requerir uno o más parámetros que representan la información adicional que necesita para realizar su tarea. La llamada a una función proporciona valores (llamados **argumentos**) para cada uno de los parámetros de esa función. Por ejemplo, para realizar un depósito en una cuenta bancaria, suponga que una función miembro llamada `depositar` de una clase `Cuenta` especifica un parámetro que representa el monto a depositar.

Cuando se hace una llamada a la función miembro `deposito`, se copia al parámetro de la función miembro un valor como argumento, que representa el monto a depositar. Después, la función miembro suma esa cantidad al saldo de la cuenta.

Definición y prueba de la clase `LibroCalificaciones`

Nuestro siguiente ejemplo (figura 3.3) redefine la clase `LibroCalificaciones` (líneas 14 a 23) con una función miembro `mostrarMensaje` (líneas 18 a 22) que muestra el nombre del curso como parte del mensaje de bienvenida. La nueva versión de `mostrarMensaje` requiere un parámetro (`nombreCurso` en la línea 18) que representa el nombre del curso a imprimir en pantalla.

```
1 // Fig. 3.3: fig03_03.cpp
2 // Define la clase LibroCalificaciones con una función miembro que recibe un parámetro;
3 // Crea un objeto LibroCalificaciones y llama a su función mostrarMensaje.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include <string> // el programa usa la clase string estándar de C++
10 using std::string;
11 using std::getline;
12
13 // definición de la clase LibroCalificaciones
14 class LibroCalificaciones
15 {
16 public:
17     // función que muestra un mensaje de bienvenida para el usuario de LibroCalificaciones
18     void mostrarMensaje( string nombreCurso )
19     {
20         cout << "Bienvenido al libro de calificaciones para\n" << nombreCurso << "!"
21             << endl;
22     } // fin de la función mostrarMensaje
23 }; // fin de la clase LibroCalificaciones
24
25 // la función main empieza la ejecución del programa
26 int main()
27 {
28     string nombreDelCurso; // cadena de caracteres que almacena el nombre del curso
29     LibroCalificaciones miLibroCalificaciones; // crea un objeto LibroCalificaciones llamado mi
30     LibroCalificaciones
31
32     // pide y recibe el nombre del curso como entrada
33     cout << "Escriba el nombre del curso:" << endl;
34     getline( cin, nombreDelCurso ); // lee el nombre de un curso con espacios en blanco
35     cout << endl; // imprime una línea en blanco
36
37     // llama a la función mostrarMensaje de miLibroCalificaciones
38     // y pasa nombreDelCurso como argumento
39     miLibroCalificaciones.mostrarMensaje( nombreDelCurso );
40     return 0; // indica que terminó correctamente
41 } // fin de main
```

Escriba el nombre del curso:
CS101 Introducción a la programación en C++

Bienvenido al libro de calificaciones para
CS101 Introducción a la programación en C++!

Figura 3.3 | Define la clase `LibroCalificaciones` con una función miembro que recibe un parámetro, crea un objeto `LibroCalificaciones` y llama a su función `mostrarMensaje`.

Antes de hablar sobre las nuevas características de la clase `LibroCalificaciones`, veamos cómo se utiliza la nueva clase en `main` (líneas 26 a 40). En la línea 28 se crea una variable de tipo `string` llamada `nombreDelCurso`, la cual se utilizará para almacenar el nombre del curso que escriba el usuario. Una variable de tipo `string` representa una cadena de caracteres como "CS101 Introducción a la programación en C++". En realidad, una cadena es un objeto de la clase `string`, de la Biblioteca estándar de C++. Esta clase se define en el archivo de encabezado `<string>`, y el nombre `string` (al igual que `cout`) pertenece al espacio de nombres `std`. Para que la línea 28 se pueda compilar, en la línea 9 se incluye el archivo de encabezado `<string>`. Observe que la declaración `using` en la línea 10 nos permite escribir simplemente `string` en la línea 28, en lugar de `std::string`. Por ahora, puede considerar a las variables `string` como las variables de otros tipos tales como `int`. En la sección 3.10 aprenderá acerca de las herramientas adicionales de `string`.

En la línea 29 se crea un objeto de la clase `LibroCalificaciones`, llamado `miLibroCalificaciones`. En la línea 32 se pide al usuario que escriba el nombre de un curso. En la línea 33 se lee el nombre del usuario y se asigna a la variable `nombreDelCurso`, usando la función de biblioteca `getline` para llevar a cabo la entrada. Antes de explicar esta línea de código, veamos por qué no podemos simplemente escribir

```
cin >> nombreDelCurso;
```

para obtener el nombre del curso. En la ejecución de ejemplo de nuestro programa, utilizamos el nombre "CS101 Introducción a la programación en C++", la cual contiene varias palabras. (Recuerde que resaltamos la entrada que suministra el usuario en negrita.) Cuando se utiliza `cin` con el operador de extracción de flujo, lee caracteres hasta que se llega al primer carácter de espacio en blanco. Por ende, la instrucción anterior sólo leería "CS101". El resto del nombre del curso tendría que leerse mediante operaciones de entrada subsiguientes.

En este ejemplo, nos gustaría que el usuario escribiera el nombre completo del curso y que oprimiera *Intro* para enviarlo al programa, y nos gustaría almacenar el nombre completo del curso en la variable `string` llamada `nombreDelCurso`. La llamada a la función `getline(cin, nombreDelCurso)` en la línea 33 lee caracteres (incluyendo los caracteres de espacio que separan las palabras en la entrada) del objeto flujo de entrada estándar `cin` (es decir, el teclado) hasta encontrar el carácter de nueva línea, coloca los caracteres en la variable `string` llamada `nombreDelCurso` y descarta el carácter de nueva línea. Observe que, al oprimir *Intro* mientras se escribe la entrada del programa, se inserta una nueva línea en el flujo de entrada. Observe además que debe incluirse el archivo de encabezado `<string>` en el programa para usar la función `getline`, y que el nombre `getline` pertenece al espacio de nombres `std`.

En la línea 38 se hace una llamada a la función miembro `mostrarMensaje` de `miLibroCalificaciones`. La variable `nombreDelCurso` entre paréntesis es el argumento que se pasa a la función miembro `mostrarMensaje` para que pueda realizar su tarea. El valor de la variable `nombreDelCurso` en `main` se convierte en el valor del parámetro `nombreCurso` de la función miembro `mostrarMensaje` en la línea 18. Al ejecutar este programa, observe que la función miembro `mostrarMensaje` imprime, como parte del mensaje de bienvenida, el nombre del curso que usted escriba (en nuestra ejecución de ejemplo, CS101 Introducción a la programación en C++).

Más sobre los argumentos y los parámetros

Para especificar que una función requiere datos para realizar su tarea, hay que colocar información adicional en la lista de parámetros de la función, la cual se encuentra en los paréntesis que van después del nombre de la función. La lista de parámetros puede contener cualquier número de parámetros, incluso ninguno (lo que se representa mediante los paréntesis vacíos, como en la línea 13 de la figura 3.1), para indicar que una función no requiere parámetros. La lista de parámetros de la función miembro `mostrarMensaje` (figura 3.3, línea 18) declara que la función requiere un parámetro. Cada parámetro debe especificar un tipo y un identificador. En este caso, el tipo `string` y el identificador `nombreCurso` indican que la función miembro `mostrarMensaje` requiere un objeto `string` para realizar su tarea. El cuerpo de la función miembro utiliza el parámetro `nombreDelCurso` para acceder al valor que se pasa a la función en la llamada (línea 38 en `main`). En las líneas 20 y 21 se muestra el valor del parámetro `nombreDelCurso` como parte del mensaje de bienvenida. Observe que el nombre de la variable de parámetro (línea 18) puede ser igual o distinto al nombre de la variable de argumento (línea 38); en el capítulo 6, Funciones y una introducción a la recursividad, aprenderá por qué.

Una función puede especificar múltiples parámetros; sólo hay que separar un parámetro de otro mediante una coma (en las figuras 6.4 y 6.5 veremos un ejemplo de esto). El número y el orden de los argumentos en la llamada a una función deben coincidir con el número y orden de los parámetros en la lista de parámetros del encabezado de la función miembro que se llamó. Además, los tipos de los argumentos en la llamada a la función deben ser consistentes con los tipos de los parámetros correspondientes en el encabezado de la función. (Como veremos en capítulos posteriores, no siempre se requiere que el tipo de un argumento y el tipo de su correspondiente parámetro sean idénticos, pero deben ser "consistentes"). En nuestro ejemplo, el único argumento `string` en la llamada a la función (es decir, `nombreDelCurso`) coincide exactamente con el único parámetro `string` en la definición de la función miembro (es decir, `nombreCurso`).



Error común de programación 3.4

Colocar un punto y coma después del paréntesis derecho que encierra a la lista de parámetros de la definición de una función es un error de sintaxis.



Error común de programación 3.5

Definir el parámetro de una función de nuevo como una variable local en la función es un error de compilación.



Buena práctica de programación 3.1

Para evitar ambigüedades, no utilice los mismos nombres para los argumentos que se pasan a una función y los parámetros correspondientes en la definición de la función.



Buena práctica de programación 3.2

Elegir nombres significativos para una función, y nombres significativos para sus parámetros, mejora la legibilidad de los programas y ayuda a evitar un uso excesivo de los comentarios.

Diagrama de clases de UML actualizado para la clase LibroCalificaciones

El diagrama de clases de UML de la figura 3.4 modela la clase `LibroCalificaciones` de la figura 3.3. Al igual que la figura 3.1, esta clase `LibroCalificaciones` contiene la función miembro `public` llamada `mostrarMensaje`. Sin embargo, esta versión de `mostrarMensaje` tiene un parámetro. Para modelar un parámetro, UML lista el nombre del parámetro, seguido de dos puntos y del tipo del parámetro entre paréntesis, después del nombre de la operación. UML tiene sus propios tipos de datos similares a los de C++. UML es independiente del lenguaje (se utiliza con muchos lenguajes de programación distintos), por lo que su terminología no coincide exactamente con la de C++. Por ejemplo, el tipo `String` de UML corresponde al tipo `string` de C++. La función miembro `mostrarMensaje` de la clase `LibroCalificaciones` (figura 3.3, líneas 18 a 22) tiene un parámetro `string` llamado `nombreCurso`, por lo que en la figura 3.4 se lista a `nombreCurso : String` entre los paréntesis que van después del nombre de la operación `mostrarMensaje`. Observe que esta versión de la clase `LibroCalificaciones` aún no tiene miembros de datos.

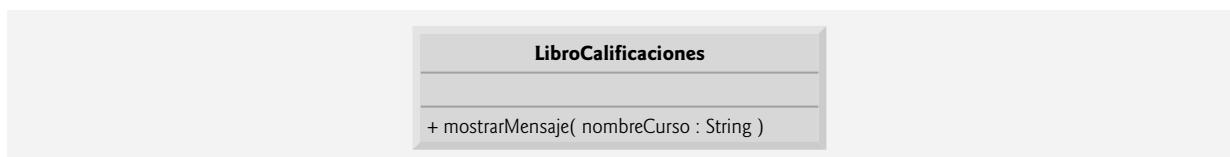


Figura 3.4 | Diagrama de clases de UML, que indica que la clase `LibroCalificaciones` tiene una operación llamada `mostrarMensaje`, con un parámetro llamado `nombreCurso` de tipo `String` de UML.

3.6 Miembros de datos, funciones establecer y funciones obtener

En el capítulo 2 declaramos todas las variables de un programa en su función `main`. Las variables que se declaran en el cuerpo de la definición de una función se conocen como **variables locales**, y sólo se pueden utilizar desde la línea de su declaración en la función, hasta la llave derecha de cierre (`}`) que le corresponda. Una variable local se debe declarar antes de poder utilizarla en una función. No se puede acceder a una variable local fuera de la función en la que está declarada. Cuando una función termina, se pierden los valores de sus variables locales. (En el capítulo 6 veremos una excepción a esto, cuando hablemos sobre las variables locales `static`). En la sección 3.2 vimos que un objeto tiene atributos que lleva con él, a medida que se utiliza en un programa. Dichos atributos existen durante toda la vida del objeto.

Por lo general, una clase consiste en una o más funciones miembro que manipulan los atributos pertenecientes a un objeto específico de la clase. Los atributos se representan como variables en la definición de una clase. Dichas variables se llaman **miembros de datos** y se declaran dentro de la definición de una clase, pero fuera de los cuerpos de las definiciones de las funciones miembro de la clase. Cada objeto de una clase mantiene su propia copia de sus atributos en memoria. El ejemplo en esta sección demuestra una clase `LibroCalificaciones`, que contiene un miembro de datos llamado `nombreCurso` para representar el nombre del curso de un objeto `LibroCalificaciones` específico.

La clase LibroCalificaciones con miembro de datos, una función establecer y una función obtener

En nuestro siguiente ejemplo, la clase `LibroCalificaciones` (figura 3.5) mantiene el nombre del curso como un miembro de datos, para que pueda usarse o modificarse en cualquier momento, durante la ejecución de un programa. Esta clase contiene las funciones miembro `establecerNombreCurso`, `obtenerNombreCurso` y `mostrarMensaje`. La función miembro `establecerNombreCurso` almacena el nombre de un curso en un miembro de datos de `LibroCalificaciones`. La función miembro `obtenerNombreCurso` obtiene el nombre del curso de ese miembro de datos. La función miembro `mostrarMensaje`, que en este caso no especifica parámetros, sigue mostrando un mensaje de bienvenida que incluye el nombre del curso. Pero como veremos más adelante, la función ahora obtiene el nombre del curso mediante una llamada a otra función en la misma clase: `obtenerNombreCurso`.

```

1 // Fig. 3.5: fig03_05.cpp
2 // Define la clase LibroCalificaciones que contiene un miembro de datos
3 // nombreCurso y funciones miembro para establecer y obtener su valor;
4 // Crea y manipula un objeto LibroCalificaciones.
5 #include <iostream>
6 using std::cout;
7 using std::cin;
8 using std::endl;
9
10 #include <string> // el programa usa la clase string estándar de C++
11 using std::string;
12 using std::getline;
13
14 // definición de la clase LibroCalificaciones
15 class LibroCalificaciones
16 {
17 public:
18     // función que establece el nombre del curso
19     void establecerNombreCurso( string nombre )
20     {
21         nombreCurso = nombre; // almacena el nombre del curso en el objeto
22     } // fin de la función establecerNombreCurso
23
24     // función que obtiene el nombre del curso
25     string obtenerNombreCurso()
26     {
27         return nombreCurso; // devuelve el nombreCurso del objeto
28     } // fin de la función obtenerNombreCurso
29
30     // función que muestra un mensaje de bienvenida
31     void mostrarMensaje()
32     {
33         // esta instrucción llama a obtenerNombreCurso para obtener el
34         // nombre del curso que representa este LibroCalificaciones
35         cout << "Bienvenido al libro de calificaciones para\n" << obtenerNombreCurso() << "!"
36         << endl;
37     } // fin de la función mostrarMensaje
38 private:
39     string nombreCurso; // nombre del curso para este LibroCalificaciones
40 }; // fin de la clase LibroCalificaciones
41
42 // la función main empieza la ejecución del programa
43 int main()
44 {
45     string nombreDelCurso; // cadena de caracteres para almacenar el nombre del curso
46     LibroCalificaciones miLibroCalificaciones; // crea un objeto LibroCalificaciones llamado
        miLibroCalificaciones

```

Figura 3.5 | Definición y prueba de la clase `LibroCalificaciones` con un miembro de datos y funciones *establecer* y *obtener*. (Parte I de 2).

```

47 // muestra el valor inicial de nombreCurso
48 cout << "El nombre inicial del curso es: " << miLibroCalificaciones.obtenerNombreCurso()
49     << endl;
50
51
52 // pide, recibe y establece el nombre del curso
53 cout << "\nEscriba el nombre del curso:" << endl;
54 getline( cin, nombreDelCurso ); // lee el nombre de un curso con espacios en blanco
55 miLibroCalificaciones.establecerNombreCurso( nombreDelCurso ); // establece el nombre del
56 curso
57 cout << endl; // imprime una línea en blanco
58 miLibroCalificaciones.mostrarMensaje(); // muestra un mensaje con el nuevo nombre del curso
59 return 0; // indica que terminó correctamente
60 } // fin de main

```

El nombre inicial del curso es:

Escriba el nombre del curso:

CS101 Introducción a la programación en C++

Bienvenido al libro de calificaciones para
CS101 Introducción a la programación en C++!

Figura 3.5 | Definición y prueba de la clase LibroCalificaciones con un miembro de datos y funciones establecer y obtener. (Parte 2 de 2).



Buena práctica de programación 3.3

Coloque una línea en blanco entre las definiciones de las funciones miembro, para mejorar la legibilidad del programa.

Un instructor típico enseña más de un curso, cada uno con su propio nombre. En la línea 39 se declara que `nombreCurso` es una variable de tipo `string`. Como la variable se declara en la definición de la clase (líneas 15 a 40) pero fuera de los cuerpos de las definiciones de las funciones miembro de ésta (líneas 19 a 22, 25 a 28 y 31 a 37), la variable es un miembro de datos. Cada instancia (es decir, objeto) de la clase `LibroCalificaciones` contiene una copia de cada uno de los miembros de datos de la clase; si hay dos objetos `LibroCalificaciones`, cada objeto tiene su propia copia de `nombreCurso` (una por cada objeto), como veremos en el ejemplo de la figura 3.7. Un beneficio de hacer de `nombreCurso` un miembro de datos es que todas las funciones miembro de la clase (en este caso, `LibroCalificaciones`) pueden manipular cualquier miembro de datos que aparezca en la definición de la clase (en este caso, `nombreCurso`).

Los especificadores de acceso `public` y `private`

La mayoría de las declaraciones de miembros de datos aparecen después de la etiqueta del especificador de accesos `private`: (línea 38). Al igual que `public`, la palabra clave `private` es un especificador de acceso. Las variables o funciones declaradas después del especificador de acceso `private` (y antes del siguiente especificador de acceso) son accesibles sólo para las funciones miembro de la clase en la que se declaran. Así, el miembro de datos `nombreCurso` sólo puede utilizarse en las funciones miembro `establecerNombreCurso`, `obtenerNombreCurso` y `mostrarMensaje` de (cada objeto de) la clase `LibroCalificaciones`. Como el miembro de datos `nombreCurso` es `private`, no lo pueden utilizar las funciones externas a la clase (como `main`), ni las funciones miembro de otras clases en el programa. Si tratáramos de tener acceso al miembro de datos `nombreCurso` en una de estas ubicaciones del programa con una expresión tal como `miLibroCalificaciones.nombreCurso`, se produciría un error de compilación con un mensaje similar a:

cannot access private member declared in class 'LibroCalificaciones'



Observación de Ingeniería de Software 3.1

Como regla empírica, los miembros de datos deben declararse como `private` y las funciones miembro deben declararse como `public`. (Más adelante veremos que es apropiado declarar ciertas funciones miembro como `private`, si sólo van a estar accesibles para otras funciones miembro de la clase).



Error común de programación 3.6

Si una función que no es miembro de una clase específica (o amiga de esa clase, como veremos en el capítulo 10, Clases: un análisis más detallado, parte 2) intenta acceder a un miembro `private` de esa clase, se produce un error de compilación.

El acceso predeterminado para los miembros de datos es `private`, de manera que todos los miembros después del encabezado de la clase y antes del primer especificador de acceso son `private`. Los especificadores de acceso `public` y `private` pueden repetirse, pero esto es innecesario y puede ser confuso.



Buena práctica de programación 3.4

A pesar del hecho de que los especificadores de acceso `public` y `private` pueden repetirse y entremezclarse, debemos listar todos los miembros `public` de una clase primero en un grupo, y después listar todos los miembros `private` en otro grupo. Esto enfoca la atención del cliente en la interfaz `public` de la clase, en lugar de hacerlo en su implementación.



Buena práctica de programación 3.5

Si elige listar los miembros `private` primero en la definición de una clase, utilice explícitamente el especificador de acceso `private` a pesar del hecho de que se asume este especificador de manera predeterminada. Esto aumenta la claridad del programa.

El proceso de declarar miembros de datos con el modificador de acceso `private` se conoce como **ocultamiento de datos**. Cuando un programa crea (instancia) un objeto de la clase `LibroCalificaciones`, el miembro de datos `nombreCurso` se encapsula (oculta) en el objeto, y sólo está accesible para las funciones miembro de la clase de ese objeto. En la clase `LibroCalificaciones`, las funciones miembro `establecerNombreCurso` y `obtenerNombreCurso` manipulan al miembro de datos `nombreCurso` directamente (y `mostrarMensaje` podría hacerlo también, si fuera necesario).



Observación de Ingeniería de Software 3.2

En el capítulo 10 aprenderá que las funciones y las clases declaradas por una clase como amigas (friend) pueden acceder a los miembros `private` de la clase.



Tip para prevenir errores 3.1

Hacer que los miembros de datos de una clase sean `private`, y que las funciones miembro de la clase sean `public` facilita la depuración, ya que los problemas con las manipulaciones de datos se localizan, ya sea con las funciones miembro de la clase, o con los amigos (friend) de esa clase.

Las funciones miembro `establecerNombreCurso` y `obtenerNombreCurso`

La función miembro `obtenerNombreCurso` (definida en las líneas 19 a 22) no devuelve datos cuando completa su tarea, por lo que su tipo de valor de retorno es `void`. La función miembro recibe un parámetro (`nombre`), el cual representa el nombre del curso que recibirá como argumento (como veremos en la línea 55 de `main`). En la línea 21 asigna `nombre` al miembro de datos `nombreCurso`. En este ejemplo, `establecerNombreCurso` no trata de validar el nombre del curso; es decir, la función no verifica que el nombre del curso se apegue a cierto formato en especial, o que siga cualquier otra regla en relación con la apariencia que debe tener un nombre de curso “válido”. Por ejemplo, suponga que una universidad puede imprimir certificados de los estudiantes que contengan los nombres de cursos de sólo 25 caracteres o menos. En este caso, es conveniente que la clase `LibroCalificaciones` asegure que su miembro de datos `nombreCurso` nunca contendrá más de 25 caracteres. En la sección 3.10 hablaremos sobre las técnicas de validación básicas.

La función miembro `obtenerNombreCurso` (definida en las líneas 25 a 28) devuelve un valor de `nombreCurso` de un objeto `LibroCalificaciones` específico. La función miembro tiene una lista de parámetros vacía, por lo que no requiere información adicional para realizar su tarea. La función especifica que devuelve un objeto `string`. Cuando se hace una llamada a una función que especifica un tipo de valor de retorno distinto de `void`, y ésta completa su tarea, la función devuelve un resultado a la función que la llamó. Por ejemplo, cuando usted va a un cajero automático (ATM) y solicita el saldo de su cuenta, espera que el ATM le devuelva un valor que representa su saldo. De manera similar, cuando una instrucción llama a la función miembro `obtenerNombreCurso` en un objeto `LibroCalificaciones`, la instrucción espera recibir el nombre del curso de `LibroCalificaciones` (en este caso, un objeto `string`, como se especifica en el

tipo de valor de retorno de la función). Si tiene una función llamada `cuadrado` que devuelve el cuadrado de su argumento, es de esperarse que la instrucción

```
resultado = cuadrado( 2 );
```

devuelva 4 de la función `cuadrado` y asigne el valor 4 a la variable `resultado`. Si tiene una función llamada `maximo` que devuelve el mayor de tres argumentos enteros, es de esperarse que la siguiente instrucción

```
mayor = maximo( 27, 114, 51 );
```

devuelva 114 de la función `maximo` y asigne 114 a la variable `mayor`.

Error común de programación 3.7



Olvidar devolver un valor de una función, que se supone debe devolver un valor, es un error de compilación.

Observe que las instrucciones en las líneas 21 y 27 utilizan la variable `nombreCurso` (línea 39), aun y cuando esta variable no se declaró en ninguna de las funciones miembro. Podemos utilizar `nombreCurso` en las funciones miembro de la clase `LibroCalificaciones`, ya que `nombreCurso` es un miembro de datos de la clase. Observe además que el orden en el que se definen las funciones miembro no determina cuándo se van a llamar en tiempo de ejecución. Por lo tanto, la función miembro `obtenerNombreCurso` podría declararse antes que la función miembro `establecerNombreDelCurso`.

La función miembro mostrarMensaje

La función miembro `mostrarMensaje` (líneas 31 a 37) no devuelve datos cuando completa su tarea, por lo que su tipo de valor de retorno es `void`. Esta función no recibe parámetros, por lo que la lista de parámetros está vacía. En las líneas 35 y 36 se imprime un mensaje de bienvenida, que incluye el valor del miembro de datos `nombreCurso`. La línea 35 llama a la función miembro `establecerNombreDelCurso` a obtener el valor de `nombreCurso`. Observe que la función miembro `mostrarMensaje` también podría acceder al miembro de datos `nombreCurso` directamente, así como las funciones miembro `establecerNombreCurso` y `obtenerNombreCurso`. En breve explicaremos por qué optamos por llamar a la función miembro `obtenerNombreCurso` para obtener el valor de `nombreCurso`.

Prueba de la clase LibroCalificaciones

La función `main` (líneas 43 a 60) crea un objeto de la clase `LibroCalificaciones` y utiliza cada una de sus funciones miembro. En la línea 46 se crea un objeto `LibroCalificaciones` llamado `miLibroCalificaciones`. En las líneas 49 a 50 se muestra el nombre inicial del curso, llamando a la función miembro `obtenerNombreCurso` del objeto. Observe que la primera línea de la salida no muestra un nombre de curso ya que, al principio, el miembro de datos `nombreCurso` del objeto (es decir, un `string`) está vacío; de manera predeterminada, el valor inicial de un objeto `string` es lo que se denomina `cadena vacía` (una cadena que no contiene caracteres). No aparece nada en la pantalla cuando se muestra una cadena vacía.

En la línea 53 se pide al usuario que escriba el nombre de un curso. La variable `string` local `nombreDelCurso` (declarada en la línea 45) se inicializa con el nombre del curso que escribió el usuario, el cual se devuelve mediante la llamada a la función `getLine` (línea 54). En la línea 55 se hace una llamada a la función miembro `establecerNombreCurso` del objeto `miLibroCalificaciones` y se provee `nombreDelCurso` como argumento para la función. Cuando se hace la llamada a la función, el valor del argumento se copia al parámetro `nombre` (línea 19) de la función miembro `establecerNombreCurso` (líneas 19 a 22). Despues, el valor del parámetro se asigna al miembro de datos `nombreCurso` (línea 21). En la línea 57 se salta una línea en la salida; después en la línea 58 se hace una llamada a la función `mostrarMensaje` del objeto `miLibroCalificaciones` para mostrar en pantalla el mensaje de bienvenida, que contiene el nombre del curso.

Ingeniería de software mediante las funciones establecer y obtener

Los miembros de datos `private` de una clase pueden manipularse sólo mediante las funciones miembro de esa clase (y por los “amigos” de la clase, como veremos en el capítulo 10). Por lo tanto, un cliente de un objeto (es decir, cualquier clase o función que llame a las funciones miembro del objeto desde su exterior) llama a las funciones miembro `public` de la clase para solicitar los servicios de la clase para objetos específicos de ésta. Esto explica por qué las instrucciones en la función `main` (figura 3.5, líneas 43 a 60) llaman a las funciones miembro `establecerNombreCurso`, `obtenerNombreCurso` y `mostrarMensaje` en un objeto `LibroCalificaciones`. A menudo, las clases proporcionan funciones miembro `public` para permitir a los clientes de la clase *establecer* (es decir, asignar valores a) u *obtener* (es decir, obtener los

valores de) miembros de datos **private**. Los nombres de estas funciones miembro no necesitan empezar con **establecer** u **obtener**, pero esta convención de nomenclatura es común. En este ejemplo, la función miembro que *establece* el miembro de datos **nombreCurso** se llama **establecerNombreCurso**, y la función miembro que *obtiene* el valor del miembro de datos **nombreCurso** se llama **obtenerNombreCurso**. Observe que las funciones *establecer* se conocen también como **mutadores** (porque mutan, o modifican, valores), y que las funciones *obtener* se conocen también como **accesores** (porque acceden a los valores).

Recuerde que al declarar miembros de datos con el especificador de acceso **private** se cumple con la ocultación de datos. Al proporcionar funciones *establecer* y *obtener* **public**, permitimos que los clientes de una clase accedan a los datos ocultos, pero sólo en forma *indirecta*. El cliente sabe que está intentando modificar u obtener los datos de un objeto, pero no sabe cómo el objeto lleva a cabo estas operaciones. En algunos casos, una clase puede representar internamente una pieza de datos de cierta forma, pero puede exponer esos datos a los clientes de una forma distinta. Por ejemplo, suponga que una clase **Reloj** representa la hora del día como un miembro de datos **private int** llamado **hora**, que almacena el número de segundos transcurridos desde media noche. Sin embargo, cuando un cliente llama a la función miembro **obtenerHora** de un objeto **Reloj**, el objeto podría devolver la hora con horas, minutos y segundos en un objeto **string**, en el formato "HH:MM:SS". De manera similar, suponga que la clase **Reloj** cuenta con una función *establecer* llamada **establecerHora**, que recibe un parámetro **string** en el formato "HH:MM:SS". Mediante el uso de las herramientas de la clase **string** que presentaremos en el capítulo 18, la función **establecerHora** podría convertir este objeto **string** en un número de segundos, que la función almacena en su miembro de datos **private**. La función *establecer* también podría verificar que el valor que recibe represente una hora válida (por ejemplo, "12:30:45" es válida, pero "42:85:70" no). Las funciones *establecer* y *obtener* permiten que un cliente interactúe con un objeto, pero los datos **private** del objeto permanecen encapsulados (ocultos) de una manera segura dentro del mismo objeto.

Las funciones *establecer* y *obtener* de una clase también deben utilizarlas otras funciones miembro dentro de la clase, para manipular los datos **private** de ésta, aunque estas funciones miembro *pueden* acceder a los datos **private** directamente. En la figura 3.5, las funciones miembro **establecerNombreCurso** y **obtenerNombreCurso** son funciones miembro **public**, por lo que están accesibles para los clientes de la clase, así como para la misma clase. La función miembro **mostrarMensaje** llama a la función miembro **obtenerNombreCurso** para obtener el valor del miembro de datos **nombreCurso** y mostrarlo en pantalla, aun y cuando **mostrarMensaje** puede acceder directamente a **nombreCurso**; al acceder a un miembro de datos a través de su función *obtener* se crea una clase más robusta y mejor (es decir, una clase que sea fácil de mantener y menos propensa a dejar de trabajar). Si decidimos cambiar el miembro de datos **nombreCurso** en cierta forma, la definición de **mostrarMensaje** no requerirá modificación; sólo los cuerpos de las funciones *establecer* y *obtener*, que manipulan directamente al miembro de datos, tendrán que cambiar. Por ejemplo, suponga que decidimos representar el nombre del curso como dos miembros de datos separados: **nombreCurso** (por ejemplo, "CS101") y **tituloCurso** (por ejemplo, "Introducción a la programación en C++"). La función miembro **mostrarMensaje** puede aun emitir una sola llamada a la función miembro **obtenerNombreCurso** para obtener el nombre completo del curso y mostrarlo como parte del mensaje de bienvenida. En este caso, **obtenerNombreCurso** necesitaría crear y devolver un objeto **string** que contenga el **nombreCurso** seguido del **tituloCurso**. La función miembro **mostrarMensaje** seguiría mostrando el título completo del curso "CS101 Introducción a la programación en C++", ya que esto no se afecta debido a la modificación en los miembros de datos de la clase. Los beneficios de llamar a una función *establecer* desde otra función miembro de la clase se volverán más claros cuando hablemos sobre la validación en la sección 3.10.



Buena práctica de programación 3.6

Trate siempre de localizar los efectos de las modificaciones a los miembros de datos de una clase, utilizando y manipulando los miembros de datos a través de sus funciones *obtener* y *establecer*. Las modificaciones al nombre de un miembro de datos o al tipo de datos usado para almacenar un miembro de datos afectarían entonces sólo a las correspondientes funciones *obtener* y *establecer*, pero no a las funciones que llamaron a esas funciones.



Observación de Ingeniería de Software 3.3

Es importante escribir programas que sean comprensibles y fáciles de mantener. El cambio es la regla, en lugar de la excepción. Debemos anticipar que nuestro código será modificado en el futuro.



Observación de Ingeniería de Software 3.4

El diseñador de la clase no necesita proporcionar funciones *establecer* u *obtener* para cada elemento de datos **private**; estas herramientas se deben proporcionar sólo cuando sea apropiado. Si un servicio es útil para el código del cliente, ese servicio debe proporcionarse generalmente en la interfaz **public** de la clase.

Diagrama de clases de UML para la clase LibroCalificaciones con un miembro de datos, y métodos establecer y obtener

La figura 3.6 contiene un diagrama de clases de UML actualizado para la versión de la clase `LibroCalificaciones` de la figura 3.5. Este diagrama modela el miembro de datos `nombreCurso` de la clase `LibroCalificaciones` como un atributo en el compartimiento intermedio. UML representa a los miembros de datos como atributos, listando el nombre del atributo, seguido de dos puntos y del tipo del atributo. El tipo de UML del atributo `nombreCurso` es `String`, que corresponde al tipo `string` en C++. El miembro de datos `nombreCurso` es `private` en C++, por lo que el diagrama de clases lista un signo menos (-) en frente del nombre del atributo correspondiente. El signo menos en UML es equivalente al especificador de acceso `private` en C++. La clase `LibroCalificaciones` contiene tres funciones miembro `public`, por lo que el diagrama de clases lista tres operaciones en el tercer compartimiento. Recuerde que el signo más (+) antes de cada nombre de operación indica que ésta es `public` en C++. La operación `establecerNombreCurso` tiene un parámetro `String` llamado `nombre`. UML indica el tipo de valor de retorno de una operación colocando dos puntos y el tipo de valor de retorno después de los paréntesis que le siguen al nombre de la operación. La función miembro `obtenerNombreCurso` de la clase `LibroCalificaciones` (figura 3.5) tiene un tipo de valor de retorno `string` en C++, por lo que el diagrama de clases muestra un tipo de valor de retorno `String` en UML. Observe que las operaciones `establecerNombreCurso` y `mostrarMensaje` no devuelven valores (es decir, devuelven `void`), por lo que el diagrama de clases de UML no especifica un tipo de valor de retorno después de los paréntesis de estas operaciones. UML no utiliza a `void` en la misma forma que C++ cuando una función no devuelve un valor.

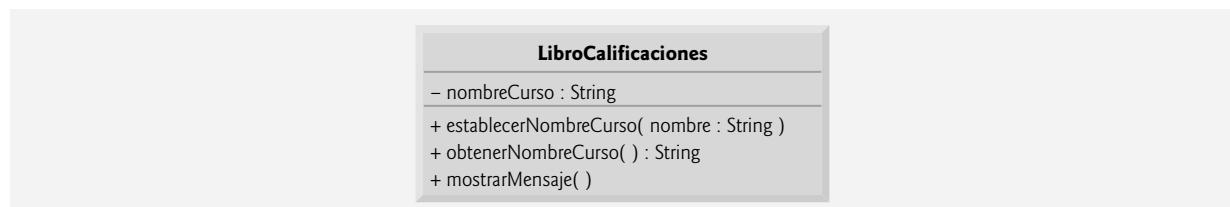


Figura 3.6 | Diagrama de clases de UML para la clase `LibroCalificaciones`, con un atributo privado `nombreCurso` y operaciones públicas `establecerNombreCurso`, `obtenerNombreCurso` y `mostrarMensaje`.

3.7 Inicialización de objetos mediante constructores

Como mencionamos en la sección 3.6, cuando se crea un objeto de la clase `LibroCalificaciones` (figura 3.5), su miembro de datos `nombreCurso` se inicializa con la cadena vacía de manera predeterminada. ¿Qué pasa si usted desea proporcionar el nombre de un curso a la hora de crear un objeto `LibroCalificaciones`? Cada clase que usted declare puede proporcionar un **constructor**, el cual puede utilizarse para inicializar un objeto de una clase al momento de crear ese objeto. Un constructor es una función miembro especial que debe definirse con el mismo nombre que el de la clase, de manera que el compilador pueda diferenciarlo de las demás funciones miembro de la clase. Una importante diferencia entre los constructores y las otras funciones es que los primeros no pueden devolver valores, por lo cual no pueden especificar un tipo de valor de retorno (ni siquiera `void`). Por lo general, los constructores se declaran como `public`. El término “constructor” se abrevia comúnmente como “ctor” en la literatura; por lo general, nosotros evitaremos las abreviaciones.

C++ requiere una llamada al constructor para cada objeto que se crea, lo cual ayuda a asegurar que cada objeto se inicialice antes de utilizarlo en un programa. La llamada al constructor ocurre de manera implícita cuando se crea el objeto. Si una clase no incluye un constructor en forma explícita, el compilador proporciona un **constructor predeterminado**, es decir, un constructor sin parámetros. Por ejemplo, cuando en la línea 46 de la figura 3.5 se crea un objeto `LibroCalificaciones`, se hace una llamada al constructor predeterminado. Este constructor proporcionado por el compilador crea un objeto `LibroCalificaciones` sin proporcionar ningún valor inicial para los miembros de datos de tipos fundamentales del objeto. [Nota: para los miembros de datos que son objetos de otras clases, el constructor llama de manera implícita al constructor predeterminado de cada miembro de datos, para asegurar que ese miembro de datos se inicialice en forma apropiada. Ésta es la razón por la cual el miembro de datos `string` llamado `nombreCurso` (en la figura 3.5) se inicializó con la cadena vacía; el constructor predeterminado para la clase `string` asigna al valor del objeto `string` la cadena vacía. En la sección 10.3 aprenderá más acerca de cómo inicializar miembros de datos que sean objetos de otras clases.]

En el ejemplo de la figura 3.7, especificamos el nombre de un curso para un objeto `LibroCalificaciones` cuando se crea el objeto (línea 49). En este caso, el argumento "CS101 Introducción a la programación en C++" se pasa al constructor del objeto `LibroCalificaciones` (líneas 17 a 20) y se utiliza para inicializar el `nombreCurso`. En la

figura 3.7 se define una clase `LibroCalificaciones` modificada, la cual contiene un constructor con un parámetro `string` que recibe el nombre inicial del curso.

```

1 // Fig. 3.7: fig03_07.cpp
2 // Creación de instancias de varios objetos de la clase LibroCalificaciones y uso
3 // del constructor de LibroCalificaciones para especificar el nombre del curso
4 // cuando se crea cada objeto LibroCalificaciones.
5 #include <iostream>
6 using std::cout;
7 using std::endl;
8
9 #include <string> // el programa usa la clase string estándar de C++
10 using std::string;
11
12 // definición de la clase LibroCalificaciones
13 class LibroCalificaciones
14 {
15 public:
16     // el constructor inicializa a nombreCurso con la cadena que se suministra como argumento
17     LibroCalificaciones( string nombre )
18     {
19         establecerNombreCurso( nombre ); // llama a la función establecer para inicializar
20         nombreCurso
21     } // fin del constructor de LibroCalificaciones
22
23     // función para establecer el nombre del curso
24     void establecerNombreCurso( string nombre )
25     {
26         nombreCurso = nombre; // almacena el nombre del curso en el objeto
27     } // fin de la función establecerNombreCurso
28
29     // función para obtener el nombre del curso
30     string obtenerNombreCurso()
31     {
32         return nombreCurso; // devuelve el nombreCurso del objeto
33     } // fin de la función obtenerNombreCurso
34
35     // muestra un mensaje de bienvenida para el usuario de LibroCalificaciones
36     void mostrarMensaje()
37     {
38         // llama a obtenerNombreCurso para obtener el nombreCurso
39         cout << "Bienvenido al libro de calificaciones para\n" << obtenerNombreCurso()
40         << "!" << endl;
41     } // fin de la función mostrarMensaje
42 private:
43     string nombreCurso; // nombre del curso para este LibroCalificaciones
44 }; // fin de la clase LibroCalificaciones
45
46 // la función main empieza la ejecución del programa
47 int main()
48 {
49     // crea dos objetos LibroCalificaciones
50     LibroCalificaciones libroCalificaciones1( "CS101 Introducción a la programación en C++" );
51     LibroCalificaciones libroCalificaciones2( "CS102 Estructuras de datos en C++" );
52
53     // muestra el valor inicial de nombreCurso para cada LibroCalificaciones
54     cout << "libroCalificaciones1 se creo para el curso: " << libroCalificaciones1.
      obtenerNombreCurso()
      << "\nlibroCalificaciones2 se creo para el curso: " << libroCalificaciones2.
      obtenerNombreCurso()

```

Figura 3.7 | Creación de instancias de varios objetos de la clase `LibroCalificaciones` y uso de su constructor para especificar el nombre del curso cuando se crea cada objeto `LibroCalificaciones`. (Parte I de 2).

```

55     << endl;
56     return 0; // indica que terminó correctamente
57 } // fin de main

```

libroCalificaciones1 se creo para el curso: CS101 Introduccion a la programacion en C++
 libroCalificaciones2 se creo para el curso: CS102 Estructuras de datos en C++

Figura 3.7 | Creación de instancias de varios objetos de la clase `LibroCalificaciones` y uso de su constructor para especificar el nombre del curso cuando se crea cada objeto `LibroCalificaciones`. (Parte 2 de 2).

Definición de un constructor

En las líneas 17 a 20 de la figura 3.7 se define un constructor para la clase `LibroCalificaciones`. Observe que el constructor tiene el mismo nombre que su clase, `LibroCalificaciones`. Un constructor especifica en su lista de parámetros los datos que requiere para realizar su tarea. Al crear un nuevo objeto, el programador coloca estos datos en los paréntesis que van después del nombre del objeto (como hicimos en las líneas 49 y 50). En la línea 17 se indica que el constructor de la clase `LibroCalificaciones` tiene un parámetro `string` llamado `nombre`. Observe que en la línea 17 no se especifica un tipo de valor de retorno, ya que los constructores no pueden devolver valores (ni siquiera `void`).

En la línea 19, en el cuerpo del constructor se pasa el parámetro `nombre` a la función miembro `establecerNombreCurso`, la cual asigna un valor al miembro de datos `nombreCurso`. La función miembro `establecerNombreCurso` (líneas 23 a 26) simplemente asigna su parámetro `nombre` al miembro de datos `nombreCurso`, por lo que tal vez usted se pregunte por qué nos tomamos la molestia de realizar la llamada a `establecerNombreCurso` en la línea 19; sin duda, el constructor podría realizar la asignación `nombreCurso = nombre`. En la sección 3.10, modificaremos `establecerNombreCurso` para llevar a cabo la validación (lo cual asegura que, en este caso, el valor de `nombreCurso` tenga una longitud de 25 caracteres o menos). En ese punto, los beneficios de llamar a `establecerNombreCurso` desde el constructor se verán con claridad. Observe que tanto el constructor (línea 17) como la función `establecerNombreCurso` (línea 23) utilizan un parámetro llamado `nombre`. Puede usar los mismos nombres de parámetros en distintas funciones, ya que los parámetros son locales para cada función; no interfieren unos con otros.

Prueba de la clase `LibroCalificaciones`

En las líneas 46 a 57 de la figura 3.7 se define la función `main` que prueba la clase `LibroCalificaciones` y demuestra cómo inicializar objetos `LibroCalificaciones` mediante el uso de un constructor. En la línea 49, en la función `main` se crea y se inicializa un objeto `LibroCalificaciones` llamado `libroCalificaciones1`. Cuando se ejecuta esta línea, C++ hace una llamada implícita al constructor de `LibroCalificaciones` (líneas 17 a 20) con el argumento "CS101 Introduccion a la programacion en C++" para inicializar el nombre del curso de `libroCalificaciones1`. En la línea 50 se repite este proceso para el objeto `LibroCalificaciones` llamado `libroCalificaciones2`, pero esta vez se pasa el argumento "CS102 Estructuras de datos en C++" para inicializar el nombre del curso de `libroCalificaciones2`. En las líneas 53 y 54 se utiliza la función miembro `obtenerNombreCurso` de cada objeto para obtener los nombres de los cursos y mostrar que, sin duda, se inicializaron al momento de crear los objetos. La salida confirma que cada objeto `LibroCalificaciones` mantiene su propia copia del miembro de datos `nombreCurso`.

Dos formas de proporcionar un constructor predeterminado para una clase

Cualquier constructor que no recibe argumentos se llama constructor predeterminado. Una clase puede recibir un constructor predeterminado en una de dos formas:

1. El compilador crea de manera implícita un constructor predeterminado en una clase que no define a un constructor. Dicho constructor predeterminado no inicializa los miembros de datos de la clase, pero llama al constructor predeterminado para cada miembro de datos que sea un objeto de otra clase. [Nota: por lo general, una variable sin inicializar contiene un valor "basura" (por ejemplo, una variable `int` sin inicializar podría contener -858993460, que probablemente sea un valor incorrecto para esa variable en la mayoría de los programas).]
2. El programador define en forma explícita un constructor que no recibe argumentos. Dicho constructor predeterminado realizará la inicialización especificada por el programador, y llamará al constructor predeterminado para cada miembro de datos que sea un objeto de otra clase.

Si define un constructor sin argumentos, C++ no creará de manera implícita un constructor predeterminado para esa clase. Observe que para cada versión de la clase `LibroCalificaciones` en las figuras 3.1, 3.3 y 3.5, el compilador definió de manera implícita un constructor predeterminado.



Tip para prevenir errores 3.2

A menos que no sea necesario inicializar los miembros de datos de su clase (casi nunca), debe proporcionar un constructor para asegurar que los miembros de datos de su clase se inicialicen con valores significativos al momento de crear cada nuevo objeto de su clase.



Observación de Ingeniería de Software 3.5

Los miembros de datos se pueden inicializar en un constructor de la clase, o sus valores pueden establecerse más adelante, después de crear el objeto. Sin embargo, es una buena práctica de ingeniería de software asegurarse que un objeto esté inicializado por completo antes de que el código cliente invoque las funciones miembro de ese objeto. En general, no debemos depender del código cliente para asegurar que un objeto se inicialice de manera apropiada.

Agregar el constructor al diagrama de clases de UML de la clase LibroCalificaciones

El diagrama de clases de UML de la figura 3.8 modela la clase `LibroCalificaciones` de la figura 3.7, la cual tiene un constructor con un parámetro `nombre` de tipo `String` (representado por el tipo `String` en UML). Al igual que las operaciones, el UML modela a los constructores en el tercer compartimiento de una clase en un diagrama de clases. Para diferenciar a un constructor de las operaciones de la clase, UML coloca la palabra “constructor” entre los signos «» y antes del nombre del constructor. Es costumbre enlistar el constructor de la clase antes de todas las operaciones en el tercer compartimiento.

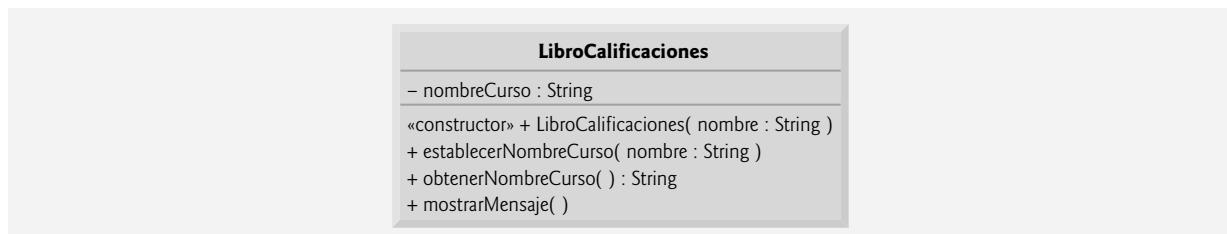


Figura 3.8 | Diagrama de clases de UML, el cual indica que la clase `LibroCalificaciones` tiene un constructor con un parámetro llamado `nombre` de tipo `String` de UML.

3.8 Colocar una clase en un archivo separado para fines de reutilización

Hemos desarrollado la clase `LibroCalificaciones` de acuerdo con nuestras necesidades por ahora, desde la perspectiva de programación, por lo que vamos a considerar ciertas cuestiones de ingeniería. Uno de los beneficios de crear definiciones de clases es que, cuando se empaquetan en forma apropiada, nuestras clases pueden ser reutilizadas por los programadores, potencialmente desde cualquier parte del mundo. Por ejemplo, podemos utilizar el tipo `string` de la Biblioteca estándar de C++ en cualquier programa en C++ al incluir el archivo de encabezado `<string>` en el programa (y, como veremos, al poder enlazarnos con el código objeto de la biblioteca).

Por desgracia, los programadores que deseen utilizar nuestra clase `LibroCalificaciones` no pueden simplemente incluir el archivo de la figura 3.7 en otro programa. Como aprendió en el capítulo 2, la función `main` empieza la ejecución de todo programa, y cada programa debe tener sólo una función `main`. Si otros programadores incluyen el código de la figura 3.7, sus programas tendrán entonces dos funciones `main`. Cuando intenten compilar sus programas, el compilador indicará un error. Por ejemplo, al tratar de compilar un programa con dos funciones `main` en Microsoft Visual C++ 2005 se produce el siguiente error:

```
error C2084: function 'int main(void)' already has a body
```

cuando el compilador trata de compilar la segunda función `main` que encuentra. De manera similar, el compilador GNU C++ produce el siguiente error:

```
redefinition of 'int main()'
```

Estos errores indican que un programa ya tiene una función `main`. Por lo tanto, al colocar `main` en el mismo archivo con una definición de clase, evitamos que esa clase pueda ser reutilizada por otros programas. En esta sección demostraremos cómo hacer la clase `LibroCalificaciones` reutilizable, al separarla de la función `main` y colocarla en otro archivo.

Archivos de encabezado

Cada uno de los ejemplos anteriores en el capítulo consiste de un solo archivo .cpp, al cual se le conoce también como **archivo de código fuente**, el cual contiene la definición de la clase `LibroCalificaciones` y una función `main`. Al construir un programa en C++ orientado a objetos, es costumbre definir el código fuente reutilizable (como una clase) en un archivo que, por convención, tiene la extensión .h; a éste se le conoce como **archivo de encabezado**. Los programas utilizan las directivas del preprocesador `#include` para incluir archivos de encabezado y aprovechar los componentes de software reutilizables, como el tipo `string` que se proporciona en la Biblioteca estándar de C++, y los tipos definidos por el usuario como la clase `LibroCalificaciones`.

En nuestro siguiente ejemplo, sepáramos el código de la figura 3.7 en dos archivos: `LibroCalificaciones.h` (figura 3.9) y `fig03_10.cpp` (figura 3.10). Cuando analice el archivo de encabezado de la figura 3.9, observe que sólo contiene la definición de la clase `LibroCalificaciones` (líneas 11 a 41) y las líneas 3 a 8, que permiten a la clase `LibroCalificaciones` usar `cout`, `endl` y el tipo `string`. La función `main` que utiliza a la clase `LibroCalificaciones` se define en el archivo de código fuente `fig03_10.cpp` (figura 3.10) en las líneas 10 a 21. Para ayudarlo a prepararse para los programas más extensos que encontrará más adelante en este libro y en la industria, a menudo utilizamos un archivo de código fuente separado que contiene la función `main` para probar nuestras clases (a éste se le conoce como **programa controlador**). Pronto aprenderá cómo un archivo de código fuente con `main` puede utilizar la definición de una clase que se encuentra en un archivo de encabezado para crear objetos de esa clase.

Incluir un archivo de encabezado que contiene una clase definida por el usuario

Un archivo de encabezado como `LibroCalificaciones.h` (figura 3.9) no puede usarse para empezar la ejecución del programa, ya que no tiene una función `main`. Si trata de compilar y enlazar `LibroCalificaciones.h` por sí solo para crear una aplicación ejecutable, Microsoft Visual C++ 2005 produce el siguiente mensaje de error del enlazador:

```
error LNK2019: unresolved external symbol _main referenced in
function _mainCRTStartup
```

Para compilar y enlazar con GNU C++ en Linux, debe incluir primero el archivo de encabezado en un archivo de código fuente .cpp, y después GNU C++ produce un mensaje de error:

```
undefined reference to 'main'
```

Este error indica que el enlazador no pudo localizar la función `main` del programa. Para probar la clase `LibroCalificaciones` (definida en la figura 3.9), debe escribir un archivo de código fuente separado que contenga una función `main` (como la figura 3.10), la cual debe instanciar y utilizar objetos de la clase.

```

1 // Fig. 3.9: LibroCalificaciones.h
2 // Definición de la clase LibroCalificaciones en un archivo separado de main.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string> // la clase LibroCalificaciones utiliza la clase string estándar de C++
8 using std::string;
9
10 // definición de la clase LibroCalificaciones
11 class LibroCalificaciones
12 {
13 public:
14     // el constructor inicializa nombreCurso con la cadena que se suministra como argumento
15     LibroCalificaciones( string nombre )
16     {
17         establecerNombreCurso( nombre ); // llama a la función establecer para inicializar
18         nombreCurso
19     } // fin del constructor de LibroCalificaciones
20
21     // función para establecer el nombre del curso
22     void establecerNombreCurso( string nombre )
23     {
```

Figura 3.9 | Definición de la clase `LibroCalificaciones`. (Parte I de 2).

```

23     nombreCurso = nombre; // almacena el nombre del curso en el objeto
24 } // fin de la función establecerNombreCurso
25
26 // función para obtener el nombre del curso
27 string obtenerNombreCurso()
28 {
29     return nombreCurso; // devuelve el nombreCurso del objeto
30 } // fin de la función obtenerNombreCurso
31
32 // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
33 void mostrarMensaje()
34 {
35     // llama a obtenerNombreCurso para obtener el nombreCurso
36     cout << "Bienvenido al libro de calificaciones para\n" << obtenerNombreCurso()
37     << "!" << endl;
38 } // fin de la función mostrarMensaje
39 private:
40     string nombreCurso; // nombre del curso para este LibroCalificaciones
41 }; // fin de la clase LibroCalificaciones

```

Figura 3.9 | Definición de la clase **LibroCalificaciones**. (Parte 2 de 2).

```

1 // Fig. 3.10: fig03_10.cpp
2 // Inclusión de la clase LibroCalificaciones del archivo LibroCalificaciones.h para usarla en main.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "LibroCalificaciones.h" // incluye la definición de la clase LibroCalificaciones
8
9 // la función main empieza la ejecución del programa
10 int main()
11 {
12     // crea dos objetos LibroCalificaciones
13     LibroCalificaciones libroCalificaciones1( "CS101 Introduccion a la programacion en C++" );
14     LibroCalificaciones libroCalificaciones2( "CS102 Estructuras de datos en C++" );
15
16     // muestra el valor inicial de nombreCurso para cada LibroCalificaciones
17     cout << "libroCalificaciones1 creado para el curso: " << libroCalificaciones1.
18         obtenerNombreCurso()
19         << "\nlibroCalificaciones2 creado para el curso: " << libroCalificaciones2.
20         obtenerNombreCurso()
21         << endl;
22     return 0; // indica que terminó correctamente
23 } // fin de main

```

libroCalificaciones1 creado para el curso: CS101 Introduccion a la programacion en C++
 libroCalificaciones2 creado para el curso: CS102 Estructuras de datos en C++

Figura 3.10 | Inclusión de la clase **LibroCalificaciones** del archivo **LibroCalificaciones.h** para usarla en **main**.

En la sección 3.4 vimos que, aunque el compilador sabe qué son los tipos de datos fundamentales (como **int**), no sabe qué es un **LibroCalificaciones** ya que es un tipo definido por el usuario. De hecho, el compilador ni siquiera conoce las clases de la Biblioteca estándar de C++. Para ayudarlo a comprender cómo usar una clase, debemos proporcionar en forma explícita al compilador la definición de la clase; ésta es la razón por la que, para que un programa pueda usarse un tipo **string**, debe incluir el archivo de encabezado **<string>**. Esto permite al compilador determinar la cantidad de memoria que debe reservar para cada objeto de la clase, y asegurar que un programa llame a las funciones miembro de la clase de una forma correcta.

Para crear los objetos **LibroCalificaciones** llamados **libroCalificaciones1** y **libroCalificaciones2** en las líneas 13 y 14 de la figura 3.10, el compilador debe conocer el tamaño de un objeto **LibroCalificaciones**. Aunque en concepto los objetos contienen miembros de datos y funciones miembro, los objetos de C++ sólo contienen datos.

El compilador sólo crea una copia de las funciones miembro de la clase y comparte esa copia entre todos los objetos de la clase. Desde luego que cada objeto necesita su propia copia de los miembros de datos de la clase, ya que su contenido puede variar de un objeto a otro (como dos objetos `CuentaBanco` distintos, que tienen dos miembros de datos `saldo` distintos). Sin embargo, el código de la función miembro no se puede modificar, por lo que puede compartirse entre todos los objetos de la clase. Al incluir a `LibroCalificaciones.h` en la línea 7, proporcionamos acceso al compilador para que utilice la información que necesita (figura 3.9, línea 40) para determinar el tamaño de un objeto `LibroCalificaciones` y determinar si los objetos de la clase se utilizan correctamente (en las líneas 13 a 14 y 17 a 18 de la figura 3.10).

En la línea 7 se indica al preprocesador de C++ que reemplace la directiva con una copia del contenido de `LibroCalificaciones.h` (es decir, la definición de la clase `LibroCalificaciones`) *antes* de compilar el programa. Cuando se compila el archivo de código fuente `fig03_10.cpp`, ahora contiene la definición de la clase `LibroCalificaciones` (debido a la instrucción `#include`), y el compilador puede determinar cómo crear objetos `LibroCalificaciones` y revisar que se hagan llamadas a sus funciones miembro en forma adecuada. Ahora que la definición de la clase está en un archivo de encabezado (sin una función `main`), podemos incluir ese encabezado en *cualquier* programa que necesite reutilizar nuestra clase `LibroCalificaciones`.

Cómo se localizan los archivos de encabezado

Observe que el nombre del archivo de encabezado `LibroCalificaciones.h` en la línea 7 de la figura 3.10 se encierra entre comillas (" ") en lugar de usar los signos < y >. Por lo general, los archivos de código fuente de un programa y los archivos de encabezado definidos por el usuario se colocan en el mismo directorio. Cuando el preprocesador encuentra el nombre de un archivo de encabezado entre comillas (por ejemplo, "LibroCalificaciones.h"), intenta localizar el archivo de encabezado en el mismo directorio que el archivo en el que aparece la directiva `#include`. Si el preprocesador no puede encontrar el archivo de encabezado en ese directorio, lo busca en la(s) misma(s) ubicación(es) que los archivos de encabezado de la Biblioteca estándar de C++. Cuando el preprocesador encuentra el nombre de un archivo de encabezado entre los signos < y > (como <iostream>), asume que el encabezado forma parte de la Biblioteca estándar de C++ y no busca en el directorio del programa que se está procesando.



Tip para prevenir errores 3.3

Para asegurar que el preprocesador pueda localizar los archivos de encabezado en forma correcta, en las directivas del preprocesador `#include` se deben colocar los nombres de los archivos de encabezado definidos por el usuario entre comillas (como "LibroCalificaciones.h"), y se deben colocar los nombres de los archivos de encabezado de la Biblioteca estándar de C++ entre los signos < y > (como <iostream>).

Cuestiones adicionales sobre Ingeniería de Software

Ahora que la clase `LibroCalificaciones` está definida en un archivo de encabezado, puede reutilizarse. Por desgracia, al colocar la definición de una clase en un archivo de encabezado como en la figura 3.9, se sigue revelando toda la implementación de la clase a los clientes de la misma; `LibroCalificaciones.h` es simplemente un archivo de texto que cualquiera puede abrir y leer. La sabiduría de la Ingeniería de Software convencional nos dice que para usar un objeto de la clase, el código cliente necesita saber sólo qué funciones miembro debe llamar, qué argumentos debe proporcionar a cada función miembro y qué tipo de valor de retorno debe esperar de cada función miembro. El código cliente no necesita saber cómo se implementan esas funciones.

Si el programador del código cliente necesita saber cómo se implementa una clase, podría escribir código cliente basado en los detalles de implementación de la clase. Lo ideal sería que, si cambia la implementación, los clientes de la clase no tengan que cambiar. Al ocultar los detalles de implementación de la clase, facilitamos la tarea de cambiar la implementación de la clase al mismo tiempo que minimizamos (y con suerte, eliminamos) los cambios al código cliente.

En la sección 3.9 le mostraremos cómo descomponer la clase `LibroCalificaciones` en dos archivos, de manera que:

1. la clase sea reutilizable,
2. los clientes de la clase sepan qué funciones miembro proporciona la clase, cómo llamarlas y qué tipo de valores de retorno esperar, y
3. los clientes no sepan cómo se implementan las funciones miembro de la clase.

3.9 Separar la interfaz de la implementación

En la sección anterior le mostramos cómo fomentar la reutilización de software al separar la definición de la clase del código cliente (por ejemplo, la función `main`) que utiliza esa clase. Ahora presentaremos otro principio fundamental de la buena Ingeniería de Software: **separar la interfaz de la implementación**.

La interfaz de una clase

Las **interfaces** definen y estandarizan las formas en las que las personas y los sistemas interactúan entre sí. Por ejemplo, los controles de una radio sirven como una interfaz entre los usuarios de la radio y sus componentes internos. Los controles permiten a los usuarios realizar un conjunto limitado de operaciones (como cambiar la estación, ajustar el volumen y elegir entre una estación en AM o una en FM). Varias radios pueden implementar estas operaciones de manera distinta; algunas proporcionan botones, otras perillas y algunas incluso soportan comandos de voz. La interfaz especifica *qué* operaciones permite realizar una radio a los usuarios, pero no especifica *cómo* se implementan estas operaciones en su interior.

De manera similar, la **interfaz de una clase** describe *qué* servicios pueden usar los clientes de la clase y *cómo solicitar* esos servicios, pero no *cómo* lleva a cabo la clase esos servicios. La interfaz de una clase consiste en las funciones miembro **public** de la clase (también conocidas como **servicios públicos**). Por ejemplo, la interfaz de la clase **LibroCalificaciones** (figura 3.9) contiene un constructor y las funciones miembro **establecerNombreCurso**, **obtenerNombreCurso** y **mostrarMensaje**. Los clientes de **LibroCalificaciones** (por ejemplo, **main** en la figura 3.10) utilizan estas funciones para solicitar los servicios de la clase. Como pronto veremos, podemos especificar la interfaz de una clase al escribir una definición de clase que sólo enliste los nombres de las funciones miembro, los tipos de los valores de retorno y los tipos de los parámetros.

Separar la interfaz de la implementación

En nuestros ejemplos anteriores, la definición de cada clase contenía las definiciones completas de las funciones miembro **public** de la clase y las declaraciones de sus miembros de datos **private**. Sin embargo, una mejor ingeniería de software es definir las funciones miembro fuera de la definición de la clase, de manera que sus detalles de implementación se puedan ocultar del código cliente. Esta práctica asegura que los programadores no escriban código cliente que dependa de los detalles de implementación de la clase. Si éste fuera el caso, sería más probable que el código cliente fallara si cambiara la implementación de la clase.

El programa de las figuras 3.11 a 3.13 separa la interfaz de **LibroCalificaciones** de su implementación; para ello divide la definición de la clase de la figura 3.9 en dos archivos: el archivo de encabezado **LibroCalificaciones.h** (figura 3.11) en el que se define la clase **LibroCalificaciones**, y el archivo de código fuente **LibroCalificaciones.cpp** (figura 3.12) en el que se definen las funciones miembro de **LibroCalificaciones**. Por convención, las definiciones de las funciones miembro se colocan en un archivo de código fuente con el mismo nombre base (por ejemplo, **LibroCalificaciones**) que el archivo de encabezado de la clase, pero con una extensión de archivo **.cpp**. El archivo de código fuente **fig03_13.cpp** (figura 3.13) define la función **main** (el código cliente). El código y la salida de la figura 3.13 son idénticos a los de la figura 3.10. En la figura 3.14 se muestra cómo se compila este programa de tres archivos, desde las perspectivas del programador de la clase **LibroCalificaciones** y del programador del código cliente; explicaremos esta figura con detalle.

LibroCalificaciones.h: definición de la interfaz de una clase mediante prototipos de funciones

El archivo de encabezado **LibroCalificaciones.h** (figura 3.11) contiene otra versión de la definición de la clase **LibroCalificaciones** (líneas 9 a 18). Esta versión es similar a la de la figura 3.9, pero las definiciones de las funciones en la figura 3.9 se reemplazan aquí con **prototipos de funciones** (líneas 12 a 15) que describen la interfaz **public** de la clase sin revelar las implementaciones de sus funciones miembro. Un prototipo de función es una declaración de una función que indica al compilador el nombre de la función, su tipo de valor de retorno y los tipos de sus parámetros. Observe que el archivo de encabezado sigue especificando el miembro de datos **private** de la clase (línea 17) también. De nuevo, el compilador debe conocer los miembros de datos de la clase para determinar cuánta memoria debe reservar para cada objeto de la misma. Al incluir el archivo de encabezado **LibroCalificaciones.h** en el código cliente (línea 8 de la figura 3.13), el compilador obtiene la información que necesita para asegurar que el código cliente llame a las funciones miembro de la clase **LibroCalificaciones** en forma correcta.

El prototipo de función en la línea 12 (figura 3.11) indica que el constructor requiere un parámetro **string**. Recuerde que los constructores no tienen tipos de valores de retorno, por lo que no aparece ningún tipo de valor de retorno en el prototipo de la función. El prototipo de la función miembro **establecerNombreCurso** (línea 13) indica que requiere un parámetro **string** y no devuelve un valor (es decir, su tipo de valor de retorno es **void**). El prototipo de la función **obtenerNombreCurso** (línea 14) indica que la función no requiere parámetros y devuelve un **string**. Por último, el prototipo de la función **mostrarMensaje** (línea 15) especifica que **mostrarMensaje** no requiere parámetros y no devuelve un valor. Estos prototipos de funciones son iguales que los correspondientes encabezados de funciones en la figura 3.9, sólo que los nombres de los parámetros (que son opcionales en los prototipos) no se incluyen y cada prototipo de función debe terminar con un punto y coma.

```

1 // Fig. 3.11: LibroCalificaciones.h
2 // Definición de la clase LibroCalificaciones. Este archivo presenta la interfaz
3 // public de LibroCalificaciones sin revelar las implementaciones de sus funciones
4 // miembro, que están definidas en LibroCalificaciones.cpp.
5 #include <string> // la clase LibroCalificaciones utiliza la clase string estándar de C++
6 using std::string;
7
8 // definición de la clase LibroCalificaciones
9 class LibroCalificaciones
10 {
11 public:
12     LibroCalificaciones( string ); // constructor que inicializa a nombreCurso
13     void establecerNombreCurso( string ); // función que establece el nombre del curso
14     string obtenerNombreCurso(); // función que obtiene el nombre del curso
15     void mostrarMensaje(); // función que muestra un mensaje de bienvenida
16 private:
17     string nombreCurso; // nombre del curso para este LibroCalificaciones
18 } // fin de la clase LibroCalificaciones

```

Figura 3.11 | Definición de la clase `LibroCalificaciones` que contiene prototipos de funciones que especifican la interfaz de la clase.



Error común de programación 3.8

Olvidar el punto y coma al final de un prototipo de función es un error de sintaxis.



Buena práctica de programación 3.7

Aunque los nombres de los parámetros en los prototipos de funciones son opcionales (el compilador los ignora), muchos programadores utilizan estos nombres para fines de documentación.



Tip para prevenir errores 3.4

Los nombres de los parámetros en un prototipo de función (que, de nuevo, el compilador los ignora) pueden provocar confusión si están incorrectos, o si se utilizan nombres confusos. Por esta razón, para crear prototipos de funciones, muchos programadores copian la primera línea de las definiciones de funciones correspondientes (cuando está disponible el código para las funciones), y después anexan un punto y coma al final de cada prototipo.

LibroCalificaciones.cpp: definir las funciones miembro en un archivo de código fuente separado

El archivo de código fuente `LibroCalificaciones.cpp` (figura 3.12) define las funciones miembro de la clase `LibroCalificaciones`, que se declararon en las líneas 12 a 15 de la figura 3.11. Las definiciones de las funciones miembro aparecen en las líneas 11 a 34 y son casi idénticas a las definiciones de las funciones miembro en las líneas 15 a 38 de la figura 3.9.

```

1 // Fig. 3.12: LibroCalificaciones.cpp
2 // Definiciones de las funciones miembro de LibroCalificaciones. Este archivo contiene
3 // implementaciones de las funciones miembro cuyo prototipo está en LibroCalificaciones.h.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "LibroCalificaciones.h" // incluye la definición de la clase LibroCalificaciones
9
10 // el constructor inicializa nombreCurso con el objeto string suministrado como argumento
11 LibroCalificaciones::LibroCalificaciones( string nombre )

```

Figura 3.12 | Las definiciones de las funciones miembro de `LibroCalificaciones` representan la implementación de la clase `LibroCalificaciones`. (Parte I de 2).

```

12 {
13     establecerNombreCurso( nombre ); // llama a la función establecer para inicializar nombreCurso
14 } // fin del constructor de LibroCalificaciones
15
16 // función para establecer el nombre del curso
17 void LibroCalificaciones::establecerNombreCurso( string nombre )
18 {
19     nombreCurso = nombre; // almacena el nombre del curso en el objeto
20 } // fin de la función establecerNombreCurso
21
22 // función para obtener el nombre del curso
23 string LibroCalificaciones::obtenerNombreCurso()
24 {
25     return nombreCurso; // devuelve el nombreCurso del objeto
26 } // fin de la función obtenerNombreCurso
27
28 // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
29 void LibroCalificaciones::mostrarMensaje()
30 {
31     // llama a obtenerNombreCurso para obtener el nombreCurso
32     cout << "Bienvenido al libro de calificaciones para\n" << obtenerNombreCurso()
33     << "!" << endl;
34 } // fin de la función mostrarMensaje

```

Figura 3.12 | Las definiciones de las funciones miembro de `LibroCalificaciones` representan la implementación de la clase `LibroCalificaciones`. (Parte 2 de 2).

Observe que al nombre de cada función miembro en los encabezados de las funciones (líneas 11, 17, 23 y 29) se le antepone el nombre de la clase y los caracteres `::`, que representan el **operador binario de resolución de ámbito** (**o alcance**). Esto “enlaza” a cada función miembro con la definición (ahora separada) de la clase `LibroCalificaciones` (figura 3.11), la cual declara las funciones miembro y los miembros de datos. Sin “`LibroCalificaciones::`” antes de cada nombre de función, el compilador no las reconocería como funciones miembro de la clase `LibroCalificaciones`; las consideraría como funciones “libres” o “sueltas”, al igual que `main`. Dichas funciones no pueden acceder a los datos `private` de `LibroCalificaciones` ni llamar a las funciones miembro de la clase, sin especificar un objeto. Por lo tanto, el compilador no podrá compilar estas funciones. Por ejemplo, en las líneas 19 y 25 que acceden a la variable `nombreCurso` se producirían errores de compilación, ya que `nombreCurso` no está declarada como una variable local en cada función; el compilador no sabría que `nombreCurso` ya está declarada como miembro de datos de la clase `LibroCalificaciones`.



Error común de programación 3.9

Al definir las funciones miembro de una clase fuera de la misma, si se omite el nombre de la clase y el operador de resolución de ámbito (::) antes de los nombres de las funciones se producen errores de compilación.

Para indicar que las funciones miembro en `LibroCalificaciones.cpp` forman parte de la clase `LibroCalificaciones`, debemos primero incluir el archivo de encabezado `LibroCalificaciones.h` (línea 8 de la figura 3.12). Esto nos permite acceder al nombre de la clase `LibroCalificaciones` en el archivo `LibroCalificaciones.cpp`. Al compilar `LibroCalificaciones.cpp`, el compilador utiliza la información en `LibroCalificaciones.h` para asegurar que

1. la primera línea de cada función miembro (líneas 11, 17, 23 y 29) coincide con su prototipo en el archivo `LibroCalificaciones.h`; por ejemplo, el compilador asegura que `NombreCurso` no acepte parámetros y devuelva un valor `string`, y que
2. cada función miembro sepa acerca de los miembros de datos y otras funciones miembro de la clase; por ejemplo, en las líneas 19 y 25 se puede acceder a la variable `nombreCurso`, ya que está declarada en `LibroCalificaciones.h` como miembro de datos de la clase `LibroCalificaciones`, y en las líneas 13 y 32 se puede llamar a las funciones `establecerNombreCurso` y `obtenerNombreCurso`, respectivamente, ya que cada una se declara como función miembro de la clase en `LibroCalificaciones.h` (y debido a que estas llamadas se conforman con los prototipos correspondientes).

Prueba de la clase LibroCalificaciones

En la figura 3.13 se realizan las mismas manipulaciones de objetos `LibroCalificaciones` que en la figura 3.10. Al separar la interfaz de `LibroCalificaciones` de la implementación de sus funciones miembro, no se ve afectada la forma en que este código cliente utiliza la clase. Sólo afecta la forma en que se compila y enlaza el programa, lo cual veremos en breve.

```

1 // Fig. 3.13: fig03_13.cpp
2 // Demostración de la clase LibroCalificaciones después de separar
3 // su interfaz de su implementación.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "LibroCalificaciones.h" // incluye la definición de la clase LibroCalificaciones
9
10 // la función main empieza la ejecución del programa
11 int main()
12 {
13     // crea dos objetos LibroCalificaciones
14     LibroCalificaciones libroCalificaciones1( "CS101 Introducción a la programación en C++" );
15     LibroCalificaciones libroCalificaciones2( "CS102 Estructuras de datos en C++" );
16
17     // muestra el valor inicial de courseName para cada LibroCalificaciones
18     cout << "libroCalificaciones1 creado para el curso: " << libroCalificaciones1.
19     obtenerNombreCurso()
20         << endl;
21     return 0; // indica que terminó correctamente
22 } // fin de main

```

```

libroCalificaciones1 creado para el curso: CS101 Introducción a la programación en C++
libroCalificaciones2 creado para el curso: CS102 Estructuras de datos en C++

```

Figura 3.13 | Demostración de la clase `LibroCalificaciones` después de separar su interfaz de la implementación.

Al igual que en la figura 3.10, en la línea 8 de la figura 3.13 se incluye el archivo de encabezado `LibroCalificaciones.h`, de manera que el compilador pueda asegurar que los objetos `LibroCalificaciones` se creen y manipulen correctamente en el código cliente. Antes de ejecutar este programa, deben compilarse los archivos de código fuente de las figuras 3.12 y 3.13, y después se deben enlazar; es decir, las llamadas a las funciones miembro en el código cliente necesitan enlazarse con las implementaciones de estas funciones miembro de la clase; un trabajo que realiza el enlazador.

El proceso de compilación y enlace

El diagrama de la figura 3.14 muestra el proceso de compilación y enlace que produce una aplicación `LibroCalificaciones` ejecutable, que los instructores pueden utilizar. Lo común es que un programador cree y compile la interfaz e implementación de una clase, y que otro programador implemente el código cliente para utilizar esa clase. Así, el diagrama muestra lo que requieren tanto el programador de la implementación de la clase, como el programador del código cliente. Las líneas punteadas en el diagrama muestran las piezas requeridas por el programador de la implementación de la clase, el programador del código cliente y el usuario de la aplicación `LibroCalificaciones`, respectivamente. [Nota: la figura 3.14 no es un diagrama de UML.]

El programador de la implementación de una clase, responsable de crear una clase `LibroCalificaciones` reutilizable, crea el archivo de encabezado `LibroCalificaciones.h` y el archivo de código fuente `LibroCalificaciones.cpp` que incluye (mediante `#include`) el archivo de encabezado, y después compila el archivo de código fuente para crear el código objeto de `LibroCalificaciones`. Para ocultar los detalles de la implementación de las funciones miembro de `LibroCalificaciones`, el programador de la implementación de la clase proporciona al programador del código cliente el archivo de encabezado `LibroCalificaciones.h` (que especifica la interfaz y los miembros de datos de la clase) y el código objeto para la clase `LibroCalificaciones` (que contiene las instrucciones en lenguaje máquina que representan

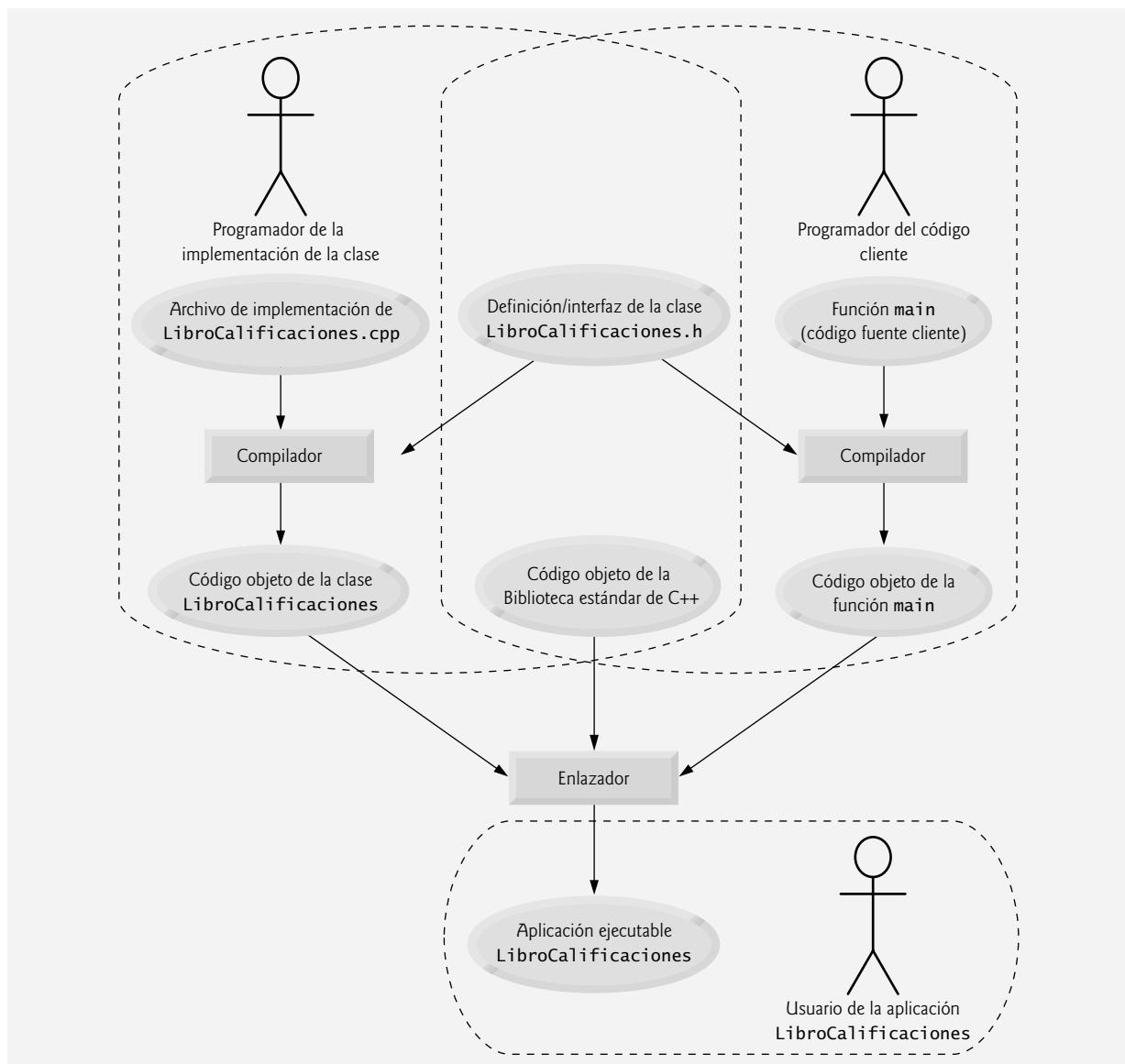


Figura 3.14 | Proceso de compilación y enlace que produce una aplicación ejecutable.

a las funciones miembro de `LibroCalificaciones`). El programador del código cliente no recibe `LibroCalificaciones.cpp`, por lo que desconoce cómo se implementan las funciones miembro de `LibroCalificaciones`.

El código cliente sólo necesita conocer la interfaz de `LibroCalificaciones` para usar la clase, y debe tener la capacidad de enlazar su código objeto. Como la interfaz de la clase es parte de su definición en el archivo de encabezado `LibroCalificaciones.h`, el programador del código cliente debe tener acceso a este archivo e incluirlo (mediante `#include`) en el archivo de código fuente del cliente. Cuando se compila el código cliente, el compilador usa la definición de la clase en `LibroCalificaciones.h` para asegurar que la función `main` cree y manipule objetos de la clase `LibroCalificaciones` correctamente.

Para crear la aplicación `LibroCalificaciones` ejecutable y que los instructores puedan usarla, el último paso es enlazar

1. el código objeto para la función `main` (es decir, el código cliente)
2. el código objeto para las implementaciones de las funciones miembro de la clase `LibroCalificaciones`.
3. el código objeto de la Biblioteca estándar de C++ para las clases de C++ (como `string`) que utilicen tanto el programador de la implementación de la clase, como el programador del código cliente.

La salida del enlazador es la aplicación *LibroCalificaciones* ejecutable, que los instructores pueden utilizar para administrar las calificaciones de los estudiantes.

Para obtener más información acerca de cómo compilar programas con varios archivos de código fuente, consulte la documentación de su compilador. En nuestro Centro de recursos de C++ en www.deitel.com/cplusplus/ proporcionamos vínculos a varios compiladores de C++.

3.10 Validación de datos mediante funciones *establecer*

En la sección 3.6 presentamos funciones *establecer* para permitir a los clientes de una clase modificar el valor de un miembro de datos *private*. En la figura 3.5, la clase *LibroCalificaciones* define la función miembro *establecerNombreCurso* sólo para asignar un valor recibido en su parámetro *nombre* al miembro de datos *nombreCurso*. Esta función miembro no asegura que el nombre del curso se adhiera a algún formato específico, o que siga cualquier otra regla en relación con la apariencia que debe tener un nombre de curso “válido”. Como dijimos antes, suponga que una universidad puede imprimir certificados de los estudiantes que contengan nombres de cursos con 25 caracteres o menos. Si la universidad utiliza un sistema que contenga objetos *LibroCalificaciones* para generar los certificados, tal vez sea conveniente que la clase *LibroCalificaciones* asegure que su miembro de datos *nombreCurso* nunca contenga más de 25 caracteres. El programa de las figuras 3.15 a 3.17 mejora la función miembro *establecerNombreCurso* de la clase *LibroCalificaciones* para realizar esta **validación** (lo que también se conoce como **verificación de validez**).

Definición de la clase *LibroCalificaciones*

Observe que la definición de la clase *LibroCalificaciones* (figura 3.15) (y por ende, su interfaz) es idéntica a la de la figura 3.11. Como la interfaz permanece sin cambios, los clientes de esta clase no necesitan modificarse a la hora que se modifica la definición de la función *establecerNombreCurso*. Esto permite a los clientes aprovechar la clase *LibroCalificaciones* mejorada, con sólo enlazar el código cliente con el código objeto actualizado de *LibroCalificaciones*.

```

1 // Fig. 3.15: LibroCalificaciones.h
2 // Definición de la clase LibroCalificaciones que presenta la interfaz pública de la
3 // clase. Las definiciones de las funciones miembro aparecen en LibroCalificaciones.cpp.
4 #include <string> // el programa usa la clase string estándar de C++
5 using std::string;
6
7 // definición de la clase LibroCalificaciones
8 class LibroCalificaciones
9 {
10 public:
11     LibroCalificaciones( string ); // constructor que inicializa un objeto LibroCalificaciones
12     void establecerNombreCurso( string ); // función que establece el nombre del curso
13     string obtenerNombreCurso(); // función que obtiene el nombre del curso
14     void mostrarMensaje(); // función que muestra un mensaje de bienvenida
15 private:
16     string nombreCurso; // nombre del curso para este LibroCalificaciones
17 };// fin de la clase LibroCalificaciones

```

Figura 3.15 | Definición de la clase *LibroCalificaciones*.

Validar el nombre del curso con la función miembro *establecerNombreCurso* de *LibroCalificaciones*

La mejora a la clase *LibroCalificaciones* está en la definición de *establecerNombreCurso* (figura 3.16, líneas 18 a 31). La instrucción *if* en las líneas 20 y 21 determina si el parámetro *nombre* contiene un nombre de curso válido (es decir, un *string* de 25 caracteres o menos). Si el nombre del curso es válido, en la línea 21 se almacena el nombre del curso en el miembro de datos *nombreCurso*. Observe la expresión *nombre.length()* en la línea 20. Ésta es una llamada a la función miembro, justo igual que *miLibroCalificaciones.mostrarMensaje*. La clase *string* de la Biblioteca estándar de C++ define a una función miembro *length* que devuelve el número de caracteres en un objeto *string*. El parámetro *nombre* es un objeto *string*, por lo que la llamada a *nombre.length()* devuelve el número de caracteres en *nombre*. Si este valor es menor o igual que 25, *nombre* es válido y la línea 21 se ejecuta.

La instrucción *if* en las líneas 23 a 30 se encarga del caso en el que *establecerNombreCurso* recibe un nombre de curso inválido (es decir, un nombre que tenga más de 25 caracteres). Aun si el parámetro *nombre* es demasiado largo,

de todas formas queremos que el objeto `LibroCalificaciones` quede en un **estado consistente**; es decir, un estado en el que el miembro de datos `nombreCurso` del objeto contenga un valor válido (un `string` de 25 caracteres o menos). Por ende, truncamos (reducimos) el nombre del curso especificado y asignamos los primeros 25 caracteres de `nombre` al miembro de datos `nombreCurso` (por desgracia, esto podría truncar el nombre del curso de una manera extraña). La clase `string` estándar proporciona la función miembro `substr` (“substring”, o subcadena), la cual devuelve un nuevo objeto `string` que se crea al copiar parte de un objeto `string` existente. La llamada en la línea 26 (es decir, `nombre.substr(0, 25)`) pasa dos enteros (0 y 25) a la función miembro `substr` de `nombre`. Estos argumentos indican la porción de la cadena `nombre` que `substr` debe devolver. El primer argumento especifica la posición inicial en el objeto `string` original desde el que se van a copiar los caracteres; en todas las cadenas se considera que el primer carácter se encuentra en la posición 0. El segundo argumento especifica el número de caracteres a copiar. Por lo tanto, la llamada en la línea 26 devuelve una subcadena de 25 caracteres de `nombre`, empezando en la posición 0 (es decir, los primeros 25 caracteres en `nombre`). Por ejemplo, si `nombre` contiene el valor “CS101 Introducción a la programación en C++”, `substr` devuelve “CS101 Introducción a la p”. Después de la llamada a `substr`, en la línea 26 se asigna la subcadena devuelta por `substr` al miembro de datos `nombreCurso`. De esta forma, la función miembro `establecerNombreCurso` asegura que a `nombreCurso` siempre se le asigne una cadena que contenga 25 caracteres o menos. Si la función miembro tiene que truncar el nombre del curso para hacerlo válido, en las líneas 28 y 29 se muestra un mensaje de advertencia.

Observe que si la instrucción `if` en las líneas 23 a 30 contiene dos instrucciones en su cuerpo (una para establecer el `nombreCurso` a los primeros 25 caracteres del parámetro `nombre` y una para imprimir un mensaje complementario al usuario). Deseamos que ambas instrucciones se ejecuten cuando `nombre` sea demasiado largo, por lo que las colocaremos en un par de llaves, `{ }`. En el capítulo 2 vimos que esto crea un bloque. En el capítulo 4 aprenderá más acerca de cómo colocar varias instrucciones en el cuerpo de una instrucción de control.

```

1 // Fig. 3.16: LibroCalificaciones.cpp
2 // Implementaciones de las definiciones de las funciones miembro de LibroCalificaciones.
3 // La función establecerNombreCurso realiza la validación.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "LibroCalificaciones.h" // incluye la definición de la clase LibroCalificaciones
9
10 // el constructor inicializa nombreCurso con la cadena que se suministra como argumento
11 LibroCalificaciones::LibroCalificaciones( string nombre )
12 {
13     establecerNombreCurso( nombre ); // valida y almacena nombreCurso
14 } // fin del constructor de LibroCalificaciones
15
16 // función que establece el nombre del curso;
17 // asegura que el nombre del curso tenga como máximo 25 caracteres
18 void LibroCalificaciones::establecerNombreCurso( string nombre )
19 {
20     if ( nombre.length() <= 25 ) // si nombre tiene 25 caracteres o menos
21         nombreCurso = nombre; // almacena el nombre del curso en el objeto
22
23     if ( nombre.length() > 25 ) // si nombre tiene más de 25 caracteres
24     {
25         // establece nombreCurso a los primeros 25 caracteres del parámetro nombre
26         nombreCurso = nombre.substr( 0, 25 ); // empieza en 0, longitud de 25
27
28         cout << "El nombre \'" << nombre << "\' excede la longitud maxima (25).\n"
29             << "Se limitó nombreCurso a los primeros 25 caracteres.\n" << endl;
30     } // fin de if
31 } // fin de la función establecerNombreCurso
32
33 // función para obtener el nombre del curso
34 string LibroCalificaciones::obtenerNombreCurso()

```

Figura 3.16 | Definiciones de las funciones miembro para la clase `LibroCalificaciones`, con una función `establecer` que valida la longitud del miembro de datos `nombreCurso`. (Parte I de 2).

```

35  {
36      return nombreCurso; // devuelve el nombreCurso del objeto
37  } // fin de la función obtenerNombreCurso
38
39 // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
40 void LibroCalificaciones::mostrarMensaje()
41 {
42     // llama a obtenerNombreCurso para obtener el nombreCurso
43     cout << "Bienvenido al libro de calificaciones para\n" << obtenerNombreCurso()
44     << "!" << endl;
45 } // fin de la función mostrarMensaje

```

Figura 3.16 | Definiciones de las funciones miembro para la clase `LibroCalificaciones`, con una función *establecer* que valida la longitud del miembro de datos `nombreCurso`. (Parte 2 de 2).

Observe que la instrucción en las líneas 28 y 29 también podría aparecer sin un operador de inserción de flujo al principio de la segunda línea de la instrucción, como en

```

cout << "El nombre \" " << nombre << "\" excede la longitud maxima (25). \n"
    "Se limito nombreCurso a los primeros 25 caracteres.\n" << endl;

```

El compilador de C++ combina las literales de cadena adyacentes, aun si aparecen en líneas separadas de un programa. Por ende, en la instrucción anterior, el compilador de C++ combinaría las literales de cadena "`"\\" excede la longitud maxima (25). \n"` y "`"Se limito nombreCurso a los primeros 25 caracteres.\n"`" en una sola literal de cadena que produzca una salida idéntica a la de las líneas 28 y 29 de la figura 3.16. Este comportamiento nos permite imprimir cadenas extensas, al descomponerlas en varias líneas en nuestro programa, sin necesidad de incluir operaciones de inserción de flujo adicionales.

Prueba de la clase `LibroCalificaciones`

En la figura 3.17 se demuestra la versión modificada de la clase `LibroCalificaciones` (figuras 3.15 a 3.16) que incluye la validación. En la línea 14 se crea un objeto `LibroCalificaciones` llamado `libroCalificaciones1`. Recuerde que el

```

1 // Fig. 3.17: fig03_16.cpp
2 // Crea y manipula un objeto LibroCalificaciones; ilustra la validación.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "LibroCalificaciones.h" // incluye la definición de la clase LibroCalificaciones
8
9 // la función main empieza la ejecución del programa
10 int main()
11 {
12     // crea dos objetos LibroCalificaciones;
13     // el nombre inicial del curso de libroCalificaciones1 es demasiado largo
14     LibroCalificaciones libroCalificaciones1( "CS101 Introducción a la programación en C++" );
15     LibroCalificaciones libroCalificaciones2( "CS102 C++:Estruct de datos" );
16
17     // muestra el nombreCurso de cada LibroCalificaciones
18     cout << "el nombre inicial del curso de libroCalificaciones1 es: "
19         << libroCalificaciones1.obtenerNombreCurso()
20         << "\nel nombre inicial del curso de libroCalificaciones2 es: "
21         << libroCalificaciones2.obtenerNombreCurso() << endl;
22
23     // modifica el nombreCurso de libroCalificaciones1 (con una cadena con longitud válida)
24     libroCalificaciones1.establecerNombreCurso( "CS101 Programación en C++" );
25

```

Figura 3.17 | Creación y manipulación de un objeto `LibroCalificaciones` en el que el nombre del curso está limitado a una longitud de 25 caracteres. (Parte 1 de 2).

```

26 // muestra el nombreCurso de cada LibroCalificaciones
27 cout << "\nel nombre del curso de libroCalificaciones1 es: "
28     << libroCalificaciones1.obtenerNombreCurso()
29     << "\nel nombre del curso de libroCalificaciones2 es: "
30     << libroCalificaciones2.obtenerNombreCurso() << endl;
31 return 0; // indica que terminó correctamente
32 } // fin de main

```

El nombre "CS101 Introduccion a la programacion en C++" excede la longitud maxima (25). Se limito nombreCurso a los primeros 25 caracteres.

el nombre inicial del curso de libroCalificaciones1 es: **CS101 Introduccion a la pro**
el nombre inicial del curso de libroCalificaciones2 es: CS102 C++:Estruc de datos

el nombre del curso de libroCalificaciones1 es: CS101 Programacion en C++
el nombre del curso de libroCalificaciones2 es: CS102 C++:Estruc de datos

Figura 3.17 | Creación y manipulación de un objeto *LibroCalificaciones* en el que el nombre del curso está limitado a una longitud de 25 caracteres. (Parte 2 de 2).

constructor de *LibroCalificaciones* llama a *establecerNombreCurso* para inicializar el miembro de datos *nombreCurso*. En versiones anteriores de la clase, el beneficio de llamar a *establecerNombreCurso* en el constructor no era evidente. Sin embargo, ahora el constructor aprovecha la validación que proporciona *establecerNombreCurso*. El constructor simplemente llama a *establecerNombreCurso*, en lugar de duplicar su código de validación. Cuando en la línea 14 de la figura 3.17 se pasa un nombre inicial para el curso de "CS101 Introduccion a la programacion en C++" al constructor de *LibroCalificaciones*, el constructor pasa este valor a *establecerNombreCurso*, donde ocurre la inicialización en sí. Como este nombre contiene más de 25 caracteres, se ejecuta el cuerpo de la segunda instrucción *if*, lo cual hace que *nombreCurso* se inicialice con el nombre truncado del curso de 25 caracteres de "CS101 Introduccion a la p" (la parte truncada se resalta con rojo en la línea 14). Observe que la salida en la figura 3.17 contiene el mensaje de advertencia que producen las líneas 28 y 29 de la figura 3.16 en la función miembro *establecerNombreCurso*. En la línea 15 se crea otro objeto *LibroCalificaciones* llamado *libroCalificaciones2*; el nombre válido del curso que se pasa al constructor es exactamente de 25 caracteres.

En las líneas 18 a 21 de la figura 3.17 se muestra el nombre del curso truncado para *libroCalificaciones1* [resaltamos esto en negritas en la salida del programa (en rojo en su pantalla)] y el nombre del curso para *libroCalificaciones2*. En la línea 24 se hace una llamada a la función miembro *establecerNombreCurso* de *libroCalificaciones1* directamente, para modificar el nombre del curso en el objeto *LibroCalificaciones* a un nombre más corto, que no necesite truncarse. Después, en las líneas 17 a 30 se imprimen los nombres de los cursos para los objetos *LibroCalificaciones* de nuevo.

Observaciones adicionales acerca de las funciones establecer

Una función *establecer* pública tal como *establecerNombreCurso* debe escudriñar cuidadosamente cualquier intento por modificar el valor de un miembro de datos (por ejemplo, *nombreCurso*) para asegurar que el nuevo valor sea apropiado para ese elemento de datos. Por ejemplo, un intento por *establecer* el día del mes en 37 debe rechazarse, un intento por *establecer* el peso de una persona en cero o en un valor negativo debe rechazarse, un intento por *establecer* una calificación en un examen en 185 (cuando el rango apropiado es de cero a 100) debe rechazarse, y así en lo sucesivo.



Observación de Ingeniería de Software 3.6

Hacer los miembros de datos private y controlar el acceso, en especial el acceso de escritura, a esos miembros de datos a través de funciones miembro public, ayuda a asegurar la integridad de los datos.



Tip para prevenir errores 3.5

Los beneficios de la integridad de datos no son automáticos sólo porque los miembros de datos se hacen private; debemos proporcionar una verificación de validez apropiada y reportar los errores.



Observación de Ingeniería de Software 3.7

Las funciones miembro que establecen los valores de los datos private deben verificar que los nuevos valores deseados sean apropiados; si no lo son, las funciones establecer deben colocar los miembros de datos private en un estado apropiado.

Las funciones *establecer* de una clase pueden devolver valores a los clientes de la clase, indicando que se realizaron intentos por asignar datos inválidos a los objetos de la clase. Un cliente de la clase puede probar el valor de retorno de una función *establecer* para determinar si el intento del cliente por modificar el objeto fue exitoso, y para realizar la acción apropiada. En el capítulo 16 demostraremos cómo se puede notificar a los clientes de una clase a través del mecanismo de manejo de excepciones, cuando se hace un intento por modificar un objeto con un valor inapropiado. Para mantener simple el programa de las figuras 3.15 a 3.17 en este punto en el libro, `establecerNombreCurso` en la figura 3.16 sólo imprime un mensaje apropiado en la pantalla.

3.11 (Opcional) Ejemplo práctico de Ingeniería de Software: identificación de las clases en la especificación de requerimientos del ATM

Ahora empezaremos a diseñar el sistema ATM que presentamos en el capítulo 2. En esta sección identificaremos las clases necesarias para crear el sistema ATM, analizando los sustantivos y las frases nominales que aparecen en la especificación de requerimientos. Presentaremos los diagramas de clases de UML para modelar las relaciones entre estas clases. Este primer paso es importante para definir la estructura de nuestro sistema.

Identificación de las clases en un sistema

Para comenzar nuestro proceso de DOO, vamos a identificar las clases requeridas para crear el sistema ATM. Más adelante describiremos estas clases mediante el uso de los diagramas de clases de UML y las implementaremos en C++. Primero debemos revisar la especificación de requerimientos de la sección 2.8 e identificar los sustantivos y frases nominales clave que nos ayuden a identificar las clases que conformarán el sistema ATM. Tal vez decidamos que algunos de estos sustantivos y frases nominales sean atributos de otras clases en el sistema. Tal vez también concluyamos que algunos de los sustantivos no corresponden a ciertas partes del sistema y, por ende, no deben modelarse. A medida que avancemos por el proceso de diseño podemos ir descubriendo clases adicionales.

En la figura 3.18 se enlistan los sustantivos y frases nominales que se encontraron en la especificación de requerimientos. Los enlistamos de izquierda a derecha en el orden en que aparecen en la especificación de requerimientos. Sólo enlistaremos la forma singular de cada sustantivo o frase nominal.

Vamos a crear clases sólo para los sustantivos y frases nominales que tengan importancia en el sistema ATM. No modelamos “banco” como una clase, ya que el banco no es una parte del sistema ATM; el banco sólo quiere que nosotros construyamos el ATM. “Cliente” y “usuario” también representan entidades fuera del sistema; son importantes debido a que interactúan con nuestro sistema ATM, pero no necesitamos modelarlos como clases en el software del ATM. Recuerde que modelamos un usuario del ATM (es decir, un cliente del banco) como el actor en el diagrama de caso de uso de la figura 2.18.

No necesitamos modelar “billete de \$20” ni “sobre de depósito” como clases. Éstos son objetos físicos en el mundo real, pero no forman parte de lo que se va a automatizar. Podemos representar en forma adecuada la presencia de billetes en el sistema, mediante el uso de un atributo de la clase que modela el dispensador de efectivo (en la sección 4.13 asignaremos atributos a las clases). Por ejemplo, el dispensador de efectivo mantiene un conteo del número de billetes que contiene. La especificación de requerimientos no dice nada acerca de lo que debe hacer el sistema con los sobres de depósito después de recibirlos. Podemos suponer que con sólo admitir la recepción de un sobre (una operación que realiza la clase que modela la ranura de depósito) es suficiente para representar la presencia de un sobre en el sistema (en la sección 6.22 asignaremos operaciones a las clases).

Sustantivos y frases nominales en la especificación de requerimientos		
banco	dinero / fondos	número de cuenta
ATM	pantalla NIP	
usuario	teclado numérico	base de datos del banco
cliente	dispensador de efectivo	solicitud de saldo
transacción	billete de \$20 / efectivo	retiro
cuenta	ranura de depósito	depósito
saldo	sobre de depósito	

Figura 3.18 | Sustantivos y frases nominales en la especificación de requerimientos.

En nuestro sistema ATM simplificado, que representa varios montos de “dinero” (incluyendo el “saldo” de una cuenta) como atributos de otras clases, parece ser lo más apropiado. De igual forma, los sustantivos “número de cuenta” y “NIP” representan piezas importantes de información en el sistema ATM. Son atributos importantes de una cuenta bancaria. Sin embargo, no exhiben comportamientos. Por ende, podemos modelarlos de la manera más apropiada como atributos de una clase de cuenta.

Aunque, con frecuencia, la especificación de requerimientos describe una “transacción” en un sentido general, no modelaremos la amplia noción de una transacción financiera en este momento. En lugar de ello, modelaremos los tres tipos de transacciones (es decir, “solicitud de saldo”, “retiro” y “depósito”) como clases individuales. Estas clases poseen los atributos específicos necesarios para ejecutar las transacciones que representan. Por ejemplo, para un retiro se necesita conocer el monto de dinero que el usuario desea retirar. Sin embargo, una solicitud de saldo no requiere datos adicionales. Lo que es más, las tres clases de transacciones exhiben comportamientos únicos. Para un retiro se requiere entregar efectivo al usuario, mientras que para un depósito se requiere recibir sobres de depósito del usuario. [Nota: en la sección 13.10, “factorizaremos” las características comunes de todas las transacciones en una clase de “transacción” general, mediante el uso de los conceptos orientados a objetos de las clases abstractas y la herencia.]

Vamos a determinar las clases para nuestro sistema con base en los sustantivos y frases nominales de las frases restantes de la figura 3.18. Cada una de ellas se refiere a uno o varios de los siguientes elementos:

- ATM
- pantalla
- teclado numérico
- dispensador de efectivo
- ranura de depósito
- cuenta
- base de datos del banco
- solicitud de saldo
- retiro
- depósito

Es probable que los elementos de esta lista sean clases que necesitaremos implementar en nuestro sistema.

Ahora podemos modelar las clases en nuestro sistema, con base en la lista que hemos creado. En el proceso de diseño escribimos los nombres de las clases con la primera letra en mayúscula (una convención de UML), como lo haremos cuando escribamos el código de C++ para implementar nuestro diseño. Si el nombre de una clase contiene más de una palabra, juntaremos todas las palabras y escribiremos la primera letra de cada una de ellas en mayúscula (por ejemplo, `NombreConVariasPalabras`). Utilizando esta convención, vamos a crear las clases `ATM`, `Pantalla`, `Teclado`, `DispensadorEfectivo`, `RanuraDeposito`, `Cuenta`, `BaseDatosBanco`, `SolicitudSaldo`, `Retiro` y `Deposito`. Construiremos nuestro sistema mediante el uso de todas estas clases como bloques de construcción. Sin embargo, antes de empezar a construir el sistema, debemos comprender mejor la forma en que las clases se relacionan entre sí.

Modelado de las clases

UML nos permite modelar, a través de los **diagramas de clases**, las clases en el sistema ATM y sus interrelaciones. La figura 3.19 representa a la clase `ATM`. En UML, cada clase se modela como un rectángulo con tres compartimentos. El compartimiento superior contiene el nombre de la clase, centrado horizontalmente y en negrita. El compartimiento intermedio contiene los atributos de la clase (en las secciones 4.13 y 5.11 hablaremos sobre los atributos). El compartimiento inferior contiene las operaciones de la clase (que veremos en la sección 6.22). En la figura 3.19, los compartimientos intermedio e inferior están vacíos, ya que no hemos determinado los atributos y operaciones de esta clase todavía.

Los diagramas de clases también muestran las relaciones entre las clases del sistema. En la figura 3.20 se muestra cómo nuestras clases `ATM` y `Retiro` se relacionan una con la otra. Por el momento vamos a modelar sólo este subconjunto de las clases del ATM, por cuestión de simplicidad; más adelante en esta sección, presentaremos un diagrama de clases más completo. Observe que los rectángulos que representan a las clases en este diagrama no están subdivididos en compartimentos. UML permite suprimir los atributos de las clases y sus operaciones de esta forma, cuando sea apropiado, para crear diagramas más legibles. Un diagrama de este tipo se denomina **diagrama con elementos omitidos (elided diagram)**: su información, tal como el contenido de los compartimentos segundo y tercero, no se modela. En las secciones 4.13 y 6.22 colocaremos información en estos compartimentos.



Figura 3.19 | Representación de una clase en UML mediante un diagrama de clases.

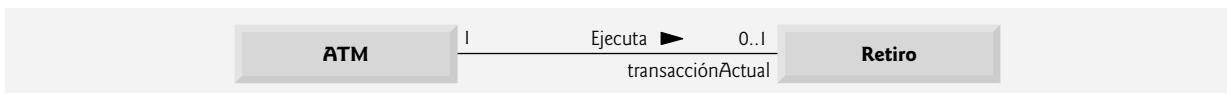


Figura 3.20 | Diagrama de clases que muestra una asociación entre clases.

En la figura 3.20, la línea sólida que conecta a las dos clases representa una **asociación**: una relación entre clases. Los números cerca de cada extremo de la línea son valores de **multiplicidad**; éstos indican cuántos objetos de cada clase participan en la asociación. En este caso, al seguir la línea de un extremo al otro se revela que, en un momento dado, un objeto ATM participa en una asociación con cero o con un objeto Retiro; cero si el usuario actual no está realizando una transacción o si ha solicitado un tipo distinto de transacción, y uno si el usuario ha solicitado un retiro. UML puede modelar muchos tipos de multiplicidad. En la figura 3.21 se enlistan y explican los tipos de multiplicidad.

Una asociación puede tener nombre. Por ejemplo, la palabra Ejecuta por encima de la línea que conecta a las clases ATM y Retiro en la figura 3.20 indica el nombre de esa asociación. Esta parte del diagrama se lee así: “un objeto de la clase ATM ejecuta cero o un objeto de la clase Retiro”. Los nombres de las asociaciones son direccionales, como lo indica la punta de flecha rellena; por lo tanto, sería inapropiado, por ejemplo, leer la anterior asociación de derecha a izquierda como “cero o un objeto de la clase Retiro ejecuta un objeto de la clase ATM”.

La palabra transaccionActual en el extremo de Retiro de la línea de asociación en la figura 3.20 es un **nombre de rol**, el cual identifica el rol que desempeña el objeto Retiro en su relación con el ATM. Un nombre de rol agrega significado a una asociación entre clases, ya que identifica el rol que desempeña una clase dentro del contexto de una asociación. Una clase puede desempeñar varios roles en el mismo sistema. Por ejemplo, en un sistema de personal de una universidad, una persona puede desempeñar el rol de “profesor” respecto a los estudiantes. La misma persona puede desempeñar el rol de “colega” cuando participa en una asociación con otro profesor, y de “entrenador” cuando entrena a los atletas estudiantes. En la figura 3.20, el nombre de rol transaccionActual indica que el objeto Retiro que participa en la asociación Ejecuta con un objeto de la clase ATM representa a la transacción que está procesando el ATM en ese momento. En otros contextos, un objeto Retiro puede desempeñar otros roles (por ejemplo, la transacción anterior). Observe que no especificamos un nombre de rol para el extremo del ATM de la asociación Ejecuta. A menudo, los nombres de los roles se omiten en los diagramas de clases, cuando el significado de una asociación está claro sin ellos.

Símbolo	Significado
0	Ninguno
1	Uno
m	Un valor entero
0..1	Cero o uno
m, n	m o n
$m..n$	Cuando menos m , pero no más que n
*	Cualquier entero no negativo (cero o más)
0..*	Cero o más (idéntico a *)
1..*	Uno o más

Figura 3.21 | Tipos de multiplicidad.

Además de indicar relaciones simples, las asociaciones pueden especificar relaciones más complejas, como cuando los objetos de una clase están compuestos de objetos de otras clases. Considere un cajero automático real. ¿Qué “piezas” reúne un fabricante para construir un ATM funcional? Nuestra especificación de requerimientos nos indica que el ATM está compuesto de una pantalla, un teclado, un dispensador de efectivo y una ranura de depósito.

En la figura 3.22, los **diamantes sólidos** que se adjuntan a las líneas de asociación de la clase ATM indican que esta clase tiene una relación de **composición** con las clases Pantalla, Teclado, DispensadorEfectivo y RanuraDeposito. La composición implica una relación en todo/en parte. La clase que tiene el símbolo de composición (el diamante sólido) en su extremo de la línea de asociación es el todo (en este caso, ATM), y las clases en el otro extremo de las líneas de asociación son las partes; en este caso, las clases Pantalla, Teclado, DispensadorEfectivo y RanuraDeposito. Las composiciones en la figura 3.22 indican que un objeto de la clase ATM está formado por un objeto de la clase Pantalla, un objeto de la clase DispensadorEfectivo, un objeto de la clase Teclado y un objeto de la clase RanuraDeposito. El ATM “tiene una” pantalla, un teclado, un dispensador de efectivo y una ranura de depósito. La **relación “tiene un”** define la composición (en la sección del Ejemplo práctico de Ingeniería de Software del capítulo 13 veremos que la relación “*es un*” define la herencia).

De acuerdo con la especificación de UML, las relaciones de composición tienen las siguientes propiedades:

1. Sólo una clase en la relación puede representar el todo (es decir, el diamante puede colocarse sólo en un extremo de la línea de asociación). Por ejemplo, la pantalla es parte del ATM o el ATM es parte de la pantalla, pero la pantalla y el ATM no pueden representar ambos el “todo” dentro de la relación.
2. Las partes en la relación de composición existen sólo mientras exista el todo, y el todo es responsable de la creación y destrucción de sus partes. Por ejemplo, el acto de construir un ATM incluye la manufactura de sus partes. Lo que es mas, si el ATM se destruye, también se destruyen su pantalla, teclado, dispensador de efectivo y ranura de depósito.
3. Una parte puede pertenecer sólo a un todo a la vez, aunque esa parte puede quitarse y unirse a otro todo, el cual entonces asumirá la responsabilidad de esa parte.

Los diamantes sólidos en nuestros diagramas de clases indican las relaciones de composición que cumplen con estas tres propiedades. Si una relación “tiene un” no satisface uno o más de estos criterios, UML especifica que se deben adjuntar diamantes sin relleno a los extremos de las líneas de asociación para indicar una **agregación**: una forma más débil de la composición. Por ejemplo, una computadora personal y un monitor de computadora participan en una relación de agregación: la computadora “tiene un” monitor, pero las dos partes pueden existir en forma independiente, y el mismo monitor puede conectarse a varias computadoras a la vez, con lo cual se violan las propiedades segunda y tercera de la composición.

La figura 3.23 muestra un diagrama de clases para el sistema ATM. Este diagrama modela la mayoría de las clases que identificamos antes en esta sección, así como las asociaciones entre ellas que podemos inferir de la especificación de requerimientos. [Nota: las clases SolicitudSaldo y Deposito participan en asociaciones similares a las de la clase Retiro, por lo que preferimos omitirlas en este diagrama por cuestión de simpleza. En el capítulo 13 expandiremos nuestro diagrama de clases para incluir todas las clases en el sistema ATM.]

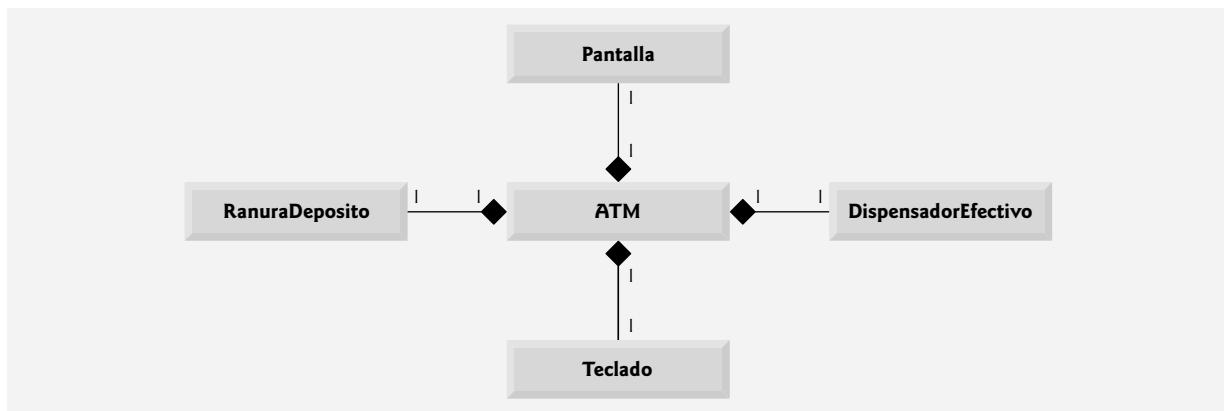


Figura 3.22 | Diagrama de clases que muestra las relaciones de composición.

La figura 3.23 presenta un modelo gráfico de la estructura del sistema ATM. Este diagrama de clases incluye a las clases **BaseDatosBanco** y **Cuenta**, junto con varias asociaciones que no presentamos en las figuras 3.20 o 3.22. El diagrama de clases muestra que la clase **ATM** tiene una **relación de uno a uno** con la clase **BaseDatosBanco**: un objeto ATM autentica a los usuarios en base a un objeto **BaseDatosBanco**. En la figura 3.23 también modelamos el hecho de que la base de datos del banco contiene información sobre muchas cuentas; un objeto de la clase **BaseDatosBanco** participa en una relación de composición con cero o más objetos de la clase **Cuenta**. Recuerde que en la figura 3.21 se muestra que el valor de multiplicidad $0..*$ en el extremo de la clase **Cuenta**, de la asociación entre las clases **BaseDatosBanco** y **Cuenta**, indica que cero o más objetos de la clase **Cuenta** participan en la asociación. La clase **BaseDatosBanco** tiene una **relación de uno a varios** con la clase **Cuenta**; **BaseDatosBanco** puede contener muchos objetos **Cuenta**. De manera similar, la clase **Cuenta** tiene una **relación de varios a uno** con la clase **BaseDatosBanco**; puede haber muchos objetos **Cuenta** en **BaseDatosBanco**. [Nota: si recuerda la figura 3.21, el valor de multiplicidad * es idéntico a $0..*$. Incluimos $0..*$ en nuestros diagramas de clases por cuestión de claridad.]

La figura 3.23 también indica que si el usuario va a realizar un retiro, “un objeto de la clase **Retiro** accede a/modifica un saldo de cuenta a través de un objeto de la clase **BaseDatosBanco**”. Podríamos haber creado una asociación directamente entre la clase **Retiro** y la clase **Cuenta**. No obstante, la especificación de requerimientos indica que el “ATM debe interactuar con la base de datos de información de las cuentas del banco” para realizar transacciones. Una cuenta de banco contiene información delicada, por lo que los ingenieros de sistemas deben considerar siempre la seguridad de los datos personales al diseñar un sistema. Por ello, sólo **BaseDatosBanco** puede acceder a una cuenta y manipularla en forma directa. Todas las demás partes del sistema deben interactuar con la base de datos para obtener o actualizar la información de las cuentas (por ejemplo, el saldo de una cuenta).

El diagrama de clases de la figura 3.23 también modela las asociaciones entre la clase **Retiro** y las clases **Pantalla**, **DispensadorEfectivo** y **Teclado**. Una transacción de retiro implica pedir al usuario que seleccione el monto a retirar; también implica recibir entrada numérica. Estas acciones requieren el uso de la pantalla y del teclado, respectivamente. Además, para entregar efectivo al usuario se requiere acceso al dispensador de efectivo.

Aunque no se muestran en la figura 3.23, las clases **SolicitudSaldo** y **Depósito** participan en varias asociaciones con las otras clases del sistema ATM. Al igual que la clase **Retiro**, cada una de estas clases se asocia con las clases **ATM** y **BaseDatosBanco**. Un objeto de la clase **SolicitudSaldo** también se asocia con un objeto de la clase **Pantalla** para mostrar al usuario el saldo de una cuenta. La clase **Depósito** se asocia con las clases **Pantalla**, **Teclado** y **RanuraDepósito**. Al igual que los retiros, las transacciones de depósito requieren el uso de la pantalla y el teclado para mostrar

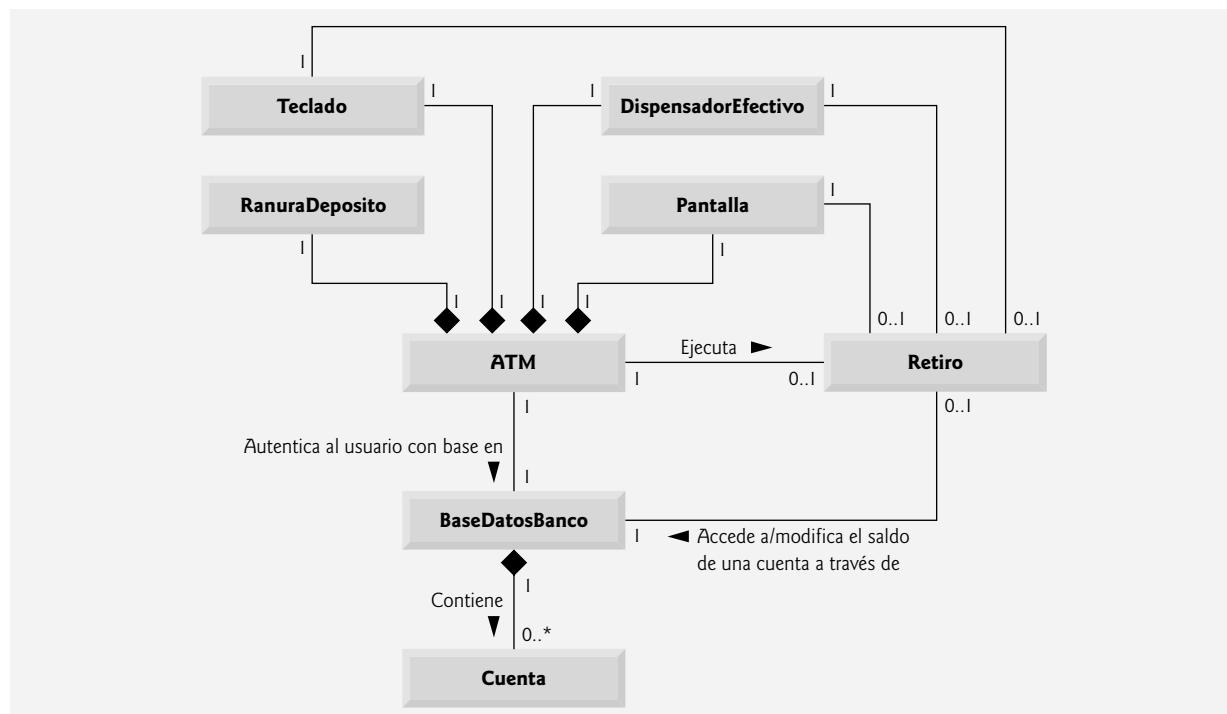


Figura 3.23 | Diagrama de clases para el modelo del sistema ATM.

mensajes y recibir datos de entrada, respectivamente. Para recibir sobres de depósito, un objeto de la clase `Deposito` accede a la ranura de depósitos.

Ya hemos identificado las clases en nuestro sistema ATM (aunque tal vez descubramos otras, a medida que avanzemos con el diseño y la implementación). En la sección 4.13 vamos a determinar los atributos para cada una de estas clases, y en la sección 5.11 utilizaremos estos atributos para examinar la forma en que cambia el sistema con el tiempo. En la sección 6.22, determinaremos las operaciones de las clases en nuestro sistema.

Ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

3.1 Suponga que tenemos una clase llamada `Auto`, la cual representa a un auto. Piense en algunas de las distintas piezas que podría reunir un fabricante para producir un auto completo. Cree un diagrama de clases (similar a la figura 3.22) que modele algunas de las relaciones de composición de la clase `Auto`.

3.2 Suponga que tenemos una clase llamada `Archivo`, la cual representa un documento electrónico en una computadora independiente, sin conexión de red, representada por la clase `Computadora`. ¿Qué tipo de asociación existe entre la clase `Computadora` y la clase `Archivo`?

- La clase `Computadora` tiene una relación de uno a uno con la clase `Archivo`.
- La clase `Computadora` tiene una relación de varios a uno con la clase `Archivo`.
- La clase `Computadora` tiene una relación de uno a varios con la clase `Archivo`.
- La clase `Computadora` tiene una relación de varios a varios con la clase `Archivo`.

3.3 Indique si la siguiente aseveración es *verdadera* o *falsa*. Si es *falsa*, explique por qué: un diagrama de clases de UML, en el que no se modelan los comportamientos segundo y tercero de una clase, se denomina diagrama con elementos omitidos (elided diagram).

3.4 Modifique el diagrama de clases de la figura 3.23 para incluir la clase `Deposito`, en lugar de la clase `Retiro`.

Respuestas a los ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

3.1 [Nota: las respuestas de los estudiantes pueden variar.] La figura 3.24 presenta un diagrama de clases que muestra algunas de las relaciones de composición de una clase `Auto`.

3.2 c. [Nota: en una computadora con conexión de red, esta relación podría ser de varios a varios.]

3.3 Verdadera.

3.4 La figura 3.25 presenta un diagrama de clases para el ATM, en el cual se incluye la clase `Deposito` en lugar de la clase `Retiro` (como en la figura 3.23). Observe que la clase `Deposito` no se asocia con la clase `DispensadorEfectivo`, sino que se asocia con la clase `RanuraDeposito`.

3.12 Repaso

En este capítulo aprendió a crear clases definidas por el usuario, y a crear y utilizar objetos de esas clases. Declaramos miembros de datos de una clase para mantener los datos para cada objeto de la misma. También definimos funciones miembro que operan con esos datos. Aprendió a llamar a las funciones miembro de un objeto para solicitar los servicios que éste proporciona, y cómo pasar datos a esas funciones miembro como argumentos. Hablamos sobre la dife-

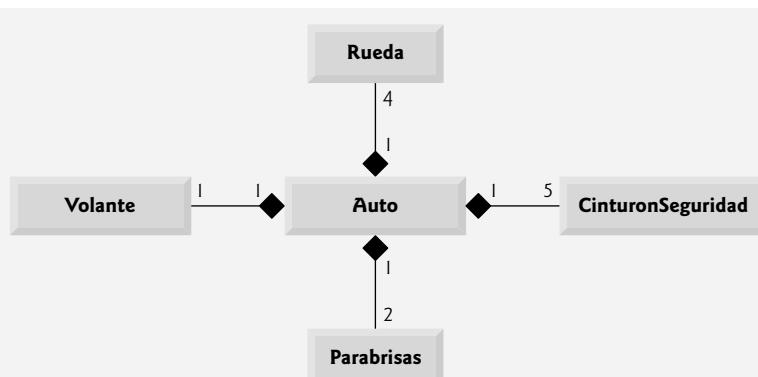


Figura 3.24 | Diagrama de clases que muestra algunas relaciones de composición de una clase `Auto`.

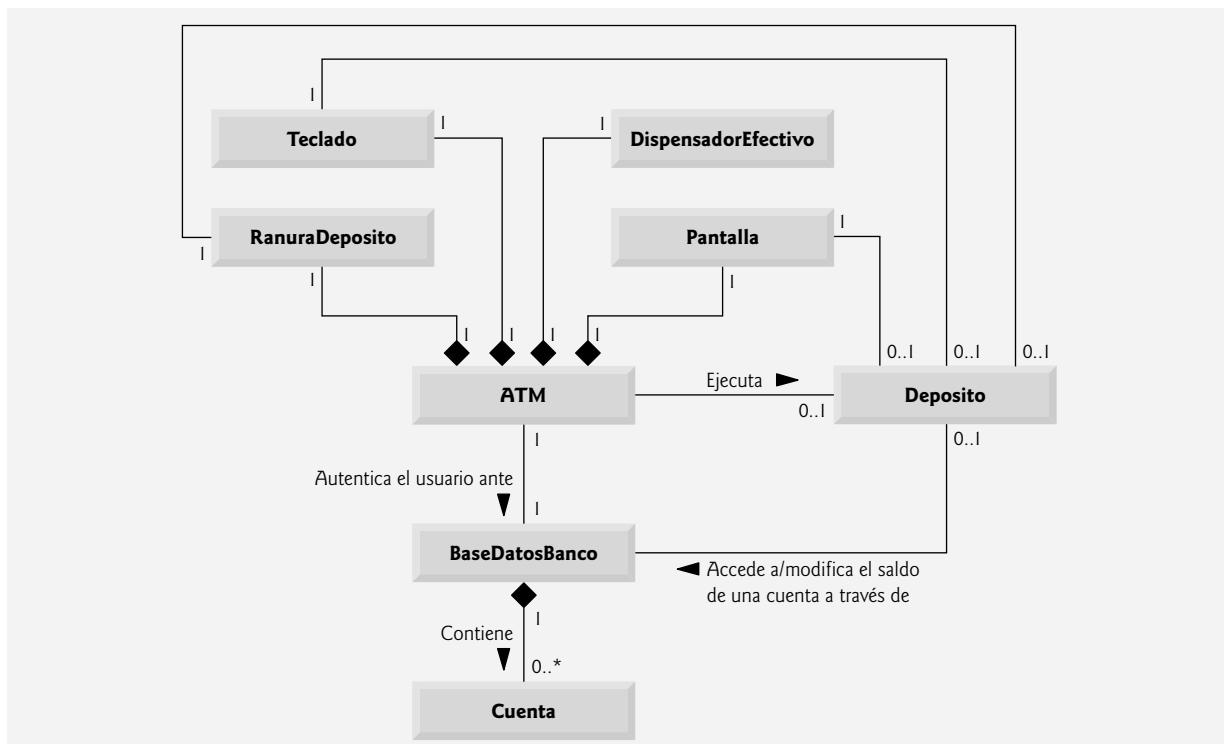


Figura 3.25 | Diagrama de clases para el modelo del sistema ATM, incluyendo la clase **Depósito**.

rencia entre una variable local de una función miembro y un miembro de datos de una clase. También le mostramos cómo usar un constructor para especificar los valores iniciales para los miembros de datos de un objeto. Aprendió a separar la interfaz de una clase de su implementación, para fomentar la buena ingeniería de software. Presentamos un diagrama que muestra los archivos que necesitan los programadores de la implementación de la clase y los programadores del código cliente para compilar el código que escriben. Demostramos cómo se pueden utilizar las funciones *establecer* para validar los datos de un objeto y asegurar que los objetos se mantengan en un estado consistente. Además, se utilizaron diagramas de clases de UML para modelar las clases y sus constructores, las funciones miembro y los miembros de datos. En el siguiente capítulo empezaremos nuestra introducción a las instrucciones de control, las cuales especifican el orden en el que se realizan las acciones de una función.

Resumen

Sección 3.2 Clases, objetos, funciones miembro y miembros de datos

- Para realizar una tarea en un programa se requiere una función. La función oculta al usuario las tareas complejas que realiza.
- Una función en una clase se conoce como función miembro, y realiza una de las tareas de esa clase.
- Para que un programa pueda realizar las tareas que describe la clase, debemos crear un objeto de esa clase. Ésta es una de las razones por las que C++ se conoce como lenguaje de programación orientado a objetos.
- Cada mensaje que se envía a un objeto es una llamada a una función miembro, la cual le indica al objeto que realice una tarea.
- Un objeto tiene atributos que se acarrean con el objeto, a medida que éste se utiliza en un programa. Estos atributos se especifican como miembros de datos en la clase del objeto.

Sección 3.4 Definición de una clase con una función miembro

- La definición de una clase contiene los datos miembro y las funciones miembro que definen los atributos y comportamientos de esa clase, respectivamente.

- La definición de una clase empieza con la palabra clave `class`, seguida inmediatamente por el nombre de la clase.
- Por convención, el nombre de una clase definida por el usuario empieza con una letra mayúscula, y por legibilidad, cada palabra subsiguiente en el nombre de la clase empieza con una letra mayúscula.
- El cuerpo de cada clase va encerrado entre un par de llaves izquierda y derecha (`{` y `}`), y termina con un punto y coma.
- Las funciones miembro que aparecen después del especificador de acceso `public` pueden ser llamadas por otras funciones en un programa, y por las funciones miembro de otras clases.
- Los especificadores de acceso siempre van seguidos de un punto y coma (`;`).
- La palabra clave `void` es un tipo de valor de retorno especial, el cual indica que una función realizará una tarea, pero no devolverá datos a la función que la llamó cuando complete su tarea.
- Por convención, los nombres de las funciones empiezan con la primera letra en minúscula, y todas las palabras subsiguientes en el nombre empiezan con letra mayúscula.
- Un conjunto vacío de paréntesis después del nombre de una función indica que ésta no requiere datos adicionales para realizar su tarea.
- El cuerpo de toda función está delimitado por las llaves izquierda y derecha (`{` y `}`).
- Por lo general, no se puede llamar a una función miembro sino hasta que se crea un objeto de su clase.
- Cada nueva clase que creamos se convierte en un nuevo tipo en C++, que se puede utilizar para declarar variables y crear objetos. Ésta es una razón por la que C++ se conoce como lenguaje extensible.
- En UML, cada clase se modela en un diagrama de clases como un rectángulo con tres compartimientos. El compartimiento superior contiene el nombre de la clase, centrado horizontalmente y en negrita. El compartimiento intermedio contiene los atributos de la clase (miembros de datos en C++). El compartimiento inferior contiene las operaciones de la clase (funciones miembro y constructores en C++).
- Para modelar las operaciones, UML enlista el nombre de la operación, seguido de un conjunto de paréntesis. Un signo más (+) enfrente del nombre de la operación indica que ésta es una operación `public` en UML (es decir, una función miembro `public` en C++).

Sección 3.5 Definición de una función miembro con un parámetro

- Una función miembro puede requerir uno o más parámetros para representar los datos adicionales que necesita para realizar su tarea. La llamada a una función suministra los argumentos para cada uno de los parámetros de esa función.
- Para llamar a una función miembro, se coloca después del nombre del objeto un operador punto (`.`), el nombre de la función y un conjunto de paréntesis que contienen los argumentos de la misma.
- Una variable de la clase `string` de la Biblioteca estándar de C++ representa a una cadena de caracteres. Esta clase se define en el archivo de encabezado `<string>`, y el nombre `string` pertenece al espacio de nombres `std`.
- La función `getline` (del encabezado `<string>`) lee caracteres de su primer argumento hasta encontrar un carácter de nueva línea, y después coloca los caracteres (sin incluir la nueva línea) en la variable `string` que se especifica como su segundo argumento. El carácter de nueva línea se descarta.
- Una lista de parámetros puede contener cualquier número de parámetros, incluyendo ninguno (lo cual se representa por paréntesis vacíos) para indicar que una función no requiere parámetros.
- El número de argumentos en la llamada a una función debe coincidir con el número de parámetros en la lista de parámetros del encabezado de la función miembro a la que se llamó. Además, los tipos de los argumentos en la llamada a la función deben ser consistentes con los tipos de los parámetros correspondientes en el encabezado de la función.
- Para modelar un parámetro de una operación, UML enlista el nombre del parámetro, seguido de dos puntos y del tipo del parámetro entre los paréntesis que van después del nombre de la operación.
- UML tiene sus propios tipos de datos. No todos los tipos de datos de UML tienen los mismos nombres que los tipos de C++ correspondientes. El tipo `String` de UML corresponde al tipo `string` de C++.

Sección 3.6 Miembros de datos, funciones establecer y funciones obtener

- Las variables que se declaran en el cuerpo de una función son variables locales y sólo pueden utilizarse desde el punto en el que se declararon en la función, hasta la llave de cierre derecha (`}`) correspondiente. Cuando termina una función, se pierden los valores de sus variables locales.
- Una variable local debe declararse antes de poder utilizarse en una función. Una variable local no puede utilizarse fuera de la función en la que está declarada.
- Por lo general, los miembros de datos son `private`. Las variables o funciones que se declaran `private` son accesibles sólo para las funciones miembro de la clase en la que están declaradas, o para las funciones amigas de la clase.
- Cuando un programa crea (instancia) un objeto de una clase, sus miembros de datos `private` se encapsulan (ocultan) en el objeto y sólo las funciones miembro de la clase del objeto pueden utilizarlos.
- Cuando se hace una llamada a una función que especifica un tipo de valor de retorno distinto de `void` y ésta completa su tarea, la función devuelve un resultado a la función que la llamó.

- De manera predeterminada, el valor inicial de un objeto `string` es la cadena vacía; es decir, una cadena que no contenga caracteres. No aparece nada en la pantalla cuando se muestra una cadena vacía.
- A menudo, las clases proporcionan funciones miembro `public` para permitir que los clientes de la clase *establezcan u obtengan* miembros de datos privados. Los nombres de esas funciones miembro comúnmente empiezan con *establecer u obtener* (*set o get*, en inglés).
- Las funciones *establecer y obtener* permiten a los clientes de una clase utilizar de manera indirecta los datos ocultos. El cliente no sabe cómo realiza el objeto estas operaciones.
- Las funciones *establecer y obtener* de una clase deben ser utilizadas por otras funciones miembro de la clase para manipular los datos `private` de esa clase. Si la representación de los datos de la clase cambia, las funciones miembro que acceden a los datos sólo a través de las funciones *establecer y obtener* no requerirán modificación.
- Una función `set` pública debe escudriñar cuidadosamente cualquier intento por modificar el valor de un miembro de datos, para asegurar que el nuevo valor sea apropiado para ese elemento de datos.
- UML representa a los miembros de datos como atributos, para lo cual enlista el nombre del atributo, seguido de dos puntos y del tipo del atributo. En UML, se coloca un signo menos (-) antes de los atributos privados.
- Para indicar el tipo de valor de retorno de una operación en UML, se coloca un signo de dos puntos y el tipo de valor de retorno después de los paréntesis que siguen del nombre de la operación.
- Los diagramas de clases de UML no especifican tipos de valores de retorno para las operaciones que no devuelven valores.

Sección 3.7 Inicialización de objetos mediante constructores

- Cada clase debe proporcionar un constructor para inicializar un objeto de la clase cuando el objeto se crea. Un constructor se debe definir con el mismo nombre que la clase.
- Una diferencia entre los constructores y las funciones es que los constructores no pueden devolver valores, por lo que no pueden especificar un tipo de valor de retorno (ni siquiera `void`). Por lo general, los constructores se declaran como `public`.
- C++ requiere la llamada a un constructor al momento en que se crea un objeto, lo cual ayuda a asegurar que todo objeto se inicialice antes de usarlo en un programa.
- Un constructor sin parámetros es un constructor predeterminado. Si el programador no proporciona un constructor, el compilador proporciona un constructor predeterminado. También podemos definir un constructor predeterminado de manera explícita. Si define un constructor para una clase, C++ no creará un constructor predeterminado.
- UML modela a los constructores como operaciones en el tercer compartimiento de un diagrama de clases. Para diferenciar a un constructor de las operaciones de una clase, UML coloca la palabra “constructor” entre los signos «» y » antes del nombre del constructor.

Sección 3.8 Colocar una clase en un archivo separado para fines de reutilización

- Cuando las definiciones de clases se empaquetan en forma apropiada, los programadores de todo el mundo pueden reutilizarlas.
- Es costumbre definir una clase en un archivo de encabezado que tenga una extensión `.h`.
- Si cambia la implementación de la clase, no se requiere que los clientes de la clase cambien.
- Las interfaces definen y estandarizan las formas en que interactúan las personas y los sistemas.
- La interfaz de una clase describe las funciones miembro `public` que están disponibles para los clientes de la clase. La interfaz describe *qué* servicios pueden utilizar los clientes y cómo *solicitar* esos servicios, pero no especifica *cómo* la clase lleva a cabo los servicios.

Sección 3.9 Separar la interfaz de la implementación

- Al separar la interfaz de la implementación se facilita la modificación de los programas. Los cambios en la implementación de la clase no afectan al cliente, mientras que la interfaz de la clase permanezca sin cambios.
- El prototipo de una función contiene el nombre de una función, su tipo de valor de retorno y el número, tipos y orden de los parámetros que la función espera recibir.
- Una vez que se define una clase y se declaran sus funciones miembro (a través de los prototipos de función), las funciones miembro deben definirse en un archivo de código fuente separado.
- Para cada función miembro definida fuera de su correspondiente definición de clase, hay que anteponer al nombre de la función el nombre de la clase y el operador binario de resolución de ámbito (`::`).

Sección 3.10 Validación de datos mediante funciones establecer

- La función miembro `length` de la clase `string` devuelve el número de caracteres en un objeto `string`.
- La función miembro `substr` de la clase `string` devuelve un nuevo objeto `string` que contiene una copia de parte de un objeto `string` existente. El primer argumento especifica la posición inicial en el objeto `string` original. El segundo argumento especifica el número de caracteres a copiar.

Terminología

accesor	llamada a una función
archivo de código fuente	llamada a una función miembro
archivo de encabezado	más (+), signo de UML
argumento	menos (-), signo de UML
atributo (UML)	mensaje (enviar a un objeto)
cadena vacía	miembro de datos
cliente de un objeto o clase	mutador
código objeto	nomenclatura de camello
compartimiento en un diagrama de clases (UML)	ocultamiento de datos
constructor	operación (UML)
constructor predeterminado	operador binario de resolución de ámbito (::)
cuerpo de la definición de una clase	operador punto (.)
definición de clase	parámetro
definición de una clase	parámetro de operación (UML)
diagrama de clases (UML)	private, especificador de acceso
diagrama de clases de UML	programa controlador
encabezado de función	programador de la implementación de una clase
especificador de acceso	programador del código cliente
estado consistente	prototipo de función
función establecer	public, especificador de acceso
función miembro	public, servicios de una clase
función obtener	separar la interfaz de la implementación
función que hace la llamada	solicitar un servicio de un objeto
getline, función de la biblioteca <string>	string, clase
.h, extensión de archivo	<string>, archivo de encabezado
implementación de una clase	substr, función miembro de la clase string
ingeniería de software	tipo de valor de retorno
instancia de una clase	tipo definido por el usuario
interfaz de una clase	validación
invocar una función miembro	variable local
length, función miembro de la clase string	verificación de validez
lenguaje extensible	void, tipo de valor de retorno
lista de parámetros	« y », signos de UML

Ejercicios de autoevaluación

3.1 Complete las siguientes oraciones:

- Una casa es para un plano de construcción lo que un(a) _____ para una clase.
- Cada declaración de clase contiene la palabra clave _____, seguida inmediatamente por el nombre de la clase.
- Por lo general, la definición de una clase se almacena en un archivo con la extensión de archivo _____.
- Cada parámetro en un encabezado de función debe especificar un(a) _____ y un(a) _____.
- Cuando cada objeto de una clase mantiene su propia copia de un atributo, la variable que representa a este atributo se conoce también como _____.
- La palabra clave public es un(a) _____.
- El tipo de valor de retorno _____ indica que una función realizará una tarea, pero no devolverá información cuando complete su tarea.
- La función _____ de la biblioteca <string> lee caracteres hasta encontrar una nueva línea, y después copia esos caracteres en el objeto string especificado.
- Cuando se define una función miembro fuera de la definición de una clase, el encabezado de la función debe incluir el nombre de la clase y el _____, seguido del nombre de la función para “enlazar” la función miembro con la definición de la clase.
- El archivo de código fuente, y cualquier otro archivo que utilice una clase, pueden incluir el archivo de encabezado de la clase mediante una directiva del preprocesador _____.

- 3.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
- Por convención, los nombres de las funciones empiezan con la primera letra en mayúscula y todas las palabras subsiguientes en el nombre empiezan con la primera letra en mayúscula.
 - Los paréntesis vacíos que van después del nombre de una función en un prototipo de función indican que ésta no requiere parámetros para realizar su tarea.
 - Los miembros de datos o las funciones miembro que se declaran con el modificador de acceso `private` son accesibles para las funciones miembro de la clase en la que se declaran.
 - Las variables que se declaran en el cuerpo de una función miembro específico se conocen como miembros de datos, y pueden utilizarse en todas las funciones miembro de la clase.
 - El cuerpo de toda función está delimitado por las llaves izquierda y derecha (`{` y `}`).
 - Cualquier archivo de código fuente que contenga `int main()` puede usarse para ejecutar un programa.
 - Los tipos de los argumentos en la llamada a una función deben ser consistentes con los tipos de los parámetros correspondientes en la lista de parámetros del prototipo de la función.
- 3.3 ¿Cuál es la diferencia entre una variable local y un miembro de datos?
- 3.4 Explique el propósito de un parámetro de una función. ¿Cuál es la diferencia entre un parámetro y un argumento?

Respuestas a los ejercicios de autoevaluación

- 3.1 a) objeto. b) `class`. c) h. d) tipo, nombre. e) miembro de datos. f) especificador de acceso. g) `void`. h) `getLine`. i) operador binario de resolución de ámbito `(::)`. j) `#include`.
- 3.2 a) Falso. Los nombres de las funciones empiezan con una primera letra en minúscula y todas las palabras subsiguientes en el nombre empiezan con una letra en mayúscula. b) Verdadero. c) Verdadero. d) Falso. Dichas variables son variables locales y sólo pueden usarse en la función miembro en la que están declaradas. e) Verdadero. f) Verdadero. g) Verdadero.
- 3.3 Una variable local se declara en el cuerpo de una función, y sólo puede utilizarse desde el punto en el que se declaró, hasta la llave de cierre correspondiente. Un miembro de datos se declara en una clase, pero no en el cuerpo de alguna de las funciones miembro de la clase. Cada objeto de una clase tiene una copia separada de los miembros de datos de la clase. Los miembros de datos están accesibles para todas las funciones miembro de la clase.
- 3.4 Un parámetro representa la información adicional que requiere una función para realizar su tarea. Cada parámetro requerido por una función está especificado en el encabezado de la función. Un argumento es el valor que se suministra en la llamada a la función. Cuando se llama a la función, el valor del argumento se pasa al parámetro de la función, para que ésta pueda realizar su tarea.

Ejercicios

- 3.5 Explique la diferencia entre un prototipo de función y la definición de una función.
- 3.6 ¿Qué es un constructor predeterminado? ¿Cómo se inicializan los miembros de datos de un objeto, si una clase sólo tiene un constructor predeterminado definido en forma implícita?
- 3.7 Explique el propósito de un miembro de datos.
- 3.8 ¿Qué es un archivo de encabezado? ¿Qué es un archivo de código fuente? Hable sobre el propósito de cada uno.
- 3.9 Explique cómo puede usar un programa una clase `string` sin insertar una declaración `using`.
- 3.10 Explique por qué una clase podría proporcionar una función `establecer` y una función `obtener` para un miembro de datos.
- 3.11 (*Modificación de la clase LibroCalificaciones*) Modifique la clase `LibroCalificaciones` (figuras 3.11 a 3.12) de la siguiente manera:
- Incluya un segundo miembro de datos `string`, que represente el nombre del instructor del curso.
 - Proporcione una función `establecer` para modificar el nombre del instructor, y una función `obtener` para obtenerlo.
 - Modifique el constructor para especificar dos parámetros: uno para el nombre del curso y otro para el nombre del instructor.
 - Modifique la función `mostrarMensaje`, de tal forma que primero imprima el mensaje de bienvenida y el nombre del curso, y que después imprima "Este curso es presentado por: ", seguido del nombre del instructor.
- Use su clase modificada en un programa de prueba que demuestre las nuevas capacidades de la clase.

3.12 (Clase Cuenta) Cree una clase llamada `Cuenta` que podría ser utilizada por un banco para representar las cuentas bancarias de sus clientes. Incluya un miembro de datos de tipo `int` para representar el saldo de la cuenta. [Nota: en los siguientes capítulos, utilizaremos números que contienen puntos decimales (por ejemplo, 2.75), a los cuales se les conoce como valores de punto flotante, para representar montos en dólares.] Proporcione un constructor que reciba un saldo inicial y lo utilice para inicializar el miembro de datos. El constructor debe validar el saldo inicial para asegurar que sea mayor o igual que 0. De no ser así, establezca el saldo en 0 y muestre un mensaje de error, indicando que el saldo inicial era inválido. Proporcione tres funciones miembro. La función miembro `credit` debe agregar un monto al saldo actual. La función miembro `cargar` deberá retirar dinero del objeto `Cuenta` y asegurarse que el monto a cargar no exceda el saldo de `Cuenta`. Si lo hace, el saldo debe permanecer sin cambio y la función debe imprimir un mensaje que indique "El monto a cargar excede el saldo de la cuenta." La función miembro `obtenerSaldo` debe devolver el saldo actual. Cree un programa que cree dos objetos `Cuenta` y evalúe las funciones miembro de la clase `Cuenta`.

3.13 (Clase Factura) Cree una clase llamada `Factura`, que una ferretería podría utilizar para representar una factura por un artículo vendido en la tienda. Una `Factura` debe incluir cuatro piezas de información como miembros de datos: un número de pieza (tipo `string`), la descripción de la pieza (tipo `string`), la cantidad de artículos de ese tipo que se van a comprar (tipo `int`) y el precio por artículo (tipo `int`). [Nota: en los siguientes capítulos, utilizaremos números que contienen puntos decimales (por ejemplo, 2.75), a los cuales se les conoce como valores de punto flotante, para representar montos en dólares.] Su clase debe tener un constructor que inicialice los cuatro miembros de datos. Proporcione una función `establecer` y una función `obtener` para cada miembro de datos. Además, proporcione una función miembro llamada `obtenerMontoFactura`, que calcule el monto de la factura (es decir, que multiplique la cantidad por el precio por artículo) y después devuelva ese monto como un valor `int`. Si la cantidad no es positiva, debe establecerse en 0. Si el precio por artículo no es positivo, debe establecerse en 0. Escriba un programa de prueba que demuestre las capacidades de la clase `Factura`.

3.14 (Clase Empleado) Cree una clase llamada `Empleado`, que incluya tres piezas de información como miembros de datos: un primer nombre (tipo `string`), un apellido paterno (tipo `string`) y un salario mensual (tipo `int`). Nota: en los siguientes capítulos, utilizaremos números que contienen puntos decimales (por ejemplo, 2.75), a los cuales se les conoce como valores de punto flotante, para representar montos en dólares.] Su clase debe tener un constructor que inicialice los tres miembros de datos. Proporcione una función `establecer` y una función `obtener` para cada miembro de datos. Si el salario mensual no es positivo, establezcalo en 0. Escriba un programa de prueba que demuestre las capacidades de la clase `Empleado`. Cree dos objetos `Empleado` y muestre el salario *anual* de cada objeto. Después, proporcione a cada `Empleado` un aumento del 10% y muestre el salario anual de cada `Empleado` otra vez.

3.15 (Clase Fecha) Cree una clase llamada `Fecha`, que incluya tres piezas de información como miembros de datos: un mes (tipo `int`), un día (tipo `int`) y un año (tipo `int`). Su clase debe tener un constructor con tres parámetros, los cuales debe utilizar para inicializar los tres miembros de datos. Para los fines de este ejercicio, suponga que los valores que se proporcionan para el año y el día son correctos, pero asegúrese que el valor del mes se encuentre en el rango de 1 a 12; de no ser así, establezca el mes en 1. Proporcione una función `establecer` y una función `obtener` para cada miembro de datos. Proporcione una función miembro `mostrarFecha`, que muestre el mes, día y año, separados por barras diagonales (/). Escriba un programa de prueba que demuestre las capacidades de la clase `Fecha`.

4



Desplacémonos un lugar.

—Lewis Carroll

La rueda se convirtió en un círculo completo.

—William Shakespeare

¡Cuántas manzanas tuvieron que caer en la cabeza de Newton antes de que entendiera el suceso!

—Robert Frost

Toda la evolución que conocemos procede de lo vago a lo definido.

—Charles Sanders Peirce

Instrucciones de control: parte I

OBJETIVOS

En este capítulo aprenderá a:

- Comprender las técnicas básicas para solucionar problemas.
- Desarrollar algoritmos mediante el proceso de mejoramiento de arriba a abajo, paso a paso.
- Utilizar las estructuras de selección `if` e `if...else` para elegir entre distintas acciones alternativas.
- Utilizar la estructura de repetición `while` para ejecutar instrucciones de manera repetitiva dentro de un programa.
- Comprender la repetición controlada por un contador y la repetición controlada por un centinela.
- Utilizar los operadores de incremento, decremento y asignación.

- 4.1 Introducción
- 4.2 Algoritmos
- 4.3 Seudocódigo
- 4.4 Estructuras de control
- 4.5 Instrucción de selección **if**
- 4.6 Instrucción de selección doble **if...else**
- 4.7 Instrucción de repetición **while**
- 4.8 Cómo formular algoritmos: repetición controlada por un contador
- 4.9 Cómo formular algoritmos: repetición controlada por un centinela
- 4.10 Cómo formular algoritmos: instrucciones de control anidadas
- 4.11 Operadores de asignación
- 4.12 Operadores de incremento y decremento
- 4.13 (Opcional) Ejemplo práctico de Ingeniería de Software: identificación de los atributos de las clases en el sistema ATM
- 4.14 Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)

4.1 Introducción

Antes de escribir un programa que dé solución a un problema, es necesario tener una comprensión detallada de todo el problema, además de una metodología cuidadosamente planeada para resolverlo. Al escribir un programa, también debemos comprender los tipos de bloques de construcción disponibles, y emplear las técnicas comprobadas para construir programas. En este capítulo y en el capítulo 5, Instrucciones de control: parte 2, hablaremos sobre estas cuestiones cuando presentemos la teoría y los principios de la programación estructurada. Los conceptos aquí presentados son imprescindibles para crear clases efectivas y manipular objetos.

En este capítulo presentamos las instrucciones **if**, **if...else** y **while** de C++, tres de los bloques de construcción que permiten a los programadores especificar la lógica requerida para que las funciones miembro realicen sus tareas. Dedicamos una parte de este capítulo (y de los capítulos 5 y 7) para desarrollar más la clase **LibroCalificaciones** que presentamos en el capítulo 3. En especial, agregamos una función miembro a la clase **LibroCalificaciones** que utiliza instrucciones de control para calcular el promedio de un conjunto de calificaciones de estudiantes. Otro ejemplo demuestra formas adicionales de combinar instrucciones de control para resolver un problema similar. Presentamos los operadores de asignación de C++, y exploramos los operadores de incremento y decremento. Estos operadores adicionales abrevian y simplifican muchas instrucciones de los programas.

4.2 Algoritmos

Cualquier problema de computación puede resolverse ejecutando una serie de acciones en un orden específico. Un **procedimiento** para resolver un problema en términos de

1. las **acciones** a ejecutar y
2. el **orden** en el que se ejecutan estas acciones

se conoce como un **algoritmo**. El siguiente ejemplo demuestra que es importante especificar de manera correcta el orden en el que se ejecutan las acciones.

Considere el “algoritmo para levantarse y arreglarse” que sigue un ejecutivo para levantarse de la cama e ir a trabajar: (1) levantarse; (2) quitarse la pijama; (3) bañarse; (4) vestirse; (5) desayunar; (6) transportarse al trabajo. Esta rutina logra que el ejecutivo llegue al trabajo bien preparado para tomar decisiones críticas. Suponga que los mismos pasos se realizan en un orden ligeramente distinto: (1) levantarse; (2) quitarse la pijama; (3) vestirse; (4) bañarse; (5) desayunar; (6) transportarse al trabajo. En este caso nuestro ejecutivo llegará al trabajo todo mojado. Al proceso de especificar el orden en el que se ejecutan las instrucciones (acciones) en un programa de computadora, se le llama **control del programa**. En este capítulo investigaremos el control de los programas mediante el uso de las **instrucciones de control** de C++.

4.3 Seudocódigo

El **seudocódigo** (o “imitación” de código) es un lenguaje informal que ayuda a los programadores a desarrollar algoritmos sin tener que preocuparse por los estrictos detalles de la sintaxis del lenguaje C++. El seudocódigo que presentaremos aquí es especialmente útil para desarrollar algoritmos que se convertirán en porciones estructuradas de programas en C++. El seudocódigo es similar al lenguaje cotidiano; es conveniente y amigable con el usuario, aunque no es realmente un lenguaje de programación de computadoras.

El seudocódigo no se ejecuta en las computadoras. En lugar de ello, ayuda al programador a “organizar” un programa antes de intentar escribirlo en un lenguaje de programación como C++. Este capítulo presenta varios ejemplos de cómo utilizar el seudocódigo para desarrollar programas en C++.

El estilo de seudocódigo que presentaremos consiste solamente en caracteres, de manera que los programadores pueden escribir el seudocódigo convenientemente, utilizando cualquier programa editor de texto. La computadora puede producir una copia recién impresa de un programa de seudocódigo bajo demanda. Un programa en seudocódigo preparado de manera cuidadosa puede convertirse fácilmente en su correspondiente programa en C++. En muchos casos, esto requiere tan sólo reemplazar las instrucciones en seudocódigo con sus instrucciones equivalentes en C++.

Por lo general, el seudocódigo describe sólo las **instrucciones ejecutables**, que representan las acciones que ocurren después de que un programador convierte un programa de seudocódigo a C++, y el programa se ejecuta en una computadora. Las declaraciones (que no tienen inicializadores, o que no implican llamadas a un constructor) no son instrucciones ejecutables. Por ejemplo, la declaración

```
int i;
```

indica al compilador el tipo de la variable *i* y lo instruye para que reserve espacio en memoria para esa variable. Esta declaración no hace que ocurra ninguna acción (como una operación de entrada, salida o un cálculo) cuando el programa se ejecuta. Por lo general no incluimos las declaraciones de variables en nuestro seudocódigo. Sin embargo, algunos programadores optan por enlistar las variables y mencionar sus propósitos al principio de sus programas en seudocódigo.

Veamos un ejemplo de seudocódigo que se puede escribir para ayudar a un programador a crear el programa de suma de la figura 2.5. Este seudocódigo (figura 4.1) corresponde al algoritmo que recibe como entrada dos enteros del usuario, suma estos enteros y muestra su suma en pantalla. Aunque mostramos aquí el listado completo en seudocódigo, le mostraremos cómo crear seudocódigo a partir de un problema más adelante en este capítulo.

Las líneas 1 y 2 corresponden a las instrucciones 13 y 14 de la figura 2.5. Observe que las instrucciones en seudocódigo son simplemente instrucciones en lenguaje cotidiano que representan la tarea que se debe realizar en C++. De igual forma, las líneas 4 y 5 corresponden a las instrucciones en las líneas 16 y 17 de la figura 2.5, y las líneas 7 y 8 corresponden a las instrucciones en las líneas 19 y 21 de la figura 2.5.

- 1** Pedir al usuario que introduzca el primer entero
- 2** Recibir como entrada el primer entero
- 3**
- 4** Pedir al usuario que introduzca el segundo entero
- 5** Recibir como entrada el segundo entero
- 6**
- 7** Sumar el primer entero con el segundo entero, almacenar el resultado
- 8** Mostrar el resultado en pantalla

Figura 4.1 | Seudocódigo para el programa de suma de la figura 2.5.

Hay algunos aspectos importantes acerca del seudocódigo de la figura 4.1. Observe que el seudocódigo corresponde sólo al código de la función *main*. Esto ocurre debido a que el seudocódigo se utiliza normalmente para los algoritmos, no para los programas completos. En este caso, el seudocódigo representa el algoritmo. La función en la que se coloca este código no es importante para el algoritmo en sí. Por la misma razón, la línea 23 de la figura 2.5 (la instrucción *return*) no se incluye en el seudocódigo; esta instrucción *return* se coloca al final de toda función *main* y no es importante para el algoritmo. Por último, las líneas 9 a 11 de la figura 2.5 tampoco se incluyen en el seudocódigo, ya que estas declaraciones de variables no son instrucciones ejecutables.

4.4 Estructuras de control

Por lo general, en un programa las instrucciones se ejecutan una después de otra, en el orden en que están escritas. Este proceso se conoce como **ejecución secuencial**. Varias instrucciones en C++ que pronto veremos, permiten al programador especificar que la siguiente instrucción a ejecutarse tal vez no sea la siguiente en la secuencia. Esto se conoce como **transferencia de control**.

Durante la década de los sesentas, se hizo evidente que el uso indiscriminado de las transferencias de control era el origen de muchas de las dificultades que experimentaban los grupos de desarrollo de software. A quien se señaló como culpable fue a la **instrucción goto**, la cual permite al programador especificar la transferencia de control a uno de los muchos posibles destinos dentro de un programa (creando lo que se conoce comúnmente como “código espagueti”). La noción de la llamada **programación estructurada** se hizo casi un sinónimo de la “eliminación del goto”.

Las investigaciones de Böhm y Jacopini¹ demostraron que los programas podían escribirse sin instrucciones goto. El reto de la época para los programadores fue cambiar sus estilos a una “programación sin goto”. No fue sino hasta la década de los setentas cuando los programadores tomaron en serio la programación estructurada. Los resultados han sido impresionantes, ya que los grupos de desarrollo de software han reportado reducciones en los tiempos de desarrollo, mayor incidencia de entregas de sistemas a tiempo y más proyectos de software finalizados sin salirse del presupuesto. La clave para estos logros es que los programas estructurados son más claros, más fáciles de depurar, probar y modificar, y hay más probabilidad de que estén libres de errores desde el principio.

El trabajo de Böhm y Jacopini demostró que todos los programas podían escribirse en términos de tres **estructuras de control** solamente: la **estructura de secuencia**, la **estructura de selección** y la **estructura de repetición**. El término “estructuras de control” proviene del campo de las ciencias computacionales. Cuando presentemos las implementaciones en C++ de las estructuras de control, nos referiremos a ellas en la terminología del documento del estándar de C++² como “instrucciones de control”.

Estructura de secuencia en C++

La estructura de secuencia está integrada en C++. A menos que se le indique lo contrario, la computadora ejecuta las instrucciones en C++ una después de otra, en el orden en que estén escritas, es decir, en secuencia. El **diagrama de actividad** en Lenguaje unificado de modelado (UML) de la figura 4.2 ilustra una estructura de secuencia típica, en la que se realizan dos cálculos en orden. C++ permite tantas acciones como deseemos en una estructura de secuencia. Como veremos pronto, donde quiera que se coloque una sola acción, podrán colocarse varias acciones en secuencia.

En esta figura, las dos instrucciones implican sumar una calificación a una variable llamada **total**, y sumar el valor 1 a una variable llamada **contador**. Dichas instrucciones podrían aparecer en un programa que obtenga el promedio de varias calificaciones de estudiantes. Para calcular un promedio, el total de las calificaciones que se van a promediar se divide entre el número de calificaciones. Podría usarse una variable contador para llevar la cuenta del número de valores a promediar. En el programa de la sección 4.8 verá instrucciones similares.

Los diagramas de actividad son parte del UML. Un diagrama de actividad modela el **flujo de trabajo** (también conocido como la **actividad**) de una parte de un sistema de software. Dichos flujos de trabajo pueden incluir una porción de un algoritmo, como la estructura de secuencia de la figura 4.2. Los diagramas de actividad están compuestos por símbolos de propósito especial, como los **símbolos de estado de acción** (rectángulos cuyos lados izquierdo y derecho se reemplazan con arcos hacia afuera), **rombos (diamantes)** y **pequeños círculos**; estos símbolos se conectan mediante **flechas de transición**, que representan el flujo de la actividad.

Al igual que el seudocódigo, los diagramas de actividad ayudan a los programadores a desarrollar y representar algoritmos, aunque muchos de ellos prefieren el seudocódigo. Los diagramas de actividad muestran claramente cómo operan las estructuras de control.

Considere el diagrama de actividad para la estructura de secuencia de la figura 4.2. Este diagrama contiene dos **estados de acción** que representan las acciones a realizar. Cada estado de acción contiene una **expresión de acción** (por ejemplo, “sumar calificación a total” o “sumar 1 al contador”), que especifica una acción particular a realizar. Otras acciones podrían incluir cálculos u operaciones de entrada/salida. Las flechas en el diagrama de actividad se llaman **flechas de transición**. Estas flechas representan **transiciones**, las cuales indican el orden en el que ocurren las acciones representadas

-
1. Bohm, C. y G. Jacopini, “Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules,” *Communications of the ACM*, Vol. 9, No. 5, Mayo de 1996, páginas 336-371.
 2. Este documento se conoce más específicamente como *INCITS/ISO/IEC 14882-2003 Programming Languages—C++*, y está disponible para descargarlo (por una cuota) en: webstore.ansi.org/ansidocstore/product.asp?sku=INCITS%2FiSO%2FiEC+14882%2D2003.

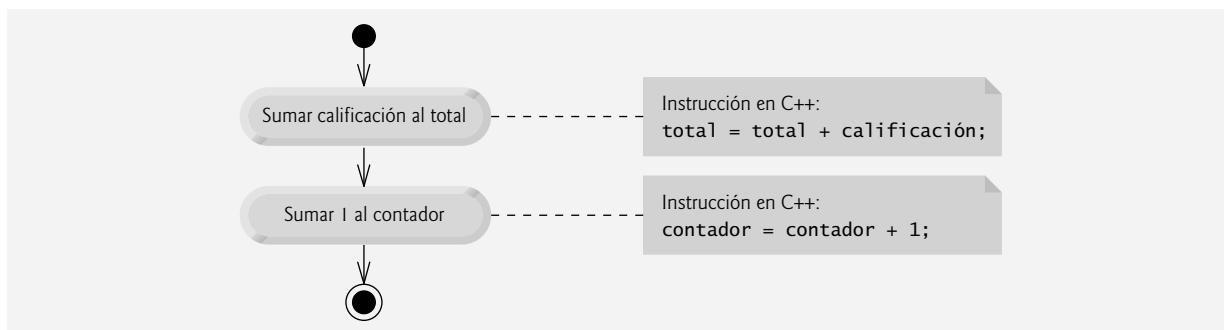


Figura 4.2 | Diagrama de actividad de una estructura de secuencia.

por los estados de acción; el programa que implementa las actividades ilustradas por el diagrama de la figura 4.2 primero suma `calificación` a `total`, y después suma 1 a `contador`.

El **círculo relleno** que se encuentra en la parte superior del diagrama de actividad representa el **estado inicial** de la actividad: el inicio del flujo de trabajo antes de que el programa realice las actividades modeladas. El círculo sólido rodeado por una circunferencia que aparece en la parte inferior del diagrama representa el **estado final**, es decir, el final del flujo de trabajo después de que el programa realiza sus acciones.

La figura 4.2 también incluye rectángulos que tienen la esquina superior derecha doblada. En UML, a estos rectángulos se les llama **notas**. Las notas son comentarios con explicaciones que describen el propósito de los símbolos en el diagrama. Las notas se pueden usar en cualquier diagrama de UML, no sólo en los diagramas de actividad. La figura 4.2 utiliza las notas de UML para mostrar el código en C++ asociado con cada uno de los estados de acción en el diagrama de actividad. Una **línea punteada** conecta cada nota con el elemento que ésta describe. Por lo general, los diagramas de actividad no muestran el código de C++ que implementa a cada actividad. En este libro utilizamos las notas con este propósito, para mostrar cómo se relaciona el diagrama con el código en C++. Para obtener más información sobre UML, vea nuestro ejemplo práctico opcional, que aparece en las secciones tituladas Ejemplo práctico de Ingeniería de Software al final de los capítulos 1 a 7, 9, 10, 12 y 13, o visite www.uml.org.

Instrucciones de selección en C++

C++ tiene tres tipos de instrucciones de selección (las cuales se describen en este capítulo y en el siguiente). La instrucción de selección `if` realiza (selecciona) una acción si la condición (predicado) es verdadera, o evita la acción si la condición es falsa. La instrucción de selección `if...else` realiza una acción si la condición es verdadera, o realiza una acción distinta si la condición es falsa. La instrucción de selección `switch` (capítulo 5) realiza una de entre varias acciones distintas, dependiendo del valor de una expresión entera.

La instrucción de selección `if` es una **instrucción de selección simple**, ya que selecciona o ignora una sola acción (o, como pronto veremos, un solo grupo de acciones). La instrucción `if...else` se conoce como **instrucción de selección doble**, ya que selecciona entre dos acciones distintas (o grupos de acciones). La instrucción `switch` es una **estructura de selección múltiple**, ya que selecciona entre diversas acciones (o grupos de acciones).

Instrucciones de repetición en C++

C++ cuenta con tres instrucciones de repetición (también llamadas **instrucciones de ciclo** o **ciclos**) que permiten a los programas ejecutar instrucciones en forma repetida, siempre y cuando una condición (llamada la **condición de continuación del ciclo**) siga siendo verdadera. Las instrucciones de repetición se implementan con las instrucciones `while`, `do...while` y `for`. (En el capítulo 5 presentaremos las instrucciones `do..while` y `for`.) Las instrucciones `while` y `for` realizan la acción (o grupo de acciones) en sus cuerpos, cero o más veces; si la condición de continuación del ciclo es inicialmente falsa, no se ejecutará la acción (o grupo de acciones). La instrucción `do...while` realiza la acción (o grupo de acciones) en su cuerpo, una o más veces.

Las palabras `if`, `else`, `switch`, `while`, `do` y `for` son palabras clave en C++. Estas palabras las reserva el lenguaje de programación C++ para implementar varias características, como las instrucciones de control. Las palabras clave no pueden usarse como identificadores, como los nombres de variables. En la figura 4.3 aparece una lista completa de las palabras clave en C++.

Palabras clave de C++				
<i>Palabras clave comunes para los lenguajes de programación C y C++</i>				
auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			
<i>Palabras clave sólo de C++</i>				
and	and_eq	asm	bitand	bitor
bool	catch	class	compl	const_cast
delete	dynamic_cast	explicit	export	false
friend	inline	mutable	namespace	new
not	not_eq	operator	or	or_eq
private	protected	public	reinterpret_cast	static_cast
template	this	throw	true	try
typeid	typename	using	virtual	wchar_t
xor	xor_eq			

Figura 4.3 | Palabras clave de C++.



Error común de programación 4.1

Usar una palabra clave como identificador es un error de sintaxis.



Error común de programación 4.2

Deletrear una palabra clave con cualquier letra mayúscula es un error de sintaxis. Todas las palabras clave de C++ contienen sólo letras minúsculas.

Resumen de las instrucciones de control en C++

C++ sólo tiene tres tipos de estructuras de control, a las cuales nos referiremos de aquí en adelante como instrucciones de control: la instrucción de secuencia, las instrucciones de selección (tres tipos: `if`, `if...else` y `switch`) y las instrucciones de repetición (tres tipos: `while`, `for` y `do...while`). Cada programa de C++ se forma combinando tantas instrucciones de secuencia, selección y repetición como sea apropiado para el algoritmo que implemente el programa. Al igual que con la instrucción de secuencia de la figura 4.2, podemos modelar cada una de las instrucciones de control como un diagrama de actividad. Cada diagrama contiene un estado inicial y final, los cuales representan el punto de entrada y salida de la instrucción de control, respectivamente. Las **instrucciones de control de una sola entrada/una sola salida** facilitan la creación de programas; las instrucciones de control están unidas entre sí mediante la conexión del punto de salida de una instrucción de control, al punto de entrada de la siguiente. Este procedimiento es similar a la manera en que un niño apila los bloques de construcción, así que a esto le llamamos **apilamiento de instrucciones de control**. En breve aprenderemos que sólo hay una manera alternativa de conectar las instrucciones de control: el **anidamiento de instrucciones de control**, en el cual una instrucción de control aparece dentro de otra. Por lo tanto, los algoritmos en los programas en C++ se crean a partir de sólo tres tipos de instrucciones de control, que se combinan solamente de dos formas. Ésta es la esencia de la simpleza.



Observación de Ingeniería de Software 4.1

Cualquier programa de C++ que se deseé crear puede construirse a partir de sólo siete tipos distintos de instrucciones de control (secuencia, `if`, `if...else`, `switch`, `while`, `do...while` y `for`), combinadas en sólo dos formas (apilamiento de instrucciones de control y anidamiento de instrucciones de control).

4.5 Instrucción de selección if

Los programas utilizan instrucciones de selección para elegir entre los cursos alternativos de acción. Por ejemplo, suponga que la calificación para aprobar un examen es de 60. La instrucción en seudocódigo

*Si la calificación del estudiante es mayor o igual a 60
Imprimir "Aprobado"*

determina si la condición “la calificación del estudiante es mayor o igual a 60” es verdadera (`true`) o falsa (`false`). Si la condición es verdadera se imprime “Aprobado”, y se “ejecuta” en orden la siguiente instrucción en seudocódigo (recuerde que el seudocódigo no es un verdadero lenguaje de programación). Si la condición es falsa se ignora la instrucción para imprimir, y se ejecuta en orden la siguiente instrucción en seudocódigo. Observe que la segunda línea de esta instrucción de selección tiene sangría. Dicha sangría es opcional, pero se recomienda ya que enfatiza la estructura inherente de los programas estructurados. Al convertir seudocódigo en código de C++, el compilador ignora los caracteres de espacio en blanco (como espacios, tabuladores y caracteres de nueva línea) que se utilizan para aplicar sangría y espacio vertical.



Buena práctica de programación 4.1

Al aplicar con consistencia una sangría razonable en todos sus programas, se mejora enormemente la legibilidad de los mismos. Sugerimos tres espacios en blanco por sangría. Algunas personas prefieren usar tabuladores, pero éstos pueden variar de un editor de texto a otro, lo cual provoca que un programa escrito en un editor de texto se alinee de forma distinta cuando se utilice en otro editor de texto.

La instrucción anterior `if` en seudocódigo puede escribirse en C++ de la siguiente manera:

```
if ( calificacionEstudiante >= 60 )
    cout << "Aprobado";
```

Observe que el código en C++ corresponde en gran medida con el seudocódigo. Ésta es una de las propiedades que hace del seudocódigo una herramienta de desarrollo de programas tan útil.

La figura 4.4 muestra la instrucción `if` de selección simple. Esta figura contiene lo que quizá sea el símbolo más importante en un diagrama de actividad: el rombo o **símbolo de decisión**, el cual indica que se va a tomar una decisión. Un símbolo de decisión indica que el flujo de trabajo continuará a lo largo de una ruta determinada por las **condiciones de guardia** asociadas de ese símbolo, que pueden ser verdaderas o falsas. Cada flecha de transición que sale de un símbolo de decisión tiene una condición de guardia (especificada entre corchetes, por encima o a un lado de la flecha de transición). Si cierta condición de guardia es verdadera, el flujo de trabajo entra al estado de acción al que apunta la flecha de transición. En la figura 4.4, si la calificación es mayor o igual a 60, el programa imprime “Aprobado” en la pantalla, y luego se dirige al estado final de esta actividad. Si la calificación es menor a 60, el programa se dirige inmediatamente al estado final sin mostrar ningún mensaje.

En el capítulo 2 aprendimos que las decisiones se pueden basar en condiciones que contengan operadores de igualdad o relacionales. En realidad, en C++ una decisión se puede basar en cualquier expresión: si la expresión se evalúa como cero, se considera como falsa; si la expresión se evalúa como un número distinto de cero, se considera verdadera. C++ proporciona el tipo de datos `bool` para las variables que sólo pueden contener los valores `true` y `false`; cada una de éstas es una palabra clave de C++.

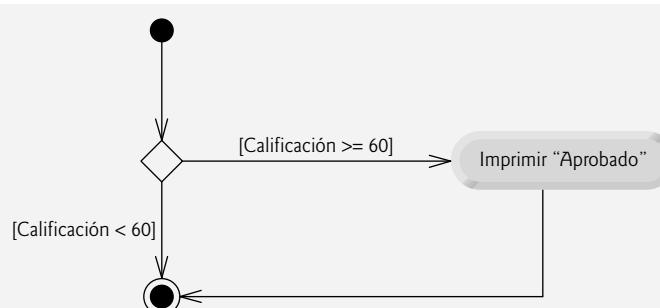


Figura 4.4 | Diagrama de actividad de la instrucción `if` de selección simple.



Tip de portabilidad 4.1

Para tener compatibilidad con versiones anteriores de C, en las que se utilizan enteros para los valores booleanos, el valor `bool true` también se puede representar mediante cualquier valor distinto de cero (por lo general, los compiladores utilizan 1) y el valor `bool false` también se puede representar como el valor cero.

Observe que la instrucción `if` es una instrucción de control de una sola entrada/una sola salida. Pronto veremos que los diagramas de actividad para las instrucciones de control restantes también contienen estados iniciales, flechas de transición, estados de acción que indican las acciones a realizar, símbolos de decisión (con sus condiciones de guardia asociadas) que indican las decisiones a tomar, y estados finales. Esto es consistente con el **modelo de programación acción/decisión** que hemos estado enfatizando.

Podemos imaginar siete cajones, cada uno de los cuales contiene diagramas de actividad de UML vacíos de uno de los siete tipos de instrucciones de control. Su tarea es ensamblar un programa a partir de los diagramas de actividad de tantas instrucciones de control de cada tipo como lo requiera el algoritmo, combinando esas instrucciones de control en sólo dos formas posibles (apilando o anidando), y después llenando los estados de acción y las decisiones con expresiones de acción y condiciones de guardia, en una manera que sea apropiada para formar una implementación estructurada para el algoritmo. Hablaremos sobre la variedad de formas en que pueden escribirse las acciones y las decisiones.

4.6 Instrucción de selección doble `if...else`

La instrucción `if` de selección simple realiza una acción indicada solamente cuando la condición es verdadera (`true`); de no ser así, se evita dicha acción. La instrucción `if...else` de selección doble permite al programador especificar una acción a realizar cuando la condición es verdadera, y otra distinta cuando la condición es falsa (`false`). Por ejemplo, la instrucción en seudocódigo

Si la calificación del estudiante es mayor o igual a 60

Imprimir "Aprobado"

De lo contrario

Imprimir "Reprobado"

imprime “Aprobado” si la calificación del estudiante es mayor o igual a 60, e imprime “Reprobado” si la calificación del estudiante es menor a 60. En cualquier caso, después de que ocurre la impresión se “ejecuta” la siguiente instrucción en seudocódigo en la secuencia.

La instrucción anterior `if...Else` en seudocódigo puede escribirse en C++ como

```
if ( calificacion >= 60 )
    cout << "Aprobado";
else
    cout << "Reprobado";
```

Observe que el cuerpo de la instrucción `else` también tiene sangría.



Buena práctica de programación 4.2

Cualquiera que sea la convención de sangría que usted elija, debe aplicarla consistentemente en todos sus programas. Es difícil leer programas que no obedecen las convenciones de espaciado uniformes.



Buena práctica de programación 4.3

Utilice sangría en ambos cuerpos de instrucciones de una estructura `if...else`.



Buena práctica de programación 4.4

Si hay varios niveles de sangría, en cada nivel debe aplicarse la misma cantidad de espacio adicional para promover la legibilidad y facilidad de mantenimiento.

La figura 4.5 muestra el flujo de control en la instrucción `if...else`. Una vez más (además del estado inicial, las flechas de transición y el estado final), los otros símbolos en el diagrama de actividad representan estados de acción y decisiones. Nosotros seguimos enfatizando este modelo de computación acción/decisión. Imagine de nuevo un cajón profundo de diagramas de actividad de UML vacíos de instrucciones de selección doble; tantas como se pudiera necesitar para apilar o anidar con los diagramas de actividad de otras instrucciones de control, para formar una implementación

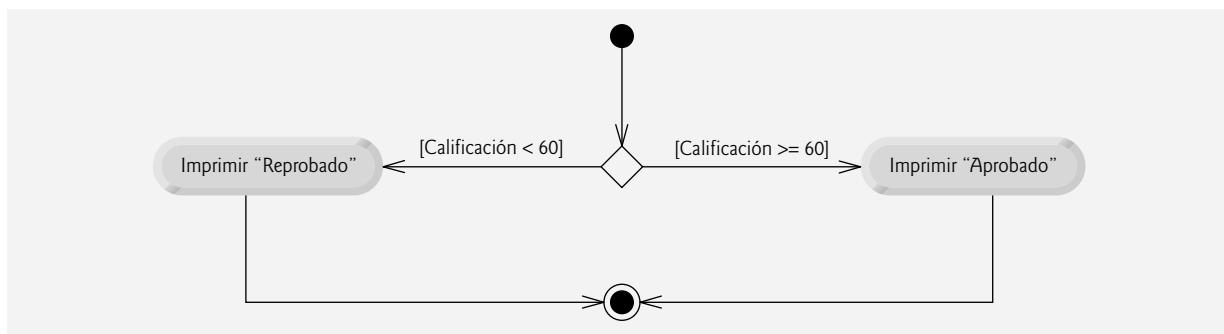


Figura 4.5 | Diagrama de actividad de la instrucción if...else de selección doble.

estructurada de un algoritmo. Usted debe llenar los estados de acción y los símbolos de decisión con expresiones de acción y condiciones de guardia que sean apropiadas para el algoritmo.

Operador condicional (?:)

C++ cuenta con el **operador condicional (?:)**, que está estrechamente relacionado con la instrucción if...else. Éste es el único **operador ternario** de C++, es decir, que recibe tres operandos. En conjunto, los operandos y el operador condicional forman una **expresión condicional**. El primer operando es una condición, el segundo es el valor de la expresión condicional si la condición es `true`, y el tercer operando es el valor de toda la expresión condicional si la condición es `false`. Por ejemplo, la instrucción de salida

```
cout << ( calificacionEstudiante >= 60 ? "Aprobado" : "Reprobado" );
```

contiene una expresión condicional, `calificacionEstudiante >= 60 ? "Aprobado" : "Reprobado"`, que se evalúa como la cadena "Aprobado" si la condición `calificacionEstudiante >= 60` es `true`, pero se evalúa como la cadena "Reprobado" si la condición es `false`. Por lo tanto, la instrucción con el operador condicional realiza en esencia la misma función que la instrucción if...else que se mostró anteriormente. Como veremos más adelante, la precedencia del operador condicional es baja, por lo cual los paréntesis en la expresión anterior son obligatorios.



Tip para prevenir errores 4.1

Para evitar problemas de precedencia (y por claridad), coloque las expresiones condicionales (que aparezcan en expresiones más grandes) entre paréntesis.

Los valores en una expresión condicional también pueden ser acciones a ejecutar. Por ejemplo, la siguiente expresión condicional también imprime "Aprobado" o "Reprobado":

```
calificacionEstudiante >= 60 ? cout << "Aprobado" : cout << "Reprobado";
```

La expresión condicional anterior se lee como: "si `calificacionEstudiante` es mayor o igual que 60, entonces `cout << "Aprobado"`"; en caso contrario, `cout << "Reprobado"`". Esto también puede compararse con la instrucción if...else anterior. Las expresiones condicionales pueden aparecer en algunas partes de los programas en las que no se pueden utilizar instrucciones if...else.

Instrucciones if...else anidadas

Las **instrucciones if...else anidadas** pueden evaluar varios casos, al colocar instrucciones de selección if...else dentro de otras instrucciones if...else. Por ejemplo, la siguiente instrucción if...else en pseudocódigo imprime A para las calificaciones de exámenes mayores o iguales a 90, B para las calificaciones en el rango de 80 a 89, C para las calificaciones en el rango de 70 a 79, D para las calificaciones en el rango de 60 a 69 y F para todas las demás calificaciones:

Si la calificación del estudiante es mayor o igual a 90

Imprimir "A"

de lo contrario

Si la calificación del estudiante es mayor o igual a 80

Imprimir "B"

de lo contrario

Si la calificación del estudiante es mayor o igual a 70

Imprimir "C"

de lo contrario

Si la calificación del estudiante es mayor o igual a 60

Imprimir "D"

de lo contrario

Imprimir "F"

Este seudocódigo puede escribirse en C++ como

```
if ( calificacionEstudiante >= 90 ) // 90 o más recibe una "A"
    cout << "A";
else
    if ( calificacionEstudiante >= 80 ) // 80 a 89 recibe una "B"
        cout << "B";
    else
        if ( calificacionEstudiante >= 70 ) // 70 a 79 recibe "C"
            cout << "C";
        else
            if ( calificacionEstudiante >= 60 ) // 60 a 69 recibe "D"
                cout << "D";
            else // menos de 60 recibe "F"
                cout << "F";
```

Si `calificacionEstudiante` es mayor o igual a 90, las primeras cuatro condiciones serán `true`, pero sólo se ejecutará la instrucción en la parte `if` de la primera instrucción `if...else`. Después de que se ejecute esa instrucción, se evita la parte `else` de la instrucción `if...else` más “externa”. La mayoría de los programadores en C++ prefieren escribir la instrucción `if...else` anterior así:

```
if ( calificacionEstudiante >= 90 ) // 90 o más recibe una "A"
    cout << "A";
else if ( calificacionEstudiante >= 80 ) // 80 a 89 recibe una "B"
    cout << "B";
else if ( calificacionEstudiante >= 70 ) // 70 a 79 recibe "C"
    cout << "C";
else if ( calificacionEstudiante >= 60 ) // 60 a 69 recibe "D"
    cout << "D";
else // menos de 60 recibe "F"
    cout << "F";
```

Las dos formas son idénticas, excepto por el espaciado y la sangría, que el compilador ignora. La segunda forma es más popular ya que evita usar mucha sangría hacia la derecha en el código. Dicha sangría a menudo deja poco espacio en una línea de código, forzando a que las líneas se dividan y empeorando la legibilidad del programa.



Tip de rendimiento 4.1

Una instrucción `if...else` anidada puede ejecutarse con mucha más rapidez que una serie de instrucciones `if` de selección simple, debido a la posibilidad de salir antes de tiempo, una vez que se cumple una de las condiciones.



Tip de rendimiento 4.2

En una instrucción `if...else` anidada, debemos evaluar las condiciones que tengan más probabilidades de ser `true` al principio de la instrucción `if...else` anidada. Esto permite que la instrucción `if...else` anidada se ejecute con más rapidez, al salir antes de lo esperado si se evalúan primero los casos que ocurren con menos frecuencia.

Problema del `else` suelto

El compilador de C++ siempre asocia un `else` con el `if` que le precede inmediatamente, a menos que se le indique otra cosa mediante la colocación de llaves (`{ y }`). Este comportamiento puede ocasionar lo que se conoce como el **problema del `else` suelto**. Por ejemplo,

```
if ( x > 5 )
    if ( y > 5 )
        cout << "x y y son > 5";
else
    cout << "x es <= 5" );
```

parece indicar que si x es mayor que 5, la instrucción `if` anidada determina si y es también mayor que 5. De ser así, se produce como resultado la cadena " x y y son > 5". De lo contrario, parece ser que si x no es mayor que 5, la instrucción `else` que es parte del `if...else` produce como resultado la cadena " x es <= 5".

¡Cuidado! Esta instrucción `if...else` anidada no se ejecuta como parece ser. El compilador en realidad interpreta la instrucción así:

```
if ( x > 5 )
    if ( y > 5 )
        cout << "x y y son > 5";
else
    cout << "x es <= 5";
```

donde el cuerpo del primer `if` es un `if...else` anidado. La instrucción `if` más externa evalúa si x es mayor que 5. De ser así, la ejecución continúa evaluando si y es también mayor que 5. Si la segunda condición es verdadera, se muestra la cadena apropiada (" x y y son > 5"). No obstante, si la segunda condición es falsa se muestra la cadena " x es <= 5", aun y cuando sabemos que x es mayor que 5.

Para forzar a que la instrucción `if...else` anidada se ejecute como se tenía pensado originalmente, podemos escribirla de la siguiente manera:

```
if ( x > 5 )
{
    if ( y > 5 )
        cout << "x y y son > 5";
}
else
    cout << "x es <= 5";
```

Las llaves (`{}`) indican al compilador que la segunda instrucción `if` se encuentra en el cuerpo del primer `if`, y que el `else` está asociado con el primer `if`. Los ejercicios 4.23 a 4.24 investigan con más detalle el problema del `else` suelto.

Bloques

La instrucción de selección `if` normalmente espera sólo una instrucción en su cuerpo. De manera similar, las partes `if` y `else` de una instrucción `if...else` esperan sólo una instrucción en su cuerpo. Para incluir varias instrucciones en el cuerpo de un `if` (o en el cuerpo del `else` en una instrucción `if...else`), encierre las instrucciones entre llaves (`{` y `}`). A un conjunto de instrucciones contenidas dentro de un par de llaves se le llama **instrucción compuesta** o **bloque**. De aquí en adelante utilizaremos el término "bloque".



Observación de Ingeniería de Software 4.2

Un bloque puede colocarse en cualquier parte de un programa donde pueda colocarse una sola instrucción.

El siguiente ejemplo incluye un bloque en la parte `else` de una instrucción `if...else`:

```
if ( calificacionEstudiante >= 60 )
    cout << "Aprobado.\n";
else
{
    cout << "Reprobado.\n";
    cout << "Debe tomar este curso otra vez.\n";
}
```

En este caso, si `calificacionEstudiante` es menor que 60, el programa ejecuta ambas instrucciones en el cuerpo del `else` e imprime

Reprobado.
Debe tomar este curso otra vez.

Observe las llaves que rodean a las dos instrucciones en la cláusula `else`. Estas llaves son importantes. Sin ellas, la instrucción

```
cout << "Debe tomar este curso otra vez.\n";
```

estaría fuera del cuerpo de la parte `else` de la instrucción `if` y se ejecutaría sin importar que la calificación fuera menor a 60. Éste es un ejemplo de un error lógico.



Error común de programación 4.3

Olvidar una o las dos llaves que delimitan un bloque puede provocar errores de sintaxis o errores lógicos en un programa.



Buena práctica de programación 4.5

Colocar siempre las llaves en una instrucción if...else (o cualquier estructura de control) ayuda a evitar que se omitan de manera accidental, en especial, cuando posteriormente se agregan instrucciones a una cláusula if o else. Para evitar que esto suceda, algunos programadores prefieren escribir las llaves inicial y final de los bloques antes de escribir las instrucciones individuales dentro de ellas.

Así como un bloque puede colocarse en cualquier parte donde pueda colocarse una sola instrucción individual, también es posible no tener instrucción alguna; a ésta se le conoce como **instrucción nula** (o **instrucción vacía**). Para representar a la instrucción nula, se coloca un punto y coma (;) donde normalmente iría una instrucción.



Error común de programación 4.4

Colocar un punto y coma después de la condición en una instrucción if produce un error lógico en las instrucciones if de selección simple, y un error de sintaxis en las instrucciones if...else de selección doble (cuando la parte del if contiene una instrucción en el cuerpo).

4.7 Instrucción de repetición while

Una **instrucción de repetición** (también llamada **instrucción de ciclo**, o un **ciclo**) permite al programador especificar que un programa debe repetir una acción mientras cierta condición sea verdadera. La instrucción en pseudocódigo

Mientras existan más artículos en mi lista de compras

Comprar el siguiente artículo y quitarlo de mi lista

describe la repetición que ocurre durante una salida de compras. La condición “existan más artículos en mi lista de compras” puede ser verdadera o falsa. Si es verdadera, entonces se realiza la acción “Comprar el siguiente artículo y quitarlo de mi lista”. Esta acción se realizará en forma repetida mientras la condición sea verdadera. La instrucción contenida en la instrucción de repetición *Mientras (while)* constituye el cuerpo de esta estructura, el cual puede ser una sola instrucción o un bloque. En algún momento, la condición se hará falsa (cuando el último artículo de la lista de compras sea adquirido y eliminado de la lista). En este punto la repetición terminará y se ejecutará la primera instrucción que esté después de la instrucción de repetición.

Como ejemplo de la instrucción de repetición *while* en C++, considere un segmento de programa diseñado para encontrar la primera potencia de 3 que sea mayor a 100. Suponga que la variable *producto* de tipo *int* se inicializa en 3. Cuando la siguiente instrucción *while* termine de ejecutarse, *producto* contendrá el resultado:

```
int producto = 3;
while ( producto <= 100 )
    producto = 3 * producto;
```

Cuando esta instrucción *while* comienza a ejecutarse, el valor de *producto* es 3. Cada repetición de la instrucción *while* multiplica a *producto* por 3, por lo que *producto* toma los valores de 9, 27, 81 y 243, sucesivamente. Cuando *producto* se vuelve 243, la condición de la instrucción *while* (*producto <= 100*) se torna falsa. Esto termina la repetición, por lo que el valor final de *producto* es 243. En este punto, la ejecución del programa continúa con la siguiente instrucción después de la instrucción *while*.



Error común de programación 4.5

Si no se proporciona, en el cuerpo de una instrucción while, una acción que ocasione que en algún momento la condición del while se torne falsa, por lo general se producirá un error lógico conocido como ciclo infinito, en el que el ciclo nunca terminará. Esto puede hacer que un programa parezca “quedarse colgado” o “congelarse” si el cuerpo del ciclo no contiene instrucciones que interactúen con el usuario.

El diagrama de actividad de UML de la figura 4.6 muestra el flujo de control que corresponde a la instrucción *while* anterior. Una vez más, los símbolos en el diagrama (aparte del estado inicial, las flechas de transición, un estado final y tres notas) representan un estado de acción y una decisión. Este diagrama también introduce el **símbolo de fusión** de UML, el cual une dos flujos de actividad en un solo flujo de actividad. UML representa tanto al símbolo de fusión como al símbolo de decisión como rombos. En este diagrama, el símbolo de fusión une las transiciones del estado inicial y del estado de acción, de manera que ambas fluyan en la decisión que determina si el ciclo debe empezar a

ejecutarse (o seguir ejecutándose). Los símbolos de decisión y de fusión pueden diferenciarse por el número de flechas de transición “entrantes” y “salientes”. Un símbolo de decisión tiene una flecha de transición que apunta hacia el rombo y dos o más flechas de transición que apuntan hacia afuera del rombo, para indicar las posibles transiciones desde ese punto. Además, cada flecha de transición que apunta hacia afuera de un símbolo de decisión tiene una condición de guardia junto a ella. Un símbolo de fusión tiene dos o más flechas de transición que apuntan hacia el rombo, y sólo una flecha de transición que apunta hacia afuera del rombo, para indicar múltiples flujos de actividad que se fusionan para continuar la actividad. Observe que, a diferencia del símbolo de decisión, el de fusión no tiene su contraparte en el código de C++. Ninguna de las flechas de transición asociadas con un símbolo de fusión tienen condiciones de guardia.

El diagrama de la figura 4.6 muestra claramente la repetición de la instrucción `while` que vimos antes en esta sección. La flecha de transición que emerge del estado de acción apunta a la fusión, desde la cual regresa a la decisión que se evalúa en cada iteración del ciclo, hasta que la condición de guardia `producto > 100` se vuelva verdadera. Entonces, la instrucción `while` termina (llega a su estado final) y el control pasa a la siguiente instrucción en la secuencia del programa.

Imagine un recipiente profundo de diagramas de actividad de instrucciones de repetición `while` de UML vacíos; todos los que podría necesitar para apilar y anidar con los diagramas de actividad de otras instrucciones de control, para formar una implementación estructurada de un algoritmo. Usted llena los estados de acción y los símbolos de decisión con expresiones de acción y las condiciones de guardia apropiadas para el algoritmo.

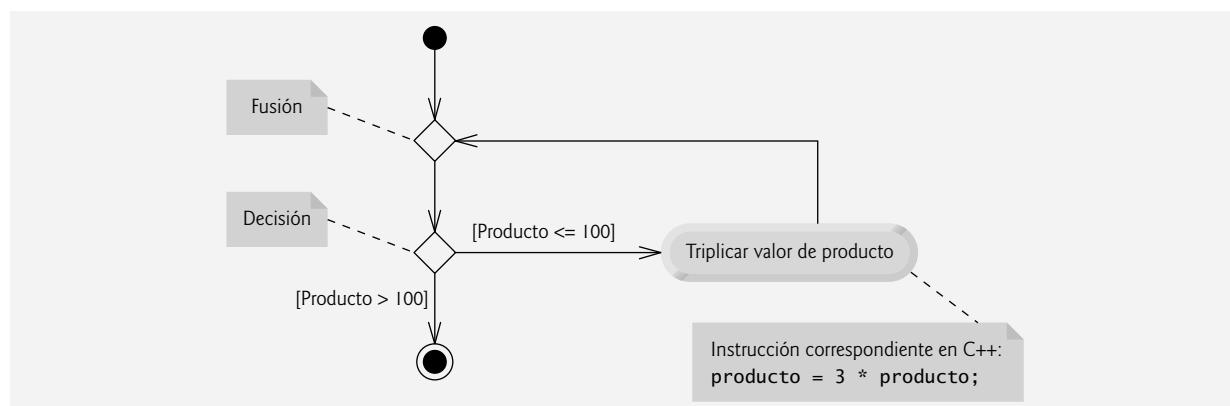


Figura 4.6 | Diagrama de actividad de UML de la instrucción de repetición `while`.



Tip de rendimiento 4.3

Muchos de los Tips de rendimiento que mencionamos en este texto sólo producen mejoras pequeñas, por lo que tal vez usted esté tentado a ignorarlos. Sin embargo, una pequeña mejora en el rendimiento para el código que se ejecuta muchas veces en un ciclo puede producir una mejora considerable en el rendimiento en general.

4.8 Cómo formular algoritmos: repetición controlada por un contador

Para ilustrar la forma en que los programadores desarrollan los algoritmos, en esta sección y en la sección 4.9 resolveremos dos variantes de un problema que promedia las calificaciones de una clase. Considere el siguiente enunciado del problema:

A una clase de diez estudiantes se les aplicó un examen. Las calificaciones (enteros en el rango de 0 a 100) de este examen están disponibles para su análisis. Calcule y muestre el total de las calificaciones de todos los estudiantes y el promedio de la clase para este examen.

El promedio de la clase es igual a la suma de las calificaciones, dividida entre el número de estudiantes. El algoritmo para resolver este problema en una computadora debe recibir como entrada cada una de las calificaciones, calcular el promedio e imprimir el resultado.

Algoritmo de seudocódigo con repetición controlada por un contador

Emplearemos seudocódigo para enlistar las acciones a ejecutar y especificar el orden en el que deben ocurrir. Usaremos una **repetición controlada por contador** para introducir las calificaciones, una por una. Esta técnica utiliza una variable

llamada **contador** para controlar el número de veces que debe ejecutarse un conjunto de instrucciones (a lo cual se le conoce también como el número de **iteraciones** del ciclo).

A la repetición controlada por contador se le llama comúnmente **repetición definida**, ya que el número de repeticiones se conoce antes de que el ciclo empiece a ejecutarse. En este ejemplo, la repetición termina cuando el contador excede a 10. Esta sección presenta un algoritmo de seudocódigo completamente desarrollado (figura 4.7), y una versión de la clase **LibroCalificaciones** (figuras 4.8 y 4.9) que implementa el algoritmo en una función miembro de C++. Después presentamos una aplicación (figura 4.10) que demuestra el algoritmo en acción. En la sección 4.9 demostraremos cómo utilizar el seudocódigo para desarrollar dicho algoritmo desde cero.



Observación de Ingeniería de software 4.3

La experiencia ha demostrado que la parte más difícil para la resolución de un problema en una computadora es desarrollar el algoritmo para la solución. Por lo general, una vez que se ha especificado el algoritmo correcto, el proceso de producir un programa funcional en C++ a partir de dicho algoritmo es relativamente sencillo.

Observe las referencias en el algoritmo de seudocódigo de la figura 4.7 para un total y un contador. Un **total** es una variable que se utiliza para acumular la suma de varios valores. Un **contador** es una variable que se utiliza para contar; en este caso, el contador de calificaciones indica cuál de las 10 calificaciones está a punto de escribir el usuario. Por lo general, las variables que se utilizan para guardar totales deben inicializarse en cero antes de utilizarse en un programa; de lo contrario, la suma incluiría el valor anterior almacenado en la ubicación de memoria del total.

- 1 Asignar a **total** el valor de cero
- 2 Asignar al **contador de calificaciones** el valor de uno
- 3
- 4 Mientras que el **contador de calificaciones** sea menor o igual a diez
 - 5 Pedir al usuario que introduzca la siguiente calificación
 - 6 Obtener como entrada la siguiente calificación
 - 7 Sumar la calificación al **total**
 - 8 Sumar uno al **contador de calificaciones**
- 9
- 10 Asignar al promedio de la clase el **total dividido entre diez**
- 11 Imprimir el **total de las calificaciones para todos los estudiantes en la clase**
- 12 Imprimir el **promedio de la clase**

Figura 4.7 | Algoritmo en seudocódigo que utiliza la repetición controlada por contador para resolver el problema del promedio de una clase.

```

1 // Fig. 4.8: LibroCalificaciones.h
2 // Definición de la clase LibroCalificaciones que determina el promedio de una clase.
3 // Las funciones miembro se definen en LibroCalificaciones.cpp
4 #include <string> // el programa usa la clase string estándar de C++
5 using std::string;
6
7 // definición de la clase LibroCalificaciones
8 class LibroCalificaciones
9 {
10 public:
11     LibroCalificaciones( string ); // el constructor inicializa el nombre del curso
12     void establecerNombreCurso( string ); // función para establecer el nombre del curso
13     string obtenerNombreCurso(); // función para obtener el nombre del curso
14     void mostrarMensaje(); // muestra un mensaje de bienvenida
15     void determinarPromedioClase(); // promedia las calificaciones escritas por el usuario
16 private:
17     string nombreCurso; // nombre del curso para este LibroCalificaciones
18 };// fin de la clase LibroCalificaciones

```

Figura 4.8 | Problema del promedio de una clase utilizando la repetición controlada por contador: archivo de encabezado de **LibroCalificaciones**.

```

1 // Fig. 4.9: LibroCalificaciones.cpp
2 // Definiciones de funciones miembro para la clase LibroCalificaciones que resuelve
3 // el problema del promedio de la clase con repetición controlada por contador.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include "LibroCalificaciones.h" // incluye la definición de la clase LibroCalificaciones
10
11 // el constructor inicializa a nombreCurso con la cadena que se suministra como argumento
12 LibroCalificaciones::LibroCalificaciones( string nombre )
13 {
14     establecerNombreCurso( nombre ); // valida y almacena nombreCurso
15 } // fin del constructor de LibroCalificaciones
16
17 // función para establecer el nombre del curso;
18 // asegura que el nombre del curso tenga cuando menos 25 caracteres
19 void LibroCalificaciones::establecerNombreCurso( string nombre )
20 {
21     if ( nombre.length() <= 25 ) // si nombre tiene 25 caracteres o menos
22         nombreCurso = nombre; // almacena el nombre del curso en el objeto
23     else // si nombre es mayor de 25 caracteres
24     { // establece nombreCurso a los primeros 25 caracteres del parámetro nombre
25         nombreCurso = nombre.substr( 0, 25 ); // selecciona los primeros 25 caracteres
26         cout << "El nombre \"'" << nombre << "\" excede la longitud maxima (25).\\n"
27             << "Se limito nombreCurso a los primeros 25 caracteres.\\n" << endl;
28     } // fin de if...else
29 } // fin de la función establecerNombreCurso
30
31 // función para obtener el nombre del curso
32 string LibroCalificaciones::obtenerNombreCurso()
33 {
34     return nombreCurso;
35 } // fin de la función obtenerNombreCurso
36
37 // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
38 void LibroCalificaciones::mostrarMensaje()
39 {
40     cout << "Bienvenido al libro de calificaciones para \\n" << obtenerNombreCurso() << "!\n"
41         << endl;
42 } // fin de la función mostrarMensaje
43
44 // determina el promedio de la clase, con base en las 10 calificaciones escritas por el usuario
45 void LibroCalificaciones::determinarPromedioClase()
46 {
47     int total; // suma de las calificaciones introducidas por el usuario
48     int contadorCalif; // número de la calificación a introducir a continuación
49     int calificacion; // el valor de la calificación introducida por el usuario
50     int promedio; // promedio de calificaciones
51
52     // fase de inicialización
53     total = 0; // inicializa el total
54     contadorCalif = 1; // inicializa el contador del ciclo
55
56     // fase de procesamiento
57     while ( contadorCalif <= 10 ) // itera 10 veces
58     {
59         cout << "Escriba una calificación: "; // pide la entrada
60         cin >> calificacion; // recibe como entrada la siguiente calificación
61         total = total + calificacion; // suma la calificación al total
62         contadorCalif = contadorCalif + 1; // incrementa el contador por 1

```

Figura 4.9 | Problema del promedio de una clase utilizando la repetición controlada por contador: archivo de código fuente de LibroCalificaciones. (Parte I de 2).

```

63 } // fin de while
64
65 // fase de terminación
66 promedio = total / 10; // la división de enteros produce un resultado entero
67
68 // muestra el total y el promedio de las calificaciones
69 cout << "\nEl total de las 10 calificaciones es " << total << endl;
70 cout << "El promedio de la clase es " << promedio << endl;
71 } // fin de la función determinarPromedioClase

```

Figura 4.9 | Problema del promedio de una clase utilizando la repetición controlada por contador: archivo de código fuente de *LibroCalificaciones*. (Parte 2 de 2).

Mejora a la validación de *LibroCalificaciones*

Antes de hablar sobre la implementación del algoritmo para obtener el promedio de la clase, vamos a considerar una mejora que hicimos a nuestra clase *LibroCalificaciones*. En la figura 3.16, la forma en que nuestra función miembro *establecerNombreCurso* validaría el nombre del curso sería probar primero si la longitud del mismo es menor o igual a 25 caracteres, mediante el uso de una instrucción *if*. Si esto fuera cierto, se establecería el nombre del curso. Después, a este código le seguiría otra instrucción *if* que evaluaría si la longitud del nombre del curso es mayor que 25 caracteres (en cuyo caso, el nombre del curso se reduciría). Observe que la condición de la segunda instrucción *if* es el opuesto exacto de la condición de la primera instrucción *if*. Si una condición se evalúa como *true*, la otra se debe evaluar como *false*. Dicha situación es ideal para una instrucción *if...else*, por lo que hemos modificado nuestro código, reemplazando las dos instrucciones *if* por una instrucción *if...else* (líneas 21 a 28 de la figura 4.9).

Implementación de la repetición controlada por contador en la clase *LibroCalificaciones*

La clase *LibroCalificaciones* (figuras 4.8 y 4.9) contiene un constructor (declarado en la línea 11 de la figura 4.8 y definido en las líneas 12 a 15 de la figura 4.9) que asigna un valor a la variable de instancia *nombreDelCurso* (declarada en la línea 17 de la figura 4.8) de la clase. En las líneas 19 a 29, 32 a 35 y 38 a 42 de la figura 4.9 se definen las funciones miembro *establecerNombreCurso*, *obtenerNombreCurso* y *mostrarMensaje*, respectivamente. En las líneas 45 a 71 se declara la función miembro *determinarPromedioClase*, la cual implementa el algoritmo para sacar el promedio de la clase, descrito por el seudocódigo de la figura 4.7.

En las líneas 47 a 50 se declaran las variables *total*, *contadorCalif*, *calificacion* y *promedio* de tipo *int*. La variable *calificacion* almacena la entrada del usuario. Observe que las anteriores declaraciones aparecen en el cuerpo de la función miembro *determinarPromedioClase*.

En las versiones de la clase *LibroCalificaciones* en este capítulo, simplemente leemos y procesamos un conjunto de calificaciones. El cálculo del promedio se realiza en el método *determinarPromedioClase*, usando variables locales; no preservamos información acerca de las calificaciones de los estudiantes en las variables de instancia de la clase. En el capítulo 7, Arreglos y vectores, modificaremos la clase *LibroCalificaciones* para mantener las calificaciones en memoria, utilizando una variable de instancia que hace referencia a una estructura de datos conocida como arreglo. Esto permite que un objeto *LibroCalificaciones* realice varios cálculos sobre el mismo conjunto de calificaciones, sin requerir que el usuario escriba las calificaciones varias veces.



Buena práctica de programación 4.6

Separé las declaraciones de las otras instrucciones en las funciones con una línea en blanco, para mejorar la legibilidad.

En las líneas 53 y 54 se inicializan *total* a 0 y *contadorCalif* a 1. Observe que las variables *total* y *contadorCalif* se inicializan antes de usarlas en un cálculo. Por lo general, las variables contador se inicializan con cero o uno, dependiendo de su uso (presentaremos ejemplos para demostrar cada una de estas posibilidades). Una variable no inicializada contiene un **valor “basura”** (también conocido como **valor indefinido**): el último valor almacenado en la ubicación de memoria reservada para esa variable. Las variables *calificacion* y *promedio* (para la entrada del usuario y el promedio calculado, respectivamente) no necesitan inicializarse aquí; sus valores se asignarán a medida que se introduzcan o se calculen más adelante en el método.



Error común de programación 4.6

Si no se inicializan los contadores y los totales, se pueden producir errores lógicos.



Tip para prevenir errores 4.2

Inicialice cada contador y total, ya sea en su declaración o en una instrucción de asignación. Por lo general, los totales se inicializan con 0. Los contadores comúnmente se inicializan con 0 o 1, dependiendo de cómo se utilicen (más adelante veremos ejemplos de cuándo usar 0 y cuándo usar 1).



Buena práctica de programación 4.7

Declare cada variable en una línea separada con su propio comentario, para mejorar la legibilidad de los programas.

La línea 57 indica que la instrucción `while` debe continuar ejecutando el ciclo (lo que también se conoce como iterar), siempre y cuando el valor de `contadorCalif` sea menor o igual a 10. Mientras esta condición sea verdadera, la instrucción `while` ejecutará en forma repetida las instrucciones entre las llaves que delimitan su cuerpo (líneas 58 a 63).

En la línea 59 se muestra el indicador "Escriba la calificación: ". Esta línea corresponde a la instrucción en seudocódigo "Pedir al usuario que introduzca la siguiente calificación". En la línea 60 se lee la calificación escrita por el usuario y se asigna a la variable `calificacion`. Esta línea corresponde a la instrucción en seudocódigo "Obtener como entrada la siguiente calificación". Recuerde que la variable `calificacion` no se inicializó antes en el programa, ya que éste obtiene el valor de `calificacion` del usuario durante cada iteración del ciclo. En la línea 61 se suma la nueva `calificacion` escrita por el usuario al `total`, y se asigna el resultado a `total`, que sustituye su valor anterior.

En la línea 62 se suma 1 a `contadorCalif` para indicar que el programa ha procesado una calificación y está listo para recibir la siguiente calificación del usuario. Al incrementar a `contadorCalif` en cada iteración, en un momento dado su valor excederá a 10. En ese momento, el ciclo `while` termina debido a que su condición (línea 57) se vuelve falsa.

Cuando el ciclo termina, en la línea 66 se realiza el cálculo del promedio y se asigna su resultado a la variable `promedio`. En la línea 69 se muestra el texto "El total de las 10 calificaciones es ", seguido del valor de la variable `total`. Después, en la línea 70 se muestra el texto "El promedio de la clase es ", seguido del valor de la variable `promedio`. A continuación, la función miembro `determinarPromedioClase` devuelve el control a la función que hizo la llamada (es decir, a `main` en la figura 4.10).

Demostración de la clase LibroCalificaciones

La figura 4.10 contiene la función `main` de esta aplicación, la cual crea un objeto de la clase `LibroCalificaciones` y demuestra sus capacidades. En la línea 9 de la figura 4.10 se crea un nuevo objeto llamado `miLibroCalificaciones`.

```

1 // Fig. 4.10: fig04_10.cpp
2 // Crea un objeto LibroCalificaciones e invoca a su función determinarPromedioClase.
3 #include "LibroCalificaciones.h" // incluye la definición de la clase LibroCalificaciones
4
5 int main()
6 {
7     // crea un objeto LibroCalificaciones llamado miLibroCalificaciones y
8     // pasa el nombre del curso al constructor
9     LibroCalificaciones miLibroCalificaciones( "CS101 Programacion en C++" );
10
11    miLibroCalificaciones.mostrarMensaje(); // muestra el mensaje de bienvenida
12    miLibroCalificaciones.determinarPromedioClase(); // busca el promedio de 10 calificaciones
13    return 0; // indica que el programa terminó correctamente
14 } // fin de main

```

Bienvenido al libro de calificaciones para
CS101 Programacion en C++!

Escriba una calificacion: 67
 Escriba una calificacion: 78
 Escriba una calificacion: 89
 Escriba una calificacion: 67
 Escriba una calificacion: 87

Figura 4.10 | Problema del promedio de una clase utilizando la repetición controlada por contador: creación de un objeto de la clase `LibroCalificaciones` (figuras 4.8 y 4.9) e invocación de su función miembro `determinarPromedioClase`. (Parte I de 2).

```

Escriba una calificacion: 98
Escriba una calificacion: 93
Escriba una calificacion: 85
Escriba una calificacion: 82
Escriba una calificacion: 100

El total de las 10 calificaciones es 846
El promedio de la clase es 84

```

Figura 4.10 | Problema del promedio de una clase utilizando la repetición controlada por contador: creación de un objeto de la clase `LibroCalificaciones` (figuras 4.8 y 4.9) e invocación de su función miembro `determinarPromedioClase`. (Parte 2 de 2).

La cadena en la línea 9 se pasa al constructor de `LibroCalificaciones` (líneas 12 a 15 de la figura 4.9). En la línea 11 de la figura 4.10 se hace una llamada a la función miembro `mostrarMensaje` de `miLibroCalificaciones` para mostrar un mensaje de bienvenida al usuario. Después, en la línea 11 se hace una llamada a la función miembro `determinarPromedioClase` de `miLibroCalificaciones` para permitir que el usuario introduzca 10 calificaciones, para las cuales la función miembro posteriormente calcula e imprime el promedio; la función miembro ejecuta el algoritmo que se muestra en el seudocódigo de la figura 4.7.

Observaciones acerca de la división de enteros y el truncamiento

El cálculo del promedio realizado por la función miembro `determinarPromedioClase`, en respuesta a la llamada a la función en la línea 12 de la figura 4.10, produce un resultado entero. La salida del programa indica que la suma de los valores de las calificaciones en la ejecución de ejemplo es 846 que, al dividirse entre 10, debe producir 84.6; un número con un punto decimal. Sin embargo, el resultado del cálculo `total / 10` (línea 66 de la figura 4.9) es el entero 84, ya que `total` y 10 son enteros. Al dividir dos enteros se produce una división entera: se pierde cualquier parte fraccionaria del cálculo (es decir, se *trunca*). En la siguiente sección veremos cómo obtener un resultado que incluye un punto decimal a partir del cálculo del promedio.

Error común de programación 4.7



Suponer que la división entera redondea (en lugar de truncar) y puede producir resultados erróneos. Por ejemplo, $7 \div 4$, que produce 1.75 en la aritmética convencional, se trunca a 1 en la aritmética entera, en lugar de redondearse a 2.

En la figura 4.9, si en la línea 66 se utilizó `contadorCalif` en lugar de 10 para el cálculo, el resultado para este programa mostraría un valor incorrecto, 76. Esto ocurriría debido a que en la iteración final de la instrucción `while`, `contadorCalif` se incrementó al valor 11 en la línea 62.

Error común de programación 4.8



El uso de una variable de control tipo contador de un ciclo en un cálculo después del ciclo produce un error lógico, conocido como error de desplazamiento en 1. En un ciclo controlado por contador que cuenta en uno cada vez que recorre el ciclo, éste termina cuando el valor del contador es uno más que su último valor legítimo (es decir, 11 en el caso de contar del 1 al 10).

4.9 Cómo formular algoritmos: repetición controlada por un centinela

Generalicemos el problema para los promedios de una clase. Considere el siguiente problema:

Desarrollar un programa que calcule el promedio de una clase y que procese las calificaciones para un número arbitrario de estudiantes cada vez que se ejecute.

En el ejemplo anterior del promedio de una clase, el enunciado del problema especificó el número de estudiantes, por lo que se conocía el número de calificaciones (10) por adelantado. En este ejemplo no se indica cuántas calificaciones va a introducir el usuario durante la ejecución del programa. El programa debe procesar un número arbitrario de calificaciones. ¿Cómo puede el programa determinar cuándo terminar de introducir calificaciones? ¿Cómo sabrá cuándo calcular e imprimir el promedio de la clase?

Una manera de resolver este problema es utilizar un valor especial denominado **valor centinela** (también llamado **valor de señal**, **valor de prueba** o **valor de bandera**) para indicar el “fin de la introducción de datos”. El usuario escribe

calificaciones hasta que se haya introducido el número correcto de ellas. Después, el usuario escribe el valor centinela para indicar que no se van a introducir más calificaciones. A la repetición controlada por centinela a menudo se le llama **repetición indefinida**, ya que el número de repeticiones no se conoce antes de que comience la ejecución del ciclo.

Evidentemente, debe elegirse un valor centinela de tal forma que no pueda confundirse con un valor de entrada permitido. Las calificaciones de un examen son enteros positivos, por lo que -1 es un valor centinela aceptable para este problema. Por lo tanto, una ejecución del programa para promediar una clase podría procesar una cadena de entradas como 95, 96, 75, 74, 89 y -1 . El programa entonces calcularía e imprimiría el promedio de la clase para las calificaciones 95, 96, 75, 74 y 89. Como -1 es el valor centinela, no debe entrar en el cálculo del promedio.



Error común de programación 4.9

Seleccionar un valor centinela que sea también un valor de datos permitido es un error lógico.

Desarrollo del algoritmo en seudocódigo con el método de mejoramiento de arriba a abajo, paso a paso: la primera mejora (cima)

Desarrollamos el programa para promediar clases con una técnica llamada **mejoramiento de arriba a abajo, paso a paso**, la cual es esencial para el desarrollo de programas bien estructurados. Comenzamos con una representación en seudocódigo de la **cima**, una sola instrucción que transmite la función del programa en general:

Determinar el promedio de la clase para el examen, para un número arbitrario de estudiantes

La cima es, en efecto, la representación *completa* de un programa. Desafortunadamente, la cima (como es éste el caso) pocas veces transmite los detalles suficientes como para escribir un programa. Por lo tanto, ahora comenzaremos el proceso de mejora. Dividiremos la cima en una serie de tareas más pequeñas y las enlistaremos en el orden en el que se van a realizar. Esto arroja como resultado la siguiente **primera mejora**:

Inicializar variables

Introducir, sumar y contar las calificaciones del examen

Calcular e imprimir el total de las calificaciones de todos los estudiantes y el promedio de la clase

Esta mejora utiliza sólo la estructura de secuencia; los pasos aquí mostrados deben ejecutarse en orden, uno después del otro.



Observación de Ingeniería de Software 4.4

Cada mejora, así como la cima en sí, es una especificación completa del algoritmo; sólo varía el nivel del detalle.



Observación de Ingeniería de Software 4.5

Muchos programas pueden dividirse lógicamente en tres fases: una fase de inicialización, en la que se inicializan las variables del programa; una fase de procesamiento, en la que se introducen los valores de los datos y se ajustan las variables del programa (como contadores y totales) según sea necesario; y una fase de terminación, que calcula y produce los resultados finales.

Cómo proceder a la segunda mejora

La anterior *Observación de Ingeniería de Software* es a menudo todo lo que usted necesita para la primera mejora en el proceso de arriba a abajo. Para avanzar al siguiente nivel de mejora, es decir, la **segunda mejora**, nos comprometemos a usar variables específicas. En este ejemplo necesitamos el total actual de los números, una cuenta de cuántos números se han procesado, una variable para recibir el valor de cada calificación, a medida que el usuario las vaya introduciendo, y una variable para almacenar el promedio calculado. La instrucción en seudocódigo

Inicializar las variables

puede mejorarse como sigue:

Inicializar total en cero

Inicializar contador en cero

Sólo las variables *total* y *contador* necesitan inicializarse antes de que puedan utilizarse. Las variables *promedio* y *calificación* (para el promedio calculado y la entrada del usuario, respectivamente) no necesitan inicializarse, ya que sus valores se reemplazarán a medida que se calculen o introduzcan.

La instrucción en seudocódigo

Introducir, sumar y contar las calificaciones del examen

requiere una estructura de repetición (es decir, un ciclo) que introduzca cada calificación en forma sucesiva. No sabemos de antemano cuántas calificaciones van a procesarse, por lo que utilizaremos la repetición controlada por centinela. El usuario

introduce las calificaciones una por una. Después de introducir la última calificación, el usuario introduce el valor centinela. El programa evalúa el valor centinela después de la introducción de cada calificación, y termina el ciclo cuando el usuario introduce el valor centinela. Entonces, la segunda mejora de la instrucción anterior en seudocódigo sería

*Pedir al usuario que introduzca la primera calificación
Recibir como entrada la primera calificación (puede ser el centinela)*

*While (mientras) el usuario no haya introducido aún el centinela
 Sumar esta calificación al total actual
 Sumar uno al contador de calificaciones
 Pedir al usuario que introduzca la siguiente calificación
 Recibir como entrada la siguiente calificación (puede ser el centinela)*

En seudocódigo no utilizamos llaves alrededor de las instrucciones que forman el cuerpo de la estructura *While*. Simplemente aplicamos sangría a las instrucciones bajo el *While* para mostrar que pertenecen a esta instrucción. De nuevo, el seudocódigo es solamente una herramienta informal para desarrollar programas.

La instrucción en seudocódigo

Calcular e imprimir el total de las calificaciones de todos los estudiantes y el promedio de la clase

puede redefinirse de la siguiente manera:

*Si el contador no es igual a cero
 Asignar al promedio el total dividido entre el contador
 Imprimir el total de las calificaciones para todos los estudiantes en la clase
 Imprimir el promedio de la clase
De lo contrario
 Imprimir "No se introdujeron calificaciones"*

Aquí tenemos cuidado de evaluar la posibilidad de una división entre cero; por lo general esto es un **error lógico fatal** que, si no se detecta, haría que el programa fallara o produjera resultados inválidos (a lo que a menudo se le conoce como “bombing” o “crashing”). La segunda mejora completa del seudocódigo para el problema del promedio de una clase se muestra en la figura 4.11.



Error común de programación 4.10

Un intento de dividir entre cero produce generalmente un error fatal en tiempo de ejecución.

```

1 Inicializar total en cero
2 Inicializar contador en cero
3
4 Pedir al usuario que introduzca la primera calificación
5 Recibir como entrada la primera calificación (puede ser el centinela)
6
7 Mientras el usuario no haya introducido aún el centinela
8     Sumar esta calificación al total actual
9     Sumar uno al contador de calificaciones
10    Pedir al usuario que introduzca la siguiente calificación
11    Recibir como entrada la siguiente calificación (puede ser el centinela)
12
13 Si el contador no es igual a cero
14     Asignar al promedio el total dividido entre el contador
15     Imprimir el total de las calificaciones para todos los estudiantes de la clase
16     Imprimir el promedio
17 De lo contrario
18     Imprimir "No se introdujeron calificaciones"

```

Figura 4.11 | Algoritmo en seudocódigo del problema para promediar una clase, con una repetición controlada por centinela.



Tip para prevenir errores 4.3

Al realizar una división entre una expresión cuyo valor pudiera ser cero, debemos evaluar explícitamente esta posibilidad y manejárla de manera apropiada en el programa (como imprimir un mensaje de error), en lugar de permitir que ocurra el error fatal.

En las figuras 4.7 y 4.11 incluimos algunas líneas en blanco y sangría en el seudocódigo para facilitar su lectura. Las líneas en blanco separan los algoritmos en seudocódigo en sus diversas fases, y la sangría enfatiza los cuerpos de las instrucciones de control.

El algoritmo en seudocódigo en la figura 4.11 resuelve el problema más general para promediar una clase. Este algoritmo se desarrolló después de aplicar dos niveles de mejoramiento. En ocasiones, se requieren más niveles de mejoramiento.



Observación de Ingeniería de Software 4.6

Termine el proceso de mejoramiento de arriba a abajo, paso a paso, cuando haya especificado el algoritmo en seudocódigo con el detalle suficiente como para poder convertir el seudocódigo en C++. Por lo general, la implementación del programa en C++ después de esto es mucho más sencilla.



Observación de Ingeniería de Software 4.7

Algunos programadores experimentados escriben programas sin utilizar herramientas de desarrollo de programas como el seudocódigo. Estos programadores sienten que su meta final es resolver el problema en una computadora y que el escribir seudocódigo simplemente retarda la producción de los resultados finales. Aunque este método pudiera funcionar para problemas sencillos y conocidos, tiende a ocasionar graves errores y retrasos en proyectos grandes y complejos.

Implementación de la repetición controlada por centinela en la clase LibroCalificaciones

En las figuras 4.12 y 4.13 se muestra la clase `LibroCalificaciones` de C++ que contiene la función miembro `determinarPromedioClase`, la cual implementa el algoritmo en seudocódigo de la figura 4.11. Aunque cada calificación introducida es un valor entero, existe la probabilidad de que el cálculo del promedio produzca un número con un punto decimal; en otras palabras, un número real o **número de punto flotante** (por ejemplo, 7.33, 0.0975 o 1000.12345). El tipo `int` no puede representar un número de este tipo, por lo que esta clase debe usar otro tipo para hacerlo. C++ proporciona varios tipos de datos para almacenar números de punto flotante en la memoria, incluyendo `float` y `double`. La principal diferencia entre estos tipos es que, en comparación con las variables `float`, las variables `double` pueden almacenar comúnmente números con una mayor magnitud y un detalle más fino (es decir, más dígitos a la derecha del punto decimal; a esto se le conoce también como la **precisión** del número). Este programa introduce un operador especial llamado **operador de conversión**, para forzar a que el cálculo del promedio produzca un resultado numérico de punto flotante. Explicaremos estas características con detalle, al hablar sobre el programa.

```

1 // Fig. 4.12: LibroCalificaciones.h
2 // Definición de la clase LibroCalificaciones que determina el promedio de una clase.
3 // Las funciones miembro se definen en LibroCalificaciones.cpp
4 #include <string> // el programa usa la clase string estándar de C++
5 using std::string;
6
7 // definición de la clase LibroCalificaciones
8 class LibroCalificaciones
9 {
10 public:
11     LibroCalificaciones( string ); // el constructor inicializa el nombre del curso
12     void establecerNombreCurso( string ); // función para establecer el nombre del curso
13     string obtenerNombreCurso(); // función para obtener el nombre del curso
14     void mostrarMensaje(); // muestra un mensaje de bienvenida
15     void determinarPromedioClase(); // promedia las calificaciones escritas por el usuario
16 private:
17     string nombreCurso; // nombre del curso para este LibroCalificaciones
18 };// fin de la clase LibroCalificaciones

```

Figura 4.12 | Problema del promedio de una clase utilizando la repetición controlada por centinela: archivo de encabezado de `LibroCalificaciones`.

En este ejemplo vemos que las estructuras de control pueden apilarse una encima de otra (en secuencia), al igual que un niño apila bloques de construcción. La instrucción `while` (líneas 67 a 75 de la figura 4.13) va seguida por una instrucción `if...else` (líneas 78 a 90) en secuencia. La mayor parte del código en este programa es igual al código de la figura 4.9, por lo que nos concentraremos en las nuevas características y conceptos.

```

1 // Fig. 4.13: LibroCalificaciones.cpp
2 // Definiciones de funciones miembro para la clase LibroCalificaciones que resuelve
3 // el problema del promedio de la clase con repetición controlada por centinela.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8 using std::fixed; // asegura que se muestre el punto decimal
9
10 #include <iomanip> // manipuladores de flujo parametrizados
11 using std::setprecision; // establece la precisión numérica de salida
12
13 // incluye la definición de la clase LibroCalificaciones de LibroCalificaciones.h
14 #include "LibroCalificaciones.h"
15
16 // el constructor inicializa a nombreCurso con la cadena que se suministra como argumento
17 LibroCalificaciones::LibroCalificaciones( string nombre )
18 {
19     establecerNombreCurso( nombre ); // valida y almacena nombreCurso
20 } // fin del constructor de LibroCalificaciones
21
22 // función para establecer el nombre del curso;
23 // asegura que el nombre del curso tenga cuando mucho 25 caracteres
24 void LibroCalificaciones::establecerNombreCurso( string nombre )
25 {
26     if ( nombre.length() <= 25 ) // si el nombre tiene 25 caracteres o menos
27         nombreCurso = nombre; // almacena el nombre del curso en el objeto
28     else // si el nombre es mayor de 25 caracteres
29         { // establece nombreCurso a los primeros 25 caracteres del parámetro nombre
30             nombreCurso = nombre.substr( 0, 25 ); // selecciona los primeros 25 caracteres
31             cout << "El nombre \" " << nombre << "\" excede la longitud maxima (25).\n"
32                 << "Se limitó nombreCurso a los primeros 25 caracteres.\n" << endl;
33         } // fin de if...else
34 } // fin de la función establecerNombreCurso
35
36 // función para obtener el nombre del curso
37 string LibroCalificaciones::obtenerNombreCurso()
38 {
39     return nombreCurso;
40 } // fin de la función obtenerNombreCurso
41
42 // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
43 void LibroCalificaciones::mostrarMensaje()
44 {
45     cout << "Bienvenido al libro de calificaciones para\n" << obtenerNombreCurso() << "!\n"
46         << endl;
47 } // fin de la función mostrarMensaje
48
49 // determina el promedio de la clase con base en las 10 calificaciones escritas por el usuario
50 void LibroCalificaciones::determinarPromedioClase()
51 {
52     int total; // suma de las calificaciones introducidas por el usuario
53     int contadorCalif; // número de calificaciones introducidas
54     int calificacion; // valor de la calificación

```

Figura 4.13 | Problema del promedio de una clase utilizando la repetición controlada por centinela: archivo de código fuente de `LibroCalificaciones`. (Parte I de 2).

```

55     double promedio; // número con punto decimal para el promedio
56
57     // fase de inicialización
58     total = 0; // inicializa el total
59     contadorCalif = 0; // inicializa el contador del ciclo
60
61     // fase de procesamiento
62     // pide la entrada y lee la calificación del usuario
63     cout << "Escriba la calificación o -1 para salir: ";
64     cin >> calificacion; // recibe como entrada la calificación o el valor centinela
65
66     // itera hasta leer el valor centinela del usuario
67     while ( calificacion != -1 ) // mientras calificación no sea -1
68     {
69         total = total + calificacion; // suma la calificación al total
70         contadorCalif = contadorCalif + 1; // incrementa el contador
71
72         // pide la entrada y lee la siguiente calificación del usuario
73         cout << "Escriba la calificación o -1 para salir: ";
74         cin >> calificacion; // recibe como entrada la calificación o el valor centinela
75     } // fin de while
76
77     // fase de terminación
78     if ( contadorCalif != 0 ) // si el usuario introdujo al menos una calificación...
79     {
80         // calcula el promedio de todas las calificaciones introducidas
81         promedio = static_cast< double >( total ) / contadorCalif;
82
83         // muestra el total y el promedio (con dos dígitos de precisión)
84         cout << "\nEl total de las " << contadorCalif << " calificaciones introducidas es "
85             << total << endl;
86         cout << "El promedio de la clase es " << setprecision( 2 ) << fixed << promedio
87             << endl;
88     } // fin de if
89     else // no se introdujeron calificaciones, por lo que imprime el mensaje apropiado
90         cout << "No se introdujeron calificaciones" << endl;
91 } // fin de la función determinarPromedioClase

```

Figura 4.13 | Problema del promedio de una clase utilizando la repetición controlada por centinela: archivo de código fuente de *LibroCalificaciones*. (Parte 2 de 2).

```

1 // Fig. 4.14: fig04_14.cpp
2 // Crea un objeto LibroCalificaciones e invoca a su función determinarPromedioClase.
3
4 // incluye la definición de la clase LibroCalificaciones de LibroCalificaciones.h
5 #include "LibroCalificaciones.h"
6
7 int main()
8 {
9     // crea el objeto LibroCalificaciones llamado miLibroCalificaciones y
10    // pasa el nombre del curso al constructor
11    LibroCalificaciones miLibroCalificaciones( "CS101 Programación en C++" );
12
13    miLibroCalificaciones.mostrarMensaje(); // muestra el mensaje de bienvenida
14    miLibroCalificaciones.determinarPromedioClase(); // busca el promedio de 10 calificaciones
15    return 0; // indica que el programa terminó correctamente
16 } // fin de main

```

Figura 4.14 | Problema del promedio de una clase utilizando la repetición controlada por centinela: creación de un objeto de la clase *LibroCalificaciones* (figuras 4.12 y 4.13) e invocación de su función miembro *determinarPromedioClase*. (Parte 1 de 2).

```
Bienvenido al libro de calificaciones para
CS101 Programacion en C++!
Escriba la calificacion o -1 para salir: 97
Escriba la calificacion o -1 para salir: 88
Escriba la calificacion o -1 para salir: 72
Escriba la calificacion o -1 para salir: -1
El total de las 3 calificaciones introducidas es 257
El promedio de la clase es 85.67
```

Figura 4.14 | Problema del promedio de una clase utilizando la repetición controlada por centinela: creación de un objeto de la clase LibroCalificaciones (figuras 4.12 y 4.13) e invocación de su función miembro determinerPromedioClase. (Parte 2 de 2).

En la línea 55 se declara la variable `promedio` de tipo `double`. Recuerde que utilizamos una variable `int` en el ejemplo anterior para almacenar el promedio de la clase. El uso del tipo `double` en el ejemplo actual nos permite guardar el resultado del cálculo del promedio de la clase como un número de punto flotante. En la línea 59 se inicializa `contadorCalif` en 0, ya que todavía no se han introducido calificaciones. Recuerde que este programa utiliza la repetición controlada por centinela. Para mantener un registro preciso del número de calificaciones introducidas, el programa incrementa `contadorCalif` sólo cuando el usuario introduce un valor permitido para la calificación (es decir, que no sea el valor centinela) y el programa completa el procesamiento de la calificación. Por último, observe que antes de ambas instrucciones (líneas 64 y 74) se coloca una instrucción de salida que pide al usuario los datos de entrada.



Buena práctica de programación 4.8

Pida al usuario cada dato de entrada del teclado. El indicador debe indicar la forma de la entrada y cualquier valor de entrada especial. Por ejemplo, en un ciclo controlado por centinela, los indicadores que solicitan datos de entrada deben recordar explícitamente al usuario cuál es el valor centinela.

Comparación entre la lógica del programa para la repetición controlada por centinela, y la repetición controlada por contador

Compare la lógica de esta aplicación para la repetición controlada por centinela con la repetición controlada por contador en la figura 4.9. En la repetición controlada por contador, cada iteración de la instrucción `while` (líneas 57 a 63 de la figura 4.9) lee un valor del usuario, para el número especificado de iteraciones. En la repetición controlada por centinela, el programa lee el primer valor (líneas 63 y 64 de la figura 4.13) antes de llegar al `while`. Este valor determina si el flujo de control del programa debe entrar al cuerpo del `while`. Si la condición del `while` es falsa, el usuario introdujo el valor centinela, por lo que el cuerpo del `while` no se ejecuta (es decir, no se introdujeron calificaciones). Si, por otro lado, la condición es verdadera, el cuerpo comienza a ejecutarse y el ciclo suma el valor de `calificacion` al `total` (línea 69). Después, en las líneas 73 y 74 en el cuerpo del ciclo se pide y recibe el siguiente valor del usuario. A continuación, el control del programa se acerca a la llave derecha de terminación (`}`) del cuerpo del ciclo en la línea 75, por lo que la ejecución continúa con la evaluación de la condición del `while` (línea 67). La condición utiliza el valor más reciente de `calificacion` que acaba de introducir el usuario, para determinar si el cuerpo de la instrucción `while` debe ejecutarse otra vez. Observe que el valor de la variable `calificacion` siempre lo introduce el usuario inmediatamente antes de que el programa evalúe la condición del `while`. Esto permite al programa determinar si el valor que acaba de introducir el usuario es el valor centinela, *antes* de que el programa procese ese valor (es decir, que lo sume al `total` e incremente `contadorCalif`). Si el valor introducido es el valor centinela, el ciclo termina y el programa no suma `-1` al `total`.

Una vez que termina el ciclo, se ejecuta la instrucción `if...else` en las líneas 78 a 90. La condición en la línea 78 determina si se introdujeron calificaciones o no. Si no se introdujo ninguna, se ejecuta la parte del `else` (líneas 89 y 90) de la instrucción `if...else` y muestra el mensaje "No se introdujeron calificaciones", y la función miembro devuelve el control a la función que la llamó.

Observe el bloque de la instrucción `while` en la figura 4.13. Sin las llaves, las últimas tres instrucciones en el bloque quedarían fuera del cuerpo del ciclo, ocasionando que la computadora interprete el código incorrectamente, como se muestra a continuación:

```
// itera hasta leer el valor centinela del usuario
while ( calificacion != -1 )
    total = total + calificacion; // suma calificacion al total
    contadorCalif = contadorCalif + 1; // incrementa el contador
```

```
// pide la entrada y lee la siguiente calificación del usuario
cout << "Escriba calificación o -1 para terminar: ";
cin >> calificacion;
```

El código anterior ocasionaría un ciclo infinito en el programa, si el usuario no introduce el centinela `-1` para la primera calificación (en línea 64).



Error común de programación 4.11

Omitir las llaves que delimitan a un bloque puede provocar errores lógicos, como ciclos infinitos. Para prevenir este problema, algunos programadores encierran el cuerpo de todas las instrucciones de control con llaves, aun si el cuerpo sólo contiene una instrucción.

Precisión de los números de punto flotante y requerimientos de memoria

Las variables de tipo `float` representan **números de punto flotante con precisión simple** y tienen siete dígitos significativos en la mayoría de los sistemas de 32 bits. Las variables de tipo `double` representan **números de punto flotante con precisión doble**. Éstos requieren el doble de memoria que las variables `float` y pueden proporcionar 15 dígitos significativos en la mayoría de los sistemas de 32 bits: aproximadamente el doble de precisión que las variables `float`. Para el rango de valores requeridos por la mayoría de los programas debe ser suficiente usar variables de tipo `float`, pero podemos usar variables `double` para “estar a la segura”. En algunos programas, aun las variables de tipo `double` serán inadecuadas; dichos programas están más allá del alcance de este libro. La mayoría de los programadores representan a los números de punto flotante con el tipo `double`. De hecho, C++ considera a todos los números de punto flotante que escribimos en el código fuente de un programa (como 7.33 y 0.0975) como valores `double` de manera predeterminada. Dichos valores en el código fuente se conocen como **constants de punto flotante**. En el apéndice C, Tipos fundamentales, podrá consultar los rangos de valor para los números `float` y `double`.

A menudo, los números de punto flotante ocurren a través de la división. En la aritmética convencional, cuando dividimos 10 entre 3, el resultado es 3.3333333..., donde la secuencia de números se repite en forma infinita. La computadora asigna sólo una cantidad fija de espacio para guardar ese valor, por lo que evidentemente el valor de punto flotante guardado almacenado sólo puede ser una aproximación.



Error común de programación 4.12

Usar números de punto flotante de una forma que suponga que se representan con precisión de manera exacta (por ejemplo, usándolos en las comparaciones de igualdad) puede producir resultados imprecisos. Los números de punto flotante se representan sólo en forma aproximada en la mayoría de las computadoras.

Aunque los números de punto flotante no siempre son 100 por ciento precisos, tienen numerosas aplicaciones. Por ejemplo, cuando hablamos de una temperatura corporal “normal” de 36.7, no necesitamos tener una precisión con una gran cantidad de dígitos. Cuando leemos la temperatura en un termómetro como 36.7, en realidad podría ser 36.6999473210643. Considerar a este número simplemente como 36.7 está bien para la mayoría de las aplicaciones en las que se utilizan temperaturas corporales. Debido a la naturaleza imprecisa de los números de punto flotante, se prefiere el tipo `double` al tipo `float`, ya que las variables `double` pueden representar números de punto flotante con más precisión. Por esta razón, usamos el tipo `double` en el resto del libro.

Conversión explícita e implícita entre los tipos fundamentales

La variable `promedio` se declara como de tipo `double` (línea 55 de la figura 4.13) para capturar el resultado fraccionario de nuestro cálculo. Sin embargo, `total` y `contadorCalif` son variables enteras. Recuerde que al dividir dos enteros se produce una división entera, en la cual cualquier parte fraccionaria del cálculo se pierde (es decir, se **trunca**). En la siguiente instrucción

```
promedio = total / contadorCalif;
```

primero se realiza el cálculo de la división, por lo que se pierde la parte fraccionaria del resultado antes de asignarlo a `promedio`. Para realizar un cálculo de punto flotante con valores enteros, debemos crear valores temporales que sean números de punto flotante para el cálculo. C++ cuenta con el **operador unario de conversión de tipo** para llevar a cabo esta tarea. En la línea 81 se utiliza el operador de conversión de tipo `static_cast<double>(total)` para crear una copia de punto flotante *temporal* de su operando entre paréntesis: `total`. Utilizar un operador de conversión de tipo de esta forma es un proceso que se denomina **conversión explícita**. El valor almacenado en `total` sigue siendo un entero.

El cálculo ahora consiste de un valor de punto flotante (la versión temporal `double` de `total`) dividido por el entero `contadorCalif`. El compilador de C++ sabe cómo evaluar sólo expresiones en las que los tipos de datos de los operandos sean idénticos. Para asegurar que los operandos sean del mismo tipo, el compilador realiza una operación llamada **promoción** (o **conversión implícita**) en los operandos seleccionados. Por ejemplo, en una expresión que contenga valores de los tipos de datos `int` y `double`, C++ promueve los operandos `int` a valores `double`. En nuestro ejemplo, tratamos a `total` como `double` (mediante el operador unario de conversión de tipos), por lo que el compilador promueve el valor de `contadorCalif` al tipo `double`, con lo cual permite realizar el cálculo; el resultado de la división de punto flotante se asigna a `promedio`. En el capítulo 6, Funciones y una introducción a la recursividad, hablaremos sobre todos los tipos de datos fundamentales y su orden de promoción.



Error común de programación 4.13

El operador de conversión de tipo puede utilizarse para convertir entre los tipos numéricos fundamentales, como int y double, y para convertir entre los tipos de clases relacionados (como lo describiremos en el capítulo 13, Programación orientada a objetos: polimorfismo). La conversión al tipo incorrecto puede ocasionar errores de compilación o errores en tiempo de ejecución.

Los operadores de conversión de tipo están disponibles para cualquier tipo, además de los tipos de clases. El operador `static_cast` se forma colocando la palabra clave `static_cast` entre los signos `<` y `>`, alrededor del nombre de un tipo de datos. Este operador es un **operador unario** (es decir, un operador que utiliza sólo un operando). En el capítulo 2 estudiamos los operadores aritméticos binarios. C++ también soporta las versiones unarias de los operadores de suma (+) y resta (-), por lo que el programador puede escribir expresiones como `-7` o `+5`. Los operadores de conversión de tipo tienen mayor precedencia que los demás operadores unarios, como `+ y -`. Esta precedencia es un nivel mayor que la de los **operadores de multiplicación** `*`, `/` y `%`, y menor que la de los paréntesis. En nuestras tablas de precedencia, indicamos el operador de conversión de tipos con la notación `static_cast< tipo >()` (por ejemplo, vea la figura 4.22).

Formato para los números de punto flotante

Aquí veremos brevemente las herramientas de formato en la figura 4.13, y las explicaremos con detalle en el capítulo 15, Entrada y salida de flujos. La llamada a `setprecision` en la línea 86 (con un argumento de 2) indica que la variable `double` llamada `promedio` debe imprimirse con dos dígitos de precisión a la derecha del punto decimal (por ejemplo, 92.37). A esta llamada se le conoce como **manipulador de flujo parametrizado** (debido al 2 entre paréntesis). Los programas que utilizan estas llamadas deben contener la siguiente directiva del preprocesador (línea 10):

```
#include <iomanip>
```

En la línea 11 se especifica el nombre del archivo de encabezado `<iomanip>` que se utiliza en este programa. Observe que `endl` es un **manipulador de flujos no parametrizado** (ya que no va seguido de un valor o expresión entre paréntesis), por lo cual no requiere el archivo de encabezado `<iomanip>`. Si no se especifica la precisión, los valores de punto flotante se imprimen generalmente con seis dígitos de precisión (es decir, la **precisión predeterminada** en la mayoría de los sistemas de 32 bits actuales), aunque en un momento veremos una excepción a esto.

El manipulador de flujo `fixed` (línea 86) indica que los valores de punto flotante deben imprimirse en lo que se denomina **formato de punto fijo**, en oposición a la **notación científica**. La notación científica es una forma de mostrar un número como valor de punto flotante entre los valores de 1.0 y 10.0, multiplicado por una potencia de 10. Por ejemplo, el valor 3,100.0 se mostraría en notación científica como $3.1 \cdot 10^3$. La notación científica es útil cuando se muestran valores muy grandes o muy pequeños. En el capítulo 15 hablaremos sobre el formato mediante el uso de la notación científica. Por otra parte, el formato de punto fijo se utiliza para forzar a que un número de punto flotante muestre un número específico de dígitos. Al especificar el formato de punto fijo también forzamos a que se imprima el punto decimal y los ceros a la derecha, aun si el valor es una cantidad entera, como 88.00. Sin la opción de formato de punto fijo, dicho valor se imprime en C++ como 88, sin los ceros a la derecha ni el punto decimal. Al utilizar los manipuladores de flujo `fixed` y `setprecision` en un programa, el valor impreso se redondea al número de posiciones decimales indicado por el valor que se pasa a `setprecision` (por ejemplo, el valor 2 en la línea 86), aunque el valor en memoria permanece sin cambios. Por ejemplo, los valores 87.946 y 67.543 se imprimen como 87.95 y 67.54, respectivamente. Observe que también es posible forzar a que aparezca un punto decimal mediante el uso del manipulador `showpoint`. Si se especifica `showpoint` sin `fixed`, entonces no se imprimirán ceros a la derecha. Al igual que `endl`, los manipuladores `fixed` y `showpoint` no están parametrizados y no requieren el archivo de encabezado `<iomanip>`. Ambos se encuentran en el encabezado `<iostream>`.

En las líneas 86 y 87 de la figura 4.13 se imprime el promedio de la clase. En este ejemplo mostramos el promedio redondeado a la centésima más cercana, y lo imprimimos con sólo dos dígitos a la derecha del punto decimal. El manipulador de flujo parametrizado (línea 86) indica que el valor de la variable `promedio` se debe mostrar con dos dígitos de precisión

a la derecha del punto decimal; esto se indica mediante `setprecision(2)`. Las tres calificaciones introducidas durante la ejecución de ejemplo del programa de la figura 4.14 dan un total de 257, que a su vez produce el promedio 85.66666... El manipulador de flujo parametrizado `setprecision` hace que el valor se redondee al número especificado de dígitos. En este programa, el promedio se redondea a la posición de las centésimas y se muestra como 85.67.

4.10 Cómo formular algoritmos: instrucciones de control anidadas

En el siguiente ejemplo formularemos una vez más un algoritmo utilizando seudocódigo y el mejoramiento de arriba a abajo, paso a paso, y después escribiremos el correspondiente programa en C++. Hemos visto que las instrucciones de control pueden apilarse una encima de otra (en secuencia), de igual forma que un niño apila bloques de construcción. En este ejemplo práctico examinaremos la otra forma en la que pueden conectarse las instrucciones de control, a saber, mediante el **anidamiento** de una instrucción de control dentro de otra.

Consideré el siguiente enunciado de un problema:

Una universidad ofrece un curso que prepara a los estudiantes para el examen estatal de certificación del estado como corredores de bienes raíces. El año pasado, diez de los estudiantes que completaron este curso tomaron el examen. La universidad desea saber qué tan bien se desempeñaron sus estudiantes en el examen. A usted se le ha pedido que escriba un programa para sintetizar los resultados. Se le dio una lista de estos 10 estudiantes. Junto a cada nombre hay un 1 escrito, si el estudiante aprobó el examen, o un 2 si lo reprobó.

Su programa debe analizar los resultados del examen de la siguiente manera:

1. *Introducir cada resultado de la prueba (es decir, un 1 o un 2). Mostrar el mensaje “Escriba el resultado” en la pantalla, cada vez que el programa solicite otro resultado de la prueba.*
2. *Contar el número de resultados de la prueba, de cada tipo.*
3. *Mostrar un resumen de los resultados de la prueba, indicando el número de estudiantes que aprobaron y el número de estudiantes que reprobaron.*
4. *Si más de ocho estudiantes aprobaron el examen, imprimir el mensaje “Aumentar la colegiatura”.*

Después de leer el enunciado del programa cuidadosamente, hacemos las siguientes observaciones:

1. El programa debe procesar los resultados de la prueba para 10 estudiantes. Puede usarse un ciclo controlado por contador, ya que el número de resultados de la prueba se conoce de antemano.
2. Cada resultado de la prueba tiene un valor numérico, ya sea 1 o 2. Cada vez que el programa lee un resultado de la prueba, debe determinar si el número es 1 o 2. Nosotros evaluamos un 1 en nuestro algoritmo. Si el número no es 1, suponemos que es un 2. (El ejercicio 4.20 considera las consecuencias de esta suposición.)
3. Dos contadores se utilizan para llevar el registro de los resultados del examen: uno para contar el número de estudiantes que aprobaron el examen y uno para contar el número de estudiantes que reprobaron el examen.
4. Una vez que el programa ha procesado todos los resultados, debe decidir si más de ocho estudiantes aprobaron el examen.

Veamos ahora el mejoramiento de arriba a abajo, paso a paso. Comencemos con la representación del seudocódigo de la cima:

Analizar los resultados del examen y decidir si debe aumentarse la colegiatura o no.

Una vez más, es importante enfatizar que la cima es una representación *completa* del programa, pero es probable que se necesiten varias mejoras antes de que el seudocódigo pueda evolucionar de manera natural en un programa en C++.

Nuestra primera mejora es

Iniciar variables

Introducir las 10 calificaciones del examen, contar los aprobados y reprobados

Imprimir un resumen de los resultados del examen y decidir si debe aumentarse la colegiatura

Aquí también, aun y cuando tenemos una representación completa del programa, es necesario mejorarla. Ahora nos comprometemos con variables específicas. Se necesitan contadores para registrar los aprobados y reprobados; utilizaremos un contador para controlar el proceso de los ciclos y necesitaremos una variable para guardar la entrada del usuario. La última variable no se inicializa, ya que su valor proviene del usuario durante cada iteración del ciclo.

La instrucción en seudocódigo

Iniciar variables

puede mejorarse de la siguiente manera:

Iniciar aprobados en cero

Iniciar reprobados en cero

Iniciar contador de estudiantes en uno

Observe que sólo se inicializan los contadores al principio del algoritmo.

La instrucción en seudocódigo

Introducir las 10 calificaciones del examen, y contar los aprobados y reprobados

requiere un ciclo en el que se introduzca sucesivamente el resultado de cada examen. Sabemos de antemano que hay precisamente 10 resultados del examen, por lo que es apropiado utilizar un ciclo controlado por contador. Dentro del ciclo (es decir, **anidado** dentro del ciclo), una instrucción **if...else** determinará si cada resultado del examen es aprobado o reprobado, e incrementará el contador apropiado. Entonces, la mejora al seudocódigo anterior es

Mientras el contador de estudiantes sea menor o igual a 10

Pedir al usuario que introduzca el siguiente resultado del examen

Recibir como entrada el siguiente resultado del examen

Si el estudiante aprobó

Sumar uno a aprobados

De lo contrario

Sumar uno a reprobados

Sumar uno al contador de estudiantes

Nosotros utilizamos líneas en blanco para aislar la estructura de control *Si...De lo contrario*, lo cual mejora la legibilidad.

La instrucción en seudocódigo

Imprimir un resumen de los resultados de los exámenes y decidir si debe aumentarse la colegiatura

puede mejorarse de la siguiente manera:

Imprimir el número de aprobados

Imprimir el número de reprobados

Si más de ocho estudiantes aprobaron

Imprimir "Aumentar la colegiatura"

La segunda mejora completa aparece en la figura 4.15. Observe que también se utilizan líneas en blanco para separar la estructura *Mientras* y mejorar la legibilidad del programa. Este seudocódigo está ahora lo suficientemente mejorado para su conversión a C++.

```

1  Iniciar aprobados en cero
2  Iniciar reprobados en cero
3  Iniciar contador de estudiantes en uno
4
5  Mientras el contador de estudiantes sea menor o igual a 10
6    Pedir al usuario que introduzca el siguiente resultado del examen
7    Recibir como entrada el siguiente resultado del examen
8
9    Si el estudiante aprobó
10      Sumar uno a aprobados
11    De lo contrario
12      Sumar uno a reprobados
13
14    Sumar uno al contador de estudiantes
15
16  Imprimir el número de aprobados
17  Imprimir el número de reprobados
18
19  Si más de ocho estudiantes aprobaron
20    Imprimir "Aumentar colegiatura"

```

Figura 4.15 | El seudocódigo para el problema de los resultados del examen.

Conversión a la clase Analysis

La clase de C++ (*Analysis*) que implementa el algoritmo en seudocódigo se muestra en las figuras 4.16 y 4.17, y en la figura 4.18 aparecen dos ejecuciones de ejemplo.

```

1 // Fig. 4.16: Analisis.h
2 // Definición de la clase Analisis para analizar los resultados de un examen.
3 // La función miembro está definida en Analisis.cpp
4
5 // definición de la clase Analisis
6 class Analisis
7 {
8 public:
9     void procesarResultadosExamen(); // procesa los resultado del examen de 10 estudiantes
10}; // fin de la clase Analisis

```

Figura 4.16 | Problema de los resultados de un examen: archivo de encabezado de *Analisis*.

```

1 // Fig. 4.17: Analisis.cpp
2 // Definiciones de las funciones miembro para la clase
3 // Analisis que analiza los resultados de un examen.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 // incluye la definición de la clase Analisis de Analisis.h
10#include "Analisis.h"
11
12// procesa los resultados del examen de 10 estudiantes
13void Analisis::procesarResultadosExamen()
14{
15    // inicialización de variables en las declaraciones
16    int aprobados = 0; // número de aprobados
17    int reprobados = 0; // número de reprobados
18    int contadorEstudiantes = 1; // contador de estudiantes
19    int resultado; // resultado de un examen (1 = aprobado, 2 = reprobado)
20
21    // procesa 10 estudiantes usando el ciclo controlado por contador
22    while ( contadorEstudiantes <= 10 )
23    {
24        // pide datos de entrada y obtiene el valor el usuario
25        cout << "Escriba el resultado (1 = aprobado, 2 = reprobado): ";
26        cin >> resultado; // recibe como entrada el resultado
27
28        // if...else anidado en la instrucción while
29        if ( resultado == 1 )           // si resultado es 1,
30            aprobados = aprobados + 1; // incrementa aprobados;
31        else                         // else resultado no es 1, por lo que
32            reprobados = reprobados + 1; // incrementa reprobados
33
34        // incrementa contadorEstudiantes para que el ciclo termine en cierto momento
35        contadorEstudiantes = contadorEstudiantes + 1;
36    } // fin de while
37
38    // fase de terminación; muestra el número de aprobados y reprobados
39    cout << "Aprobados " << aprobados << "\nReprobados " << reprobados << endl;
40

```

Figura 4.17 | Problema de los resultados de un examen: instrucciones de control anidadas en el archivo de código fuente de *Analisis*. (Parte I de 2).

```

41 // determina si aprobaron más de ocho estudiantes
42 if (aprobados > 8)
43     cout << "Aumentar colegiatura " << endl;
44 } // fin de la función procesarResultadosExamen

```

Figura 4.17 | Problema de los resultados de un examen: instrucciones de control anidadas en el archivo de código fuente de *Analisis*. (Parte 2 de 2).

```

1 // Fig. 4.18: fig04_18.cpp
2 // Programa de prueba para la clase Analisis.
3 #include "Analisis.h" // incluye la definición de la clase Analisis
4
5 int main()
6 {
7     Analisis aplicacion; // crea un objeto Analisis
8     aplicacion.procesarResultadosExamen(); // llama a la función para procesar resultados
9     return 0; // indica que el programa terminó correctamente
10 } // fin de main

```

```

Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Aprobados 9
Reprobados 1
Aumentar colegiatura

```

```

Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Aprobados 6
Reprobados 4

```

Figura 4.18 | Programa de prueba para la clase *Analisis*.

Las líneas 16 a 18 de la figura 4.12 declaran las variables que utiliza la función miembro `procesarResultadosExamen` de la clase *Analisis* para procesar los resultados del examen. Observe que hemos aprovechado una característica de C++ que permite incorporar la inicialización de variables en las declaraciones (a `aprobados` se le asigna 0, a `reprobados` se le asigna 0 y a `contadorEstudiantes` se le asigna 1). Los programas con ciclos pueden requerir de la inicialización al principio de cada repetición; por lo general, dicha reinicialización se realiza mediante instrucciones de asignación, en lugar de hacerlo en las declaraciones, o moviendo las declaraciones fuera de los cuerpos de ciclo.

La instrucción `while` (líneas 22 a 36) itera 10 veces. Durante cada iteración, el ciclo recibe y procesa un resultado del examen. Observe que la instrucción `if...else` (líneas 29 a 32) para procesar cada resultado se anida en la instrucción `while`. Si el resultado es 1, la instrucción `if...else` incrementa a `aprobados`; en caso contrario, asume que `resultado` es 2 e incrementa `reprobados`. La línea 35 incrementa `contadorEstudiantes` antes de que se evalúe otra vez la condi-

ción del ciclo, en la línea 22. Después de introducir 10 valores, el ciclo termina y en la línea 39 se muestra el número de aprobados y de reprobados. La instrucción `if` de las líneas 42 a 43 determina si más de ocho estudiantes aprobaron el examen y, de ser así, imprime el mensaje "Aumentar colegiatura".

Demostración de la clase *Analisis*

En la figura 4.18 se crea un objeto `Analisis` (línea 7) y se invoca la función miembro `procesarResultadosExamen` (línea 8) de ese objeto para procesar un conjunto de resultados de un examen, introducidos por el usuario. En la figura 4.18 se muestra la entrada y la salida de dos ejecuciones de ejemplo del programa. Durante la primera ejecución de ejemplo, la condición en la línea 42 de la función miembro `procesarResultadosExamen` de la figura 4.17 es verdadera: más de ocho estudiantes aprobaron el examen, por lo que el programa imprime un mensaje indicando que se debe aumentar la colegiatura.

4.11 Operadores de asignación

C++ cuenta con varios **operadores de asignación** para abreviar las expresiones de asignación. Por ejemplo, la instrucción

```
c = c + 3;
```

puede abreviarse mediante el **operador de asignación de suma**, `+=`, de la siguiente manera:

```
c += 3;
```

El operador `+=` suma el valor de la expresión que está a la derecha del operador, al valor de la variable que está a la izquierda del operador, y almacena el resultado en la variable que está a la izquierda del operador. Cualquier instrucción de la forma

$$\text{variable} = \text{variable operador expresión};$$

donde la misma `variable` aparece en ambos lados del operador de asignación y `operador` es uno de los operadores binarios `+`, `-`, `*`, `/` o `%` (u otros que veremos más adelante en el libro), puede escribirse de la siguiente forma:

$$\text{variable operador= expresión};$$

Por lo tanto, la expresión de asignación `c += 3` suma 3 a `c`. La figura 4.19 muestra los operadores de asignación aritméticos, algunas expresiones de ejemplo en las que se utilizan los operadores y las explicaciones de lo que estos operadores hacen.

Operador de asignación	Expresión de ejemplo	Explicación	Asigna
<i>Suponer que: int c = 3, d = 5, e = 4, f = 6, g = 12;</i>			
<code>+=</code>	<code>c += 7</code>	<code>c = c + 7</code>	10 a c
<code>-=</code>	<code>d -= 4</code>	<code>d = d - 4</code>	1 a d
<code>*=</code>	<code>e *= 5</code>	<code>e = e * 5</code>	20 a e
<code>/=</code>	<code>f /= 3</code>	<code>f = f / 3</code>	2 a f
<code>%=</code>	<code>g %= 9</code>	<code>g = g % 9</code>	3 a g

Figura 4.19 | Operadores de asignación aritméticos.

4.12 Operadores de incremento y decremento

Además de los operadores de asignación aritméticos, C++ proporciona dos operadores unarios para sumar 1, o restar 1, al valor de una variable numérica. Estos operadores son el **operador de incremento unario**, `++`, y el **operador de decremento unario**, `--`, los cuales se sintetizan en la figura 4.20. Un programa puede incrementar en 1 el valor de una variable llamada `c`, utilizando el operador de incremento, `++`, en lugar de usar la expresión `c = c + 1` o `c+= 1`. A un operador de incremento o decremento que se coloca antes de una variable se le llama **operador de preincremento** o **predecremento**, respectivamente. A un operador de incremento o decremento que se coloca después de una variable se le llama **operador de postincremento** o **postdecremento**, respectivamente.

Operador	Llamado	Expresión de ejemplo	Explicación
<code>++</code>	preincremento	<code>++a</code>	Incrementar <code>a</code> en 1, después utilizar el nuevo valor de <code>a</code> en la expresión en la que esta variable reside.
<code>++</code>	postincremento	<code>a++</code>	Usar el valor actual de <code>a</code> en la expresión en la que esta variable reside, después incrementar <code>a</code> en 1.
<code>--</code>	predecremento	<code>--b</code>	Decrementar <code>b</code> en 1, después utilizar el nuevo valor de <code>b</code> en la expresión en la que esta variable reside.
<code>--</code>	postdecremento	<code>b--</code>	Usar el valor actual de <code>b</code> en la expresión en la que esta variable reside, después decrementar <code>b</code> en 1.

Figura 4.20 | Los operadores de incremento y decreto.

Al proceso de utilizar el operador de preincremento (o predecremento) para sumar (o restar) 1 a una variable, se le conoce como **preincrementar** (o **predecrementar**) la variable. Al preincrementar (o predecrementar) una variable, ésta se incrementa (o decremente) en 1, y después el nuevo valor de la variable se utiliza en la expresión en la que aparece. Al proceso de utilizar el operador de postincremento (o postdecremento) para sumar (o restar) 1 a una variable, se le conoce como **postincrementar** (o **postdecrementar**) la variable. Al postincrementar (o postdecrementar) una variable, el valor actual de la variable se utiliza en la expresión en la que aparece y después el valor de la variable se incrementa (o decremente) en 1.



Buena práctica de programación 4.9

*A diferencia de los operadores binarios, los operadores unarios de incremento y decreto deben colocarse **enseguida** de sus operandos, sin espacios entre ellos.*

En la figura 4.21 se demuestra la diferencia entre la versión de preincremento y la versión de predecremento del operador de incremento `++`. El operador de decremento `--` funciona de manera similar. Observe que este ejemplo no contiene una clase, sino un archivo de código fuente donde la función `main` realiza todo el trabajo de la aplicación. En este capítulo y en el capítulo 3, usted ha visto ejemplos que consisten en una clase (incluyendo los archivos de código fuente y de encabezado para esta clase), así como otro archivo de código fuente para probar la clase. Este archivo de código fuente contenía la función `main`, la cual creaba un objeto de la clase y llamaba a sus funciones miembro. En este ejemplo simplemente queremos mostrarle la mecánica del operador `++`, por lo que sólo usaremos un archivo de código fuente con la función `main`. Algunas veces, cuando no tenga sentido tratar de crear una clase reutilizable para demostrar un concepto simple, utilizaremos un ejemplo mecánico contenido completamente dentro de la función `main` de un solo archivo de código fuente.

En la línea 12 se inicializa la variable `c` con 5, y en la línea 13 se imprime el valor inicial de `c`. En la línea 14 se imprime el valor de la expresión `c++`. Esta expresión postincrementa la variable `c`, por lo que se imprime el valor original de `c` (5), y después el valor de `c` se incrementa. Por ende, en la línea 14 se imprime el valor inicial de `c` (5) otra vez. En la línea 15 se imprime el nuevo valor de `c` (6) para demostrar que, sin duda, se incrementó el valor de la variable en la línea 14.

```

1 // Fig. 4.21: fig04_21.cpp
2 // Preincremento y postdecremento.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int c;
10

```

Figura 4.21 | Preincrementar y postdecrementar. (Parte 1 de 2).

```

11 // demuestra el postincremento
12 c = 5; // asigna 5 a c
13 cout << c << endl; // print 5
14 cout << c++ << endl; // imprime 5 y después postincrementa
15 cout << c << endl; // imprime 6
16
17 cout << endl; // salta una línea
18
19 // demuestra el preincremento
20 c = 5; // asigna 5 a c
21 cout << c << endl; // imprime 5
22 cout << ++c << endl; // preincrementa y después imprime 6
23 cout << c << endl; // imprime 6
24 return 0; // indica que terminó correctamente
25 } // fin de main

```

```

5
5
6

5
6
6

```

Figura 4.21 | Preincrementar y postdecrementar. (Parte 2 de 2).

En la línea 20 se restablece el valor de `c` a 5, y en la línea 21 se imprime ese valor. En la línea 22 se imprime el valor de la expresión `++c`. Esta expresión preincrementa a `c`, por lo que su valor se incrementa y después se imprime el nuevo valor (6). En la línea 23 se imprime el valor de `c` otra vez, para mostrar que sigue siendo 6 después de que se ejecuta la línea 22.

Los operadores de asignación aritméticos y los operadores de incremento y decreto pueden utilizarse para simplificar las instrucciones de los programas. Las tres instrucciones de asignación de la figura 4.17:

```

aprobados = aprobados + 1;
reprobados = reprobados + 1;
contadorEstudiantes = contadorEstudiantes + 1;

```

se pueden escribir en forma más concisa con operadores de asignación, de la siguiente manera:

```

aprobados += 1;
reprobados += 1;
contadorEstudiantes += 1;

```

con operadores de preincremento de la siguiente forma:

```

++aprobados;
++reprobados;
++contadorEstudiantes;

```

o con operadores de postincremento de la siguiente forma:

```

aprobados++;
reprobados++;
contadorEstudiantes++;

```

Observe que, al incrementar (++) o decrementar (--) una variable que se encuentre en una instrucción por sí sola, las formas preincremento y postincremento tienen el mismo efecto, al igual que las formas predecremento y postdecremento. Solamente cuando una variable aparece en el contexto de una expresión más grande es cuando los operadores preincremento y postdecremento tienen distintos efectos (y lo mismo se aplica a los operadores de predecremento y postdecremento).

Error común de programación 4.14



Tratar de usar el operador de incremento o decremento en una expresión que no sea un nombre de variable o referencia que pueda modificarse [por ejemplo, escribir `++(x + 1)`] es un error de sintaxis.

En la figura 4.22 se muestra la precedencia y la asociatividad de los operadores que se han presentado hasta este punto. Los operadores se muestran de arriba a abajo, en orden descendente de precedencia. La segunda columna describe la asociatividad de los operadores en cada nivel de precedencia. El operador condicional (?:), los operadores unarios de preincremento (++) y predecremento (--), suma (+) y resta (-), y los operadores de asignación =, +=, -=, *=, /= y %= se asocian de derecha a izquierda. Todos los demás operadores en la tabla de precedencia de operadores de la figura 4.22 se asocian de izquierda a derecha. La tercera columna enumera los diversos grupos de operadores.

Operadores	Asociatividad	Tipo
::	izquierda a derecha	resolución de ámbito
()	izquierda a derecha	paréntesis
++ -- static_cast< tipo >()	izquierda a derecha	postfijo unario
++ -- + -	derecha a izquierda	prefijo unario
* / %	izquierda a derecha	multiplicativo
+ -	izquierda a derecha	aditivo
<< >>	izquierda a derecha	inserción/extracción
< <= > >=	izquierda a derecha	relacional
== !=	izquierda a derecha	igualdad
?:	derecha a izquierda	condicional
= += -= *= /= %=	derecha a izquierda	asignación

Figura 4.22 | Precedencia de los operadores vistos hasta ahora en el libro.

4.13 (Opcional) Ejemplo práctico de Ingeniería de Software: identificación de los atributos de las clases en el sistema ATM

En la sección 3.11 empezamos la primera etapa de un diseño orientado a objetos (OO) para nuestro sistema ATM: analizar la especificación de requerimientos e identificar las clases necesarias para implementar el sistema. Enlistamos los sustantivos y las frases nominales en la especificación de requerimientos e identificamos una clase separada para cada uno de ellos, que desempeña un papel importante en el sistema ATM. Después modelamos las clases y sus relaciones en un diagrama de clases de UML (figura 3.23). Las clases tienen atributos (datos) y operaciones (comportamientos). En los programas en C++, los atributos de las clases se implementan como miembros de datos, y las operaciones de las clases se implementan como funciones miembro. En esta sección determinaremos muchos de los atributos necesarios en el sistema ATM. En el capítulo 5 examinaremos cómo estos atributos representan el estado de un objeto. En el capítulo 6 determinaremos las operaciones de las clases.

Identificación de los atributos

Considere los atributos de algunos objetos reales: los atributos de una persona incluyen su altura, peso y si es zurdo, diestro o ambidiestro. Los atributos de un radio incluyen la estación seleccionada, el volumen seleccionado y si está en AM o FM. Los atributos de un auto incluyen las lecturas de su velocímetro y odómetro, la cantidad de gasolina en su tanque y la velocidad de marcha en la que se encuentra. Los atributos de una computadora personal incluyen su fabricante (por ejemplo, Dell, Sun, Apple o IBM), el tipo de pantalla (por ejemplo, LCD o CRT), el tamaño de su memoria principal y el de su disco duro.

Podemos identificar muchos atributos de las clases en nuestro sistema, analizando las palabras y frases descriptivas en la especificación de requerimientos. Para cada palabra o frase que descubramos desempeña un rol importante en el sistema ATM, creamos un atributo y lo asignamos a una o más de las clases identificadas en la sección 3.11. También creamos atributos para representar los datos adicionales que pueda necesitar una clase, ya que dichas necesidades se van aclarando a lo largo del proceso de diseño.

La figura 4.23 enumera las palabras o frases de la especificación de requerimientos que describen a cada una de las clases. Para formar esta lista, leemos la especificación de requerimientos e identificamos cualquier palabra o frase que haga referencia a las características de las clases en el sistema. Por ejemplo, la especificación de requerimientos describe los pasos que se llevan a cabo para obtener un “monto de retiro”, por lo que listamos “monto” enseguida de la clase *Retiro*.

Clase	Palabras y frases descriptivas
ATM	el usuario es autenticado
SolicitudSaldo	número de cuenta
Retiro	número de cuenta monto
Deposito	número de cuenta monto
BaseDatosBanco	[no hay palabras o frases descriptivas]
Cuenta	número de cuenta NIP saldo
Pantalla	[no hay palabras o frases descriptivas]
Teclado	[no hay palabras o frases descriptivas]
DispensadorEfectivo	empieza cada día cargado con 500 billetes de \$20
RanuraDeposito	[no hay palabras o frases descriptivas]

Figura 4.23 | Palabras y frases descriptivas del documento de requerimientos del ATM.

La figura 4.23 nos conduce a crear un atributo de la clase ATM. Esta clase mantiene información acerca del estado del ATM. La frase “el usuario es autenticado” describe un estado del ATM (en la sección 5.11 presentaremos los estados), por lo que incluimos `usuarioAutenticado` como un atributo Booleano (es decir, un atributo que tiene un valor de `true` o `false`) en la clase ATM. El tipo Booleano en UML es equivalente al tipo `bool` en C++. Este atributo indica si el ATM autenticó con éxito al usuario actual o no; `usuarioAutenticado` debe ser `true` para que el sistema permita al usuario realizar transacciones y acceder a la información de la cuenta. Este atributo nos ayuda a cerciorarnos de la seguridad de los datos en el sistema.

Las clases `SolicitudSaldo`, `Retiro` y `Deposito` comparten un atributo. Cada transacción requiere un “número de cuenta” que corresponde a la cuenta del usuario que realiza la transacción. Asignamos el atributo entero `numeroCuenta` a cada clase de transacción para identificar la cuenta a la que se aplica un objeto de la clase.

Las palabras y frases descriptivas en la especificación de requerimientos también sugieren ciertas diferencias en los atributos requeridos por cada clase de transacción. La especificación de requerimientos indica que para retirar efectivo o depositar fondos, los usuarios deben introducir un “monto” específico de dinero para retirar o depositar, respectivamente. Por ende, asignamos a las clases `Retiro` y `Deposito` un atributo llamado `monto` para almacenar el valor suministrado por el usuario. Los montos de dinero relacionados con un retiro y un depósito son características que definen estas transacciones, que el sistema requiere para que se lleven a cabo. Sin embargo, la clase `SolicitudSaldo` no necesita datos adicionales para realizar su tarea; sólo requiere un número de cuenta para indicar la cuenta cuyo saldo hay que obtener.

La clase `Cuenta` tiene varios atributos. La especificación de requerimientos establece que cada cuenta de banco tiene un “número de cuenta” y un “NIP”, que el sistema utiliza para identificar las cuentas y autenticar a los usuarios. A la clase `Cuenta` le asignamos dos atributos enteros: `numeroCuenta` y `nip`. La especificación de requerimientos también especifica que una cuenta debe mantener un “saldo” del monto de dinero que hay en la cuenta, y que el dinero que el usuario deposita no estará disponible para su retiro sino hasta que el banco verifique la cantidad de efectivo en el sobre de depósito y cualquier cheque que contenga. Sin embargo, una cuenta debe registrar de todas formas el monto de dinero que deposita un usuario. Por lo tanto, decidimos que una cuenta debe representar un saldo utilizando dos atributos de tipo `double` de UML: `saldoDisponible` y `saldoTotal`. El atributo `saldoDisponible` rastrea el monto de dinero que un usuario puede retirar de la cuenta. El atributo `saldoTotal` se refiere al monto total de dinero que el usuario tiene “en depósito” (es decir, el monto de dinero disponible, más el monto de depósitos en efectivo o la cantidad de cheques esperando a ser verificados). Por ejemplo, suponga que un usuario del ATM deposita \$50.00 en efectivo, en una cuenta vacía. El atributo `saldoTotal` se incrementaría a \$50.00 para registrar el depósito, pero el `saldoDisponible` permanecería en \$0. [Nota: estamos suponiendo que el banco actualiza el atributo `saldoDisponible` de una `Cuenta` poco después de que se realiza la transacción del ATM, en respuesta a la confirmación de que se encontró un monto equivalente a \$50.

en efectivo o cheques en el sobre de depósito. Asumimos que esta actualización se realiza a través de una transacción que realiza el empleado del banco mediante el uso de un sistema bancario distinto al del ATM. Por ende, no hablaremos sobre esta transacción en nuestro ejemplo práctico.]

La clase **DispensadorEfectivo** tiene un atributo. La especificación de requerimientos establece que el dispensador de efectivo “empieza cada día cargado con 500 billetes de \$20”. El dispensador de efectivo debe llevar el registro del número de billetes que contiene para determinar si hay suficiente efectivo disponible para satisfacer la demanda de los retiros. Asignamos a la clase **DispensadorEfectivo** el atributo entero **conteo**, el cual se establece al principio en 500.

Para los verdaderos problemas en la industria, no existe garantía alguna de que la especificación de requerimientos será lo suficientemente robusta y precisa como para que el diseñador de sistemas orientados a objetos determine todos los atributos, o inclusive todas las clases. La necesidad de más (o menos) clases, atributos y comportamientos puede irse aclarando a medida que avance el proceso de diseño. A medida que progresemos a través de este ejemplo práctico, nosotros también seguiremos agregando, modificando y eliminando información acerca de las clases en nuestro sistema.

Modelado de los atributos

El diagrama de clases de la figura 4.24 enumera algunos de los atributos para las clases en nuestro sistema; las palabras y frases descriptivas en la figura 4.23 nos ayudaron a identificar estos atributos. Por cuestión de simpleza, la figura 4.24 no muestra las asociaciones entre las clases; en la figura 3.23 mostramos estas asociaciones. Ésta es una práctica común de los diseñadores de sistemas, a la hora de desarrollar los diseños. En la sección 3.11 vimos que en UML los atributos de una clase se colocan en el compartimiento intermedio del rectángulo de la clase. Enlistamos el nombre de cada atributo y su tipo, separados por un signo de dos puntos (:), seguido en algunos casos de un signo de igual (=) y de un valor inicial.

Consideré el atributo **usuarioAutenticado** de la clase **ATM**:

```
usuarioAutenticado : Boolean = false
```

La declaración de este atributo contiene tres piezas de información acerca del atributo. El **nombre del atributo** es **usuarioAutenticado**. El **tipo del atributo** es **Boolean**. En C++, un atributo puede representarse mediante un tipo fundamental, tal como **bool**, **int** o **double**, o por un tipo de clase (como vimos en el capítulo 3). Hemos optado por modelar sólo los atributos de tipo primitivo en la figura 4.24; en breve hablaremos sobre el razonamiento detrás de esta decisión.



Figura 4.24 | Clases con atributos.

[Nota: la figura 4.24 enumera los tipos de datos de UML para los atributos. Cuando implementemos el sistema, asociaremos los tipos Boolean, Integer y Double en el diagrama de UML con los tipos fundamentales bool, int y double en C++, respectivamente.]

También podemos indicar un valor inicial para un atributo. El atributo `usuarioAutenticado` en la clase ATM tiene un valor inicial de `false`. Esto indica que al principio el sistema no considera que el usuario está autenticado. Si no se especifica un valor inicial para un atributo, sólo se muestran su nombre y tipo (separados por dos puntos). Por ejemplo, el atributo `numeroCuenta` de la clase `SolicitudSaldo` es un entero. Aquí no mostramos un valor inicial, ya que el valor de este atributo es un número que todavía no conocemos; se determinará en tiempo de ejecución, con base en el número de cuenta introducido por el usuario actual del ATM.

La figura 4.24 no incluye atributos para las clases Pantalla, Teclado y RanuraDepósito. Éstos son componentes importantes de nuestro sistema, para los cuales nuestro proceso de diseño aún no ha revelado ningún atributo. No obstante, tal vez descubramos algunos en las fases restantes de diseño, o cuando implementemos estas clases en C++. Esto es perfectamente normal para el proceso iterativo de ingeniería de software.



Observación de Ingeniería de software 4.8

En las primeras fases del proceso de diseño, a menudo las clases carecen de atributos (y operaciones). Sin embargo, esas clases no deben eliminarse, ya que los atributos (y las operaciones) pueden hacerse evidentes en las fases posteriores de diseño e implementación.

Observe que la figura 4.24 tampoco incluye atributos para la clase `BaseDatosBanco`. En el capítulo 3 vimos que en C++, los atributos pueden representarse mediante los tipos fundamentales o los tipos por referencia. Hemos optado por incluir sólo los atributos de tipo fundamental en el diagrama de clases de la figura 4.24 (y en los diagramas de clases similares a lo largo del ejemplo práctico). Un atributo de tipo de clase se modela con más claridad como una asociación (en particular, una composición) entre la clase con el atributo y la clase del objeto del cual el atributo es una instancia. Por ejemplo, el diagrama de clases de la figura 3.23 indica que la clase `BaseDatosBanco` participa en una relación de composición con cero o más objetos `Cuenta`. De esta composición podemos determinar que, cuando implementemos el sistema ATM en C++, tendremos que crear un atributo de la clase `BaseDatosBanco` para almacenar cero o más objetos `Cuenta`. De manera similar, podemos asignar atributos a la clase ATM que correspondan a sus relaciones de composición con las clases `Pantalla`, `Teclado`, `DispensadorEfectivo` y `RanuraDepósito`. Estos atributos basados en composiciones serían redundantes si los modeláramos en la figura 4.24, ya que las composiciones modeladas en la figura 3.23 transmiten de antemano el hecho de que la base de datos contiene información acerca de cero o más cuentas, y que un ATM está compuesto por una pantalla, un teclado, un dispensador de efectivo y una ranura para depósitos. Por lo general, los desarrolladores de software modelan estas relaciones de todo/parte como asociaciones de composición, en lugar de modelarlas como atributos requeridos para implementar las relaciones.

El diagrama de clases de la figura 4.24 proporciona una base sólida para la estructura de nuestro modelo, pero no está completo. En la sección 5.11 identificaremos los estados y las actividades de los objetos en el modelo, y en la sección 6.22 identificaremos las operaciones que realizan los objetos. A medida que presentemos más acerca de UML y del diseño orientado a objetos, continuaremos reforzando la estructura de nuestro modelo.

Ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

4.1 Por lo general, identificamos los atributos de las clases en nuestro sistema mediante el análisis de _____ en la especificación de requerimientos.

- los sustantivos y las frases nominales.
- las palabras y frases descriptivas.
- los verbos y las frases verbales.
- Todo lo anterior.

4.2 ¿Cuál de los siguientes no es un atributo de un aeroplano?

- longitud.
- envergadura.
- volar.
- número de asientos.

4.3 Describa el significado de la siguiente declaración de un atributo de la clase `DispensadorEfectivo` en el diagrama de clases de la figura 4.24:

```
conteo : Integer = 500
```

Respuestas a los ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

- 4.1 b.
- 4.2 c. Volar es una operación o comportamiento de un aeroplano, no un atributo.
- 4.3 Esta declaración indica que el atributo `conteo` es de tipo `Integer`, con un valor inicial de 500. Este atributo lleva la cuenta del número de billetes disponibles en el `DispensadorEfectivo`, en cualquier momento dado.

4.14 Repaso

Este capítulo presentó las estrategias básicas de solución de problemas, que los programadores utilizan para crear clases y desarrollar funciones miembro para estas clases. Demostramos cómo construir un algoritmo (es decir, una metodología para resolver un problema) en seudocódigo, y después cómo refinar el algoritmo a través de diversas fases de desarrollo de seudocódigo, lo cual produce código en C++ que puede ejecutarse como parte de una función. En este capítulo aprendió a utilizar el método de mejoramiento de arriba a abajo, paso a paso, para planear las acciones específicas que debe realizar una función, y el orden en el que debe realizar estas acciones.

Aprendió que sólo hay tres tipos de estructuras de control (secuencia, selección y repetición) necesarias para desarrollar cualquier algoritmo. Demostramos dos de las instrucciones de selección de C++: la instrucción de selección simple `if` y la instrucción de selección doble `if...else`. La instrucción `if` se utiliza para ejecutar un conjunto de instrucciones basadas en una condición; si la condición es verdadera, se ejecutan las instrucciones; si no, se omiten. La instrucción de selección doble `if...else` se utiliza para ejecutar un conjunto de instrucciones si se cumple una condición, y otro conjunto de instrucciones si la condición es falsa. Después vimos la instrucción de repetición `while`, donde un conjunto de instrucciones se ejecutan de manera repetida, mientras que una condición sea verdadera. Utilizamos el apilamiento de instrucciones de control para calcular el total y el promedio de un conjunto de calificaciones de estudiantes, mediante la repetición controlada por un contador y controlada por un centinela, y utilizamos el anidamiento de instrucciones de control para analizar y tomar decisiones con base en un conjunto de resultados de un examen. Vimos una introducción a los operadores de asignación, que pueden utilizarse para abreviar instrucciones. Presentamos los operadores de incremento y decremento, los cuales se pueden utilizar para sumar o restar el valor 1 de una variable. En el capítulo 5, Instrucciones de control: parte 2, continuaremos nuestra discusión acerca de las instrucciones de control, donde presentaremos las instrucciones `for`, `do...while` y `switch`.

Resumen**Sección 2 Algoritmos**

- Un algoritmo es un procedimiento para resolver un problema, en términos de las acciones a ejecutar y el orden en el que se ejecutan.
- El proceso de especificar el orden en el que se ejecutan las instrucciones (acciones) en un programa se denomina control del programa.

Sección 4.3 Seudocódigo

- El seudocódigo ayuda al programador a “idear” un programa antes de intentar escribirlo en un lenguaje de programación.
- Los diagramas de actividad forman parte del Lenguaje de modelado unificado (UML): un estándar en la industria para modelar sistemas de software.

Sección 4.4 Estructuras de control

- Un diagrama de actividad modela el flujo de trabajo (también conocido como la actividad) de una parte de un sistema de software.
- Los diagramas de actividad se componen de símbolos de propósito especial, como los símbolos de estados de acción, rombos y pequeños círculos. Estos símbolos se conectan mediante flechas de transición, las cuales representan el flujo de la actividad.
- Al igual que el seudocódigo, los diagramas nos ayudan a desarrollar y representar algoritmos.
- Un estado de acción se representa como un rectángulo en el que sus lados izquierdo y derecho se sustituyen con arcos que se curvan hacia afuera. La expresión de acción aparece dentro del estado de acción.
- Las flechas en un diagrama de actividad representan las transiciones, que indican el orden en el que ocurren las acciones representadas por los estados de acción.
- El círculo relleno que se encuentra en la parte superior de un diagrama de actividad representa el estado inicial: el comienzo del flujo de trabajo antes de que el programa realice las acciones modeladas.
- El círculo sólido rodeado por una circunferencia, que aparece en la parte inferior del diagrama, representa el estado final: el término del flujo de trabajo después de que el programa realiza sus acciones.

- Los rectángulos con las esquinas superiores derechas dobladas se llaman notas en UML: comentarios que describen el propósito de los símbolos en el diagrama. Una línea punteada conecta a cada nota con el elemento que ésta describe.
- Un rombo o símbolo de decisión en un diagrama de actividad indica que se debe realizar una decisión. El flujo de trabajo continuará a lo largo de una ruta determinada por las condiciones de guardia asociadas con el símbolo, que pueden ser verdaderas o falsas. Cada flecha de transición que emerge de un símbolo de decisión tiene una condición de guardia (especificada entre corchetes, a un lado de la flecha de transición). Si una condición de guardia es verdadera, el flujo de trabajo entra al estado de acción al que apunta la flecha de transición.
- Un rombo en un diagrama de actividad también representa el símbolo de fusión, el cual une dos flujos de actividad en uno. Un símbolo de fusión tiene dos o más flechas de transición que apuntan al rombo, y sólo una flecha de transición que apunta hacia el rombo, para indicar que varios flujos de actividad se fusionan para continuar la actividad.
- El mejoramiento de nivel superior, paso a paso, es un proceso para refinar pseudocódigo, para lo cual se mantiene una representación completa del programa durante cada mejoramiento.
- Existen tres tipos de estructuras de control: secuencia, selección y repetición.
- La estructura de secuencia está integrada en C++; de manera predeterminada, las instrucciones se ejecutan en el orden en el que aparecen.
- Una estructura de selección elige uno de varios cursos alternativos de acción.

Sección 4.5 Instrucción de selección `if`

- La instrucción `if` de selección simple ejecuta (selecciona) una acción si una condición es verdadera, o ignora la acción si la condición es falsa.

Sección 4.6 Instrucción de selección doble `if...else`

- La instrucción `if...else` de selección doble ejecuta (selecciona) una acción cuando la condición es verdadera, y otra acción distinta cuando la condición es falsa.
- Para incluir varias instrucciones en el cuerpo del `if` (o en el cuerpo del `else` para una instrucción `if...else`), encierre las instrucciones entre llaves (`{ y }`). A un conjunto de instrucciones contenidas dentro de un par de llaves se le llama bloque. Un bloque puede colocarse en cualquier parte de un programa, donde se pueda colocar una sola instrucción.
- Una instrucción nula, la cual indica que no puede realizarse ninguna acción, se indica mediante un punto y coma (`;`).

Sección 4.7 Instrucción de repetición `while`

- Una instrucción de repetición repite una acción mientras cierta condición sea verdadera.
- Un valor que contiene una parte fraccionaria se conoce como número de punto flotante, y se representa aproximadamente mediante los tipos de datos tales como `float` y `double`.

Sección 4.8 Cómo formular algoritmos: repetición controlada por un contador

- La repetición controlada por un contador se utiliza cuando se conoce el número de repeticiones antes de que un ciclo empiece a ejecutarse; es decir, cuando hay una repetición definida.
- El operador unario de conversión de tipos `static_cast` se puede utilizar para crear una copia temporal de punto flotante del operando.
- Los operadores unarios sólo reciben un operando; los operadores binarios reciben dos.
- El manipulador de flujo parametrizado `setprecision` indica el número de dígitos de precisión que deben mostrarse a la derecha del punto decimal.
- El manipulador de flujo `fixed` indica que los valores de punto flotante se deben imprimir en lo que se denomina formato de punto fijo, en oposición a la notación científica.

Sección 4.9 Cómo formular algoritmos: repetición controlada por un centinela

- La repetición controlada por un centinela se utiliza cuando no se conoce el número de repeticiones antes de que un ciclo se empiece a ejecutar; es decir, cuando hay repetición indefinida.

Sección 4.10 Cómo formular algoritmos: instrucciones de control anidadas

- Una instrucción de control anidada aparece en el cuerpo de otra instrucción de control.

Sección 4.11 Operadores de asignación

- C++ cuenta con los operadores de asignación `+=`, `-=`, `*=`, `/=` y `%=` para abreviar las expresiones de asignación.

Sección 4.12 Operadores de incremento y decremento

- El operador de incremento, `++`, y el operador de decremento, `--`, incrementan o decrementan una variable en 1, respectivamente. Si el operador se coloca antes de la variable, ésta se incrementa o decremente en 1 primero, y después su nuevo valor se utiliza en la expresión en la que aparece. Si el operador se coloca después de la variable, ésta se utiliza primero en la expresión en la que aparece, y después su valor se incrementa o decremente en 1.

Terminología

acción	iteración
algoritmo	iteraciones de un ciclo
anidamiento de estructuras de control	línea punteada
apilamiento de estructuras de control	manipulador de flujo
aproximación de los números de punto flotante	manipulador de flujo no parametrizado
asociar de derecha a izquierda	manipulador de flujo parametrizado
asociar de izquierda a derecha	mejoramiento de arriba a abajo, paso a paso
bloque	modelo de programación acción/decisión
“bombing”	nota en UML
<code>bool</code> , tipo fundamental	notación científica
ciclo	número de punto flotante
ciclo anidado dentro de un ciclo	número de punto flotante con precisión doble
ciclo, iteraciones	número de punto flotante con precisión simple
cima	operador aritmético binario
condición de continuación de ciclo	operador condicional (?:)
constante de punto flotante	operador de asignación de suma (+=)
contador	operador de conversión de tipos
control del programa	operador de conversión de tipos unario
conversión explícita	operador de decremento (--)
conversión implícita	operador de incremento (++)
“crashing”	operador de postdecremento
diagrama de actividad	operador de postincremento
diagrama de actividad de estructura de secuencia	operador de predecremento
diseño orientado a objetos (OO)	operador de preincremento
división entera	operador ternario
división entre cero	operador unario
<code>double</code> , tipo de datos	operador unario de resta (-)
ejecución secuencial	operador unario de suma (+)
error de desplazamiento en 1	operadores de asignación
error lógico fatal	operadores de asignación aritméticos
estado de acción	operando
estado final	orden en el que deben ejecutarse las acciones
estado inicial	palabras clave
estructura de secuencia	postdecremento
expresión condicional	postincremento
expresión de acción	precedencia de operadores
<code>fixed</code> , manipulador de flujo	precisión
<code>float</code> , tipo de datos	precisión predeterminada
flujo de trabajo de una porción de un sistema de software	predecremento
formato de punto fijo	preincremento
<code>goto</code> , eliminación	primera mejora
<code>goto</code> , instrucción	problema del <code>else</code> suelto
<code>if...else</code> , instrucción de selección doble	procedimiento
instrucción compuesta	programación estructurada
instrucción de ciclo	promediar un cálculo
instrucción de control	promoción
instrucción de control anidada	promoción de enteros
instrucción de control de una sola entrada/una sola salida	redondeo
instrucción de repetición	repetición controlada por centinela
instrucción de selección	repetición controlada por un contador
instrucción de selección doble	repetición definida
instrucción de selección <code>if</code> simple	repetición indefinida
instrucción de selección múltiple	segunda mejora
instrucción ejecutable	<code>setprecision</code> , manipulador de flujo
instrucción nula	seudocódigo
instrucción vacía	<code>showpoint</code> , manipulador de flujo

símbolo de círculo relleno	transferencia de control
símbolo de decisión	transición
símbolo de estado de acción	truncar
símbolo de flecha	valor “basura”
símbolo de flecha de transición	valor centinela
símbolo de fusión	valor de bandera
símbolo de pequeño círculo	valor de prueba
símbolo de rombo	valor de señal
static_cast , operador	valor indefinido
total	while , instrucción de repetición

Ejercicios de autoevaluación

4.1 Complete los siguientes enunciados:

- a) Todos los programas pueden escribirse en términos de tres tipos de estructuras de control: _____, _____ y _____.
- b) La instrucción de selección _____ se utiliza para ejecutar una acción cuando una condición es verdadera, y otra acción cuando esa condición es falsa.
- c) Al proceso de repetir un conjunto de instrucciones un número específico de veces se le llama repetición _____.
- d) Cuando no se sabe de antemano cuántas veces se repetirá un conjunto de instrucciones, se puede usar un valor _____ para terminar la repetición.

4.2 Escriba cuatro instrucciones distintas en C++, donde cada una sume 1 a la variable entera **x**.

4.3 Escriba instrucciones en C++ para realizar cada una de las siguientes tareas:

- a) En una instrucción, asignar la suma del valor actual de **x** y a **z**, y postincrementar el valor de **x**.
- b) Evaluar si la variable **cuenta** es mayor que 10. De ser así, imprimir "Cuenta es mayor que 10".
- c) Predecrementar la variable **x** en 1, luego restarla a la variable **tota1**.
- d) Calcular el residuo después de dividir **q** entre **divisor**, y asignar el resultado a **q**. Escriba esta instrucción de dos maneras distintas.

4.4 Escriba instrucciones en C++ para realizar cada una de las siguientes tareas:

- a) Declarar las variables **suma** y **x** como de tipo **int**.
- b) Asignar 1 a la variable **x**.
- c) Asignar 0 a la variable **suma**.
- d) Sumar la variable **x** a **suma** y asignar el resultado a la variable **suma**.
- e) Imprimir la cadena "La suma es: ", seguida del valor de la variable **suma**.

4.5 Combine las instrucciones que escribió en el ejercicio 4.4 para formar un programa que calcule e imprima la suma de los enteros del 1 al 10. Use una instrucción **while** para iterar a través de las instrucciones de cálculo e incremento. El ciclo debe terminar cuando el valor de **x** se vuelva 11.

4.6 Determine el valor de cada una de las variables en la siguiente instrucción, después de realizar el cálculo. Suponga que, cuando se empieza a ejecutar cada una de las instrucciones, todas las variables tienen el valor 5 entero.

- a) **producto** *= **x++**;
- b) **cociente** /= **++x**;

4.7 Escriba instrucciones individuales de C++ o porciones de instrucciones que realicen lo siguiente:

- a) Recibir como entrada la variable entera **x** con **cin** y **>>**.
- b) Recibir como entrada la variable entera **y** con **cin** y **>>**.
- c) Establecer la variable entera **i** a 1.
- d) Establecer la variable entera **potencia** a 1.
- e) Multiplicar la variable **potencia** por **x** y asignar el resultado a **potencia**.
- f) Preincrementar la variable **i** en 1.
- g) Determinar si **i** es menor o igual a **y**.
- h) Imprimir la variable entera **potencia** con **cout** y **<<**.

4.8 Escriba un programa en C++ que utilice las instrucciones del ejercicio 4.7 para calcular **x** elevada a la **y** potencia. El programa debe tener una instrucción de repetición **while**.

150 Capítulo 4 Instrucciones de control: parte I**4.9** Identifique y corrija los errores en cada uno de los siguientes fragmentos de código:

a) `while (c <= 5)`
 {
 producto *= c;
 ++c;
 }

b) `cin << valor;`

c) `if (genero == 1)`
 `cout << "Mujer" << endl;`
 `else;`
 `cout << "Hombre" << endl;`

4.10 ¿Qué está mal en la siguiente instrucción de repetición `while`?

```
while ( z >= 0 )  
    suma *= z;
```

Respuestas a los ejercicios de autoevaluación**4.1** a) Secuencia, selección y repetición. b) `if...else`. c) Controlada por contador o definida. d) Centinela, de señal, de bandera o de prueba.**4.2**

```
x = x + 1;  
x += 1;  
++x;  
x++;
```

4.3

a) `z = x++ + y;`
b) `if (cuenta > 10)`
 `cout << "Cuenta es mayor que 10" << endl;`
c) `total -= --x;`
d) `q %= divisor;`
 `q = q % divisor;`

4.4

a) `int suma;`
 `int x;`
b) `x = 1;`
c) `suma = 0;`
d) `suma += x;`
 o
 `suma = suma + x;`
e) `cout << "La suma es: " << suma << endl;`

4.5 Vea el siguiente código:

```
1 Solucion al ejercicio 4.5: ej04_05.cpp  
2 // Calcula la suma de los enteros del 1 al 10.  
3 #include <iostream>  
4 using std::cout;  
5 using std::endl;  
6  
7 int main()  
8 {  
9     int suma; // almacena la suma de los enteros del 1 al 10  
10    int x; // contador  
11  
12    x = 1; // cuenta desde 1  
13    suma = 0; // inicializa suma  
14  
15    while ( x <= 10 ) // itera 10 veces  
16    {  
17        suma += x; // suma x a suma  
18        ++x; // incrementa x  
19    } // fin de while
```

```

20
21     cout << "La suma es: " << suma << endl;
22     return 0; // indica que terminó correctamente
23 } // fin de main

```

La suma es: 55

- 4.6** a) producto = 25, x = 6;
 b) cociente = 0, x = 6;

```

1 // Solución al ejercicio 4.6: ej04_06.cpp
2 // Calcula el valor del producto y el cociente.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int x = 5;
10    int producto = 5;
11    int cociente = 5;
12
13    // parte a
14    producto *= x++; // instrucción de la parte a
15    cout << "Valor de producto despues del calculo: " << producto << endl;
16    cout << "Valor de x despues del calculo: " << x << endl << endl;
17
18    // parte b
19    x = 5; // restablece el valor de x
20    cociente /= ++x; // instrucción de la parte b
21    cout << "Valor de cociente despues del calculo: " << cociente << endl;
22    cout << "Valor de x despues del calculo: " << x << endl << endl;
23    return 0; // indica que terminó correctamente
24 } // fin de main

```

Valor de producto despues del calculo: 25
 Valor de x despues del calculo: 6

Valor de cociente despues del calculo: 0
 Valor de x despues del calculo: 6

- 4.7** a) cin >> x;
 b) cin >> y;
 c) i = 1;
 d) potencia = 1;
 e) potencia *= x;
 o
 potencia = potencia * x;
 f) ++i;
 g) if (i <= y)
 h) cout << potencia << endl;

- 4.8** Vea el siguiente código:

```

1 // Solución al ejercicio 4.8: ej04_08.cpp
2 // Eleva x a la y potencia.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7

```

```

8 int main()
9 {
10     int x; // base
11     int y; // exponente
12     int i; // cuenta de 1 a y
13     int potencia; // se usa para calcular x elevada a la potencia y
14
15     i = 1; // inicializa i para empezar a contar desde 1
16     potencia = 1; // inicializa potencia
17
18     cout << "Escriba la base como un entero: "; // pide la base
19     cin >> x; // recibe la base como entrada
20
21     cout << "Escriba el exponente como un entero: "; // pide el exponente
22     cin >> y; // recibe el exponente como entrada
23
24     // cuenta de 1 a y y multiplica potencia por x cada vez
25     while ( i <= y )
26     {
27         potencia *= x;
28         ++i;
29     } // fin de while
30
31     cout << potencia << endl; // muestra el resultado
32     return 0; // indica que terminó correctamente
33 } // fin de main

```

```

Escriba la base como un entero: 2
Escriba el exponente como un entero: 3
8

```

- 4.9 a) *Error:* falta la llave derecha de cierre del cuerpo de la instrucción `while`.

Corrección: Agregar una llave derecha de cierre después de la instrucción `c++`.

- b) *Error:* Se utiliza inserción de flujo, en lugar de extracción de flujo.

Corrección: cambie `<< a >>`.

- c) *Error:* El punto y coma después de `else` produce un error lógico. La segunda instrucción de salida siempre se ejecutará.

Corrección: Quitar el punto y coma después de `else`.

- 4.10 El valor de la variable `z` nunca se cambia en la instrucción `while`. Por lo tanto, si la condición de continuación de ciclo (`z >= 0`) es en un principio verdadera, se crea un ciclo infinito. Para evitar que ocurra un ciclo infinito, `z` debe decrementarse de manera que eventualmente se vuelva menor que 0.

Ejercicios

- 4.11 Identifique y corrija los errores en cada uno de los siguientes fragmentos de código:

```

a) if ( edad >= 65 );
    cout << "Edad es mayor o igual que 65" << endl;
    else
        cout << "Edad es menor que 65 << endl";
b) if ( edad >= 65 );
    cout << "Edad es mayor o igual que 65 << endl";
    else;
        cout << "Edad es menor que 65 << endl";
c) int x = 1, total;
    while ( x <= 10 )
    {
        total += x;
        ++x;
    }

```

```

d) while ( x <= 100 )
    total += x;
    ++x;
e) while ( y > 0 )
{
    cout << y << endl;
    ++y;
}

```

4.12 ¿Qué es lo que imprime el siguiente programa?

```

1 // Ejercicio 4.12: ej04_12.cpp
2 // ¿Qué imprime este programa?
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int y; // declara y
10    int x = 1; // inicializa x
11    int total = 0; // inicializa el total
12
13    while ( x <= 10 ) // itera 10 veces
14    {
15        y = x * x; // realiza el cálculo
16        cout << y << endl; // imprime el resultado
17        total += y; // suma y al total
18        x++; // incrementa el contador x
19    } // fin de while
20
21    cout << "El total es " << total << endl; // muestra el resultado
22    return 0; // indica que terminó correctamente
23 } // fin de main

```

Para los ejercicios 4.13 a 4.16, realice cada uno de los siguientes pasos:

- Lea el enunciado del problema.
- Formule el algoritmo utilizando seudocódigo y el proceso de mejoramiento de arriba a abajo, paso a paso.
- Escriba un programa en C++.
- Pruebe, depure y ejecute el programa en C++.

4.13 Los conductores se preocupan acerca del kilometraje de sus automóviles. Un conductor ha llevado el registro de varios reabastecimientos de gasolina, registrando los kilómetros conducidos y los litros usados en cada reabastecimiento. Desarrolle un programa en C++ que utilice una instrucción `while` para recibir como entrada los kilómetros conducidos y los litros usados (ambos como enteros) por cada reabastecimiento. El programa debe calcular y mostrar los kilómetros por litro obtenidos en cada reabastecimiento, y debe imprimir el total de kilómetros por litro obtenidos en todos los reabastecimientos hasta este punto.

Escriba los kilómetros usados (-1 para salir): 287

Escriba los litros: 13

KPL en este reabastecimiento: 22.076923

Total KPL: 22.07693

Escriba los kilómetros usados (-1 para salir): 200

Escriba los litros: 10

KPL en este reabastecimiento: 20.000000

Total KPL: 21.173913

Escriba los kilómetros usados (-1 para salir): 120

Escriba los litros: 5

KPL en este reabastecimiento: 24.000000

Total KPL: 21.678571

Escriba los kilómetros usados (-1 para salir): -1

4.14 Desarrolle una aplicación en C++ que determine si alguno de los clientes de una tienda de departamentos se ha excedido del límite de crédito en una cuenta. Para cada cliente se tienen los siguientes datos:

- Número de cuenta (un entero)
- Saldo al inicio del mes
- Total de todos los artículos cargados por el cliente en el mes
- Total de todos los créditos aplicados a la cuenta del cliente en el mes
- Límite de crédito permitido.

El programa debe usar una instrucción `while` para recibir como entrada cada uno de estos datos, debe calcular el nuevo saldo (= saldo inicial + cargos – créditos) y determinar si éste excede el límite de crédito del cliente. Para los clientes cuyo límite de crédito sea excedido, el programa debe mostrar el número de cuenta del cliente, su límite de crédito, el nuevo saldo y el mensaje "Se excedio el limite de su credito".

```
Introduzca el numero de cuenta (o -1 para salir): 100
Introduzca el saldo inicial: 5394.78
Introduzca los cargos totales: 1000.00
Introduzca los créditos totales: 500.00
Introduzca el límite de crédito: 5500.00
El nuevo saldo es 5894.78
Cuenta: 100
Límite de crédito: 5500.00
Saldo: 5894.78
Se excedio el limite de su credito.

Introduzca el numero de cuenta (o -1 para salir): 200
Introduzca el saldo inicial: 1000.00
Introduzca los cargos totales: 123.45
Introduzca los créditos totales: 321.00
Introduzca el límite de crédito: 1500.00
El nuevo saldo es 802.45

Introduzca el numero de cuenta (o -1 para salir): 300
Introduzca el saldo inicial: 500.00
Introduzca los cargos totales: 274.73
Introduzca los créditos totales: 100.00
Introduzca el límite de crédito: 800.00
El nuevo saldo es 674.73

Introduzca el numero de cuenta (o -1 para salir): -1
```

4.15 Una gran empresa de químicos paga a sus vendedores mediante comisiones. Los vendedores reciben \$200 por semana, más el 9% de sus ventas brutas durante esa semana. Por ejemplo, un vendedor que vende \$5000 de mercancía en una semana, recibe \$200 más el 9% de \$5000, o un total de \$650. Desarrolle un programa en C++ que utilice una instrucción `while` para recibir como entrada las ventas brutas de cada vendedor de la semana anterior, y que calcule y muestre los ingresos de ese vendedor. Procése las cifras de un vendedor a la vez.

```
Introduzca las ventas en dolares (-1 para salir): 5000.00
El salario es: $650.00

Introduzca las ventas en dolares (-1 para salir): 6000.00
El salario es: $740.00

Introduzca las ventas en dolares (-1 para salir): 7000.00
El salario es: $830.00

Introduzca las ventas en dolares (-1 para salir): -1
```

4.16 Desarrolle un programa en C++ que utilice una instrucción `while` para determinar el sueldo bruto para cada uno de varios empleados. La empresa paga la cuota normal en las primeras 40 horas de trabajo de cada empleado, y paga cuota y media en todas las horas trabajadas que excedan de 40. Usted recibe una lista de los empleados de la empresa, el número de horas que trabajó cada empleado la semana pasada y la tarifa por horas de cada empleado. Su programa debe recibir como entrada esta información para cada empleado, debe determinar y mostrar el sueldo bruto de cada empleado.

```
Introduzca las horas trabajadas (-1 para salir): 39
Introduzca la tarifa por horas del empleado ($00.00): 10.00
El salario es $390.00
```

```
Introduzca las horas trabajadas (-1 para salir): 40
Introduzca la tarifa por horas del empleado ($00.00): 10.00
El salario es $400.00
```

```
Introduzca las horas trabajadas (-1 para salir): 41
Introduzca la tarifa por horas del empleado ($00.00): 10.00
El salario es $415.00
```

```
Introduzca las horas trabajadas (-1 para salir): -1
```

4.17 El proceso de encontrar el número más grande (es decir, el máximo de un grupo de números) se utiliza frecuentemente en aplicaciones de computadora. Por ejemplo, un programa para determinar el ganador de un concurso de ventas recibe como entrada el número de unidades vendidas por cada vendedor. El vendedor que haya vendido más unidades es el que gana el concurso. Escriba un programa en pseudocódigo y después una aplicación en C++ que utilice una instrucción `while` para determinar e imprimir el mayor número de una serie de 10 números introducidos por el usuario. Su programa debe utilizar tres variables, como se muestra a continuación:

- | | |
|-----------|---|
| contador: | Un contador para contar hasta 10 (es decir, para llevar el registro de cuántos números se han introducido, y para determinar cuando se hayan procesado los 10 números). |
| numero: | El número actual que se introduce al programa. |
| mayor: | El número más grande encontrado hasta ahora. |

4.18 Escriba una aplicación en C++ que utilice una instrucción `while` y la secuencia de escape de tabulación `\t` para imprimir la siguiente tabla de valores:

N	10*N	100*N	1000*N
1	10	100	1000
2	20	200	2000
3	30	300	3000
4	40	400	4000
5	50	500	5000

4.19 Utilizando una metodología similar a la del ejercicio 4.17, encuentre los *dos* valores más grandes de los 10 que se introdujeron. [Nota: debe introducir cada número sólo una vez.]

4.20 El programa de resultados de un examen de las figuras 4.16 a 4.18 asume que cualquier valor introducido por el usuario que no sea un 1 debe ser un 2. Modifique la aplicación para validar sus entradas. Para cualquier entrada, si el valor introducido es distinto de 1 o 2, debe seguir iterando hasta que el usuario introduzca un valor correcto.

4.21 ¿Qué es lo que imprime el siguiente programa?

```

1 // Ejercicio 4.21: ej04_21.cpp
2 // ¿Qué es lo que imprime este programa?
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int cuenta = 1; // inicializa cuenta
10
11    while ( cuenta <= 10 ) // itera 10 veces
12    {
13        // imprime una línea de texto
14        cout << ( cuenta % 2 ? "*****" : "+++++++" ) << endl;
15        ++cuenta; // incrementa cuenta
16    } // fin de while
17
18    return 0; // indica que terminó correctamente
19 } // fin de main

```

4.22 ¿Qué es lo que imprime el siguiente programa?

```

1 // Ejercicio 4.22: ej04_22.cpp
2 // ¿Qué es lo que imprime este programa?
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int fila = 10; // inicializa fila
10    int columna; // declara columna
11
12    while ( fila >= 1 ) // itera hasta que fila < 1
13    {
14        columna = 1; // establece columna a 1 cuando empieza la iteración
15
16        while ( columna <= 10 ) // itera 10 veces
17        {
18            cout << ( fila % 2 ? "<" : ">" ); // salida
19            ++columna; // incrementa columna
20        } // fin de while interior
21
22        --fila; // decrementa fila
23        cout << endl; // empieza nueva línea de salida
24    } // fin de while exterior
25
26    return 0; // indica que terminó correctamente
27 } // fin de main

```

4.23 (Problema del else suelto) Determine la salida de cada uno de los siguientes conjuntos de código, cuando x es 9 y y es 11, y cuando x es 11 y y es 9. Observe que el compilador ignora la sangría en un programa en C++. El compilador de C++ siempre asocia un `else` con el `if` que le precede inmediatamente, a menos que se le indique de otra forma mediante la colocación de llaves (`{}`). A primera vista, el programador tal vez no esté seguro de cuál `if` corresponde a cuál `else`; esta situación se conoce como el “problema del `else` suelto”. Hemos eliminado la sangría del siguiente código para hacer el problema más retador. [Sugerencia: aplique las convenciones de sangría que ha aprendido.]

a) `if (x < 10)
 if (y > 10)
 cout << "*****" << endl;
 else
 cout << "#####" << endl;
 cout << "$$$$$" << endl;`

b) `if (x < 10)
{
 if (y > 10)
 cout << "*****" << endl;
 }
 else
 {
 cout << "#####" << endl;
 cout << "$$$$$" << endl;
 }`

4.24 (Otro problema de else suelto) Modifique el siguiente código para producir la salida que se muestra. Utilice las técnicas de sangría apropiadas. No debe hacer modificaciones en el código, sólo insertar llaves o modificar la sangría del código. El compilador ignora la sangría en un programa en C++. Hemos eliminado la sangría en el código dado, para hacer el problema más retador. [Nota: es posible que no se requieran modificaciones.]

```

if ( y == 8 )
if ( x == 5 )
cout << "@@@@" << endl;

```

```

else
cout << "#####" << endl;
cout << "$$$$$" << endl;
cout << "&&&&&" << endl;

```

- a) Suponiendo que $x = 5$ y $y = 8$, se produce la siguiente salida:

```

AAAAA
$$$$$ 
&&&&&

```

- b) Suponiendo que $x = 5$ y $y = 8$, se produce la siguiente salida:

```

AAAAA

```

- c) Suponiendo que $x = 5$ y $y = 8$, se produce la siguiente salida:

```

AAAAA
&&&&&

```

- d) Suponiendo que $x = 5$ y $y = 7$, se produce la siguiente salida. [Nota: las tres últimas instrucciones de salida después del `else` forman parte de un bloque.]

```

#####
$$$$$ 
&&&&&

```

4.25 Escriba un programa que pida al usuario que introduzca el tamaño del lado de un cuadrado y que muestre un cuadrado hueco de ese tamaño, compuesto de asteriscos y espacios en blanco. Su programa debe funcionar con cuadrados que tengan lados de todas las longitudes entre 1 y 20. Por ejemplo, si su programa lee un tamaño de 5, debe imprimir

```

*****
*   *
*   *
*   *
*****

```

4.26 Un palíndromo es un número o una frase de texto que se lee igual al derecho y al revés. Por ejemplo, cada uno de los siguientes enteros de cinco dígitos es un palíndromo: 12321, 55555, 45554 y 11611. Escriba una aplicación que lea un entero de cinco dígitos y determine si es un palíndromo. [Sugerencia: use los operadores de división y módulo para separar el número en sus dígitos individuales.]

4.27 Escriba un programa que reciba como entrada un entero que contenga sólo 0s y 1s (es decir, un entero “binario”), y que imprima su equivalente decimal. Use los operadores módulo y división para elegir los dígitos del número “binario” uno a la vez, de derecha a izquierda. En forma parecida al sistema numérico decimal, donde el dígito más a la derecha tiene un valor posicional de 1 y el siguiente dígito a la izquierda tiene un valor posicional de 10, después 100, después 1000, etcétera, en el sistema numérico binario, el dígito más a la derecha tiene un valor posicional de 1, el siguiente dígito a la izquierda tiene un valor posicional de 2, luego 4, luego 8, etcétera. Así, el número decimal 234 se puede interpretar como $2 * 100 + 3 * 10 + 4 * 1$. El equivalente decimal del número binario 1101 es $1 * 1 + 0 * 2 + 1 * 4 + 1 * 8$, o $1 + 0 + 4 + 8$, o 13. [Nota: para aprender más acerca de los números binarios, consulte el apéndice D.]

4.28 Escriba un programa que muestre el patrón de tablero de damas que se muestra a continuación. Su programa debe utilizar sólo tres instrucciones de salida, una para cada una de las siguientes formas:

```

cout << "* ";
cout << ' ';
cout << endl;

```

```
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
* * * * * * * *
```

4.29 Escriba un programa que imprima las potencias del entero 2; a saber, 2, 4, 8, 16, 32, 64, etcétera. Su ciclo `while` no debe terminar (es decir, debe crear un ciclo infinito). Para ello, simplemente use la palabra clave `true` como la expresión para la instrucción `while`. ¿Qué ocurre cuando ejecuta este programa?

4.30 Escriba un programa que lea el radio de un círculo (como un valor `double`), calcule e imprima el diámetro, la circunferencia y el área. Use el valor 3.14159 para π .

4.31 ¿Qué está mal con la siguiente instrucción? Proporcione la instrucción correcta para realizar lo que probablemente el programador trataba de hacer.

```
cout << ++( x + y );
```

4.32 Escriba un programa que lea tres valores `double` distintos de cero, y que determine e imprima si podrían representar los lados de un triángulo.

4.33 Escriba un programa que lea tres enteros distintos de cero, y que determine e imprima si podrían ser los lados de un triángulo recto.

4.34 (*Criptografía*) Una compañía desea transmitir datos a través del teléfono, pero le preocupa que sus teléfonos puedan estar intervenidos. Todos los datos se transmiten como enteros de cuatro dígitos. La compañía le ha pedido a usted que escriba un programa que cifre sus datos, de manera que éstos puedan transmitirse con más seguridad. Su programa debe leer un entero de cuatro dígitos introducido por el usuario y cifrarlo de la siguiente manera: reemplace cada dígito con (*el resultado de sumar 7 al dígito*) *módulo 10*. Luego intercambie el primer dígito con el tercero, e intercambie el segundo dígito con el cuarto. Despues imprima el entero cifrado. Escriba un programa separado que reciba como entrada un entero de cuatro dígitos cifrado, y que lo descifre para formar el número original.

4.35 El factorial de un entero n no negativo se escribe como $n!$ (n factorial) y se define de la siguiente manera:

$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1$ (para valores de n mayores o iguales a 1)

y

$n! = 1$ (para $n = 0$ o $n = 1$).

Por ejemplo, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, que es 120. Use instrucciones `while` en cada uno de los siguientes casos:

- Escriba una aplicación que lea un entero no negativo, que calcule e imprima su factorial.
- Escriba un programa que estime el valor de la constante matemática e , utilizando la fórmula:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

Pida al usuario la precisión deseada de e (es decir, el número de términos en la suma).

- Escriba una aplicación que calcule el valor de e^x , utilizando la fórmula

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

Pida al usuario la precisión deseada de e (es decir, el número de términos en la suma).

4.36 [Nota: este ejercicio corresponde a la sección 4.13, una parte de nuestro Ejemplo práctico de Ingeniería de Software.] Describa en 200 palabras o menos qué es un automóvil y qué hace. Enliste los sustantivos y verbos por separado. En el texto, indicamos que cada sustantivo podría corresponder a un objeto que habrá que construir para implementar un sistema, en este caso un auto. Elija cinco de los objetos que enlistó y, para cada uno, enliste varios atributos y comportamientos. Describa brevemente cómo interactúan estos objetos entre sí, y con los demás objetos en su descripción. Acaba de realizar varios de los pasos clave en un típico diseño orientado a objetos.

5



No todo lo que puede contarse cuenta, y no todo lo que cuenta puede contarse.

—Albert Einstein

¿Quién puede controlar su destino?

—William Shakespeare

La llave usada siempre brilla.

—Benjamín Franklin

Inteligencia... es la facultad de hacer que los objetos artificiales, en especial las herramientas, crean herramientas.

—Henri Bergson

Toda ventaja en el pasado se juzga a la luz de la cuestión final.

—Demóstenes

Instrucciones de control: parte 2

OBJETIVOS

En este capítulo aprenderá a:

- Conocer los fundamentos de la repetición controlada por un contador.
- Aprender a usar las instrucciones de repetición `for` y `do...while` para ejecutar instrucciones repetidas veces en una aplicación.
- Implementar el funcionamiento de la selección múltiple mediante el uso de la instrucción de selección `switch`.
- Aprender a usar las instrucciones de control de programa `break` y `continue` para alterar el flujo de control.
- Aprender a usar los operadores lógicos para formar expresiones condicionales complejas en las instrucciones de control.
- Evitar las consecuencias de confundir los operadores de igualdad y de asignación.

- 5.1 Introducción
- 5.2 Fundamentos de la repetición controlada por contador
- 5.3 Instrucción de repetición **for**
- 5.4 Ejemplos acerca del uso de la instrucción **for**
- 5.5 Instrucción de repetición **do...while**
- 5.6 Instrucción de selección múltiple **switch**
- 5.7 Instrucciones **break** y **continue**
- 5.8 Operadores lógicos
- 5.9 Confusión entre los operadores de igualdad (==) y de asignación (=)
- 5.10 Resumen de programación estructurada
- 5.11 (Opcional) Ejemplo práctico de Ingeniería de Software: cómo identificar los estados y actividades de los objetos en el sistema ATM
- 5.12 Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)

5.1 Introducción

El capítulo 4 nos introdujo a los tipos de bloques de construcción disponibles para solucionar problemas. Utilizamos dichos bloques de construcción para emplear las técnicas, ya comprobadas, de la construcción de programas. En este capítulo continuaremos nuestra presentación de la teoría y los principios de la programación estructurada, presentando el resto de las instrucciones de control en C++. Las instrucciones de control que estudiaremos aquí y las que vimos en el capítulo 4 son útiles para crear y manipular objetos. Continuaremos con nuestro énfasis anticipado sobre la programación orientada a objetos, que empezó con una discusión de los conceptos básicos en el capítulo 1, además de muchos ejemplos y ejercicios de código orientado a objetos en los capítulos 3 y 4.

En este capítulo demostraremos las instrucciones **for**, **do...while** y **switch**. A través de una serie de ejemplos cortos en los que utilizaremos las instrucciones **while** y **for**, exploraremos los fundamentos acerca de la repetición controlada por contador. Dedicaremos una parte de este capítulo a expandir la clase **LibroCalificaciones** que presentamos en los capítulos 3 y 4. En especial, crearemos una versión de la clase **LibroCalificaciones** que utiliza una instrucción **switch** para contar el número de calificaciones equivalentes de A, B, C, D y F, en un conjunto de calificaciones numéricas introducidas por el usuario. Presentaremos las instrucciones de control de programa **break** y **continue**. Hablaremos sobre los operadores lógicos, que nos permiten utilizar expresiones condicionales más complejas en las instrucciones de control. También examinaremos el error común de confundir los operadores de igualdad (==) y desigualdad (=), y cómo evitarlo. Por último, veremos un resumen de las instrucciones de control de C++ y las técnicas ya probadas de solución de problemas que presentamos en este capítulo y en el capítulo 4.

5.2 Fundamentos de la repetición controlada por contador

Esta sección utiliza la instrucción de repetición **while**, presentada en el capítulo 4, para formalizar los elementos requeridos para llevar a cabo la repetición controlada por contador. Este tipo de repetición requiere

1. el **nombre de una variable de control** (o contador de ciclo)
2. el **valor inicial** de la variable de control
3. la **condición de continuación de ciclo**, que evalúa el **valor final** de la variable de control (es decir, determina si el ciclo debe continuar o no)
4. el **incremento** (o **decremento**) con el que se modifica la variable de control cada vez que pasa por el ciclo.

Considere el programa simple de la figura 5.1, que imprime los números del 1 al 10. La declaración en la línea 9 *nombra* a la variable de control (**contador**), la declara como entera, reserva espacio para ella en memoria y la establece a un *valor inicial* de 1. Las declaraciones que requieren inicialización son, en efecto, instrucciones ejecutables. En C++ es más preciso llamar a una declaración que también reserva memoria (al igual que la declaración anterior) una *definición*.

```

1 // Fig. 5.1: fig05_01.cpp
2 // Repetición controlada por un contador.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int contador = 1; // declara e inicializa la variable de control
10
11    while ( contador <= 10 ) // condición de continuación de ciclo
12    {
13        cout << contador << " ";
14        contador++; // incrementa la variable de control en 1
15    } // fin de while
16
17    cout << endl; // imprime una nueva línea
18    return 0; // termina correctamente
19 } // fin de main

```

1 2 3 4 5 6 7 8 9 10

Figura 5.1 | Repetición controlada por contador.

Como las definiciones también son declaraciones, utilizaremos el término “declaración” excepto cuando la distinción sea importante.

La declaración e inicialización de `contador` (línea 9) también se podría haber realizado con las siguientes instrucciones:

```

int contador; // declara la variable de control
contador = 1; // inicializa la variable de control en 1

```

Utilizamos ambos métodos para inicializar variables.

En la línea 14 se *incrementa* el contador del ciclo en 1 cada vez que se ejecuta el cuerpo del ciclo. La condición de continuación de ciclo (línea 11) en la instrucción `while` determina si el valor de la variable de control es menor o igual que 10 (el valor final para el que la condición es `true`). Observe que el cuerpo de este `while` se ejecuta, aun y cuando la variable de control sea 10. El ciclo termina cuando la variable de control es mayor a 10 (es decir, cuando `contador` se convierte en 11).

La figura 5.1 se puede hacer más concisa si se inicializa `contador` con 0 y se sustituye la instrucción `while` con:

```

while ( ++contador <= 10 ) // condición de continuación de ciclo
    cout << contador << " ";

```

Este código ahorra una instrucción, ya que el incremento se realiza de manera directa en la condición del `while`, antes de evaluarla. Además, el código elimina las llaves alrededor del cuerpo del `while`, ya que éste ahora sólo contiene una instrucción. La codificación de tal forma condensada requiere cierta práctica, y puede producir programas que sean más difíciles de leer, depurar, modificar y mantener.

Error común de programación 5.1



Debido a que los valores de punto flotante pueden ser aproximados, controlar los ciclos con variables de punto flotante puede producir valores imprecisos del contador y pruebas de terminación imprecisas.

Tip para prevenir errores 5.1



Controle los ciclos de contador con valores enteros.

Buena práctica de programación 5.1



Coloque una línea en blanco por encima y debajo de cada instrucción de control para hacer que se destaque en el programa.



Buena práctica de programación 5.2

Demasiados niveles de anidamiento pueden hacer que un programa sea difícil de comprender. Como regla, trate de evitar utilizar más de tres niveles de sangría.



Buena práctica de programación 5.3

El espaciado vertical por encima y debajo de las instrucciones de control, y la sangría en los cuerpos de las instrucciones de control dentro de sus encabezados, proporciona a los programas una apariencia bidimensional que mejora en forma considerable su legibilidad.

5.3 Instrucción de repetición for

La sección 5.2 presentó los aspectos esenciales de la repetición controlada por contador. La instrucción `while` puede utilizarse para implementar cualquier ciclo controlado por un contador. C++ también cuenta con la instrucción **de repetición for**, la cual especifica los detalles de la repetición controlada por contador en una sola línea de código. Para ilustrar el poder del **for**, vamos a modificar el programa de la figura 5.1. El resultado se muestra en la figura 5.2.

Cuando la instrucción **for** (líneas 11 y 12) se empieza a ejecutar, la variable de control **contador** se declara e inicializa en 1. A continuación, el programa verifica la condición de continuación de ciclo (línea 11 entre los signos de punto y coma) **contador <= 10**. Como el valor inicial de **contador** es 1, la condición se satisface y la instrucción del cuerpo (línea 12) imprime el valor de **contador**, que es 1. Despues, la expresión **contador++** incrementa la variable de control **contador** y el ciclo empieza de nuevo, con la prueba de continuación de ciclo. Ahora la variable de control es igual a 2, por lo que no se excede del valor final y el programa ejecuta la instrucción del cuerpo otra vez. Este proceso continúa hasta que el cuerpo del ciclo se haya ejecutado 10 veces y la variable de control **contador** se incremente a 11, con lo cual falla la prueba de continuación de ciclo y termina la repetición. El programa continúa, ejecutando la primera instrucción después de la instrucción **for** (en este caso, la instrucción de salida en la línea 14).

Componentes del encabezado de la instrucción for

La figura 5.3 muestra un análisis más detallado del encabezado de la instrucción **for** (línea 11) de la figura 5.2. Observe que el encabezado de la instrucción **for** “se encarga de todo”: especifica cada uno de los elementos necesarios para la repetición controlada por contador con una variable de control. Si hay más de una instrucción en el cuerpo del **for**, se requieren llaves para encerrar el cuerpo del ciclo.

Observe que la figura 5.2 utiliza la condición de continuación de ciclo **contador <= 10**. Si usted especificara por error **contador < 10** como la condición, el ciclo sólo iteraría 9 veces. A este error lógico común se le conoce como **error por desplazamiento en 1**.

```

1 // Fig. 5.2: fig05_02.cpp
2 // Repetición controlada por contador con la instrucción for.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     // el encabezado de la instrucción for incluye la inicialización,
10    // la condición de continuación del ciclo y el incremento.
11    for ( int contador = 1; contador <= 10; contador++ )
12        cout << contador << " ";
13
14    cout << endl; // imprime una nueva línea
15    return 0; // indica que terminó correctamente
16 } // fin de main

```

1 2 3 4 5 6 7 8 9 10

Figura 5.2 | Repetición controlada por contador con la instrucción **for**.

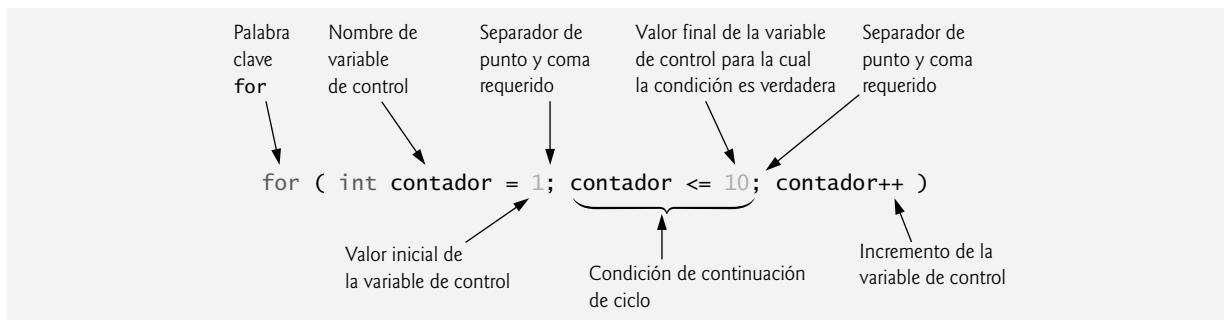


Figura 5.3 | Componentes del encabezado de la instrucción for.



Error común de programación 5.2

Utilizar un operador relacional incorrecto o un valor final incorrecto de un contador de ciclo en la condición de continuación de ciclo de una instrucción while o for puede producir errores por desplazamiento en 1.



Buena práctica de programación 5.4

Utilizar el valor final en la condición de una instrucción while o for con el operador relacional `<=` nos ayuda a evitar los errores por desplazamiento en 1. Por ejemplo, para un ciclo que imprime los valores del 1 al 10, la condición de continuación de ciclo debe ser `contador <= 10`, en lugar de `contador < 10` (lo cual produce un error por desplazamiento en uno) o `contador < 11` (que es correcto). Muchos programadores prefieren el llamado **conteo con base cero**, en el cual para contar 10 veces, `contador` se inicializaría a cero y la prueba de continuación de ciclo sería `contador < 10`.

El formato general de la instrucción for es

```
for ( inicialización; condiciónDeContinuaciónDeCiclo; incremento )
    instrucción
```

donde la expresión *inicialización* inicializa la variable de control del ciclo, la *condiciónDeContinuaciónDeCiclo* determina si el ciclo debe seguir ejecutándose (por lo general, esta condición contiene el valor final de la variable de control para la cual la condición es verdadera) y el *incremento* incrementa el valor de la variable de control. En la mayoría de los casos, la instrucción for se puede representar mediante una instrucción while equivalente, como se muestra a continuación:

```
inicialización;
while ( condiciónDeContinuaciónDeCiclo )
{
    instrucción
    incremento;
}
```

En la sección 5.7 veremos una excepción a esta regla.

Si la expresión de *inicialización* en el encabezado de la instrucción for declara la variable de control (es decir, si el tipo de la variable de control se especifica antes del nombre de la variable), la variable de control puede utilizarse sólo en el cuerpo de esa instrucción for; no existirá fuera de esta instrucción. Este uso restringido del nombre de la variable de control se conoce como el **alcance** de la variable. El alcance de una variable especifica dónde puede utilizarse en un programa. En el capítulo 6, Funciones y una introducción a la recursividad, veremos con detalle el concepto de alcance.



Error común de programación 5.3

Cuando se declara la variable de control de una instrucción for en la sección de inicialización del encabezado del for, si se utiliza la variable de control fuera del cuerpo de la instrucción se produce un error de compilación.



Tip de portabilidad 5.1

En C++ estándar, el alcance de la variable de control declarada en la sección de inicialización de una instrucción for difiere del alcance en los compiladores anteriores de C++. En los compiladores previos al estándar, el alcance de la variable de control no termina al final del bloque que define el cuerpo de la instrucción for; en lugar de ello, el alcance termina al final del bloque

que encierra a la instrucción `for`. El código de C++ creado con compiladores anteriores al C++ estándar puede tener fallas al compilarse en compiladores que se conformen al estándar. Si trabaja con compiladores previos al estándar y desea estar seguro que su código funcionará con compiladores que se conformen al estándar, hay dos estrategias de programación defensivas que puede usar: declarar las variables de control con distintos nombres en cada instrucción `for`, o si prefiere usar el mismo nombre para la variable de control en varias instrucciones `for`, declare la variable de control antes de la primera instrucción `for`.

Como veremos más adelante, las expresiones *inicialización* e *incremento* pueden ser listas de expresiones separadas por comas. Las comas, según el uso que se les da en estas expresiones, son **operadores coma**, los cuales garantizan que las listas de expresiones se evalúen de izquierda a derecha. El operador coma tiene la menor precedencia de todos los operadores de C++. El valor y tipo de una lista de expresiones separadas por comas es el valor y tipo de la expresión de la lista que está más a la derecha. El operador coma se utiliza con frecuencia en las instrucciones `for`. Su principal aplicación es permitir al programador utilizar varias expresiones de inicialización y/o varias expresiones de incremento. Por ejemplo, puede haber distintas variables de control en una sola instrucción `for` que deban inicializarse e incrementarse.



Buena práctica de programación 5.5

Coloque sólo expresiones que involucren a las variables de control en las secciones de inicialización e incremento de una instrucción `for`. Las manipulaciones de otras variables deben aparecer, ya sea antes del ciclo (si deben ejecutarse sólo una vez, al igual que las instrucciones de inicialización) o en el cuerpo del ciclo (si deben ejecutarse una vez por cada repetición, como las instrucciones de incremento o decremento).

Las tres expresiones en un encabezado `for` son opcionales (pero los dos separadores de punto y coma son obligatorios). Si se omite la *condiciónDeContinuaciónDeCiclo*, C++ asume que esta condición siempre será verdadera, con lo cual se crea un ciclo infinito. Podríamos omitir la expresión de *inicialización* si el programa inicializa la variable de control antes del ciclo. Podríamos omitir la expresión de *incremento* si el programa calcula el incremento mediante instrucciones dentro del cuerpo del `for`, o si no se necesita un incremento. La expresión de incremento en una instrucción `for` actúa como si fuera una instrucción independiente al final del cuerpo del `for`. Por lo tanto, las expresiones

```
contador = contador + 1
contador += 1
++contador
contador++
```

son todas equivalentes en la porción de incremento de la instrucción `for` (cuando no aparece ningún otro código ahí). Muchos programadores prefieren la forma `contador++`, ya que un ciclo `for` evalúa su expresión de incremento después de la ejecución de su cuerpo. Por ende, la forma de postincremento parece más natural. En este caso, la variable que se incrementa no aparece en una expresión más grande, por lo que los operadores de preincremento y postdecremento tienen en realidad el mismo efecto.



Error común de programación 5.4

El uso de comas en lugar de los dos signos de punto y coma requeridos en un encabezado `for` es un error de sintaxis.



Error común de programación 5.5

Al colocar un punto y coma justo a la derecha del paréntesis derecho del encabezado de un `for`, el cuerpo de esa instrucción `for` se convierte en una instrucción vacía. Por lo general, esto es un error lógico.

Las expresiones de inicialización, condición de continuación de ciclo e incremento de una instrucción `for` pueden contener expresiones aritméticas. Por ejemplo, si `x = 2` y `y = 10`, y además, `x` y `y` no se modifican en el cuerpo del ciclo, el siguiente encabezado de `for`:

```
for ( int j = x; j <= 4 * x * y; j += y / x )
```

es equivalente a la instrucción

```
for ( int j = 2; j <= 80; j += 5 )
```

El “incremento” de una instrucción `for` también puede ser negativo, en cuyo caso sería un decremento y el ciclo contaría en orden descendente (como se muestra en la sección 5.4).

Si al principio la condición de continuación de ciclo es falsa, el programa no ejecutará el cuerpo de la instrucción `for`, sino que la ejecución continuará con la instrucción que siga inmediatamente después del `for`.

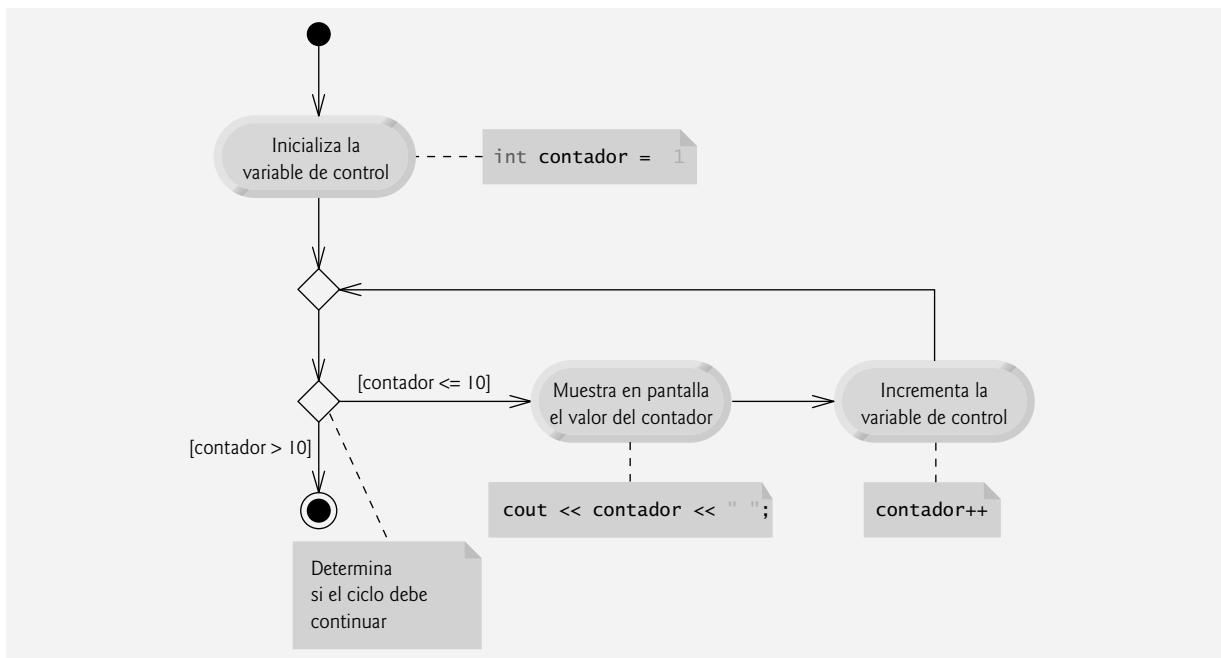


Figura 5.4 | Diagrama de actividad de UML para la instrucción **for** de la figura 5.2.

Con frecuencia, la variable de control se imprime o utiliza en cálculos dentro del cuerpo de una instrucción **for**, pero este uso no es obligatorio. Por lo general, la variable de control se utiliza para controlar la repetición sin que se le mencione dentro del cuerpo de la instrucción **for**.



Tip para prevenir errores 5.2

Aunque el valor de la variable de control puede cambiarse en el cuerpo de una instrucción for, evite hacerlo, ya que esta práctica puede llevártalo a cometer errores sutiles.

Diagrama de actividad de UML de la instrucción for

El diagrama de actividad de UML de la instrucción **for** es similar al de la instrucción **while** (figura 4.6). La figura 5.4 muestra el diagrama de actividad de la instrucción **for** de la figura 5.2. El diagrama hace evidente que la inicialización ocurre sólo una vez antes de evaluar la condición de continuación de ciclo por primera vez, y que el incremento ocurre cada vez que se realiza una iteración, *después* de que se ejecuta la instrucción del cuerpo. Observe que (además de un estado inicial, flechas de transición, una fusión, un estado final y varias notas) el diagrama sólo contiene estados de acción y una decisión. Imagine de nuevo que tiene un recipiente de diagramas de actividad de UML de instrucciones **for** vacío; tantas instrucciones como podrían ser necesarias para apilarlas y anidarlas con los diagramas de actividades de otras instrucciones de control, para formar la implementación estructurada de un algoritmo. Usted completa los estados de acción y los símbolos de decisión con expresiones de acción y condiciones de guardia apropiadas para el algoritmo.

5.4 Ejemplos acerca del uso de la instrucción for

Los siguientes ejemplos muestran métodos para modificar la variable de control en una instrucción **for**. En cada caso, escribimos el encabezado **for** apropiado. Observe el cambio en el operador relacional para los ciclos que decrementan la variable de control.

- Modificar la variable de control de 1 a 100 en incrementos de 1.

```
for ( int i = 1; i <= 100; i++ )
```

- Modificar la variable de control de 100 a 1 en incrementos de -1 (es decir, decrementos de 1).

```
for ( int i = 100; i >= 1; i-- )
```

- c) Modificar la variable de control de 7 a 77 en incrementos de 7.

```
for ( int i = 7; i <= 77; i += 7 )
```

- d) Modificar la variable de control de 20 a 2 en incrementos de -2.

```
for ( int i = 20; i >= 2; i -= 2 )
```

- e) Modificar la variable de control con la siguiente secuencia de valores: 2, 5, 8, 11, 14, 17, 20.

```
for ( int i = 2; i <= 20; i += 3 )
```

- f) Modificar la variable de control con la siguiente secuencia de valores: 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

```
for ( int i = 99; i >= 0; i -= 11 )
```



Error común de programación 5.6

No utilizar el operador relacional apropiado en la condición de continuación de un ciclo que cuente en forma regresiva (como usar incorrectamente `i <= 1` en lugar de `i >= 1` en un ciclo que cuente en forma regresiva hasta llegar a 1) es generalmente un error lógico que produce resultados incorrectos al momento de ejecutar el programa.

Aplicación: sumar los enteros pares del 2 al 20

Las dos aplicaciones de ejemplo siguientes demuestran usos simples de la instrucción `for`. El programa de la figura 5.5 utiliza una instrucción `for` para sumar los enteros pares del 2 al 20. Cada iteración del ciclo (líneas 12 y 13) suma el valor actual de la variable de control `numero` a la variable `total`.

Observe que el cuerpo de la instrucción `for` de la figura 5.5 podría mezclarse con la porción del incremento del encabezado `for` mediante el uso de una coma, como se muestra a continuación:

```
for ( int numero = 2; // inicialización
      numero <= 20; // condición de continuación de ciclo
      total += numero, numero += 2 ) // calcula el total e incrementa
; // instrucción vacía
```



Buena práctica de programación 5.6

Aunque las instrucciones antes de un `for` y las instrucciones en el cuerpo de un `for` comúnmente se pueden fusionar en el encabezado del `for`, esto podría hacer que el programa fuera más difícil de leer, mantener, modificar y depurar.



Buena práctica de programación 5.7

Límite el tamaño de los encabezados de las instrucciones de control a una sola línea, si es posible.

```

1 // Fig. 5.5: fig05_05.cpp
2 // Suma de enteros con la instrucción for.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int total = 0; // inicializa el total
10
11    // obtiene el total de los enteros pares del 2 al 20
12    for ( int numero = 2; numero <= 20; numero += 2 )
13        total += numero;
14
15    cout << "La suma es " << total << endl; // muestra los resultados
16    return 0; // terminó correctamente
17 } // fin de main

```

La suma es 110

Figura 5.5 | Suma de enteros con la instrucción `for`.

Aplicación: cálculo del interés compuesto

El siguiente ejemplo utiliza la instrucción `for` para calcular el interés compuesto. Considere el siguiente enunciado del problema:

Una persona invierte \$1000.00 en una cuenta de ahorro que produce 5 por ciento de interés. Suponiendo que todo el interés se deposita en la cuenta, calcule e imprima el monto de dinero en la cuenta al final de cada año, durante 10 años. Use la siguiente fórmula para determinar los montos:

$$c = p (1 + r)^n$$

donde

p es el monto que se invirtió originalmente (es decir, el monto principal)

t es la tasa de interés anual (por ejemplo, use 0.05 para 5%)

n es el número de años

c es la cantidad depositada al final del n-ésimo año.

Este problema implica el uso de un ciclo que realiza los cálculos indicados para cada uno de los 10 años que el dinero permanece depositado. La solución es la aplicación que se muestra en la figura 5.6.

La instrucción `for` (líneas 28 a 35) ejecuta su cuerpo 10 veces, con lo cual la variable de control varía de 1 a 10, en incrementos de 1. C++ no incluye un operador de exponenciación, por lo que utilizamos la función **pow** de la biblioteca estándar (línea 31) para este propósito. La función `pow(x, y)` calcula el valor de `x` elevado a la `y`-ésima potencia. En este ejemplo, la expresión algebraica $(1 + r)^n$ se escribe como `pow(1.0 + tasa, anio)`, donde la variable `tasa` representa a `r` y la variable `anio` representa a `n`. La función `pow` recibe dos argumentos de tipo `double` y devuelve un valor `double`.

```

1 // Fig. 5.6: fig05_06.cpp
2 // Cálculo del interés compuesto con for.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setw; // permite al programa establecer una anchura de campo
10 using std::setprecision;
11
12 #include <cmath> // biblioteca de matemáticas estándar de C++
13 using std::pow; // permite al programa utilizar la función pow
14
15 int main()
16 {
17     double monto; // monto a depositar al final de cada año
18     double principal = 1000.0; // monto inicial antes del interés
19     double tasa = .05; // tasa de interés
20
21     // muestra los encabezados
22     cout << "Anio" << setw( 21 ) << "Monto en depósito" << endl;
23
24     // establece el formato de número de punto flotante
25     cout << fixed << setprecision( 2 );
26
27     // calcula el monto en depósito para cada uno de los diez años
28     for ( int anio = 1; anio <= 10; anio++ )
29     {
30         // calcula el nuevo monto para el año especificado
31         monto = principal * pow( 1.0 + tasa, anio );
32
33         // muestra el año y el monto
34         cout << setw( 4 ) << anio << setw( 21 ) << monto << endl;
35     } // fin de for

```

Figura 5.6 | Cálculo del interés compuesto con `for`. (Parte I de 2).

```

36
37     return 0; // indica que terminó correctamente
38 } // fin de main

```

Anio	Monto en deposito
1	1050.00
2	1102.50
3	1157.63
4	1215.51
5	1276.28
6	1340.10
7	1407.10
8	1477.46
9	1551.33
10	1628.89

Figura 5.6 | Cálculo del interés compuesto con `for`. (Parte 2 de 2).

Este programa no se compilará si no se incluye el archivo de encabezado `<cmath>` (línea 12). La función `pow` requiere dos argumentos `double`. Observe que `anio` es un entero. El encabezado `<cmath>` incluye información que indica al compilador cómo convertir el valor de `anio` en una representación `double` temporal antes de llamar a la función. Esta información se incluye en el prototipo de la función `pow`. En el capítulo 6 veremos un resumen de las demás funciones matemáticas de la biblioteca.



Error común de programación 5.7

En general, olvidar incluir el archivo de encabezado apropiado al utilizar funciones de la biblioteca estándar (por ejemplo, `<cmath>` en un programa que utilice funciones matemáticas de la biblioteca) es un error de compilación.

Una advertencia en relación con el uso de los tipos `float` o `double` para cantidades monetarias

Observe que en las líneas 17 a 19 se declaran las variables `double` llamadas `monto`, `principal` y `tasa`. Hicimos esto para simplificar, ya que estamos tratando con partes fraccionarias de dólares, y necesitamos un tipo que permita puntos decimales en sus valores. Por desgracia, esto puede ocasionar problemas. He aquí una explicación simple de lo que puede salir mal al utilizar `float` o `double` para representar montos en dólares (asumiendo que se utiliza `setprecision(2)` para especificar dos dígitos de precisión a la hora de imprimir): dos montos en dólares almacenados en el equipo podrían ser 14.234 (que se imprime como 14.23) y 18.673 (que se imprime como 18.67). Al sumar estos montos, producen la suma interna 32.907, la cual se imprime como 32.91. Por ende, el resultado podría aparecer como

14.23	
+ 18.67	

32.91	

pero ¡alguien que sumara los números individuales que aparecen impresos esperaría la suma de 32.90! ¡Ya ha sido advertido!



Buena práctica de programación 5.8

No utilice variables de tipo `float` o `double` para realizar cálculos monetarios. La imprecisión de los números de punto flotante puede provocar errores que resulten en valores monetarios incorrectos. En los ejercicios vamos a explorar el uso de enteros para realizar cálculos monetarios. [Nota: algunos distribuidores de software venden bibliotecas de clases de C++ que realizan cálculos monetarios precisos.]

Uso de manipuladores de flujo para dar formato a los resultados numéricos

La instrucción de salida en la línea 25 antes del ciclo `for`, y la instrucción de salida en la línea 34 en el ciclo `for` se combinan para imprimir los valores de las variables `anio` y `monto`, con el formato especificado por los manipuladores de flujo parametrizados `setprecision` y `setw`, y por el manipulador de flujo no parametrizado `fixed`. El manipulador de flujo `setw(4)` especifica que el siguiente valor a imprimir debe aparecer en una **anchura de campo** de 4; es decir, `cout` imprime el valor con al menos 4 posiciones de caracteres. Si el valor a imprimir es menor que 4 caracteres de ancho, de manera predeterminada se imprime **justificado a la derecha**. Si el valor a imprimir es mayor que 4 caracteres, la anchura de campo se extiende

para dar cabida a todo el valor. Para indicar que los valores deben imprimirse **justificados a la izquierda**, simplemente imprima el manipulador de flujo no parametrizado `left` (que se encuentra en el encabezado `<iostream>`). La justificación a la derecha se puede restaurar al imprimir el manipulador de flujo no parametrizado `right`.

El otro formato en las instrucciones de salida indica que la variable `monto` se imprime como un valor de punto fijo con un punto decimal (especificado en la línea 25 con el manipulador de flujo `fixed`), justificado a la derecha en un campo de 21 posiciones de caracteres (especificado en la línea 34 con `setw(21)`) y dos dígitos de precisión a la derecha del punto decimal (especificado en la línea 25 con el manipulador `setprecision(2)`). Aplicamos los manipuladores de flujo `fixed` y `setprecision` al flujo de salida (es decir, `cout`) antes del ciclo `for`, ya que estas opciones de formato están en vigor hasta que se modifican; a dichas opciones se les conoce como **opciones pegajosas** y no necesitan aplicarse durante cada iteración del ciclo. Sin embargo, la anchura de campo especificada con `setw` sólo se aplica al siguiente valor que se imprime. En el capítulo 15, Entrada y salida de flujos, hablaremos sobre las poderosas herramientas de formato de entrada/salida de C++.

Observe que el cálculo `1.0 + tasa`, que aparece como argumento para la función `pow`, está contenido en el cuerpo de la instrucción `for`. De hecho, este cálculo produce el mismo resultado durante cada iteración del ciclo, por lo que repetirlo es un desperdicio; debería realizarse una vez antes del ciclo.



Tip de rendimiento 5.1

Evite colocar expresiones cuyos valores no cambien dentro de los ciclos; aun si lo hace, muchos de los compiladores optimizadores sofisticados de la actualidad colocarán de manera automática dichas expresiones fuera de los ciclos en el código de lenguaje máquina generado.



Tip de rendimiento 5.2

Muchos compiladores contienen características de optimización que mejoran el rendimiento del código que el programador escribe, pero aun así es mejor escribir buen código desde el principio.

Asegúrese de probar nuestro problema de Peter Minuit en el ejercicio 5.29. Este problema demuestra las maravillas del interés compuesto.

5.5 Instrucción de repetición do...while

La instrucción `do...while` es similar a la instrucción `while`. En la instrucción `while`, la evaluación de la condición de continuación de ciclo ocurre al principio del ciclo, antes de ejecutar su cuerpo. La instrucción `do...while` evalúa la condición de continuación de ciclo *después* de ejecutar el cuerpo del ciclo; por lo tanto, el cuerpo del ciclo siempre se ejecutará cuando menos una vez. Cuando termina una instrucción `do...while`, la ejecución continúa con la instrucción que va después de la cláusula `while`. No es necesario utilizar llaves en la instrucción `do...while` si sólo hay una instrucción en el cuerpo; sin embargo, la mayoría de los programadores incluyen las llaves para evitar la confusión entre las instrucciones `while` y `do...while`. Por ejemplo:

```
while ( condición )
```

generalmente se utiliza como encabezado de una instrucción `while`. Una instrucción `do...while` sin llaves, alrededor de un cuerpo con una sola instrucción, aparece así:

```
do
    instrucción
  while ( condición );
```

Lo cual puede ser confuso. Podríamos malinterpretar la última línea [`while(condición);`] como una instrucción `while` que contiene como cuerpo una instrucción vacía. Por ello, el `do...while` con una instrucción se escribe a menudo de la siguiente manera, para evitar confusión:

```
do
{
    instrucción
} while ( condición );
```



Buena práctica de programación 5.9

Incluir siempre las llaves en una instrucción `do...while` ayuda a eliminar la ambigüedad entre la instrucción `while` y la instrucción `do...while` que sólo contiene una instrucción.

La figura 5.7 utiliza una instrucción `do...while` para imprimir los números del 1 al 10. Al entrar a la instrucción `do...while`, en la línea 13 se imprime el valor de `contador` y en la línea 14 se incrementa `contador`. Después el programa evalúa la prueba de continuación de ciclo al final del mismo (línea 15). Si la condición es verdadera, el ciclo continúa a partir de la primera instrucción del cuerpo en la instrucción `do...while` (línea 13). Si la condición es falsa, el ciclo termina y el programa continúa con la siguiente instrucción después del ciclo (línea 17).

```

1 // Fig. 5.7: fig05_07.cpp
2 // La instrucción de repetición do...while.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int contador = 1; // inicializa contador
10
11    do
12    {
13        cout << contador << " "; // muestra contador
14        contador++; // incrementa contador
15    } while ( contador <= 10 ); // fin de do...while
16
17    cout << endl; // imprime una nueva línea
18    return 0; // indica que terminó correctamente
19 } // fin de main

```

1 2 3 4 5 6 7 8 9 10

Figura 5.7 | La instrucción de repetición `do...while`.

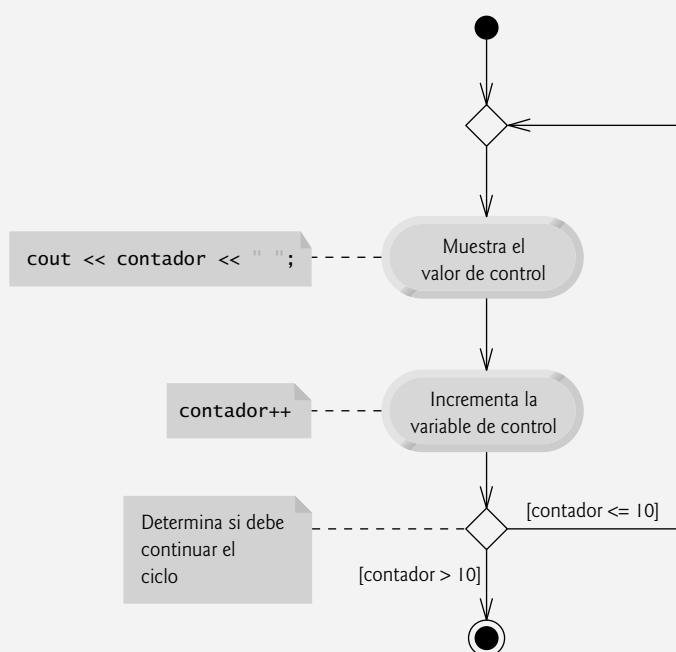


Figura 5.8 | Diagrama de actividad de UML de la instrucción de repetición `do...while` de la figura 5.7.

Diagrama de actividad de UML de la instrucción do...while

La figura 5.8 contiene el diagrama de actividad de UML para la instrucción `do...while`. Este diagrama hace evidente que la condición de continuación de ciclo no se evalúa sino hasta después que el ciclo ejecuta el estado de acción, por lo menos una vez. Compare este diagrama de actividad con el de la instrucción `while` (figura 4.6). De nuevo, observe que (además de un estado inicial, flechas de transición, una fusión, un estado final y varias notas) el diagrama sólo contiene estados de acción y una decisión. Imagine otra vez que tiene acceso a un recipiente de diagramas de actividad de UML de instrucciones `do...while` vacío; todas las que podría necesitar para apilar y anidar con los diagramas de actividad de otras instrucciones de control para formar una implementación estructurada de un algoritmo. El programador completa los estados de acción y los símbolos de decisión con expresiones de acción y condiciones de guardia apropiadas para el algoritmo.

5.6 Instrucción de selección múltiple switch

En el capítulo 4 hablamos sobre la instrucción `if` de selección simple y la instrucción `if...else` de selección doble. C++ cuenta con la instrucción `switch` de selección múltiple para realizar distintas acciones, con base en los posibles valores de una variable o expresión. Cada acción se asocia con un valor de una **expresión integral constante** (es decir, una combinación de constantes tipo carácter y constantes enteras que se evalúan como un valor entero constante) con la que se pueda evaluar la variable o expresión.

La clase LibroCalificaciones con la instrucción switch para contar las calificaciones A, B, C, D y F.

Ahora presentaremos una versión mejorada de la clase `LibroCalificaciones` que presentamos en el capítulo 3, y desarrollamos un poco más en el capítulo 4. La nueva versión de la clase pide al usuario que introduzca un conjunto de calificaciones y después muestra un resumen del número de estudiantes que recibieron cada calificación. La clase utiliza una instrucción `switch` para determinar si cada calificación introducida es el equivalente de A, B, C, D o F, y para incrementar el contador de la calificación apropiada. La clase `LibroCalificaciones` está definida en la figura 5.9, y las definiciones de sus funciones miembro aparecen en la figura 5.10. La figura 5.11 muestra la entrada y la salida de ejemplo del programa `main` que utiliza la clase `LibroCalificaciones` para procesar un conjunto de calificaciones.

Al igual que las versiones anteriores de la definición de la clase, esta definición de `LibroCalificaciones` (figura 5.9) contiene prototipos de funciones para las funciones miembro `establecerNombreCurso` (línea 13), `obtenerNombreCurso` (línea 14) y `mostrarMensaje` (línea 15), así como el constructor de la clase (línea 12). La definición de la clase también declara el miembro de datos `private` llamado `nombreCurso` (línea 19).

```

1 // Fig. 5.9: LibroCalificaciones.h
2 // Definición de la clase LibroCalificaciones que cuenta calificaciones A, B, C, D y F.
3 // Las funciones miembro se definen en LibroCalificaciones.cpp
4
5 #include <string> // el programa usa la clase string estándar de C++
6 using std::string;
7
8 // definición de la clase LibroCalificaciones
9 class LibroCalificaciones
10 {
11 public:
12     LibroCalificaciones( string ); // el constructor inicializa el nombre del curso
13     void establecerNombreCurso( string ); // función para establecer el nombre del curso
14     string obtenerNombreCurso(); // función para obtener el nombre del curso
15     void mostrarMensaje(); // muestra un mensaje de bienvenida
16     void recibirCalificaciones(); // recibe un número arbitrario de calificaciones del usuario
17     void mostrarReporteCalificaciones(); // muestra un reporte con base en las calificaciones
18 private:
19     string nombreCurso; // nombre del curso para este LibroCalificaciones
20     int aCuenta; // cuenta de calificaciones A
21     int bCuenta; // cuenta de calificaciones B
22     int cCuenta; // cuenta de calificaciones C
23     int dCuenta; // cuenta de calificaciones D
24     int fCuenta; // cuenta de calificaciones F
25 }; // fin de la clase LibroCalificaciones

```

Figura 5.9 | Definición de la clase `LibroCalificaciones`.

Ahora la clase `LibroCalificaciones` (figura 5.9) contiene cinco miembros de datos `private` adicionales (líneas 20 a 24): variables contador para cada categoría de calificación (A, B, C, D y F). La clase también contiene dos funciones miembro `public` adicionales: `recibirCalificaciones` y `mostrarReporteCalificaciones`. La función miembro `recibirCalificaciones` (declarada en la línea 16) lee un número arbitrario de calificaciones de letra del usuario mediante el uso de la repetición controlada por centinela, y actualiza el contador de calificaciones apropiado para cada calificación introducida. La función miembro `mostrarReporteCalificaciones` (declarada en la línea 17) imprime un reporte que contiene el número de estudiantes que recibieron cada calificación de letra.

El archivo de código fuente `LibroCalificaciones.cpp` (figura 5.10) contiene las definiciones de las funciones miembro para la clase `LibroCalificaciones`. Observe que en las líneas 16 a 20 en el constructor se inicializan los cinco contadores de calificaciones con 0; cuando se crea un objeto `LibroCalificaciones` por primera vez, aún no se han introducido calificaciones. Como pronto veremos, estos contadores se incrementan en la función miembro `recibirCalificaciones` a medida que el usuario introduce las calificaciones. Las definiciones de las funciones miembro `establecerNombreCurso`, `obtenerNombreCurso` y `mostrarMensaje` son idénticas a las de las versiones anteriores de la clase `LibroCalificaciones`. Vamos a considerar las nuevas funciones miembro de `LibroCalificaciones` con detalle.

```

1 // Fig. 5.10: LibroCalificaciones.cpp
2 // Definiciones de las funciones miembro para la clase LibroCalificaciones que
3 // utiliza una instrucción switch para contar calificaciones A, B, C, D y F.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include "LibroCalificaciones.h" // incluye la definición de la clase LibroCalificaciones
10
11 // el constructor inicializa nombreCurso con la cadena suministrada como argumento;
12 // inicializa los miembros de datos contadores a 0
13 LibroCalificaciones::LibroCalificaciones( string nombre )
14 {
15     establecerNombreCurso( nombre ); // valida y almacena nombreCurso
16     aCuenta = 0; // inicializa cuenta de calificaciones A con 0
17     bCuenta = 0; // inicializa cuenta de calificaciones B con 0
18     cCuenta = 0; // inicializa cuenta de calificaciones C con 0
19     dCuenta = 0; // inicializa cuenta de calificaciones D con 0
20     fCuenta = 0; // inicializa cuenta de calificaciones F con 0
21 } // fin del constructor de LibroCalificaciones
22
23 // función para establecer el nombre del curso; limita el nombre a 25 caracteres o menos
24 void LibroCalificaciones::establecerNombreCurso( string nombre )
25 {
26     if ( nombre.length() <= 25 ) // si nombre tiene 25 caracteres o menos
27         nombreCurso = nombre; // almacena el nombre del curso en el objeto
28     else // si el nombre es mayor que 25 caracteres
29     { // establece nombreCurso a los primeros 25 caracteres del parámetro nombre
30         nombreCurso = nombre.substr( 0, 25 ); // selecciona los primeros 25 caracteres
31         cout << "El nombre \" " << nombre << "\" excede la longitud maxima (25).\n"
32         << "Se limito nombreCurso a los primeros 25 caracteres.\n" << endl;
33     } // fin de if...else
34 } // fin de la función establecerNombreCurso
35
36 // función para obtener el nombre del curso
37 string LibroCalificaciones::obtenerNombreCurso()
38 {
39     return nombreCurso;
40 } // fin de la función obtenerNombreCurso
41
42 // muestra un mensaje de bienvenida para el usuario de LibroCalificaciones
43 void LibroCalificaciones::mostrarMensaje()
```

Figura 5.10 | Clase `LibroCalificaciones` que usa la instrucción `switch` para contar calificaciones con las letras A, B, C, D y F. (Parte 1 de 3).

```
44  {
45      // esta instrucción llama a obtenerNombreCurso para obtener el
46      // nombre del curso que representa este LibroCalificaciones
47      cout << "Bienvenido al libro de calificaciones para\n" << obtenerNombreCurso() << "!\n"
48      << endl;
49 } // fin de la función mostrarMensaje
50
51 // recibe un número arbitrario de calificaciones del usuario; actualiza el contador de
52 void LibroCalificaciones::recibirCalificaciones()
53 {
54     int calificacion; // calificación introducida por el usuario
55
56     cout << "Escriba las calificaciones de letra." << endl
57         << "Escriba el carácter EOF para terminar la entrada." << endl;
58
59     // itera hasta que el usuario escriba la secuencia de fin de archivo
60     while ( ( calificacion = cin.get() ) != EOF )
61     {
62         // determina cuál calificación se introdujo
63         switch ( calificacion ) // instrucción switch anidada en el while
64         {
65             case 'A': // calificación fue A mayúscula
66             case 'a': // o a minúscula
67                 aCuenta++; // incrementa aCuenta
68                 break; // es necesario salir del switch
69
70             case 'B': // calificación fue B mayúscula
71             case 'b': // o b minúscula
72                 bCuenta++; // incrementa bCuenta
73                 break; // sale del switch
74
75             case 'C': // calificación fue C mayúscula
76             case 'c': // o c minúscula
77                 cCuenta++; // incrementa cCuenta
78                 break; // sale del switch
79
80             case 'D': // calificación fue D mayúscula
81             case 'd': // o d minúscula
82                 dCuenta++; // incrementa dCuenta
83                 break; // sale del switch
84
85             case 'F': // calificación fue F mayúscula
86             case 'f': // o f minúscula
87                 fCuenta++; // incrementa fCuenta
88                 break; // sale del switch
89
90             case '\n': // ignora caracteres de nueva línea,
91             case '\t': // tabuladores
92             case ' ': // y espacios en la entrada
93                 break; // sale del switch
94
95             default: // atrapa todos los demás caracteres
96                 cout << "Se introdujo una letra de calificación incorrecta."
97                     << " Escribe una nueva calificación." << endl;
98                 break; // opcional; saldrá del switch de todas formas
99         } // fin de switch
100    } // fin de while
101 } // fin de la función recibirCalificaciones
102
103 // muestra un reporte con base en las calificaciones introducidas por el usuario
```

Figura 5.10 | Clase LibroCalificaciones que usa la instrucción switch para contar calificaciones con las letras A, B, C, D y F. (Parte 2 de 3).

```

104 void LibroCalificaciones::mostrarReporteCalificaciones()
105 {
106     // imprime resumen de resultados
107     cout << "\n\nNúmero de estudiantes que recibieron cada calificación de letra:"
108     << "\nA: " << aCuenta // muestra el número de calificaciones A
109     << "\nB: " << bCuenta // muestra el número de calificaciones B
110     << "\nC: " << cCuenta // muestra el número de calificaciones C
111     << "\nD: " << dCuenta // muestra el número de calificaciones D
112     << "\nF: " << fCuenta // muestra el número de calificaciones F
113     << endl;
114 } // fin de la función mostrarReporteCalificaciones

```

Figura 5.10 | Clase LibroCalificaciones que usa la instrucción `switch` para contar calificaciones con las letras A, B, C, D y F. (Parte 3 de 3).

Lectura de los datos de entrada tipo carácter

El usuario introduce las calificaciones de letras para un curso en la función miembro `recibirCalificaciones` (líneas 52 a 101). Dentro del encabezado del `while`, en la línea 60, la asignación entre paréntesis (`calificacion = cin.get()`) se ejecuta primero. La función `cin.get()` lee un carácter del teclado y lo almacena en la variable entera `calificacion` (declarada en la línea 54). Por lo general, los caracteres se almacenan en las variables de tipo `char`; sin embargo, los caracteres se pueden almacenar en cualquier tipo de datos entero, ya que se garantiza que los tipos `short`, `int` y `long` son por lo menos tan grandes como el tipo `char`. Por ende, podemos tratar a un carácter como entero o como carácter, dependiendo de su uso. Por ejemplo, la instrucción

```

cout << "El carácter (" << 'a' << ") tiene el valor "
    << static_cast< int > ( 'a' ) << endl;

```

imprime el carácter a y su valor entero de la siguiente manera:

```
El carácter (a) tiene el valor 97
```

El entero 97 es la representación numérica del carácter en la computadora. La mayoría de las computadoras de la actualidad usan el **conjunto de caracteres ASCII (Código estándar estadounidense para el intercambio de información)**, en el cual el 97 representa a la letra 'a' minúscula. En el apéndice B, Conjunto de caracteres ASCII se presenta una tabla de los caracteres ASCII y sus equivalentes decimales.

En su totalidad, las instrucciones de asignación tienen el valor que se asigna a la variable en el lado izquierdo del signo `=`. Por ende, el valor de la expresión de asignación `calificacion = cin.get()` es el mismo que el valor devuelto por `cin.get()` y que se asigna a la variable `calificacion`.

El hecho de que las expresiones de asignación tengan valores puede ser útil para asignar el mismo valor a distintas variables. Por ejemplo,

```
a = b = c = 0;
```

evalúa primero la asignación `c = 0` (ya que el operador `=` asocia de derecha a izquierda). Después, a la variable `b` se le asigna el valor de la asignación `c = 0` (que es 0). Después, a la variable `a` se le asigna el valor de la asignación `b = (c = 0)` (que también es 0). En el programa, el valor de la asignación `calificacion = cin.get()` se compara con el valor de `EOF` (un símbolo cuyo acrónimo significa "fin de archivo"). Utilizamos `EOF` (que por lo general tiene el valor -1) como el valor centinela. *Sin embargo, no debe escribir el valor -1, ni las letras EOF, como el valor centinela.* En lugar de ello, debe escribir una combinación de teclas dependiente del sistema, que represente el "fin de archivo" para indicar que no hay más datos que introducir. `EOF` es una constante entera simbólica definida en el archivo de encabezado `<iostream>`. Si el valor asignado a `calificacion` es igual a `EOF`, el ciclo `while` (líneas 60 a 100) termina. Hemos optado por representar los caracteres introducidos en este programa como valores `int`, ya que `EOF` tiene el tipo `int`.

En los sistemas UNIX/Linux y muchos otros, el fin de archivo se introduce escribiendo la secuencia

```
<Ctrl> d
```

en una línea por sí sola. Esta notación significa que hay que oprimir al mismo tiempo la tecla `Ctrl` y la tecla `d`. En otros sistemas como Microsoft Windows, para introducir el fin de archivo se escribe

```
<Ctrl> z
```

[*Nota:* en algunos casos, hay que oprimir `Intro` después de escribir la secuencia de teclas anterior. Además, generalmente se muestran los caracteres `^Z` en la pantalla para representar el fin de archivo, como se muestra en la figura 5.11.]



Tip de portabilidad 5.2

Las combinaciones de teclas para introducir el fin de archivo son dependientes del sistema.



Tip de portabilidad 5.3

La acción de evaluar en la condición la constante simbólica EOF en lugar de -1 hace que los programas sean más portables. El estándar ANSI/ISO de C, del cual C++ adopta la definición de EOF, establece que EOF es un valor integral entero (pero no necesariamente -1), por lo que podría tener distintos valores en distintos sistemas.

En este programa, el usuario introduce las calificaciones mediante el teclado. Cuando el usuario oprime la tecla *Intro* (o *Return*), la función `cin.get()` lee los caracteres, uno a la vez. Si el carácter introducido no es el fin de archivo, el flujo de control entra a la instrucción `switch` (líneas 63 a 99), la cual incrementa el contador de calificaciones de letras apropiado, con base en la calificación introducida.

Detalles acerca de la instrucción `switch`

La instrucción `switch` consiste en una serie de etiquetas `case` y un caso `default` opcional. Estas etiquetas se utilizan en este ejemplo para determinar qué contador incrementar, con base en una calificación. Cuando el flujo de control llega a la instrucción `switch`, el programa evalúa la expresión entre paréntesis (es decir, `calificación`) que está después de la palabra clave `switch` (línea 63). A ésta se le conoce como **expresión de control**. La instrucción `switch` compara el valor de la expresión de control con cada etiqueta `case`. Suponga que el usuario introduce la letra C como una calificación. El programa compara C con cada `case` en la instrucción `switch`. Si ocurre una coincidencia (`case 'C'` : en la línea 75), el programa ejecuta las instrucciones para esa etiqueta `case`. Para la letra C, en la línea 77 se incrementa `cCuenta` en 1. La instrucción `break` (línea 78) hace que el control del programa se reanude en la primera instrucción después del `switch`; en este programa, el control se transfiere a la línea 100. Esta línea marca el final del cuerpo del ciclo `while` que recibe las calificaciones (líneas 60 a 100), por lo que el control fluye hasta la condición del `while` (línea 60) para determinar si el ciclo debe continuar ejecutándose.

Las etiquetas `case` en nuestra instrucción `switch` evalúan explícitamente las versiones en minúscula y mayúscula de las letras A, B, C, D y F. Observe las etiquetas `case` en las líneas 65 y 66, que evalúan los valores 'A' y 'a' (ambos representan la calificación A). Al listar las etiquetas `case` de esta forma consecutiva, sin instrucciones entre ellas, pueden ejecutar el mismo conjunto de instrucciones; cuando la expresión de control se evalúe como 'A' o 'a', se ejecutarán las instrucciones de las líneas 67 y 68. Observe que cada etiqueta `case` puede tener varias instrucciones. La instrucción de selección `switch` difiere de otras instrucciones de control, en cuanto a que no requiere llaves alrededor de varias instrucciones en cada `case`.

Sin instrucciones `break`, cada vez que ocurra una concordancia en la instrucción `switch`, se ejecutarán las instrucciones para esa etiqueta `case` y todas las etiquetas `case` subsiguientes, hasta llegar a una instrucción `break` o al final de la instrucción `switch`. A menudo a esto se le conoce como que las etiquetas `case` "se pasarán" hacia las instrucciones en las etiquetas `case` subsiguientes. (Esta característica es perfecta para escribir un programa conciso, que muestre la canción iterativa "Los Doce Días de Navidad" en el ejercicio 5.28).



Error común de programación 5.8

Olvidar una instrucción `break` cuando se necesita una en una instrucción `switch` es un error lógico.



Error común de programación 5.9

Omitir el espacio entre la palabra `case` y el valor integral que se está evaluando en una instrucción `switch` puede producir un error lógico. Por ejemplo, si escribimos `case3:` en lugar de `case 3:`, simplemente se crea una etiqueta sin usar. En el apéndice E, Temas sobre código heredado de C, hablaremos más sobre esto. En esta situación, la instrucción `switch` no ejecutará las acciones apropiadas cuando su expresión de control tenga un valor de 3.

Proporcionar un caso `default`

Si no ocurre una coincidencia entre el valor de la expresión de control y una etiqueta `case`, se ejecuta el caso `default` (líneas 95 a 98). Utilizamos el caso `default` en este ejemplo para procesar todos los valores de la expresión de control que no sean calificaciones válidas o caracteres de nueva línea, tabuladores o espacios (en breve hablaremos acerca de cómo se encarga el programa de estos caracteres de espacio en blanco; esto es, todas las calificaciones de reprobado). Si no ocurre una coincidencia, se ejecuta un caso `default`, y las líneas 96-97 imprimen un mensaje de error indicando que se ha introducido una letra de calificación incorrecta. Si no ocurre una coincidencia y la instrucción `switch` no contiene un caso `default`, el control del programa simplemente continúa con la primera instrucción después de la instrucción `switch`.



Buena práctica de programación 5.10

Proporcione un caso `default` en las instrucciones `switch`. Los casos que no se evalúen en forma explícita en una instrucción `switch` sin un caso `default` deben ignorarse. Al incluir un caso `default`, nos enfocamos en la necesidad de procesar las condiciones excepcionales. Existen situaciones en las que no se necesita un procesamiento `default`. Aunque las cláusulas `case` y la cláusula del caso `default` en una instrucción `switch` pueden ocurrir en cualquier orden, es una práctica común colocar la cláusula `default` al último.



Buena práctica de programación 5.11

El último `case` en una instrucción `switch` no requiere una instrucción `break`. Algunos programadores incluyen este `break` por claridad y por simetría con otros casos.

Ignorar los caracteres de nueva línea, tabuladores y de espacio en blanco en la entrada

Observe que las líneas 90 a 93 en la instrucción `switch` de la figura 5.10 hacen que el programa omita los caracteres de nueva línea, tabuladores y espacios en blanco. Al leer los caracteres uno a la vez se pueden producir ciertos problemas. Para hacer que el programa lea los caracteres, debemos enviarlos a la computadora, oprimiendo la tecla *Intro* en el teclado. Con esto se coloca un carácter de nueva línea en la entrada, después del carácter que deseamos procesar. A menudo, este carácter de nueva línea se debe procesar de una manera especial para que el programa funcione correctamente. Al incluir las cláusulas `case` anteriores en nuestra instrucción `switch`, evitamos que se imprima el mensaje de error en el caso `default` cada vez que nos encontramos un carácter de nueva línea, tabulador o espacio.



Error común de programación 5.10

Si no se procesan los caracteres de nueva línea y demás caracteres de espacio en blanco en la entrada, al leer caracteres uno a la vez, se pueden producir errores lógicos.

Prueba de la clase LibroCalificaciones

En la figura 5.11 se crea un objeto `LibroCalificaciones` (línea 9). En la línea 11 se invoca la función miembro `mostrarMensaje` del objeto para imprimir en pantalla un mensaje de bienvenida para el usuario. En la línea 12 se invoca la función miembro `recibirCalificaciones` del objeto para leer un conjunto de calificaciones del usuario y llevar el registro de cuántos estudiantes recibieron cada calificación. Observe que la ventana de resultados en la figura 5.11 muestra un mensaje de error en respuesta a la acción de introducir una calificación inválida (es decir, E). En la línea 13 se invoca la función miembro `mostrarReporteCalificaciones` de `LibroCalificaciones` (definida en las líneas 104 a 114 de la figura 5.10), la cual imprime en pantalla un reporte con base en las calificaciones introducidas (como en los resultados de 5.11).

Diagrama de actividad de UML de la instrucción switch

La figura 5.12 muestra el diagrama de actividad de UML para la instrucción de selección múltiple `switch` general. La mayoría de las instrucciones `switch` utilizan una instrucción `break` en cada `case` para terminar la instrucción `switch` después de procesar el `case`. La figura 5.12 enfatiza esto al incluir instrucciones `break` en el diagrama de actividad.

```

1 // Fig. 5.11: fig05_11.cpp
2 // Crea un objeto LibroCalificaciones, recibe calificaciones y muestra el reporte de
   calificaciones.
3
4 #include "LibroCalificaciones.h" // incluye la definición de la clase LibroCalificaciones
5
6 int main()
7 {
8     // crea un objeto LibroCalificaciones
9     LibroCalificaciones miLibroCalificaciones( "CS101 Programacion en C++" );
10
11    miLibroCalificaciones.mostrarMensaje(); // muestra el mensaje de bienvenida
12    miLibroCalificaciones.recibirCalificaciones(); // lee las calificaciones del usuario
13    miLibroCalificaciones.mostrarReporteCalificaciones(); // muestra el reporte con base en
      las calificaciones

```

Figura 5.11 | Creación de un objeto `LibroCalificaciones` e invocación de sus funciones miembro. (Parte I de 2).

```

14     return 0; // indica que terminó correctamente
15 } // fin de main

```

Bienvenido al libro de calificaciones para
CS101 Programación en C++!

Escriba las calificaciones de letra.
Escriba el carácter EOF para terminar la entrada.

a
B
c
C
A
d
f
C
E
Se introdujo una letra de calificación incorrecta. Escribe una nueva calificación.
D
A
b
^Z

Número de estudiantes que recibieron cada calificación de letra:

A: 3
B: 2
C: 3
D: 2
F: 1

Figura 5.11 | Creación de un objeto LibroCalificaciones e invocación de sus funciones miembro. (Parte 2 de 2).

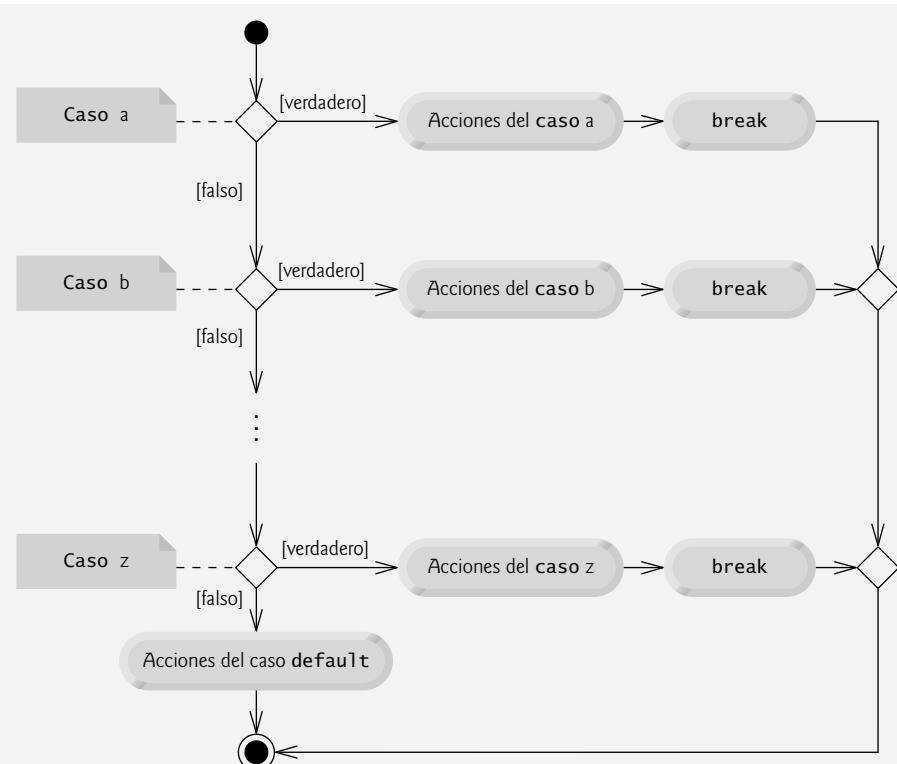


Figura 5.12 | Diagrama de actividad de UML de la instrucción `switch` de selección múltiple con instrucciones `break`.

Si la instrucción `break`, después de procesar un `case` el control no se transferiría a la primera instrucción después de la instrucción `switch`. En lugar de ello, el control se transferiría a las acciones del siguiente `case`.

El diagrama hace evidente que la instrucción `break` al final de un `case` hace que el control salga de la instrucción `switch` de inmediato. De nuevo, observe que (además de un estado inicial, flechas de transición, un estado final y varias notas) el diagrama contiene estados de acción y decisiones. Además, el diagrama utiliza símbolos de fusión para fusionar las transiciones de las instrucciones `break` hacia el estado final.

Imagine de nuevo que tiene un recipiente de diagramas de actividad de UML de instrucciones `switch` vacío; tantas instrucciones como podrían ser necesarias para apilarlas y anidarlas con los diagramas de actividades de otras instrucciones de control, para formar la implementación estructurada de un algoritmo. Usted completa los estados de acción y los símbolos de decisión con expresiones de acción y condiciones de guardia apropiadas para el algoritmo. Observe que, aun y cuando las instrucciones de control anidadas son comunes, es raro encontrar instrucciones `switch` anidadas en un programa.

Cuando utilice la instrucción `switch`, recuerde que cada `case` sólo se puede usar para evaluar una expresión integral *constante*: cualquier combinación de constantes carácter y enteras que se evalúen como un valor entero constante. Una constante carácter se representa como el carácter específico entre comillas sencillas, como '`A`'. Una constante entera es simplemente un valor entero. Además, cada etiqueta `case` puede especificar sólo una expresión integral constante.

Error común de programación 5.11

Especificar una expresión integral no constante en la etiqueta `case` de una instrucción `switch` es un error de sintaxis.

Error común de programación 5.12

Proporcionar etiquetas `case` idénticas en una instrucción `switch` es un error de compilación. Proporcionar etiquetas `case` que contengan distintas expresiones que se evalúen con el mismo valor también es un error de compilación. Por ejemplo, colocar `case 4 + 1:` y `case 3 + 2:` en la misma instrucción `switch` es un error de compilación, ya que ambas son equivalentes al `case 5:`.

En el capítulo 13 presentaremos una manera más elegante de implementar la lógica con `switch`. Utilizaremos una técnica llamada polimorfismo para crear programas que sean con frecuencia más claros, concisos, fáciles de mantener y de extender que los programas que utilizan la lógica de `switch`.

Observaciones sobre los tipos de datos

En C++, los tipos de datos tienen tamaños flexibles (vea el apéndice C, Tipos fundamentales). Por ejemplo, distintas aplicaciones podrían requerir enteros de distintos tamaños. C++ proporciona varios tipos de datos para representar enteros. El rango de valores enteros para cada tipo de datos depende del hardware específico de cada computadora. Además de los tipos `int` y `char`, C++ proporciona los tipos `short` (una abreviación de `short int`) y `long` (una abreviación de `long int`). El rango mínimo de valores para los enteros `short` es de -32,768 a 32,767. Para la amplia mayoría de los cálculos con enteros, basta con usar enteros `long`. El rango mínimo de valores para los enteros `long` es de -2,147,483,648 a 2,147,483,647. En la mayoría de las computadoras, los valores `int` son equivalentes a `short` o `long`. El rango de valores para un `int` es por lo menos el mismo que para los enteros `short`, y no más grande que para los enteros `long`. El tipo de datos `char` puede utilizarse para representar cualquiera de los caracteres en el conjunto de caracteres de la computadora. También se puede utilizar para representar enteros pequeños.

Tip de portabilidad 5.4

Como los valores `int` pueden variar en tamaño entre los sistemas, use enteros `long` si espera procesar enteros fuera del rango -32,768 a 32,767 y desea ejecutar el programa en varios sistemas computacionales distintos.

Tip de rendimiento 5.3

Si es imprescindible ahorrar memoria, podría ser conveniente usar tamaños de enteros más pequeños.

Tip de rendimiento 5.4

El uso de tamaños de enteros más pequeños puede provocar que el programa sea más lento, si las instrucciones de máquina para manipularlos no son tan eficientes como para los enteros de tamaño natural; es decir, enteros cuyo tamaño es igual al tamaño de palabra de la máquina (por ejemplo, 32 bits en una máquina de 32 bits, 64 bits en una máquina de 64 bits). Siempre hay que evaluar las "actualizaciones" de eficiencia que se estén proponiendo, para asegurar que realmente mejoren el rendimiento.

5.7 Instrucciones break y continue

Además de las instrucciones de selección y repetición, C++ proporciona las instrucciones **break** y **continue** para alterar el flujo de control. La sección anterior mostró cómo se puede utilizar **break** para terminar la ejecución de la instrucción **switch**. En esta sección veremos cómo usar **break** en una instrucción de repetición.

Instrucción break

Cuando la instrucción **break** se ejecuta en una instrucción **while**, **for**, **do...while**, o **switch**, ocasiona la salida inmediata de esa instrucción. La ejecución del programa continúa con la siguiente instrucción. Los usos comunes de la instrucción **break** son para escapar anticipadamente de ciclo, o para omitir el resto de una instrucción **switch** (como en la figura 5.10). La figura 5.13 demuestra el uso de una instrucción **break** (línea 14) para salir de un ciclo **for**.

Cuando la instrucción **if** determina que **conteo** es 5, se ejecuta la instrucción **break**. Esto termina la instrucción **for** y el programa continúa a la línea 19 (inmediatamente después de la instrucción **for**), la cual muestra un mensaje indicando el valor de la variable de control cuando terminó el ciclo. La instrucción **for** ejecuta su cuerpo por completo sólo cuatro veces en lugar de 10. Observe que la variable de control **cuenta** se define fuera del encabezado de la instrucción **for**, por lo que podemos usar la variable de control tanto en el cuerpo del ciclo como fuera de éste, una vez que complete su ejecución.

```

1 // Fig. 5.13: fig05_13.cpp
2 // instrucción break para salir de una instrucción for.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int cuenta; // la variable de control también se usa después de que termina el ciclo
10    for ( cuenta = 1; cuenta <= 10; cuenta++ ) // itera 10 veces
11    {
12        if ( cuenta == 5 ) // si cuenta es 5,
13            break;           // termina el ciclo
14
15        cout << cuenta << " ";
16    } // fin de for
17
18    cout << "\nSalio del ciclo en cuenta = " << cuenta << endl;
19    return 0; // indica que terminó correctamente
20
21 } // fin de main

```

```

1 2 3 4
Salio del ciclo en cuenta = 5

```

Figura 5.13 | Instrucción **break** para salir de una instrucción **for**.

Instrucción continue

Cuando la instrucción **continue** se ejecuta en una instrucción **while**, **for** o **do...while**, omite las instrucciones restantes en el cuerpo de esa instrucción y continúa con la siguiente iteración del ciclo. En las instrucciones **while** y **do...while**, la prueba de continuación de ciclo se evalúa justo después de que se ejecuta la instrucción **continue**. En una instrucción **for** se ejecuta la expresión de incremento y después el programa evalúa la prueba de continuación de ciclo.

La figura 5.14 utiliza la instrucción **continue** (línea 12) en una instrucción **for** para omitir la instrucción de salida (línea 14) cuando la instrucción **if** anidada (líneas 11 y 12) determina que el valor de **cuenta** es 5. Cuando se ejecuta la instrucción **continue**, el control del programa continúa con el incremento de la variable de control en el encabezado de la instrucción **for** (línea 9), e itera cinco veces más.

En la sección 5.3 dijimos que la instrucción **while** puede utilizarse, en la mayoría de los casos, para representar a la instrucción **for**. La única excepción ocurre cuando la expresión de incremento en la instrucción **while** va después de la instrucción **continue**. En este caso, el incremento no se ejecuta antes de que el programa evalúe la condición de continuación de ciclo, por lo que el **while** no se ejecuta de la misma manera que el **for**.



Buena práctica de programación 5.12

Algunos programadores sienten que las instrucciones `break` y `continue` violan la programación estructurada. Como pronto veremos, pueden lograrse los mismos efectos de estas instrucciones con las técnicas de programación estructurada, por lo que estos programadores prefieren no utilizar instrucciones `break` o `continue`. La mayoría de los programadores consideran aceptable el uso de `break` en las instrucciones `switch`.



Tip de rendimiento 5.5

Cuando las instrucciones `break` y `continue` se utilizan de manera apropiada, se ejecutan con más rapidez que las técnicas estructuradas correspondientes.



Observación de Ingeniería de Software 5.1

Existe una tensión entre lograr la ingeniería de software de calidad y lograr el software con mejor desempeño. A menudo, una de estas metas se logra a expensas de la otra. Para todas las situaciones excepto las que demanden el mayor rendimiento, aplique la siguiente regla empírica: primero, asegúrese de que su código sea simple y correcto; después hágalo rápido y pequeño, pero sólo si es necesario.

```

1 // Fig. 5.14: fig05_14.cpp
2 // instrucción continue para terminar una iteración de una instrucción for.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     for ( int cuenta = 1; cuenta <= 10; cuenta++ ) // itera 10 veces
10    {
11        if ( cuenta == 5 ) // si cuenta es 5,
12            continue;      // omite el código restante en el ciclo
13
14        cout << cuenta << " ";
15    } // fin de for
16
17    cout << "\nSe uso continue para no imprimir el 5" << endl;
18    return 0; // indica que terminó correctamente
19 } // fin de main

```

```

1 2 3 4 5 6 7 8 9 10
Se uso continue para omitir imprimir 5

```

Figura 5.14 | Instrucción `continue` para terminar una sola iteración de una instrucción `for`.

5.8 Operadores lógicos

Hasta ahora sólo hemos estudiado las **condiciones simples**, como `contador <= 10`, `total > 1000` y `numero != valorCentinela`. Expresamos esas condiciones en términos de los operadores relacionales `>`, `<`, `>=` y `<=`, y los operadores de igualdad `==` y `!=`. Cada decisión evaluó precisamente una condición. Para evaluar varias condiciones a la hora de tomar una decisión, realizamos estas pruebas en instrucciones separadas, o en instrucciones `if` o `if...else` anidadas.

C++ proporciona **operadores lógicos** que se utilizan para formar condiciones más complejas, al combinar condiciones simples. Los operadores lógicos son `&&` (AND lógico), `||` (OR lógico) y `!` (NOT lógico, también conocido como negación lógica).

Operador AND lógico (`&&`)

Suponga que deseamos asegurar en cierto punto de una aplicación que dos condiciones sean *ambas* verdaderas, antes de elegir cierta ruta de ejecución. En este caso, podemos utilizar el operador `&&` (AND lógico) de la siguiente manera:

```
if ( genero == 1 && edad >= 65 )
    mujeresMayores++;
```

Esta instrucción `if` contiene dos condiciones simples. La condición `genero == 1` se utiliza aquí para determinar si una persona es de género femenino. La condición `edad >= 65` determina si una persona es un ciudadano mayor. La condición simple a la izquierda del operador `&&` se evalúa primero. Si es necesario, la condición simple a la derecha del operador `&&` se evalúa a continuación. Como veremos en breve, el lado derecho de una expresión AND lógica se evalúa sólo si el lado izquierdo es verdadero. Después, la instrucción `if` considera la condición combinada

```
genero == 1 && edad >= 65
```

Esta condición es verdadera si y sólo si ambas condiciones simples son `true`. Por último, si esta condición combinada es evidentemente `true`, la instrucción en el cuerpo de la instrucción `if` incrementa la cuenta de `mujeresMayores`. Si alguna de esas condiciones simples es `false` (o ambas lo son), el programa omite el incremento y procede a la instrucción que va después del `if`. La condición combinada anterior puede hacerse más legible si se agregan paréntesis redundantes:

```
( genero == 1 ) && ( edad >= 65 )
```

Error común de programación 5.13



Aunque `3 < x < 7` es una condición matemáticamente correcta, no se evalúa como podría esperarse en C++. Use `(3 < x && x < 7)` para obtener la evaluación apropiada en C++.

En la figura 5.15 se sintetiza el operador `&&`. Esta tabla muestra las cuatro combinaciones posibles de valores `false` y `true` para `expresión1` y `expresión2`. A dichas tablas se les conoce comúnmente como **tablas de verdad**. C++ evalúa todas las expresiones que incluyen operadores relacionales, de igualdad y/o lógicos como `true` o `false`.

expresión1	expresión2	expresión1 && expresión2
<code>false</code>	<code>false</code>	<code>false</code>
<code>false</code>	<code>true</code>	<code>false</code>
<code>true</code>	<code>false</code>	<code>false</code>
<code>true</code>	<code>true</code>	<code>true</code>

Figura 5.15 | Tabla de verdad del operador `&&` (AND lógico).

Operador OR lógico (`||`)

Ahora considere el operador `||` (OR lógico). Suponga que deseamos asegurar en cierto punto en un programa, que de dos condiciones, una de ellas *o* ambas serán `true` antes de elegir cierta ruta de ejecución. En este caso, utilizamos el operador `||`, como en el siguiente segmento de un programa:

```
if ( ( promedioSemestre >= 90 ) || ( examenFinal >= 90 ) )
    cout << "La calificación del estudiante es A" << endl;
```

Esta instrucción también contiene dos condiciones simples. La condición simple `promedioSemestre >= 90` se evalúa para determinar si el estudiante merece una “A” en el curso, debido a que tuvo un sólido rendimiento a lo largo del semestre. La condición simple `examenFinal >= 90` se evalúa para determinar si el estudiante merece una “A” en el curso debido a un desempeño sobresaliente en el examen final. Después, la instrucción `if` considera la condición combinada

```
( promedioSemestre >= 90 ) || ( examenFinal >= 90 )
```

y otorga una “A” al estudiante si una o ambas de las condiciones simples son `true`. Observe que el mensaje “La calificación del estudiante es A” se imprime a menos que ambas condiciones simples sean `false`. La figura 5.16 es una tabla de verdad para el operador OR condicional (`||`).

El operador `&&` tiene mayor precedencia que el operador `||`. Ambos operadores se asocian de izquierda a derecha. Una expresión que contiene operadores `&&` o `||` se evalúa sólo si se conoce verdad o falsedad de la expresión. Por ende, la evaluación de la expresión

```
( genero == 1 ) && ( edad >= 65 )
```

expresión1	expresión2	expresión1 expresión2
false	false	false
false	true	true
true	false	true
true	true	true

Figura 5.16 | Tabla de verdad del operador || (OR lógico).

se detiene de inmediato si `genero` no es igual a 1 (es decir, en este punto toda la expresión es `false`) y continúa si `genero` es igual a 1 (es decir, toda la expresión podría ser aun `true` si la condición `edad >= 65` es `true`). Esta característica de rendimiento para la evaluación de las expresiones con AND lógico y OR lógico se conoce como **evaluación en corto circuito**.



Tip de rendimiento 5.6

En las expresiones que utilizan el operador `&&`, si las condiciones separadas son independientes una de otra, haga que la condición que tenga más probabilidad de ser `false` sea la condición de más a la izquierda. En expresiones que utilicen el operador `||`, haga que la condición que tenga más probabilidad de ser `true` sea la condición de más a la izquierda. Este uso en corto circuito puede reducir el tiempo de ejecución de un programa.

Operador lógico de negación (!)

C++ cuenta con el operador `!` (NOT lógico, también conocido como **negación lógica**) para que un programador pueda “invertir” el significado de una condición. A diferencia de los operadores binarios `&&` y `||`, que combinan dos condiciones, el operador lógico de negación unario sólo tiene una condición como operando. Este operador se coloca antes de una condición para elegir una ruta de ejecución si la condición original (sin el operador lógico de negación) es `false`, como en el siguiente segmento de un programa:

```
if ( ! ( calificacion == valorCentinela ) )
    cout << "La siguiente calificación es " << calificacion << endl;
```

Los paréntesis alrededor de la condición `calificacion == valorCentinela` son necesarios, ya que el operador lógico de negación tiene mayor precedencia que el operador de igualdad.

En la mayoría de los casos, puede evitar el uso de la negación lógica si expresa la condición con un operador relacional o de igualdad apropiado. Por ejemplo, la instrucción `if` anterior también puede escribirse de la siguiente manera:

```
if ( calificacion != valorCentinela )
    cout << "La siguiente calificación es " << calificacion << endl;
```

Esta flexibilidad puede ayudar a un programador a expresar una condición de una manera más “natural” o conveniente. La figura 5.17 es una tabla de verdad para el operador lógico de negación `!`.

expresión	!expresión
false	true
true	false

Figura 5.17 | Tabla de verdad del operador `!` (negación lógica).

Ejemplo de los operadores lógicos

La figura 5.18 demuestra el uso de los operadores lógicos; para ello produce sus tablas de verdad. Los resultados muestran cada expresión que se evalúa y su resultado boolean. De manera predeterminada, los valores `bool true` y `false` de las

expresiones se muestran mediante cout y el operador de inserción de flujo como 1 y 0, respectivamente. Utilizamos el manipulador de flujo `boolalpha` (un manipulador pegajoso) en la línea 11 para especificar que el valor de cada expresión `bool` se debe mostrar como la palabra “true” o la palabra “false”. Por ejemplo, el resultado de la expresión `false && false` en la línea 12 es `false`, así que la segunda línea de salida incluye la palabra “false”. Las líneas 11 a 15 producen la tabla de verdad para el `&&`. Las líneas 18 a 22 producen la tabla de verdad para el `||`. Las líneas 25 a 27 producen la tabla de verdad para el `!`.

```

1 // Fig. 5.18: fig05_18.cpp
2 // Operadores lógicos.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::boolalpha; // hace que los valores bool se impriman como "true" o "false"
7
8 int main()
9 {
10    // crea la tabla de verdad para el operador && (AND lógico)
11    cout << boolalpha << "AND logico (&&)"
12    << "\nfalse && false: " << ( false && false )
13    << "\nfalse && true: " << ( false && true )
14    << "\ntrue && false: " << ( true && false )
15    << "\ntrue && true: " << ( true && true ) << "\n\n";
16
17    // crea la tabla de verdad para el operador || (OR lógico)
18    cout << "OR logico (||)"
19    << "\nfalse || false: " << ( false || false )
20    << "\nfalse || true: " << ( false || true )
21    << "\ntrue || false: " << ( true || false )
22    << "\ntrue || true: " << ( true || true ) << "\n\n";
23
24    // crea la tabla de verdad para el operador ! (negación lógica)
25    cout << "NOT logico (!)"
26    << "\n!false: " << ( !false )
27    << "\n!true: " << ( !true ) << endl;
28    return 0; // indica que terminó correctamente
29 } // fin de main

```

```

AND logico (&&)
false && false: false
false && true: false
true && false: false
true && true: true

OR logico (||)
false || false: false
false || true: true
true || false: true
true || true: true

NOT logico (!)
!false: true
!true: false

```

Figura 5.18 | Operadores lógicos.

Resumen de precedencia y asociatividad de los operadores

La figura 5.19 agrega los operadores lógicos a la tabla de precedencia y asociatividad de los operadores. Los operadores se muestran de arriba hacia abajo, en orden descendente de precedencia.

Operadores	Asociatividad	Tipo
::	izquierda a derecha	resolución de ámbito
()	izquierda a derecha	paréntesis
++ -- static_cast< tipo >()	izquierda a derecha	unario (postfijo)
++ -- + - !	derecha a izquierda	unario (prefijo)
* / %	izquierda a derecha	multiplicativo
+ -	izquierda a derecha	aditivo
<< >>	izquierda a derecha	inserción/extracción
< <= > >=	izquierda a derecha	relacional
== !=	izquierda a derecha	igualdad
&&	izquierda a derecha	AND lógico
	izquierda a derecha	OR lógico
? :	derecha a izquierda	condicional
= += -= *= /= %=	derecha a izquierda	asignación
,	izquierda a derecha	coma

Figura 5.19 | Precedencia/asociatividad de los operadores.

5.9 Confusión entre los operadores de igualdad (==) y de asignación (=)

Hay un tipo de error que los programadores de C++, sin importar su experiencia, tienden a cometer con tanta frecuencia que creemos requiere una sección separada. Ese error es el de intercambiar accidentalmente los operadores == (igualdad) y = (asignación). Lo que hace a estos intercambios tan peligrosos es el hecho de que por lo general no producen errores sintácticos. En lugar de ello, las instrucciones con estos errores tienden a compilarse correctamente y el programa se ejecuta hasta completarse, generando a menudo resultados incorrectos a través de errores lógicos en tiempo de ejecución. [Nota: algunos compiladores generan una advertencia cuando se utiliza = en un contexto en el que normalmente se espera ==.]

Hay dos aspectos de C++ que contribuyen a estos problemas. Uno de ellos establece que cualquier expresión que produce un valor se puede utilizar en la porción correspondiente a la decisión de cualquier instrucción de control. Si el valor de la expresión es cero, se trata como `false`, y si el valor es distinto de cero, se trata como `true`. El segundo establece que las asignaciones producen un valor, a saber, el valor asignado a la variable del lado izquierdo del operador de asignación. Por ejemplo, suponga que tratamos de escribir

```
if ( codigoPago == 4 )
    cout << "Obtuvo un bono!" << endl;
```

pero accidentalmente escribimos

```
if ( codigoPago = 4 )
    cout << "Obtuvo un bono!" << endl;
```

La primera instrucción `if` otorga apropiadamente un bono a la persona cuyo `codigoPago` sea igual a 4. La segunda instrucción `if` (la del error) evalúa la expresión de asignación en la condición `if` a la constante 4. Cualquier valor distinto de cero se interpreta como `true`, por lo que la condición en esta instrucción `if` es siempre `true`, ¡y la persona siempre recibe un bono sin importar cuál sea el código de pago! Aún peor, ¡el código de pago se ha modificado, cuando se supone que sólo debía examinarse!

Error común de programación 5.14



El uso del operador == para asignación y el uso del operador = para igualdad son errores lógicos.



Tip para prevenir errores 5.3

Por lo general, los programadores escriben condiciones como `x == 7` con el nombre de la variable a la izquierda y la constante a la derecha. Al invertir éstos, de manera que la constante esté a la izquierda y el nombre de la variable a la derecha, como en `7 == x`, estará protegido por el compilador si accidentalmente sustituye el operador `==` con `=`. El compilador trata esto como un error de compilación, ya que no se puede modificar el valor de una constante. Esto evitara la potencial devastación de un error lógico en tiempo de ejecución.

Se dice que los nombres de las variables son *lvalues* (por “valores a la izquierda”), ya que pueden usarse del lado izquierdo de un operador de asignación. Se dice que las constantes son *rvalues* (por “valores a la derecha”), ya que sólo se pueden usar del lado derecho de un operador de asignación. Observe que los *lvalues* también se pueden usar como *rvalues*, pero no al revés.

Hay otra situación que es igual de incómoda. Suponga que desea asignar un valor a una variable con una instrucción simple, como:

```
x = 1;
```

pero en lugar de ello, escribe

```
x == 1;
```

Aquí podemos ver también que esto no es un error sintáctico. En lugar de ello, el compilador simplemente evalúa la expresión condicional. Si `x` es igual a 1, la condición es `true` y la expresión se evalúa con el valor `true`. Si `x` no es igual a 1, la condición es `false` y la expresión se evalúa con el valor `false`. Sin importar el valor de la expresión, no hay operador de asignación, por lo que el valor sólo se pierde. El valor de `x` permanece sin alteraciones, lo que probablemente produzca un error lógico en tiempo de ejecución. ¡Por desgracia, no tenemos un truco disponible para ayudarlo con este problema!



Tip para prevenir errores 5.4

Use su editor de texto para buscar todas las ocurrencias de `=` en su programa, y compruebe que tenga el operador de asignación u operador lógico correcto en cada lugar.

5.10 Resumen sobre programación estructurada

Así como los arquitectos diseñan edificios, empleando la sabiduría colectiva de su profesión, de igual forma los programadores deben diseñar programas. Nuestro campo es mucho más joven que la arquitectura, y nuestra sabiduría colectiva es mucho más escasa. Hemos aprendido que la programación estructurada produce programas que son más fáciles de entender, probar, depurar, modificar que los programas no estructurados, e incluso probar que son correctos en sentido matemático.

La figura 5.20 utiliza diagramas de actividad para sintetizar las instrucciones de control de C++. Los estados inicial y final indican el único punto de entrada y el único punto de salida de cada instrucción de control. Si conectamos los símbolos individuales de un diagrama de actividad en forma arbitraria, existe la posibilidad de que se produzcan programas no estructurados. Por lo tanto, la profesión de la programación utiliza sólo un conjunto limitado de instrucciones de control que pueden combinarse sólo de dos formas simples, para crear programas estructurados.

Por cuestión de simplicidad, sólo se utilizan instrucciones de control de una sola entrada/una sola salida; sólo hay una forma de entrar y una forma de salir de cada instrucción de control. Es sencillo conectar instrucciones de control en secuencia para formar programas estructurados: el estado final de una instrucción de control se conecta al estado inicial de la siguiente instrucción de control; es decir, las instrucciones de control se colocan una después de la otra en un programa. A esto le llamamos “apilamiento de instrucciones de control”. Las reglas para formar programas estructurados también permiten anidar las instrucciones de control.

La figura 5.21 muestra las reglas para formar programas estructurados. Las reglas suponen que pueden utilizarse estados de acción para indicar cualquier acción. Además, las reglas suponen que comenzamos con el diagrama de actividad más sencillo (figura 5.22), que consiste solamente de un estado inicial, un estado de acción, un estado final y flechas de transición.

Al aplicar las reglas de la figura 5.21, siempre se obtiene un diagrama de actividad con una agradable apariencia de bloque de construcción. Por ejemplo, si se aplica la regla 2 repetidamente al diagrama de actividad más sencillo, se obtiene un diagrama de actividad que contiene muchos estados de acción en secuencia (figura 5.23). La regla 2 genera una pila de estructuras de control, por lo que llamaremos a la regla 2 la **regla de apilamiento**. [Nota: las líneas punteadas verticales en la figura 5.23 no son parte de UML. Las utilizamos para separar los cuatro diagramas de actividad que demuestran cómo se aplica la regla 2 de la figura 5.21.]

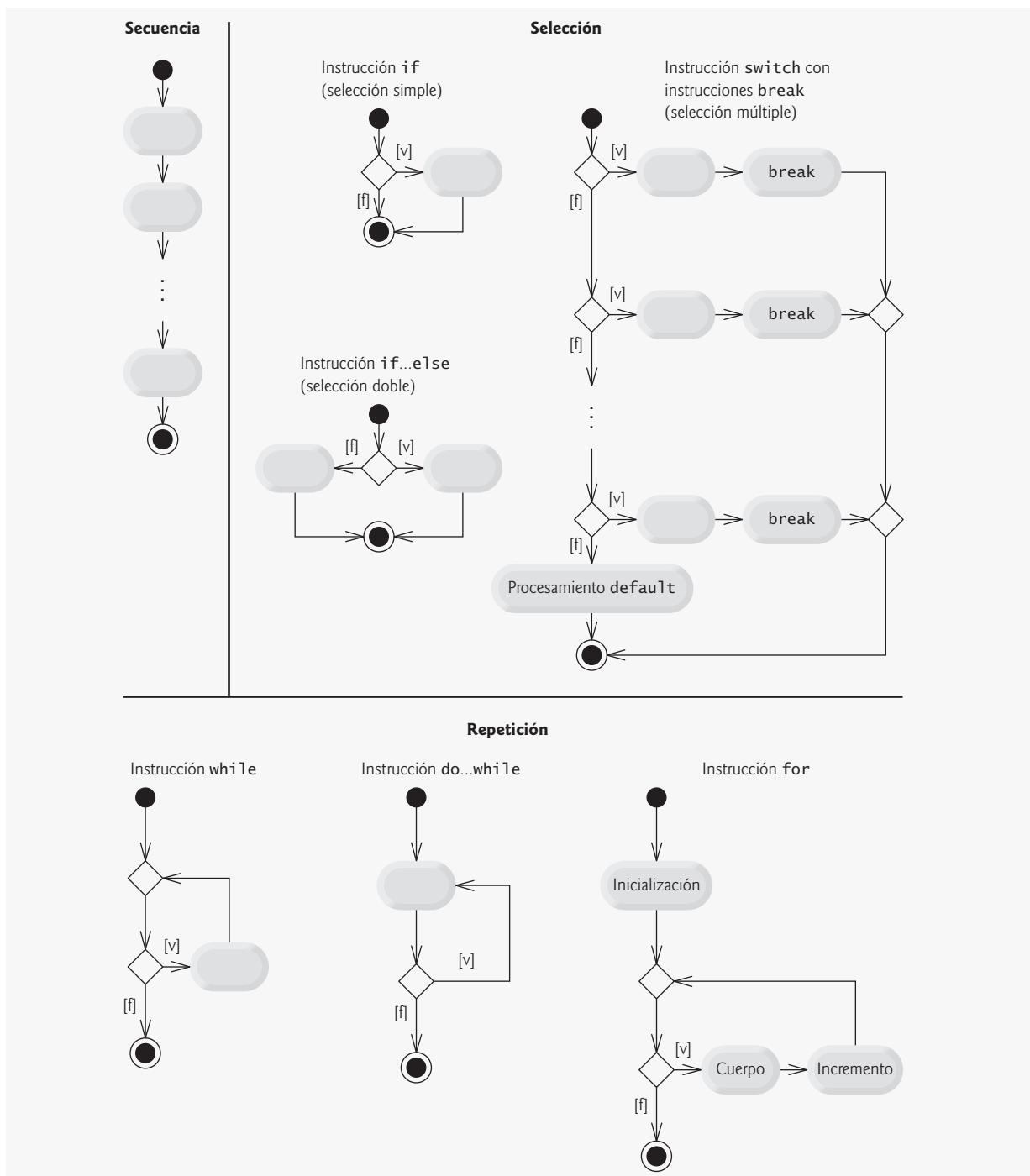


Figura 5.20 | Instrucciones de secuencia, selección y repetición de una sola entrada/una sola salida de C++.

La regla 3 se conoce como la **regla de anidamiento**. Al aplicar la regla 3 repetidamente al diagrama de actividad más sencillo, se obtiene un diagrama de actividad con instrucciones de control perfectamente anidadas. Por ejemplo, en la figura 5.24 el estado de acción en el diagrama de actividad más sencillo se reemplaza con una instrucción de selección doble (`if...else`). Luego la regla 3 se aplica otra vez a los estados de acción en la instrucción de selección doble, reemplazando cada uno de estos estados con una instrucción de selección doble. Los símbolos punteados de estado de acción alrededor de cada una de las instrucciones de selección doble, representan el estado de acción que se reemplazó en el diagrama de actividad anterior. [Nota: las flechas punteadas y los símbolos punteados de estado de acción que se muestran

Reglas para formar programas estructurados

- 1) Comenzar con el “diagrama de actividad más sencillo” (figura 5.22).
 - 2) Cualquier estado de acción puede reemplazarse por dos estados de acción en secuencia.
 - 3) Cualquier estado de acción puede reemplazarse por cualquier instrucción de control (secuencia, `if`, `if...else`, `switch`, `while`, `do...while` o `for`).
 - 4) Las reglas 2 y 3 pueden aplicarse tantas veces como se desee y en cualquier orden.

Figura 5.21 | Reglas para formar programas estructurados.

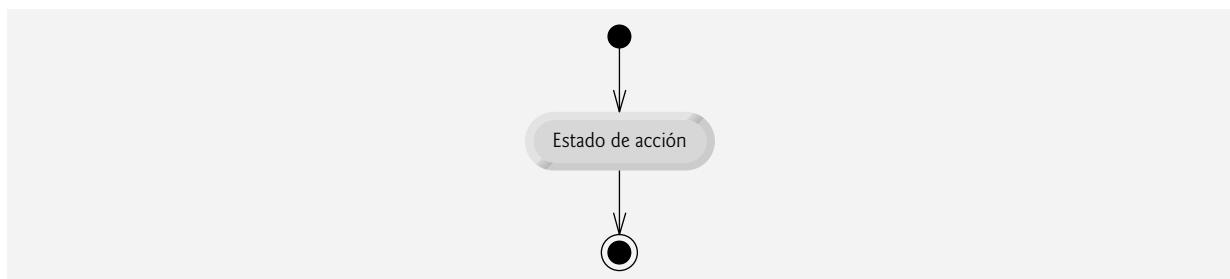


Figura 5.22 | El diagrama de actividad más sencillo.

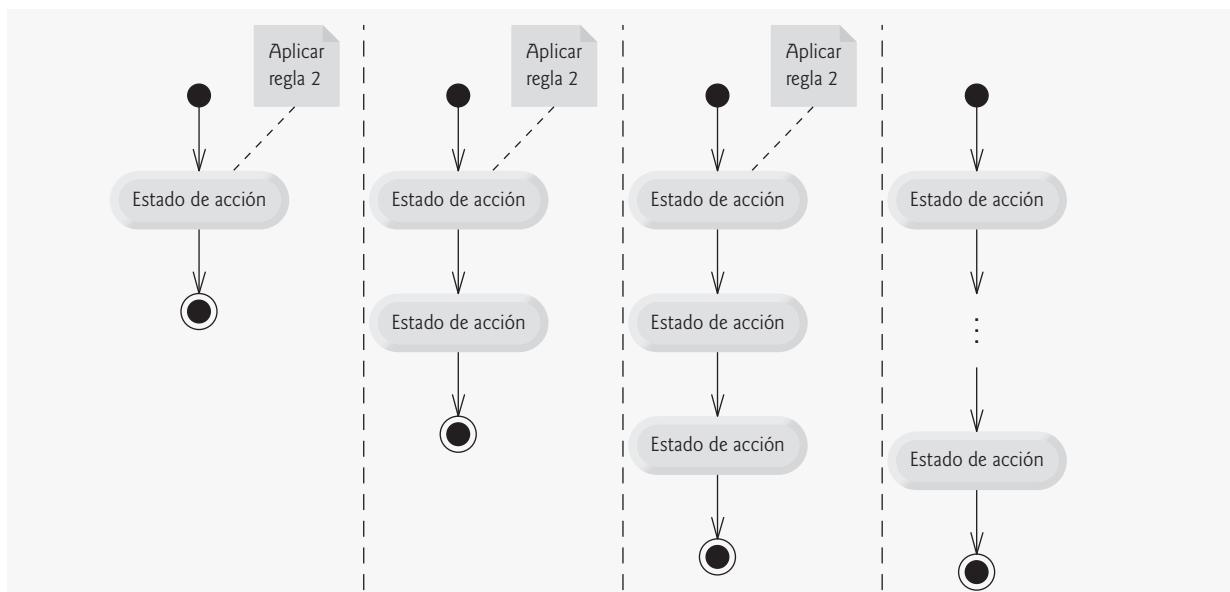


Figura 5.23 | El resultado de aplicar la regla 2 de la figura 5.21 repetidamente al diagrama de actividad más sencillo.

en la figura 5.24 no son parte de UML. Aquí se utilizan como dispositivos pedagógicos para ilustrar que cualquier estado de acción puede reemplazarse con una instrucción de control.]

La regla 4 genera instrucciones más grandes, más implicadas y más profundamente anidadas. Los diagramas que surgen debido a la aplicación de las reglas de la figura 5.21 constituyen el conjunto de todos los posibles diagramas de actividad estructurados y, por lo tanto, el conjunto de todos los posibles programas estructurados. La belleza de la metodología estructurada es que utilizamos sólo siete instrucciones de control simples de una sola entrada/una sola salida, y las ensamblamos en una de sólo dos formas simples.

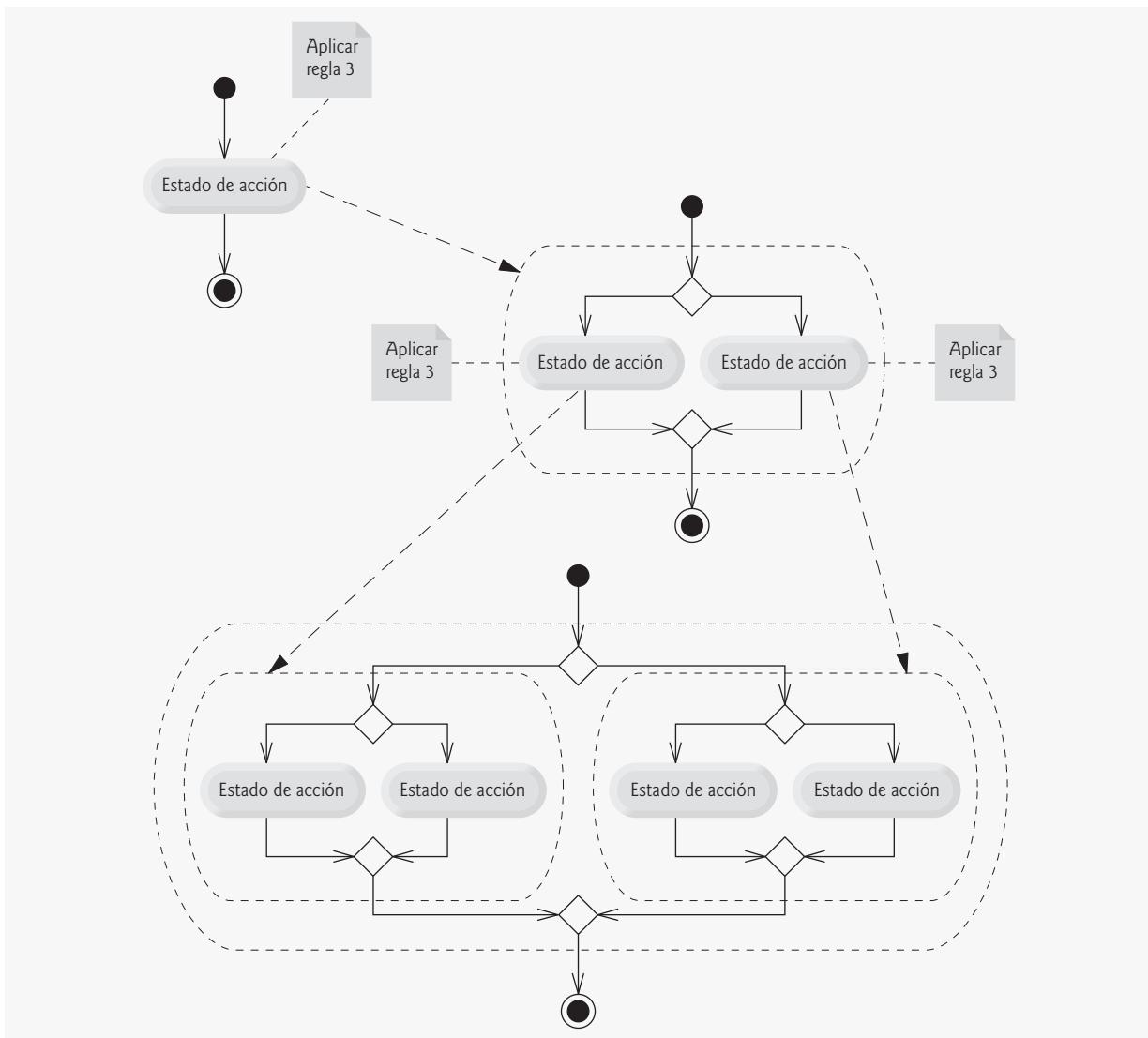


Figura 5.24 | Aplicación de la regla 3 de la figura 5.21 varias veces al diagrama de actividad más sencillo.

Si se siguen las reglas de la figura 5.21, no podrá crearse un diagrama de actividad con una sintaxis ilegal (como el de la figura 5.25). Si usted no está seguro de que cierto diagrama sea estructurado, aplique las reglas de la figura 5.21 en orden inverso para reducir el diagrama al diagrama de actividad más sencillo. Si puede reducirlo, entonces el diagrama original es estructurado; de lo contrario, no es estructurado.

La programación estructurada promueve la simpleza. Böhm y Jacopini nos han dado el resultado de que sólo se necesitan tres formas de control:

- Secuencia
- Selección
- Repetición

La estructura de secuencia es trivial. Simplemente enliste las instrucciones a ejecutar en el orden que deben ejecutarse.

La selección se implementa en una de tres formas:

- instrucción `if` (selección simple)
- instrucción `if...else` (selección doble)
- instrucción `switch` (selección múltiple)

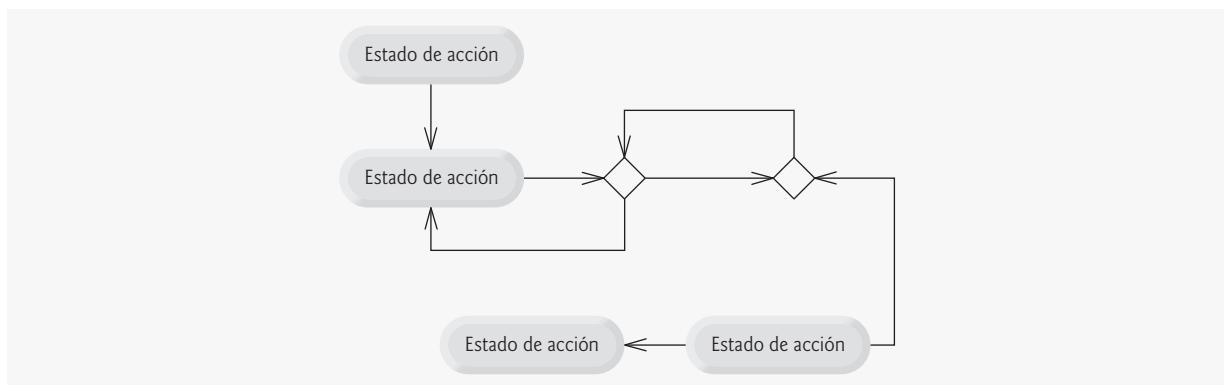


Figura 5.25 | Diagrama de actividad con sintaxis ilegal.

Es sencillo demostrar que la instrucción `if` más simple es suficiente para proporcionar cualquier forma de selección; todo lo que pueda hacerse con las instrucciones `if...else` y `switch` puede implementarse si se combinan instrucciones `if` (aunque tal vez no con tanta claridad y eficiencia).

La repetición se implementa en una de tres maneras:

- instrucción `while`
- instrucción `do...while`
- instrucción `for`

Es fácil demostrar que la instrucción `while` es suficiente para proporcionar cualquier forma de repetición. Todo lo que puede hacerse con las instrucciones `do...while` y `for`, puede hacerse también con la instrucción `while` (aunque tal vez no sea tan sencillo).

Si se combinan estos resultados, se demuestra que cualquier forma de control necesaria en un programa en C++ puede expresarse en términos de:

- secuencia
- instrucción `if` (selección)
- instrucción `while` (repetición)

y que estas tres instrucciones de control pueden combinarse en sólo dos formas: apilamiento y anidamiento. Evidentemente, la programación estructurada es la esencia de la simpleza.

5.11 (Opcional) Ejemplo práctico de Ingeniería de Software: cómo identificar los estados y actividades de los objetos en el sistema ATM

En la sección 4.13 identificamos muchos de los atributos de las clases necesarios para implementar el sistema ATM, y los agregamos al diagrama de clases de la figura 4.24. En esta sección le mostraremos la forma en que estos atributos representan el estado de un objeto. Identificaremos algunos estados clave que pueden ocupar nuestros objetos y hablaremos acerca de cómo cambian los objetos de estado, en respuesta a los diversos eventos que ocurren en el sistema. También hablaremos sobre el flujo de trabajo, o **actividades**, que realizan los objetos en el sistema ATM. En esta sección presentaremos las actividades de los objetos de transacción `SolicitudSaldo` y `Retiro`, ya que representan dos de las actividades clave en el sistema ATM.

Diagramas de máquina de estado

Cada objeto en un sistema pasa a través de una serie de estados discretos. El estado actual de un objeto se indica mediante los valores de los atributos del objeto en cualquier momento dado. Los **diagramas de máquina de estado** (que se conocen comúnmente como **diagramas de estado**) modelan los estados clave de un objeto y muestran bajo qué circunstancias el objeto cambia de estado. A diferencia de los diagramas de clases que presentamos en las secciones anteriores del ejemplo práctico, que se enfocaban principalmente en la estructura del sistema, los diagramas de estado modelan parte del comportamiento del sistema.

La figura 5.26 es un diagrama de estado simple, que modela algunos de los estados de un objeto de la clase ATM. UML representa a cada estado en un diagrama de estado como un **rectángulo redondeado** con el nombre del estado dentro de éste. Un **círculo relleno** con una punta de flecha designa el **estado inicial**. Recuerde que en el diagrama de clases de la figura 4.24 modelamos esta información de estado como el atributo Booleano de nombre `usuarioAutenticado`. Este atributo se inicializa en `false`, o en el estado “Usuario no autenticado”, de acuerdo con el diagrama de estado.

Las flechas indican las **transiciones** entre los estados. Un objeto puede pasar de un estado a otro, en respuesta a los diversos eventos que ocurren en el sistema. El nombre o la descripción del evento que ocasiona una transición se escribe cerca de la línea que corresponde a esa transición. Por ejemplo, el objeto ATM cambia del estado “Usuario no autenticado” al estado “Usuario autenticado”, una vez que la base de datos autentica al usuario. En la especificación de requerimientos vimos que para autenticar a un usuario, la base de datos compara el número de cuenta y el NIP introducidos por el usuario con los de la cuenta correspondiente en la base de datos. Si la base de datos indica que el usuario ha introducido un número de cuenta válido y el NIP correcto, el objeto ATM pasa al estado “Usuario autenticado” y cambia su atributo `usuarioAutenticado` al valor `true`. Cuando el usuario sale del sistema al seleccionar la opción “salir” del menú principal, el objeto ATM regresa al estado “Usuario no autenticado” y se prepara para el siguiente usuario del ATM.



Observación de Ingeniería de Software 5.2

Generalmente, los diseñadores de software no crean diagramas de estado que muestren todos los posibles estados y transiciones de estados para todos los atributos; simplemente hay demasiados. Lo común es que los diagramas de estado muestren sólo los estados y las transiciones de estado más importantes y complejos.

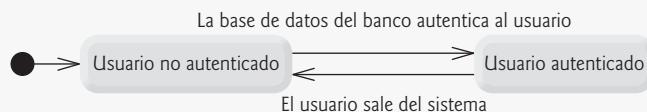


Figura 5.26 | Diagrama de estado para el objeto ATM.

Diagramas de actividad

Al igual que un diagrama de estado, un diagrama de actividad modela los aspectos del comportamiento de un sistema. A diferencia de un diagrama de estado, un diagrama de actividad modela el flujo de trabajo (secuencia de objetos) de un objeto durante la ejecución de un programa. Un diagrama de actividad modela las acciones a realizar y en qué orden las realizará el objeto. Recuerde que utilizamos diagramas de actividad de UML para ilustrar el flujo de control en las instrucciones de control que presentamos en el capítulo 4 y en este capítulo.

El diagrama de actividad de la figura 5.27 modela las acciones involucradas en la ejecución de una transacción `SolicitudSaldo`. Asumimos que ya se ha inicializado un objeto `SolicitudSaldo`, y que ya se le ha asignado un número de cuenta válido (el del usuario actual), por lo que el objeto sabe qué saldo obtener de la base de datos. El diagrama incluye las acciones que ocurren después de que el usuario selecciona la opción de solicitud de saldo del menú principal y antes de que el ATM devuelva al usuario al menú principal; un objeto `SolicitudSaldo` no realiza ni inicia estas acciones,

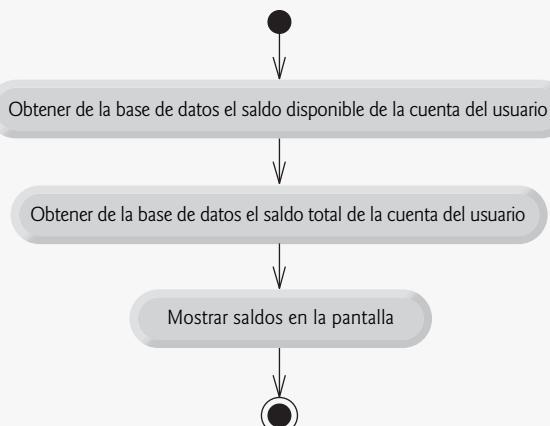


Figura 5.27 | Diagrama de actividad para un objeto `SolicitudSaldo`.

por lo que no las modelamos aquí. El diagrama empieza por obtener de la base de datos el saldo de la cuenta. Después, **SolicitudSaldo** obtiene el saldo total de la cuenta. Por último, la transacción muestra los saldos en la pantalla. Esta acción completa la ejecución de la transacción.

UML representa una acción en un diagrama de actividad como un estado de acción, el cual se modela mediante un rectángulo en el que sus lados izquierdo y derecho se sustituyen por arcos hacia afuera. Cada estado de acción contiene una expresión de acción, por ejemplo, “obtener de la base de datos el saldo de la cuenta del usuario”; eso especifica una acción a realizar. Una flecha conecta dos estados de acción, con lo cual indica el orden en el que ocurren las acciones representadas por los estados de acción. El círculo relleno (en la parte superior de la figura 5.27) representa el estado inicial de la actividad: el inicio del flujo de trabajo antes de que el objeto realice las acciones modeladas. En este caso, la transacción primero ejecuta la expresión de acción “obtener de la base de datos el saldo disponible de la cuenta de usuario”. Después, la transacción obtiene el saldo total. Por último, la transacción muestra ambos saldos en la pantalla. El círculo relleno encerrado en un círculo (en la parte inferior de la figura 5.27) representa el estado final: el fin del flujo de trabajo una vez que el objeto realiza las acciones modeladas.

La figura 5.28 muestra un diagrama de actividad para una transacción **Retiro**. Asumimos que ya se ha asignado un número de cuenta válido a un objeto **Retiro**. No modelaremos al usuario seleccionando la opción de retiro del menú principal ni al ATM devolviendo al usuario al menú principal, ya que estas acciones no las realiza un objeto **Retiro**. La transacción primero muestra un menú de montos estándar de retiro (que se muestra en la figura 2.17) y una opción para cancelar la transacción. Después la transacción recibe una selección del menú de parte del usuario. Ahora el flujo de actividad llega a un símbolo de decisión. Este punto determina la siguiente acción con base en las condiciones de guardia asociadas. Si el usuario cancela la transacción, el sistema muestra el mensaje apropiado. A continuación, el flujo de cancelación llega a un símbolo de fusión, donde este flujo de actividad se une a los otros posibles flujos de actividad de la transacción (que veremos en breve). Observe que una fusión puede tener cualquier cantidad de flechas de transición entrantes, pero sólo una flecha de transición saliente. La decisión en la parte inferior del diagrama determina si la transacción se debe repetir desde el principio. Cuando el usuario ha cancelado la transacción, la condición de guardia “se dispensó el efectivo o el usuario canceló la transacción” es verdadera, por lo que el control se pasa al estado final de la actividad.

Si el usuario selecciona un monto de retiro del menú, la transacción establece **monto** (un atributo de la clase **Retiro**, modelado originalmente en la figura 4.24) al valor elegido por el usuario. Después, la transacción obtiene el saldo disponible de la cuenta del usuario (es decir, el atributo **saldoDisponible** del objeto **Cuenta del usuario**) de la base de datos. Después el flujo de actividad llega a otra decisión. Si el monto de retiro solicitado excede al saldo disponible del usuario, el sistema muestra un mensaje de error apropiado, en el cual informa al usuario sobre el problema. Después el control se fusiona con los demás flujos de actividad antes de llegar a la decisión en la parte inferior del diagrama. La condición de guardia “no se dispensó el efectivo y el usuario no canceló” es verdadera, por lo que el diagrama de actividad regresa a la parte superior del diagrama, y la transacción pide al usuario que introduzca un nuevo monto.

Si el monto de retiro solicitado es menor o igual al saldo disponible del usuario, la transacción evalúa si el dispensador de efectivo tiene suficiente efectivo para satisfacer la solicitud de retiro. Si éste no es el caso, la transacción muestra un mensaje de error apropiado y pasa a través de la fusión, antes de llegar a la decisión final. No se dispensó el efectivo, por lo que el flujo de actividad regresa al inicio del diagrama de actividad, y la transacción pide al usuario que seleccione un nuevo monto. Si hay suficiente efectivo disponible, la transacción interactúa con la base de datos para cargar el monto retirado de la cuenta del usuario (es decir, restar el monto tanto del atributo **saldoDisponible** como del atributo **saldoTotal** del objeto **Cuenta del usuario**). Después la transacción entrega el monto deseado de efectivo e instruye al usuario para que lo tome. El flujo principal de actividad se fusiona con los dos flujos de error y con el flujo de de cancelación. En este caso se dispensó el efectivo, por lo que el flujo de actividad llega al estado final.

Hemos llevado a cabo los primeros pasos para modelar el comportamiento del sistema ATM y hemos mostrado cómo participan los atributos de un objeto para realizar las actividades del mismo. En la sección 6.22 investigaremos las operaciones de nuestras clases para crear un modelo más completo del comportamiento del sistema.

Ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

5.1 Indique si el siguiente enunciado es *verdadero* o *falso* y, si es *falso*, explique por qué: los diagramas de estado modelan los aspectos estructurales de un sistema.

5.2 Un diagrama de actividad modela las(los) _____ que realiza un objeto y el orden en el que las(los) realiza.

- a) acciones.
- b) atributos.
- c) estados.
- d) transiciones de estado.

5.3 Con base en la especificación de requerimientos, cree un diagrama de actividad para una transacción de depósito.

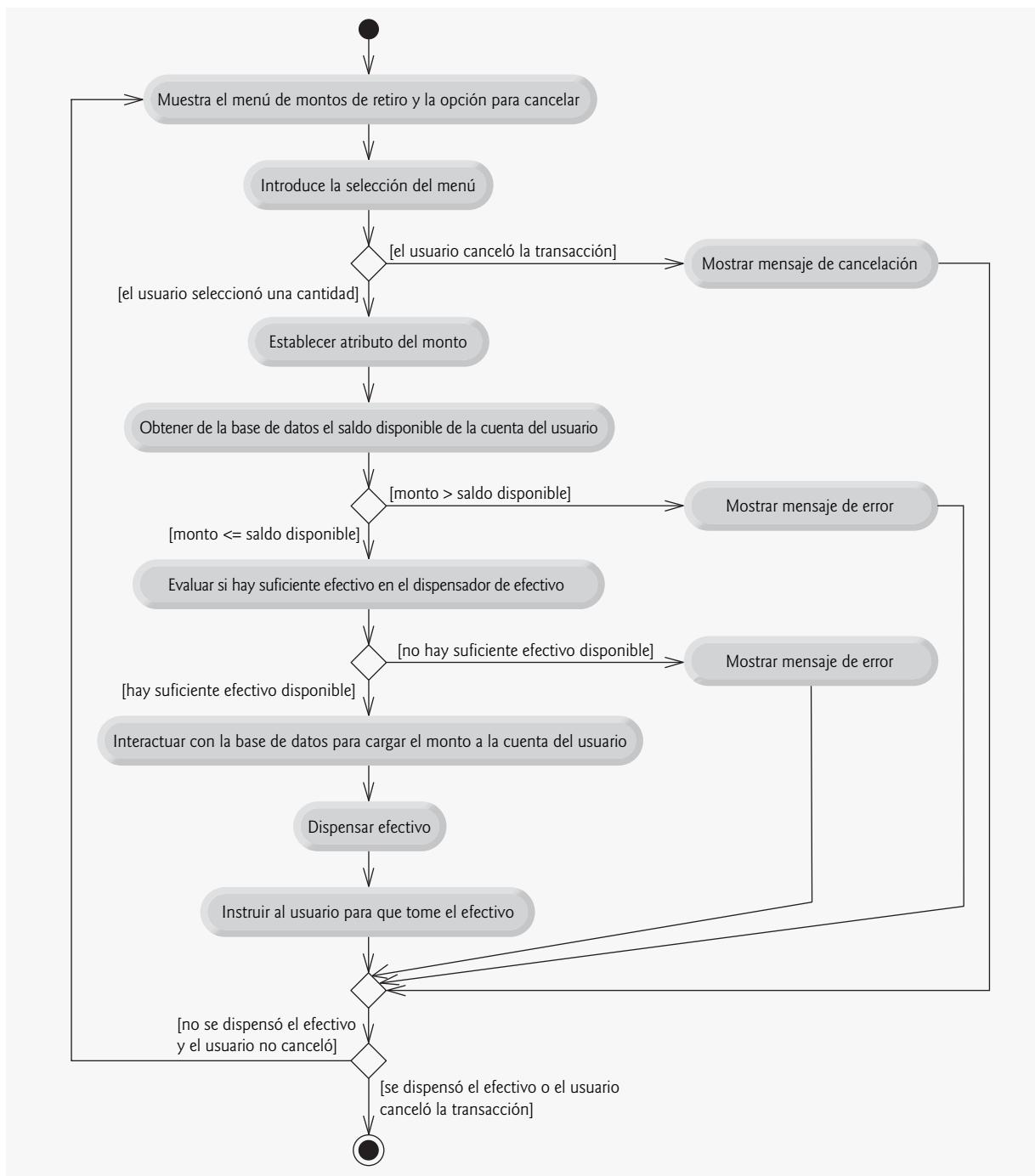


Figura 5.28 | Diagrama de actividad para una transacción **Retiro**.

Respuestas a los ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

5.1 Falso. Los diagramas de estado modelan parte del comportamiento del sistema.

5.2 a.

5.3 La figura 5.29 presenta un diagrama de actividad para una transacción de depósito. El diagrama modela las acciones que ocurren una vez que el usuario selecciona la opción de depósito del menú principal, y antes de que el ATM regrese al usuario al menú principal. Recuerde que una parte del proceso de recibir un monto de depósito de parte del usuario implica

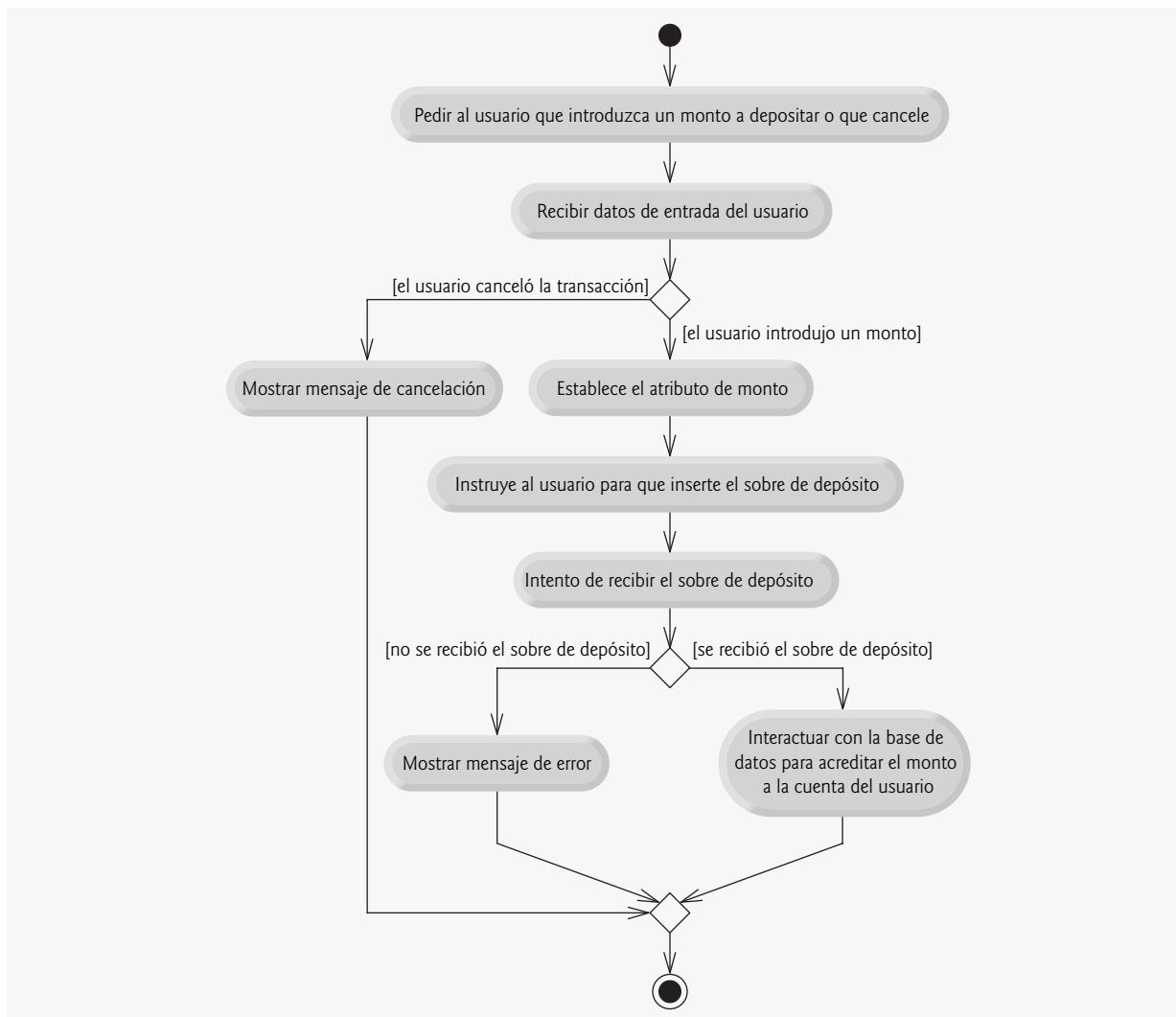


Figura 5.29 | Diagrama de actividad para una transacción Deposito.

convertir un número entero de centavos a una cantidad en dólares. Recuerde también que para acreditar un monto de depósito a una cuenta sólo hay que incrementar el atributo `saldoTotal` del objeto `Cuenta` del usuario. El banco actualiza el atributo `saldoDisponible` del objeto `Cuenta` del usuario sólo después de confirmar el monto de efectivo en el sobre de depósito y después de verificar los cheques que haya incluido; esto ocurre en forma independiente del sistema ATM.

5.12 Repaso

En este capítulo completamos nuestra introducción a las instrucciones de control de C++, las cuales nos permiten controlar el flujo de la ejecución en las funciones. El capítulo 4 trató acerca de las instrucciones de control `if`, `if...else` y `while`. En este capítulo vimos el resto de las instrucciones de control de C++: `for`, `do...while` y `switch`. Hemos demostrado que cualquier algoritmo puede desarrollarse mediante el uso de combinaciones de instrucciones de secuencia (es decir, instrucciones que se enlistan en el orden en el que deben ejecutarse), los tres tipos de instrucciones de selección (`if`, `if...else` y `switch`) y los tres tipos de instrucciones de repetición (`while`, `do...while` y `for`). En este capítulo y en el anterior hablamos acerca de cómo puede combinar estos bloques de construcción para utilizar las técnicas, ya probadas, de construcción de programas y solución de problemas. En este capítulo también se introdujeron los operadores lógicos de C++, que nos permiten utilizar expresiones condicionales más complejas en las instrucciones de control. Por último, examinamos los errores comunes al confundir los operadores de igualdad y asignación, y proporcionamos sugerencias para evitar estos errores.

En el capítulo 3 presentamos la programación en C++ con los conceptos básicos de los objetos, las clases y las funciones miembro. En este capítulo y en el anterior se introdujeron con detalle las instrucciones de control que podemos utilizar comúnmente para especificar la lógica de los programas en las funciones. En el capítulo 6 examinaremos las funciones con más detalle.

Resumen

Sección 5.2 Fundamentos de la repetición controlada por contador

- En C++, es más preciso llamar a una declaración que también reserva memoria.
- La instrucción de repetición **for** maneja todos los detalles relacionados con la repetición controlada por un contador. El formato general de la instrucción **for** es

```
for ( inicialización; condiciónDeContinuacionDeCiclo; incremento )
    instrucción
```

donde la expresión *inicialización* inicializa la variable de control del ciclo, *condiciónDeContinuaciónDeCiclo* es la condición que determina si el ciclo debe continuar su ejecución, e *incremento* incrementa la variable de control.

Sección 5.3 Instrucción de repetición **for**

- Por lo general, las instrucciones **for** se utilizan para la repetición controlada por contador y las instrucciones **while** para la repetición controlada por centinela.
- El alcance de una variable específica en qué parte de un programa se puede utilizar. Por ejemplo, una variable de control declarada en el encabezado de una instrucción **for** se puede utilizar sólo en el cuerpo de la instrucción **for**; la variable de control se desconocerá fuera de la instrucción **for**.
- Las expresiones de inicialización e incremento en el encabezado de una instrucción **for** pueden ser listas de expresiones separadas por comas. El uso que se da a las comas en estas expresiones es como operadores, lo cual garantiza que las listas de expresiones se evalúen de izquierda a derecha. El operador coma tiene la menor precedencia de todos los operadores de C++. El valor y tipo de una lista de expresiones separadas por comas es el valor y tipo de la expresión que esté más a la derecha en la lista.
- Las expresiones de inicialización, condición de continuación de ciclo e incremento de una instrucción **for** pueden contener expresiones aritméticas. Además, el incremento de una instrucción **for** puede ser negativo, en cuyo caso es realmente un decremento, y el ciclo cuenta en orden descendente.
- Si en un principio la condición de continuación de ciclo en un encabezado **for** es **false**, no se ejecuta el cuerpo de la instrucción **for**. En lugar de ello, la ejecución se reanuda en la instrucción después del **for**.

Sección 5.4 Ejemplos acerca del uso de la instrucción **for**

- La función **pow(x, y)** de la biblioteca estándar calcula el valor de **x** elevado a la **y**-ésima potencia. La función **pow** recibe dos argumentos de tipo **double** y devuelve un valor **double**.
- El manipulador de flujo parametrizado **setw** especifica la anchura de campo en la que debe aparecer el siguiente valor a imprimir. El valor se justifica a la derecha en el campo de manera predeterminada. Si el valor a imprimir es mayor que la anchura de campo, ésta se extiende para dar cabida al valor completo. El manipulador de flujo no parametrizado **left** (que se encuentra en el encabezado **<iostream>**) se puede utilizar para hacer que un valor se justifique a la izquierda en un campo, y **right** se puede usar para restaurar la justificación a la derecha.
- Las opciones pegajosas son las opciones de formato que permanecen en vigor hasta que se modifican.

Sección 5.5 Instrucción de repetición **do...while**

- La instrucción de repetición **do...while** evalúa la condición de ciclo al final del ciclo, por lo que el cuerpo del ciclo se ejecutará por lo menos una vez. El formato para la instrucción **while** es

```
do
{
    instrucción
} while ( condición );
```

Sección 5.6 Instrucción de selección múltiple **switch**

- La instrucción **switch** de selección múltiple realiza distintas acciones, con base en los posibles valores de una variable o expresión. Cada acción se asocia con el valor de una expresión integral constante (es decir, cualquier combinación de cons-

tantes tipo carácter y constantes enteras, que se evalúe como un valor constante entero) que la variable o expresión en la que se basa el `switch` puede asumir.

- La instrucción `switch` consiste de una serie de etiquetas `case` y un caso `default` opcional.
- La función `cin.get()` lee un carácter del teclado. Por lo general, los caracteres se almacenan en variables de tipo `char`; sin embargo, los caracteres se pueden almacenar en cualquier tipo de datos entero, ya que se garantiza que los tipos `short`, `int` y `long` son por lo menos tan grandes como el tipo `char`. Así, un carácter se puede tratar ya sea como entero o como carácter, dependiendo de su uso.
- El indicador de fin de archivo es una combinación de teclas dependiente del sistema, la cual indica que no hay más datos qué introducir. `EOF` es una constante entera simbólica, definida en el archivo de encabezado `<iostream>`, que indica “fin de archivo”.
- La expresión entre paréntesis después de la palabra clave `switch` se llama expresión de control del `switch`. La instrucción `switch` compara el valor de la expresión de control con cada etiqueta `case`.
- Enlistar las etiquetas `case` en forma consecutiva, sin instrucciones entre ellas, les permite ejecutar el mismo conjunto de instrucciones.
- Cada `case` puede tener varias instrucciones. La instrucción de selección `switch` difiere de otras instrucciones de control, en cuanto a que no requiere llaves alrededor de varias instrucciones en cada `case`.
- Cada `case` se puede utilizar sólo para evaluar una expresión integral constante. Una constante tipo carácter se representa como el carácter específico entre comillas sencillas, como '`A`'. Una constante entera es tan sólo un valor entero. Además, cada etiqueta `case` sólo puede especificar una expresión integral constante.
- C++ proporciona varios tipos de datos para representar enteros: `int`, `char`, `short` y `long`. El rango de valores enteros para cada tipo depende del hardware específico de cada computadora.

Sección 5.7 Instrucciones `break` y `continue`

- Cuando la instrucción `break` se ejecuta en una de las instrucciones de repetición (`for`, `while` y `do...while`), provoca la salida inmediata de esa instrucción.
- Cuando la instrucción `continue` se ejecuta en una de las instrucciones de repetición (`for`, `while` y `do...while`), omite el resto de las instrucciones en el cuerpo de la instrucción de repetición y continúa con la siguiente iteración del ciclo. En una instrucción `while` o `do...while`, la ejecución continúa con la siguiente evaluación de la condición. En una instrucción `for`, la ejecución continúa con la expresión de incremento en el encabezado de la instrucción `for`.

Sección 5.8 Operadores lógicos

- Los operadores lógicos nos permiten formar condiciones más complejas, mediante la combinación de condiciones simples. Los operadores lógicos son `&&` (AND lógico), `||` (OR lógico) y `!` (negación lógica).
- El operador `&&` (AND lógico) asegura que de dos condiciones, *ambas* sean `true` antes de elegir cierta ruta de ejecución.
- El operador `||` (OR lógico) asegura que de dos condiciones, *una o ambas* sean `true` antes de elegir cierta ruta de ejecución.
- Una expresión que contenga los operadores `&&` o `||` se evalúa sólo hasta que se conoce si la condición es verdadera o falsa. Esta característica de rendimiento para la evaluación de las expresiones AND lógico y OR lógico se conoce como evaluación de corto circuito.
- El operador `!` (NOT lógico, también conocido como negación lógica) permite a un programador “invertir” el significado de una condición. El operador de negación lógico unario se coloca antes de una condición, para elegir una ruta de ejecución si la condición original (sin el operador de negación lógico) es `false`. En la mayoría de los casos, podemos evitar el uso de la negación lógica si expresamos la condición con un operador relacional o de igualdad apropiado.
- Cuando se utiliza en una condición, cualquier valor distinto de cero se convierte de manera implícita en `true`; 0 (cero) se convierte de manera implícita en `false`.
- De manera predeterminada, `cout` muestra a los valores `bool` `true` y `false` como 1 y 0, respectivamente. El manipulador de flujo `boolalpha` (un manipulador pegajoso) especifica que el valor de cada expresión `bool` debe mostrarse, ya sea como la palabra “`true`” o la palabra “`false`”.

Sección 5.9 Confusión entre los operadores de igualdad (`==`) y de asignación (`=`)

- Cualquier expresión que produzca un valor se puede utilizar en la porción de decisión de cualquier instrucción de control. Si el valor de la expresión es cero, se trata como `false`, y si es distinto de cero, se trata como `true`.
- Una asignación produce un valor; a saber, el valor asignado a la variable del lado izquierdo del operador de asignación.

Sección 5.10 Resumen de programación estructurada

- Cualquier forma de control que se llegue a necesitar en un programa de C++, se puede expresar en términos de instrucciones de secuencia, selección y repetición, y éstas pueden combinarse sólo en dos formas: apilamiento y anidamiento.

Terminología

!, operador NOT lógico	incrementar una variable de control
&&, operador AND lógico	justificar a la derecha
, operador OR lógico	justificar a la izquierda
alcance de una variable	left, manipulador de flujo
anchura de campo	lvalue (“valor a la izquierda”)
AND lógico (&&)	negación lógica (!)
ASCII, conjunto de caracteres	nombre de una variable de control
boolalpha, manipulador de flujo	NOT lógico (!)
break, instrucción	opción pegajosa
case, etiqueta	operador coma
char, tipo fundamental	operador lógico
condición de continuación de ciclo	OR lógico ()
condición simple	pow, función de la biblioteca estándar
conteo con base cero	regla de anidamiento
decrementar una variable de control	regla de apilamiento
default, caso en una instrucción switch	right, manipulador de flujo
definición	rvalue (“valor a la derecha”)
error por desplazamiento en 1	setw, manipulador de flujo
evaluación de corto circuito	switch, instrucción de selección múltiple
expresión de control de una instrucción switch	tabla de verdad
expresión integral constante	valor final de una variable de control
for, encabezado	valor inicial de una variable de control
for, instrucción de repetición	

Ejercicios de autoevaluación

- 5.1 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
- El caso `default` es requerido en la instrucción de selección `switch`.
 - La instrucción `break` es requerida en el caso `default` de una instrucción de selección `switch`, para salir del `switch` de manera apropiada.
 - La expresión (`x > y && a < b`) es `true` si la expresión `x > y` es `true`, o si la expresión `a < b` es `true`.
 - Una expresión que contiene el operador `||` es `true` si uno o ambos de sus operandos son `true`.
- 5.2 Escriba una instrucción o un conjunto de instrucciones en C++, para realizar cada una de las siguientes tareas:
- Sumar los enteros impares entre 1 y 99, utilizando una instrucción `for`. Suponga que se han declarado las variables enteras `suma` y `cuenta`.
 - Imprimir el valor 333.546372 en una anchura de campo de 15 caracteres, con precisiones de 1, 2 y 3. Imprimir cada número en la misma línea. Justificar a la izquierda cada número en su campo. ¿Cuáles son los tres valores que se imprimen?
 - Calcular el valor de 2.5 elevado a la potencia de 3, utilizando la función `pow`. Imprimir el resultado con una precisión de 2 en una anchura de campo de 10 posiciones. ¿Qué se imprime?
 - Imprimir los enteros del 1 al 20, utilizando un ciclo `while` y la variable contador `x`. Suponga que la variable `x` se ha declarado, pero no se ha inicializado. Imprimir solamente cinco enteros por línea. [Sugerencia: use el cálculo `x % 5`. Cuando el valor de esta expresión sea 0, imprima un carácter de nueva línea; de lo contrario, imprima un carácter de tabulación.]
 - Repita el ejercicio 5.2 (d), usando una instrucción `for`.
- 5.3 Encuentre el(s) error(es), si los hay, en cada uno de los siguientes segmentos de código, y explique cómo corregirlo(s):
- ```
x = 1;
while (x <= 10);
 x++;
}
```
  - ```
for ( y = .1; y != 1.0; y += .1 )
      cout << y << endl;
```

```

c) switch ( n )
{
    case 1:
        cout << "El numero es 1" << endl;
    case 2:
        cout << "El numero es 2" << endl;
        break;
    default:
        cout << "El número no es 1 ni 2" << endl;
        break;
}
d) El siguiente código debe imprimir los valores 1 a 10:
n = 1;
while ( n < 10 )
    cout << n++ << endl;

```

Respuestas a los ejercicios de autoevaluación

- 5.1 a) Falso. El caso `default` es opcional. Sin embargo, se considera buena ingeniería de software proporcionar siempre un caso `default`.
- b) Falso. La instrucción `break` se utiliza para salir de la instrucción `switch`. La instrucción `break` no se requiere cuando el caso `default` es el último. Tampoco se requerirá la instrucción `break` si hacer que el control se reanude en el siguiente caso tiene sentido.
- c) Falso. Al utilizar el operador `&&`, ambas expresiones relacionales deben ser `true` para que toda la expresión sea `true`.
- d) Verdadero.

- 5.2 a) `suma = 0;`
`for (cuenta = 1; cuenta <= 99; cuenta += 2)`
 `suma += cuenta;`
- b) `cout << fixed << left`
 `<< setprecision(1) << setw(15) << 333.546372`
 `<< setprecision(2) << setw(15) << 333.546372`
 `<< setprecision(3) << setw(15) << 333.546372`
 `<< endl;`

La salida es:

333.5 333.55 333.546

- c) `cout << fixed << setprecision(2)`
 `<< setw(10) << pow(2.5, 3)`
 `<< endl;`

La salida es:

15.63

- d) `x = 1;`

```

while ( x <= 20 )
{
    cout << x;

    if ( x % 5 == 0 )
        cout << endl;
    else
        cout << '\t';

    x++;
}
```

```

e) for ( x = 1; x <= 20; x++ )
{
    cout << x;
    if ( x % 5 == 0 )
        cout << endl;
    else
        cout << '\t';
}

o

for ( x = 1; x <= 20; x++ )
{
    if ( x % 5 == 0 )
        cout << x << endl;
    else
        cout << x << '\t';
}

```

- 5.3 a) *Error:* el punto y coma después del encabezado `while` provoca un ciclo infinito.

Corrección: reemplazar el punto y coma por una llave izquierda ({), o eliminar tanto el punto y coma (;) como la llave derecha (}).

- b) *Error:* utilizar un número de punto flotante para controlar una instrucción de repetición `for`.

Corrección: utilice un entero, y realice el cálculo apropiado para poder obtener los valores deseados:

```

for ( y = 1; y != 10, y++ )
    cout << ( static_cast< double >( y ) / 10 ) << endl;

```

- c) *Error:* falta una instrucción `break` en el primer `case`.

Corrección: agregue una instrucción `break` al final de las instrucciones para el primer `case`. Observe que esto no es un error, si el programador desea que la instrucción del `case 2:` se ejecute siempre que lo haga la instrucción del `case 1::`

- d) *Error:* se está utilizando un operador relacional inadecuado en la condición de continuación de la instrucción de repetición `while`.

Corrección: use `<=` en lugar de `<`, o cambie 10 a 11.

Ejercicios

- 5.4 Encuentre el(los) error(es), si los hay, en cada uno de los siguientes fragmentos de código:

a) `For (x = 100, x >= 1, x++)`
`cout << x << endl;`

b) El siguiente código debe imprimirse sin importar que el entero `valor` sea par o impar:

```

switch ( valor % 2 )
{
    case 0:
        cout << "Entero par" << endl;
    case 1:
        cout << "Entero impar" << endl;
}

```

c) El siguiente código debe imprimir los enteros impares del 19 al 1:

```

for ( x = 19; x >= 1; x += 2 )
    cout << x << endl;

```

d) El siguiente código debe imprimir los enteros pares del 2 al 100:

```

contador = 2;

do
{
    cout << contador << endl;
    contador += 2;
}

```

```

    contador += 2;
} While ( contador < 100 );

```

- 5.5** Escriba un programa que utilice una instrucción **for** para sumar una secuencia de enteros. Suponga que el primer entero leído especifica el número de valores que quedan por introducir. Su programa debe leer sólo un valor por cada instrucción de entrada. Una secuencia típica de entrada podría ser

5 100 200 300 400 500

donde el 5 indica que se van a sumar los 5 valores subsiguientes.

- 5.6** Escriba un programa que utilice una instrucción **for** para calcular e imprimir el promedio de varios enteros. Suponga que el último valor leído es el valor centinela 9999. Una secuencia típica de entrada podría ser

10 8 11 7 9 9999

lo cual indica que el programa debe calcular el promedio de todos los valores antes del 9999.

- 5.7** ¿Qué hace el siguiente programa?

```

1 // Ejercicio 5.7: ej05_07.cpp
2 // ¿Qué imprime este programa?
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 int main()
9 {
10     int x; // declara x
11     int y; // declara y
12
13     // pide al usuario los datos de entrada
14     cout << "Escriba dos enteros en el rango 1 a 20: ";
15     cin >> x >> y; // lee valores para x y y
16
17     for ( int i = 1; i <= y; i++ ) // cuenta desde 1 hasta y
18     {
19         for ( int j = 1; j <= x; j++ ) // cuenta desde 1 hasta x
20             cout << '@'; // imprime @
21
22         cout << endl; // empieza nueva línea
23     } // fin de for exterior
24
25     return 0; // indica que terminó correctamente
26 } // fin de main

```

- 5.8** Escriba un programa que utilice una instrucción **for** para encontrar el menor de varios enteros. Suponga que el primer valor leído especifica el número de valores restantes.

- 5.9** Escriba un programa que utilice una instrucción **for** para calcular e imprimir el producto de los enteros impares del 1 al 15.

- 5.10** La función factorial se utiliza frecuentemente en los problemas de probabilidad. Utilizando la definición de factorial del ejercicio 4.35, escriba un programa que utilice una función **for** para evaluar los factoriales de los enteros del 1 al 5. Muestre los resultados en formato tabular. ¿Qué dificultad podría impedir que usted calculara el factorial de 20?

- 5.11** Modifique el programa de interés compuesto de la sección 5.4, repitiendo sus pasos para las tasas de interés del 5, 6, 7, 8, 9 y 10%. Use una instrucción **for** para variar la tasa de interés.

- 5.12** Escriba un programa que utilice ciclos **for** para generar los siguientes patrones por separado, uno debajo del otro. Use ciclos **for** para generar los patrones. Todos los asteriscos (*) deben imprimise mediante una sola instrucción de la forma **cout << '*' ;** (esto hace que los asteriscos se impriman uno al lado del otro). [Sugerencia: los últimos dos patrones requieren que cada línea empiece con un número apropiado de espacios en blanco. Crédito adicional: combine su código de los cuatro problemas separados en un solo programa que imprima los cuatro patrones, uno al lado del otro, haciendo un uso inteligente de los ciclos **for** anidados.]

(a)	(b)	(c)	(d)
*	*****	*****	*
**	*****	*****	**
***	*****	*****	***
****	****	****	****
*****	***	***	*****
*****	**	**	*****
*****	*	*	*****

5.13 Una aplicación interesante de las computadoras es dibujar gráficos convencionales y de barra. Escriba un programa que lea cinco números (cada uno entre 1 y 30). Suponga que el usuario sólo introduce valores válidos. Por cada número leído, su programa debe imprimir una línea que contenga ese número de asteriscos adyacentes. Por ejemplo, si su programa lee el número 7, debe mostrar *****.

5.14 Un almacén de pedidos por correo vende cinco productos distintos, cuyos precios de venta son los siguientes: producto 1, \$2.98; producto 2, \$4.50; producto 3, \$9.98; producto 4, \$4.49 y producto 5, \$6.87. Escriba un programa que lea una serie de pares de números, como se muestra a continuación:

- a) número del producto;
- b) cantidad vendida.

Su programa debe utilizar una instrucción `switch` para determinar el precio de venta de cada producto. Debe calcular y mostrar el valor total de venta de todos los productos vendidos. Use un ciclo controlado por centinela para determinar cuándo debe el programa dejar de iterar para mostrar los resultados finales.

5.15 Modifique el programa `LibroCalificaciones` de las figuras 5.9 a 5.11, de manera que calcule el promedio de puntos de calificaciones para el conjunto de calificaciones. Una calificación A vale 4 puntos, B vale 3 puntos, etcétera.

5.16 Modifique el programa de la figura 5.6, de manera que se utilicen sólo enteros para calcular el interés compuesto. [Sugerencia: trate todas las cantidades monetarias como números enteros de centavos. Luego “divida” el resultado en su porción de dólares y su porción de centavos, utilizando las operaciones de división y módulo, respectivamente. Inserte un punto.]

5.17 Suponga que `i = 1, j = 2, k = 3 y m = 2`. ¿Qué es lo que imprime cada una de las siguientes instrucciones? ¿Son necesarios los paréntesis en cada caso?

- a) `cout << (i == 1) << endl;`
- b) `cout << (j == 3) << endl;`
- c) `cout << (i >= 1 && j < 4) << endl;`
- d) `cout << (m <= 99 && k < m) << endl;`
- e) `cout << (j >= i || k == m) << endl;`
- f) `cout << (k + m < j | 3 - j >= k) << endl;`
- g) `cout << (!m) << endl;`
- h) `cout << (!(j - m)) << endl;`
- i) `cout << (!(k > m)) << endl;`

5.18 Escriba un programa que imprima una tabla de los equivalentes binario, octal y hexadecimal de los números decimales en el rango 1 al 256. Si no está familiarizado con estos sistemas numéricos,lea primero el apéndice D, Sistemas numéricos.

5.19 Calcule el valor de π a partir de la serie infinita

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \dots$$

Imprima una tabla que muestre el valor aproximado de π , después de cada uno de los primeros 1000 términos de esta serie.

5.20 (*Triples de Pitágoras*) Un triángulo recto puede tener lados cuyas longitudes sean valores enteros. Un conjunto de tres valores enteros para los lados de un triángulo recto se conoce como triple de Pitágoras. Estos tres lados deben satisfacer la relación que establece que la suma de los cuadrados de dos lados es igual al cuadrado de la hipotenusa. Encuentre todos los triples de Pitágoras para `lado1`, `lado2`, y la `hipotenusa`, que no sean mayores de 500. Use un ciclo `for` triplicemente anidado para probar todas las posibilidades. Este método es un ejemplo de la computación de **fuerza bruta**. En cursos de ciencias computacionales más avanzados aprenderá que existen muchos problemas interesantes para los cuales no hay otra metodología algorítmica conocida, más que el uso de la fuerza bruta.

5.21 Una empresa paga a sus empleados como gerentes (quienes reciben un salario semanal fijo), trabajadores por horas (que reciben un sueldo fijo por hora para las primeras 40 horas que trabajen y “tiempo y medio”, 1.5 veces su sueldo por horas, para las horas extra trabajadas), empleados por comisión (que reciben \$250 más 5.7 por ciento de sus ventas totales por semana), o trabajadores por piezas (que reciben una cantidad fija de dinero por cada artículo que producen; cada trabajador por piezas en esta empresa trabaja sólo en un tipo de artículo). Escriba un programa para calcular el sueldo semanal para cada empleado. No necesita saber cuántos empleados hay de antemano. Cada tipo de empleado tiene su propio código de pago: los gerentes tienen el código 1, los trabajadores por horas tienen el código 2, los trabajadores por comisión tienen el código 3 y los trabajadores por piezas tienen el código 4. Use una instrucción `switch` para calcular el sueldo de cada empleado, de acuerdo con el código de pago de cada uno. Dentro del `switch`, pida al usuario (es decir, el cajero de nóminas) que introduzca los hechos apropiados que su programa necesita para calcular el sueldo de cada empleado, de acuerdo con su código de pago.

5.22 (*Leyes de De Morgan*) En este capítulo, hemos hablado sobre los operadores lógicos `&&`, `||` y `!`. Algunas veces, las leyes de De Morgan pueden hacer que sea más conveniente para nosotros expresar una expresión lógica. Estas leyes establecen que la expresión `!(condición1 && condición2)` es lógicamente equivalente a la expresión `(! condición1 || !condición2)`. Además, la expresión `!(condición1 || condición2)` es lógicamente equivalente a la expresión `(! condición1 && !condición2)`. Use las leyes de De Morgan para escribir expresiones equivalentes para cada una de las siguientes expresiones, luego escriba una aplicación que demuestre que, tanto la expresión original como la nueva expresión, son equivalentes en cada caso:

- `!(x < 5) && !(y >= 7)`
- `!(a == b) || !(g != 5)`
- `!((x <= 8) && (y > 4))`
- `!((i > 4) || (j <= 6))`

5.23 Escriba un programa que imprima la siguiente figura de rombo. Puede utilizar instrucciones de salida que impriman un solo asterisco (*) o un solo espacio en blanco. Maximice el uso de la repetición (con instrucciones `for` anidadas), y minimice el número de instrucciones de salida.

```

*
 ***
 *****
 ******
 *****
 ****
 ***
 *

```

5.24 Modifique el programa que escribió en el ejercicio 5.23, para que lea un número impar en el rango de 1 a 19, de manera que especifique el número de filas en el rombo, y después muestre un rombo del tamaño apropiado.

5.25 Una crítica de las instrucciones `break` y `continue` es que ninguna es estructurada. En realidad, estas instrucciones pueden reemplazarse en todo momento por instrucciones estructuradas, aunque hacerlo podría ser inadecuado. Describa, en general, cómo eliminaría las instrucciones `break` de un ciclo en un programa, para reemplazarlas con alguna de las instrucciones estructuradas equivalentes. [Sugerencia: la instrucción `break` se sale de un ciclo desde el cuerpo de éste. La otra forma de salir es que falle la prueba de continuación de ciclo. Considere utilizar en la prueba de continuación de ciclo una segunda prueba que indique una “salida anticipada debido a una condición de ‘interrupción.’”] Use la técnica que desarrolló aquí para eliminar la instrucción `break` de la aplicación de la figura 5.13.

5.26 ¿Qué hace el siguiente segmento de programa?

```

1   for ( int i = 1; i <= 5; i++ )
2   {
3       for ( int j = 1; j <= 3; j++ )
4       {
5           for ( int k = 1; k <= 4; k++ )
6               cout << '*';
7
8           cout << endl;
9       } // fin del for interior
10
11      cout << endl;
12  } // fin del for exterior

```

5.27 Describa, en general, cómo eliminaría las instrucciones `continue` de un ciclo en un programa, para reemplazarlas con uno de sus equivalentes estructurados. Use la técnica que desarrolló aquí para eliminar la instrucción `continue` del programa de la figura 5.14.

5.28 (*Canción “Los Doce Días de Navidad”*) Escriba una aplicación que utilice instrucciones de repetición y `switch` para imprimir la canción “Los Doce Días de Navidad”. Una instrucción `switch` debe utilizarse para imprimir el día (es decir, “primer”, “segundo”, etcétera). Una instrucción `switch` separada debe utilizarse para imprimir el resto de cada verso. Visite el sitio Web http://en.wikipedia.org/wiki/Twelve_tide para obtener la letra completa de la canción.

5.29 (*Problema de Peter Minuit*) La leyenda establece que, en 1626, Peter Minuit compró la isla de Manhattan por \$24.00 en un trueque. ¿Hizo él una buena inversión? Para responder a esta pregunta, modifique el programa de interés compuesto de la figura 5.6, para empezar con un monto principal de \$24.00 y calcular el monto de interés en depósito, si ese dinero se había mantenido en depósito hasta este año (por ejemplo, 381 años hasta 2007). Coloque el ciclo `for` que realiza el cálculo del interés compuesto en un ciclo `for` exterior que varíe la tasa de interés del 5 al 10 por ciento, para observar las maravillas del interés compuesto.



La forma siempre va después de la función.

—Louis Henri Sullivan

*E pluribus unum.
(Uno compuesto de varios.)*

—Virgilio

*Oh! recuerdos del ayer,
tratar de regresar el tiempo.*

—William Shakespeare

Llámenme Ismael.

—Herman Melville

*Cuando me llames así,
¡sonríe!*

—Owen Wister

*Respóndeme en una
palabra.*

—William Shakespeare

*Hay un punto en el cual
los métodos se devoran a sí
mismos.*

—Frantz Fanon

*La vida sólo puede
comprenderse en sentido
inverso; pero debe vivirse
hacia adelante.*

—Soren Kierkegaard

Funciones y una introducción a la recursividad

OBJETIVOS

En este capítulo aprenderá a:

- Crear programas en forma modular, a partir de funciones.
- Aprender a utilizar las funciones matemáticas comunes, disponibles en la Biblioteca estándar de C++.
- Crear funciones con varios parámetros.
- Conocer los mecanismos para pasar información entre funciones y devolver resultados.
- Comprender cómo el mecanismo de llamadas a/regreso de funciones está soportado por la pila de llamadas a funciones y los registros de activación.
- Aprender a usar la generación de números aleatorios para implementar aplicaciones de juegos.
- Comprender cómo la visibilidad de los identificadores está limitada a regiones específicas de programas.
- Aprender a escribir y usar funciones recursivas; es decir, funciones que se llaman a sí mismas.

- 6.1** Introducción
- 6.2** Componentes de los programas en C++
- 6.3** Funciones matemáticas de la biblioteca
- 6.4** Definiciones de funciones con varios parámetros
- 6.5** Prototipos de funciones y coerción de argumentos
- 6.6** Archivos de encabezado de la Biblioteca estándar de C++
- 6.7** Ejemplo práctico: generación de números aleatorios
- 6.8** Ejemplo práctico: juego de probabilidad, introducción a las enumeraciones
- 6.9** Clases de almacenamiento
- 6.10** Reglas de alcance
- 6.11** La pila de llamadas a funciones y los registros de activación
- 6.12** Funciones con listas de parámetros vacías
- 6.13** Funciones en línea
- 6.14** Referencias y parámetros de referencias
- 6.15** Argumentos predeterminados
- 6.16** Operador de resolución de ámbito unario
- 6.17** Sobrecarga de funciones
- 6.18** Plantillas de funciones
- 6.19** Recursividad
- 6.20** Ejemplo sobre el uso de la recursividad: serie de Fibonacci
- 6.21** Comparación entre recursividad e iteración
- 6.22** (Opcional) Ejemplo práctico de Ingeniería de Software: identificación de las operaciones de las clases en el sistema ATM
- 6.23** Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)

6.1 Introducción

La mayoría de los programas que resuelven problemas reales son mucho más grandes que los programas que se presentan en los primeros capítulos de este libro. La experiencia ha demostrado que la mejor forma de desarrollar y mantener un programa extenso es construirlo a partir de piezas (o componentes) simples y pequeñas. A esta técnica se le conoce como **divide y vencerás**. En el capítulo 3 presentamos las funciones (como piezas de un programa). En este capítulo las estudiaremos con más detalle. Nos enfocaremos en cómo declarar y usar las funciones para facilitar el diseño, la implementación, operación y mantenimiento de programas extensos.

Veremos las generalidades de una porción de las funciones matemáticas de la Biblioteca estándar de C++, y mostraremos varias que requieren más de un parámetro. Después, el lector aprenderá a declarar una función con más de un parámetro. También presentaremos información adicional acerca de los prototipos de funciones y la forma en que el compilador los utiliza para convertir el tipo de argumento en la llamada a una función, al tipo especificado en la lista de parámetros de una función, si es necesario.

Luego, daremos un pequeño giro hacia las técnicas de simulación con la generación de números aleatorios, y desarrollaremos una versión del juego de casino conocido como “craps”, el cual utiliza la mayoría de las técnicas de programación que el lector ha aprendido hasta este punto en el libro.

Posteriormente, presentaremos las clases de almacenamiento y reglas de alcance de C++. Éstas determinan el periodo durante el cual un objeto existe en la memoria, y en dónde se puede hacer referencia a su identificador en un programa. También aprenderá cómo C++ es capaz de llevar el registro de cuál función se ejecuta en un momento dado, cómo se mantienen los parámetros y otras variables locales de las funciones en la memoria, y cómo sabe una función a dónde regresar, una vez que termina su ejecución. Hablaremos sobre dos temas que ayudan a mejorar el rendimiento de los programas: las funciones en línea que pueden eliminar la sobrecarga de la llamada a una función, y parámetros de referencia que pueden usarse para pasar elementos extensos de datos a las funciones con eficiencia.

Muchas de las aplicaciones que desarrollará tendrán más de una función con el mismo nombre. Esta técnica, llamada sobrecarga de funciones, es utilizada por los programadores para implementar funciones que realicen tareas similares para los argumentos de distintos tipos, o posiblemente para distintos números de argumentos. Vamos a considerar las plantillas de funciones: un mecanismo para definir una familia de funciones sobrecargadas. Este capítulo concluye con una discusión de las funciones que se llaman a sí mismas, ya sea en forma directa o indirecta (a través de otra función); un tema llamado recursividad, que se discute de manera extensa en cursos de ciencias computacionales de nivel superior.

6.2 Componentes de los programas en C++

Por lo general, los programas en C++ se escriben mediante la combinación de nuevas funciones y clases que escribimos con funciones “pre-empaquetadas”, y clases disponibles en la Biblioteca estándar de C++. En este capítulo, nos concentraremos en sus funciones.

La Biblioteca estándar de C++ proporciona una extensa colección de funciones para realizar cálculos matemáticos comunes, manipulaciones de cadena, manipulaciones de caracteres, entrada/salida, comprobación de errores y muchas otras operaciones útiles. Esto facilita el trabajo del programador, ya que estas funciones proporcionan muchas de las herramientas que necesita. Las funciones de la Biblioteca estándar de C++ se proporcionan como parte del entorno de programación de C++.



Observación de Ingeniería de Software 6.1

Lea la documentación de su compilador, para familiarizarse con las funciones y clases en la Biblioteca estándar de C++.

Las funciones (llamadas **métodos** o **procedimientos** en otros lenguajes de programación) permiten al programador modularizar un programa, al separar sus tareas en unidades autocontenidoas. Ya ha utilizado funciones en todos los programas que ha escrito. Algunas veces, estas funciones se denominan **funciones definidas por el usuario**, o **funciones definidas por el programador**. Las instrucciones en los cuerpos de las funciones se escriben sólo una vez, y se pueden reutilizar desde varias ubicaciones en un programa; además están ocultas de las demás funciones.

Hay varias razones para modularizar un programa con funciones. Una de ellas es la metodología **divide y vencerás**, la cual facilita el proceso de desarrollo de programas al construirlos a partir de piezas pequeñas y simples. Otra de ellas es la reutilización de software: utilizar las funciones existentes como bloques de construcción para crear nuevos programas. Por ejemplo, en los programas anteriores no tuvimos que definir cómo leer una línea de texto del teclado; C++ proporciona esta herramienta a través de la función `getline` del archivo de encabezado `<string>`. Una tercera motivación es la de evitar repetir código. Además, al dividir un programa en funciones significativas, es más fácil depurarlo y darle mantenimiento.



Observación de Ingeniería de Software 6.2

Para promover la reutilización de software, toda función debe limitarse a realizar una sola tarea bien definida, y el nombre de la función debe expresar esa tarea con efectividad. Dichas funciones facilitan la escritura, prueba, depuración y mantenimiento de los programas.



Tip para prevenir errores 6.1

Una pequeña función que realiza una tarea es más fácil de probar y depurar que una función más grande que realiza muchas tareas.



Observación de Ingeniería de Software 6.3

Si no puede elegir un nombre conciso que exprese la tarea de una función, tal vez ésta esté tratando de realizar demasiadas tareas diversas. Por lo general, es mejor descomponer dicha función en varias funciones más pequeñas.

Como sabemos, una función se invoca mediante una llamada, y cuando la función a la que se llamó completa su tarea, devuelve un resultado o simplemente devuelve el control a la función que la llamó. Una analogía a esta estructura de un programa es la forma jerárquica de la administración (figura 6.1). Un jefe (similar a la función que hace la llamada) pide a un trabajador (similar a la función que se llamó) que realice una tarea y reporte (devuelva) los resultados, después de completarla. La función jefe no sabe cómo realiza la función trabajador sus tareas designadas. El trabajador también podría llamar a otras funciones trabajador, sin que el jefe supiera. Este ocultamiento de los detalles de implementación promueve la buena ingeniería de software. La figura 6.1 muestra cómo la función `jefe` se comunica con varias funciones trabajador de una manera jerárquica. La función `jefe` divide las responsabilidades entre las diversas funciones `trabajador`. Observe que `trabajador1` actúa como “función jefe” para `trabajador4` y `trabajador5`.

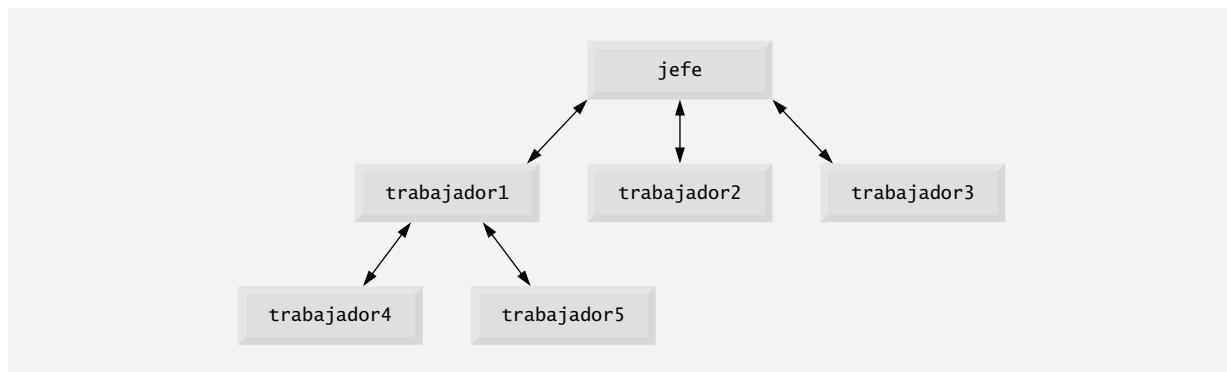


Figura 6.1 | Relación jerárquica entre la función jefe y las funciones trabajador.

6.3 Funciones matemáticas de la biblioteca

Como sabemos, una clase puede proporcionar funciones miembro que desempeñen los servicios de la clase. Por ejemplo, en los capítulos 3 a 5 hemos llamado a las funciones miembro de varias versiones de un objeto `LibroCalificaciones` para mostrar el mensaje de bienvenida del `LibroCalificaciones`, establecer el nombre del curso, obtener un conjunto de calificaciones y calcular el promedio de las mismas.

Algunas veces, las funciones no son miembros de una clase. A dichas funciones se les conoce como **funciones globales**. Al igual que las funciones miembro de una clase, los prototipos para las funciones globales se colocan en archivos de encabezado, de manera que las funciones globales se puedan reutilizar en cualquier programa que incluya el archivo de encabezado y pueda crear un enlace con el código objeto de la función. Por ejemplo, recuerde que utilizamos la función `pow` del archivo de encabezado `<cmath>` para elevar un valor a una potencia en la figura 5.6. Introduciremos aquí varias funciones del archivo de encabezado `<cmath>` para presentar el concepto de las funciones globales que no pertenecen a una clase específica. En este capítulo y en los capítulos subsiguientes, utilizaremos una combinación de funciones globales (como `main`) y clases con funciones miembro para implementar nuestros programas de ejemplo.

El archivo de encabezado `<cmath>` proporciona una colección de funciones que nos permiten realizar cálculos matemáticos comunes. Por ejemplo, puede calcular la raíz cuadrada de 900.0 con la siguiente llamada a la función:

```
sqrt( 900.0 )
```

La expresión anterior se evalúa como 30.0. La función `sqrt` recibe un argumento de tipo `double` y devuelve un resultado `double`. Observe que no hay necesidad de crear objetos antes de llamar a la función `sqrt`. Además, observe que *todas* las funciones en el archivo de encabezado `<cmath>` son funciones globales; por lo tanto, para llamar a cada una de ellas sólo hay que especificar el nombre de la función, seguido de paréntesis que contienen los argumentos de la misma.

Los argumentos de una función pueden ser constantes, variables o expresiones más complejas. Si `c = 13`, `d = 3.0` y `f = 4.0`, entonces la instrucción:

```
cout << sqrt( c + d * f ) << endl;
```

calcula e imprime la raíz cuadrada de $13.0 + 3.0 * 4.0 = 25.0$, a saber, 5.0. Algunas funciones matemáticas se sintetizan en la figura 6.2. En la figura, las variables `x` y `y` son de tipo `double`.

Función	Descripción	Ejemplo
<code>ceil(x)</code>	redondea x al entero valor más pequeño, no menor a x	<code>ceil(9.2)</code> es 10.0 <code>ceil(-9.8)</code> es -9.0
<code>cos(x)</code>	coseno trigonométrico de x (x está en radianes)	<code>cos(0.0)</code> es 1.0
<code>exp(x)</code>	función exponencial e^x	<code>exp(1.0)</code> es 2.71828 <code>exp(2.0)</code> es 7.38906

Figura 6.2 | Funciones matemáticas de la biblioteca. (Parte I de 2).

Función	Descripción	Ejemplo
<code>fabs(x)</code>	valor absoluto de x	<code>fabs(5.1)</code> es 5.1 <code>fabs(0.0)</code> es 0.0 <code>fabs(-8.76)</code> es 8.76
<code>floor(x)</code>	redondea x al entero más grande, no mayor a x	<code>floor(9.2)</code> es 9.0 <code>floor(-9.8)</code> es -10.0
<code>fmod(x, y)</code>	residuo de x/y como número de punto flotante	<code>fmod(2.6, 1.2)</code> es 0.2
<code>log(x)</code>	logaritmo natural de x (base e)	<code>log(2.718282)</code> es 1.0 <code>log(7.389056)</code> es 2.0
<code>log10(x)</code>	logaritmo de x (base 10)	<code>log10(10.0)</code> es 1.0 <code>log10(100.0)</code> es 2.0
<code>pow(x, y)</code>	x elevado a la potencia y (x^y)	<code>pow(2, 7)</code> es 128 <code>pow(9, .5)</code> es 3
<code>sin(x)</code>	seno trigonométrico de x (x en radianes)	<code>sin(0.0)</code> es 0
<code>sqrt(x)</code>	raíz cuadrada de x (donde x es un valor no negativo)	<code>sqrt(9.0)</code> es 3.0
<code>tan(x)</code>	tangente trigonométrica de x (x en radianes)	<code>tan(0.0)</code> es 0

Figura 6.2 | Funciones matemáticas de la biblioteca. (Parte 2 de 2).

6.4 Definiciones de funciones con varios parámetros

Los capítulos 3 a 5 presentaron clases que contienen funciones simples que tenían cuando mucho un parámetro. A menudo, las funciones requieren más de una pieza de información para realizar sus tareas. Ahora consideraremos las funciones con varios parámetros.

El programa en las figuras 6.3 a 6.5 modifica nuestra clase `LibroCalificaciones`, al incluir una función definida por el usuario llamada `maximo`, la cual determina y devuelve el mayor de tres valores `int`. Cuando la aplicación empieza su ejecución, la función `main` (líneas 5 a 14 de la figura 6.5) crea un objeto de la clase `LibroCalificaciones` (línea 8) y llama a la función miembro `recibirCalificaciones` del objeto (línea 11) para leer tres calificaciones enteras del usuario. En el archivo de implementación de la clase `LibroCalificaciones` (figura 6.4), en las líneas 54 y 55 de la función miembro `recibirCalificaciones` pide al usuario que introduzca tres valores enteros y los recibe de éste. En la línea 58 se hace una llamada a la función miembro `máximo` (definida en las líneas 62 a 75). La función `maximo` determina el valor más grande, y después la instrucción `return` (línea 74) devuelve el valor al punto en el cual la función `recibirCalificaciones` invocó a `maximo` (línea 58). Entonces, la función miembro `recibirCalificaciones` almacena el valor de retorno de `maximo` en el miembro de datos `calificacionMaxima`. Después, este valor se imprime llamando a la función `mostrarReporteCalificaciones` (línea 12 de la figura 6.5). [Nota: a esta función la llamamos `mostrarReporteCalificaciones`, ya que versiones subsiguientes de la clase `LibroCalificaciones` utilizará esta función para mostrar un reporte completo de calificaciones, incluyendo las calificaciones máxima y mínima.] En el capítulo 7, Arreglos y vectores, mejoraremos el `LibroCalificaciones` para procesar un número arbitrario de calificaciones.

```

1 // Fig. 6.3: LibroCalificaciones.h
2 // Definición de la clase LibroCalificaciones que encuentra el máximo de las tres
   calificaciones.
3 // Las funciones miembro están definidas en LibroCalificaciones.cpp
4 #include <string> // el programa usa la clase string estándar de C++

```

Figura 6.3 | Archivo de encabezado de `LibroCalificaciones`. (Parte 1 de 2).

```

5  using std::string;
6
7 // definición de la clase LibroCalificaciones
8 class LibroCalificaciones
9 {
10 public:
11     LibroCalificaciones( string ); // el constructor inicializa el nombre del curso
12     void establecerNombreCurso( string ); // función para establecer el nombre del curso
13     string obtenerNombreCurso(); // función para obtener el nombre del curso
14     void mostrarMensaje(); // mostrar un mensaje de bienvenida
15     void recibirCalificaciones(); // recibe las tres calificaciones del usuario
16     void mostrarReporteCalificaciones(); // muestra un reporte con base en las calificaciones
17     int maximo( int, int, int ); // determina el máximo de 3 valores
18 private:
19     string nombreCurso; // nombre del curso para este LibroCalificaciones
20     int calificacionMaxima; // valor máximo de las tres calificaciones
21 } // fin de la clase LibroCalificaciones

```

Figura 6.3 | Archivo de encabezado de *LibroCalificaciones*. (Parte 2 de 2).

```

1 // Fig. 6.4: LibroCalificaciones.cpp
2 // Definiciones de funciones miembro para la clase LibroCalificaciones
3 // que determina el máximo de tres calificaciones.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include "LibroCalificaciones.h" // incluye la definición de la clase LibroCalificaciones
10
11 // el constructor inicializa nombreCurso con la cadena suministrada como argumento;
12 // inicializa calificacionMaxima a 0
13 LibroCalificaciones::LibroCalificaciones( string nombre )
14 {
15     establecerNombreCurso( nombre ); // valida y almacena nombreCurso
16     calificacionMaxima = 0; // este valor se reemplazará por la calificación máxima
17 } // fin del constructor LibroCalificaciones
18
19 // función para establecer el nombre del curso; limita nombre a 25 o menos caracteres
20 void LibroCalificaciones::establecerNombreCurso( string nombre )
21 {
22     if ( nombre.length() <= 25 ) // si nombre tiene 25 o menos caracteres
23         nombreCurso = nombre; // almacena el nombre del curso en el objeto
24     else // si nombre es mayor que 25 caracteres
25     { // establece nombreCurso a los primeros 25 caracteres del parámetro nombre
26         nombreCurso = nombre.substr( 0, 25 ); // selecciona los primeros 25 caracteres
27         cout << "El nombre \"" << nombre << "\" excede la longitud maxima (25).\\n"
28             << "Se limitó nombreCurso a los primeros 25 caracteres.\\n" << endl;
29     } // fin de if...else
30 } // fin de la función establecerNombreCurso
31
32 // función para obtener el nombre del curso
33 string LibroCalificaciones::obtenerNombreCurso()
34 {
35     return nombreCurso;
36 } // fin de la función obtenerNombreCurso
37
38 // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
39 void LibroCalificaciones::mostrarMensaje()
40 {

```

Figura 6.4 | La clase *LibroCalificaciones* define a la función *maximo*. (Parte 1 de 2).

```

41 // esta instrucción llama a obtenerNombreCurso para obtener el
42 // nombre del curso que representa este LibroCalificaciones
43 cout << "Bienvenido al libro de calificaciones para\n" << obtenerNombreCurso() << "!\n"
44     << endl;
45 } // fin de la función mostrarMensaje
46
47 // recibe tres calificaciones del usuario; determina el valor máximo
48 void LibroCalificaciones::recibirCalificaciones()
49 {
50     int calificacion1; // primera calificación introducida por el usuario
51     int calificacion2; // segunda calificación introducida por el usuario
52     int calificacion3; // tercera calificación introducida por el usuario
53
54     cout << "Introduzca tres calificaciones enteras: ";
55     cin >> calificacion1 >> calificacion2 >> calificacion3;
56
57     // almacena el valor máximo en el miembro calificacionMaxima
58     calificacionMaxima = maximo( calificacion1, calificacion2, calificacion3 );
59 } // fin de la función recibirCalificaciones
60
61 // devuelve el máximo de sus tres parámetros enteros
62 int LibroCalificaciones::maximo( int x, int y, int z )
63 {
64     int valorMaximo = x; // supone que x es el mayor para empezar
65
66     // determina si y es mayor que valorMaximo
67     if ( y > valorMaximo )
68         valorMaximo = y; // hace a y el nuevo valorMaximo
69
70     // determina si z es mayor que valorMaximo
71     if ( z > valorMaximo )
72         valorMaximo = z; // hace a z el nuevo valorMaximo
73
74     return valorMaximo;
75 } // fin de la función maximo
76
77 // muestra un reporte con base en las calificaciones introducidas por el usuario
78 void LibroCalificaciones::mostrarReporteCalificaciones()
79 {
80     // imprime el máximo de las calificaciones introducidas
81     cout << "Calificacion maxima introducida: " << calificacionMaxima << endl;
82 } // fin de la función mostrarReporteCalificaciones

```

Figura 6.4 | La clase LibroCalificaciones define a la función maximo. (Parte 2 de 2).

```

1 // Fig. 6.5: fig06_05.cpp
2 // Crea un objeto LibroCalificaciones, recibe las calificaciones y muestra un reporte.
3 #include "LibroCalificaciones.h" // incluye la definición de la clase LibroCalificaciones
4
5 int main()
6 {
7     // crea un objeto LibroCalificaciones
8     LibroCalificaciones miLibroCalificaciones( "CS101 Programacion en C++" );
9
10    miLibroCalificaciones.mostrarMensaje(); // muestra mensaje de bienvenida
11    miLibroCalificaciones.recibirCalificaciones(); // lee calificaciones del usuario
12    miLibroCalificaciones.mostrarReporteCalificaciones(); // muestra reporte con base en las
13        calificaciones
14    return 0; // indica que terminó correctamente
15 } // fin de main

```

Figura 6.5 | Demostración de la función maximo. (Parte 1 de 2).

Bienvenido al libro de calificaciones para
CS101 Programacion en C++!

Introduzca tres calificaciones enteras: **86 67 75**
Calificacion maxima introducida: 86

Bienvenido al libro de calificaciones para
CS101 Programacion en C++!

Introduzca tres calificaciones enteras: **67 86 75**
Calificacion maxima introducida: 86

Bienvenido al libro de calificaciones para
CS101 Programacion en C++!

Introduzca tres calificaciones enteras: **67 75 86**
Calificacion maxima introducida: 86

Figura 6.5 | Demostración de la función `maximo`. (Parte 2 de 2).



Observación de Ingeniería de Software 6.4

Las comas utilizadas en la línea 58 de la figura 6.4 para separar los argumentos de la función `maximo` no son operadores coma, como vimos en la sección 5.3. El operador coma garantiza que sus operandos se evalúen de izquierda a derecha. Sin embargo, el estándar de C++ no especifica el orden de evaluación de los argumentos de una función. Por ende, distintos compiladores pueden evaluar los argumentos de una función en distintos órdenes. El estándar de C++ garantiza que todos los argumentos en la llamada a una función se evalúen antes de ejecutar la función que se va a llamar.



Tip de portabilidad 6.1

Algunas veces, cuando los argumentos de una función son expresiones más elaboradas, como las de las llamadas a otras funciones, el orden en el que el compilador evalúa los argumentos podría afectar a los valores de uno o más de los argumentos. Si el orden de evaluación cambia entre un compilador y otro, los valores de los argumentos que se pasan a la función podrían variar, lo cual produciría errores lógicos sutiles.



Tip para prevenir errores 6.2

Si tiene dudas acerca del orden de evaluación de los argumentos de una función, y de si el orden afectaría los valores que se pasan a la función, evalúe los argumentos en instrucciones de asignación separadas antes de la llamada a la función, asigne el resultado de cada expresión a una variable local y después pase esas variables como argumentos para la función.

El prototipo de la función miembro `maximo` (figura 6.3, línea 17) indica que la función devuelve un valor entero, que el nombre de la función es `maximo` y que la función requiere tres parámetros enteros para realizar su tarea. El encabezado de la función `maximo` (figura 6.4, línea 62) concuerda con el prototipo de función e indica que los nombres de los parámetros son `x`, `y` y `z`. Cuando se hace una llamada a `maximo` (figura 6.4, línea 58), el parámetro `x` se inicializa con el valor del argumento `calificacion1`, el parámetro `y` se inicializa con el valor del argumento `calificacion2` y el parámetro `z` se inicializa con el valor del argumento `calificacion3`. Debe haber un argumento en la llamada a la función para cada parámetro (también conocido como `parámetro formal`) en la definición de la función.

Observe que varios parámetros se especifican en el prototipo de función y en el encabezado de la función como una lista separada por comas. El compilador hace referencia al prototipo de la función para comprobar que las llamadas a `maximo` contengan el número y tipos de argumentos correctos, y que los tipos estén en el orden correcto. Además, el compilador usa el prototipo para asegurar que el valor devuelto por la función se pueda utilizar de manera correcta en la expresión que llamó a la función (por ejemplo, la llamada a una función que devuelve `void` no se puede utilizar como el lado derecho de una instrucción de asignación). Cada argumento debe ser consistente con el tipo del parámetro correspondiente. Por ejemplo, un parámetro de tipo `double` puede recibir valores como `7.35`, `22` o `-0.03456`, pero no una cadena como `"holá"`. Si los argumentos que se pasan a una función no concuerdan con los tipos especificados en el prototipo de la función, el compilador trata de convertir los argumentos a esos tipos. En la sección 6.5 hablaremos sobre esta conversión.

Error común de programación 6.1

Declarar los parámetros de métodos del mismo tipo como `double x, y` en lugar de `double x, double y` es un error de sintaxis; se requiere un tipo explícito para cada parámetro en la lista de parámetros.

Error común de programación 6.2

Si el prototipo de función, encabezado de función y las llamadas a la función no concuerdan en el número, tipo y orden de argumentos y parámetros, además del tipo de retorno, se producen errores de compilación.

Observación de Ingeniería de Software 6.5

Una función que tiene muchos parámetros puede estar realizando muchas tareas. Considere dividir la función en funciones más pequeñas que realicen las tareas por separado. Limite el encabezado de la función a una línea, si es posible.

Para determinar el valor máximo (líneas 62 a 75 de la figura 6.4), empezamos con la suposición de que el parámetro `x` contiene el valor más grande, por lo que en la línea 64 de la función `maximo` se declara la variable local `valorMaximo` y se inicializa con el valor del parámetro `x`. Desde luego, es posible que el parámetro `y` o `z` contenga el valor más grande actual, por lo que debemos comparar cada uno de estos valores con `valorMaximo`. La instrucción `if` en las líneas 67 y 68 determina si `y` es mayor que `valorMaximo` y, de ser así, asigna `y` a `valorMaximo`. La instrucción `if` en las líneas 71 y 72 determina si `z` es mayor que `valorMaximo` y, de ser así, asigna `z` a `valorMaximo`. En este punto, el mayor de los tres valores está en `valorMaximo`, por lo que la línea 74 devuelve ese valor a la llamada en la línea 58. Cuando el control del programa regresa al punto donde se llamó a `maximo`, los parámetros `x`, `y` y `z` de `maximo` no están accesibles ya para el programa. En la siguiente sección veremos por qué.

Hay tres formas de devolver el control al punto en el que se invocó a una función. Si la función no devuelve un resultado (es decir, si la función tiene un tipo de valor de retorno `void`), el control regresa cuando el programa llega a la llave derecha de fin de la función, o mediante la ejecución de la instrucción

```
return;
```

Si la función devuelve un resultado, la instrucción

```
return expresion;
```

evalúa `expresion` y devuelve el valor de `expresion` a la función que hizo la llamada.

6.5 Prototipos de funciones y coerción de argumentos

Un prototipo de función (también conocido como declaración de función) indica al compilador el nombre de una función, el tipo de datos devuelto por la función, el número de parámetros que la función espera recibir, los tipos de esos parámetros y el orden en el que éstos se esperan.

Observación de Ingeniería de Software 6.6

Los prototipos de función son obligatorios en C++. Use directivas `#include` del preprocesador para obtener prototipos de función para las funciones de la Biblioteca estándar de C++ de los archivos de encabezado para las bibliotecas apropiadas (por ejemplo, el prototipo para la función matemática `sqr` está en el archivo de encabezado `<cmath>`; una lista parcial de los archivos de encabezado de la Biblioteca estándar de C++ aparece en la sección 6.6). Use también `#include` para obtener archivos de encabezado que contengan prototipos de función escritos por usted, o por los miembros de su grupo.

Error común de programación 6.3

Si se define una función antes de invocarla, entonces la definición de una función también sirve como el prototipo de la misma, por lo que no es necesario un prototipo separado. Si se invoca una función antes de definirla, y esa función no tiene un prototipo de función, se produce un error de compilación.

Observación de Ingeniería de Software 6.7

Siempre debemos proporcionar prototipos de funciones, aun y cuando es posible omitirlas cuando se definen las funciones antes de utilizarlas (en cuyo caso, el encabezado de la función actúa también como el prototipo de función). Al proporcionar los prototipos, evitamos fijar el código al orden en el que se definen las funciones (lo cual puede cambiar fácilmente, a medida que un programa evoluciona).

Firmas de funciones

La porción de un prototipo de función que incluya el nombre de la función y los tipos de sus argumentos se conoce como la **firma de la función**, o simplemente **firma**. La firma de la función no especifica el tipo de valor de retorno de la función. Las funciones en el mismo alcance deben tener firmas únicas. El alcance de una función es la región del programa en la que la función se conoce y es accesible. En la sección 6.10 hablaremos más acerca del alcance.



Error común de programación 6.4

Si dos funciones en el mismo alcance tienen la misma firma, pero distintos tipos de valores de retorno, se produce un error de compilación.

En la figura 6.3, si el prototipo en la línea 17 se hubiera escrito como:

```
void maximo( int, int, int );
```

el compilador reportaría un error, ya que el tipo de valor de retorno `void` en el prototipo de la función sería distinto del tipo de valor de retorno `int` en el encabezado de la función. De manera similar, dicho prototipo haría que la instrucción

```
cout << maximo( 6, 7, 0 );
```

genere un error de compilación, ya que esa instrucción depende de `maximo` para devolver un valor que se va a mostrar en pantalla.

Coerción de argumentos

Una característica importante de los prototipos de función es la **coerción de argumentos**; es decir, obligar a que los argumentos tengan los tipos especificados por las declaraciones de los parámetros. Por ejemplo, un programa puede llamar a una función con un argumento entero, aun y cuando el prototipo de función especifica un argumento `double`; la función de todas maneras trabajará correctamente.

Reglas de promoción de argumentos

Algunas veces, los valores de los argumentos que no corresponden precisamente a los tipos de los parámetros en el prototipo de función pueden ser convertidos por el compilador al tipo apropiado, antes de que se haga una llamada a la función. Estas conversiones ocurren según lo especificado por las **reglas de promoción** de C++. Las reglas de promoción indican cómo realizar conversiones entre tipos sin perder datos. Un `int` se puede convertir en `double` sin modificar su valor. Sin embargo, un `double` convertido en `int` trunca la parte fraccionaria del valor `double`. Tenga en cuenta que las variables `double` pueden contener números de una magnitud mucho mayor que las variables `int`, por lo que la pérdida de datos puede ser considerable. Los valores también pueden modificarse al convertir tipos de enteros largos en tipos de enteros pequeños (por ejemplo, de `long` a `short`), números con signo a números sin signo, o viceversa.

Las reglas de promoción se aplican a expresiones que contienen valores de dos o más tipos de datos; dichas expresiones se conocen también como **expresiones de tipo mixto**. El tipo de cada valor en una expresión de tipo mixto se promueve al tipo “más alto” en la expresión (en realidad, se crea y se utiliza una versión temporal de cada valor para la expresión; los valores originales permanecen sin cambios). La promoción también ocurre cuando el tipo de un argumento de función no concuerda con el tipo de parámetro especificado en la definición o prototipo de la función. La figura 6.6 lista los tipos de datos fundamentales en orden del “tipo más alto” al “tipo más bajo”.

La conversión de valores a los tipos fundamentales más bajos puede producir valores incorrectos. Por lo tanto, un valor se puede convertir en un tipo fundamental menor sólo si se asigna de manera explícita el valor a una variable de tipo inferior (algunos compiladores generarán una advertencia en este caso), o mediante el uso de un operador de conversión (vea la sección 4.9). Los valores de los argumentos de una función se convierten a los tipos de los parámetros en un prototipo de función, como si se hubieran asignado de manera directa a las variables de esos tipos. Si se hace una llamada a una función `cuadrado`, que utiliza un parámetro entero, con un argumento de punto flotante, el argumento se convierte a `int` (un tipo más bajo) y `cuadrado` podría devolver un valor incorrecto. Por ejemplo, `square(4.5)` devuelve 16, no 20.25.



Error común de programación 6.5

Al convertir de un tipo de datos mayor en la jerarquía de promociones, a un tipo de datos menor, o entre números con signo y sin signo, se puede corromper el valor de datos, lo cual produce una pérdida de información.

Tipos de datos
<code>long double</code>
<code>double</code>
<code>float</code>
<code>unsigned long int</code> (sinónimo con <code>unsigned long</code>)
<code>long int</code> (sinónimo con <code>long</code>)
<code>unsigned int</code> (sinónimo con <code>unsigned</code>)
<code>int</code>
<code>unsigned short int</code> (sinónimo con <code>unsigned short</code>)
<code>short int</code> (sinónimo con <code>short</code>)
<code>unsigned char</code>
<code>char</code>
<code>bool</code>

Figura 6.6 | Jerarquía de promociones para los tipos de datos fundamentales.



Error común de programación 6.6

Si los argumentos en la llamada a una función no concuerdan con el número y tipos de los parámetros declarados en el prototipo de función correspondiente, se produce un error de compilación. También es un error si el número de argumentos en la llamada concuerda, pero los argumentos no se pueden convertir de manera implícita a los tipos esperados.

6.6 Archivos de encabezado de la Biblioteca estándar de C++

La Biblioteca estándar de C++ está dividida en muchas porciones, cada una con su propio archivo de encabezado. Los archivos de encabezado contienen los prototipos de función para las funciones relacionadas que forman cada porción de la biblioteca. Los archivos de encabezado también contienen definiciones de varios tipos de clases y funciones, así como las constantes que necesitan las funciones. Un archivo de encabezado “instruye” al compilador acerca de cómo interconectarse con los componentes de la biblioteca y los componentes escritos por el usuario.

En la figura 6.7 se listan algunos archivos de encabezado comunes de la Biblioteca estándar de C++, la mayoría de los cuales veremos más adelante en el libro. El término “macro” que se utiliza varias veces en la figura 6.7 se describe con detalle en el apéndice F, Preprocesador. Los nombres de archivos de encabezado que terminen con .h son archivos de encabezado al “estilo anterior”, los cuales han sido suplantados por los archivos de encabezado de la Biblioteca estándar de C++. Sólo utilizaremos las versiones de la Biblioteca estándar de C++ en este libro, para asegurar que nuestros ejemplos funcionen en la mayoría de los compiladores de C++ estándar.

Archivo de encabezado de la Biblioteca estándar de C++	Explicación
<code><iostream></code>	Contiene prototipos de función para las funciones de salida y entrada estándar de C++, presentadas en el capítulo 2, y que se tratan con más detalle en el capítulo 15, Entrada y salida de flujos. Este archivo de encabezado reemplaza al archivo de encabezado <code><iostream.h></code> .
<code><iomanip></code>	Contiene prototipos de función para los manipuladores de flujo que dan formato a flujos de datos. Este archivo de encabezado se utiliza en la sección 4.9 y se analiza con más detalle en el capítulo 15, Entrada y salida de flujos. Este archivo de encabezado reemplaza al archivo de encabezado <code><iomanip.h></code> .
<code><cmath></code>	Contiene prototipos de función para las funciones de la biblioteca de matemáticas (que vimos en la sección 6.3). Este archivo de encabezado reemplaza al archivo de encabezado <code><math.h></code> .

Figura 6.7 | Archivos de encabezado de la Biblioteca estándar de C++. (Parte I de 3).

Archivo de encabezado de la Biblioteca estándar de C++	Explicación
<cstdlib>	Contiene prototipos de función para las conversiones de números a texto, de texto a números, asignación de memoria, números aleatorios y varias otras funciones utilitarias. En la sección 6.7 veremos partes de este archivo de encabezado; también en el capítulo 11, Sobrecarga de operadores: objetos String y Array; en el capítulo 16, Manejo de excepciones; en el capítulo 21, Bits, caracteres, cadenas estilo C y estructuras; y en el apéndice E, Temas sobre código heredado de C. Este archivo de encabezado reemplaza al archivo de encabezado <stdlib.h>.
<ctime>	Contiene prototipos de función y tipos para manipular la hora y la fecha. Este archivo de encabezado reemplaza al archivo de encabezado <time.h>. Se utiliza en la sección 6.7.
<vector>, <list>, <deque>, <queue>, <stack>, <map>, <set>, <bitset>	Estos archivos de encabezado contienen clases que implementan los contenedores de la Biblioteca estándar de C++. Los contenedores almacenan datos durante la ejecución de un programa. El encabezado <vector> se introduce por primera vez en el capítulo 7, Arreglos y vectores. En el capítulo 22, Biblioteca de plantillas estándar (STL) hablaremos sobre todos estos archivos de encabezado.
<cctype>	Contiene prototipos de función para las funciones que evalúan caracteres con base en ciertas propiedades (por ejemplo, si el carácter es un dígito o un signo de puntuación), y prototipos de funciones que se pueden utilizar para convertir letras minúsculas a letras mayúsculas y viceversa. Este archivo de encabezado reemplaza al archivo de encabezado <ctype.h>. Hablaremos sobre estos temas en el capítulo 8, Apuntadores y cadenas basadas en apunadores, y en el capítulo 21, Bits, caracteres, cadenas estilo C y estructuras.
<cstring>	Contiene prototipos de funciones para las funciones de procesamiento de cadenas estilo C. Este archivo de encabezado reemplaza al archivo de encabezado <string.h>. Este archivo de encabezado se utiliza en el capítulo 11, Sobrecarga de operadores: objetos String y Array.
<typeinfo>	Contiene clases para la identificación de tipos en tiempo de ejecución (determinar los tipos de datos en tiempo de ejecución). Este archivo de encabezado se describe en la sección 13.8.
<exception>, <stdexcept>	Estos archivos de encabezado contienen clases que se utilizan para manejar excepciones (se describen en el capítulo 16, Manejo de excepciones).
<memory>	Contiene clases y funciones utilizadas por la Biblioteca estándar de C++ para asignar memoria a los contenedores de la Biblioteca estándar de C++. Este encabezado se utiliza en el capítulo 16, Manejo de excepciones.
<fstream>	Contiene prototipos de funciones para las funciones que realizan operaciones de entrada desde archivos en disco, y operaciones de salida hacia archivos en disco (que veremos en el capítulo 17, Procesamiento de archivos). Este archivo de encabezado reemplaza el archivo de encabezado <fstream.h>.
<string>	Contiene la definición de la clase string de la Biblioteca estándar de C++ (que veremos en el capítulo 18, La clase string y el procesamiento de flujos de cadena).
<iostream>	Contiene prototipos de función para las funciones que realizan operaciones de entrada a partir de cadenas en memoria, y operaciones de salida hacia cadenas en memoria (que veremos en el capítulo 18, La clase string y el procesamiento de flujos de cadena).
<functional>	Contiene las clases y funciones utilizadas por algoritmos de la Biblioteca estándar de C++. Este archivo de encabezado se utiliza en el capítulo 22.
<iterator>	Contiene clases para acceder a los datos en los contenedores de la Biblioteca estándar de C++. Este archivo de encabezado se utiliza en el capítulo 22, Biblioteca de plantillas estándar (STL).
<algorithm>	Contiene las funciones para manipular los datos en los contenedores de la Biblioteca estándar de C++. Este archivo de encabezado se utiliza en el capítulo 22.

Figura 6.7 | Archivos de encabezado de la Biblioteca estándar de C++. (Parte 2 de 3).

Archivo de encabezado de la Biblioteca estándar de C++	Explicación
<code><cassert></code>	Contiene macros para agregar diagnósticos que ayuden a depurar programas. Esto reemplaza al archivo de encabezado <code><assert.h></code> del C++ previo al estándar. Este archivo de encabezado se utiliza en el apéndice F, Preprocesador.
<code><cfloat></code>	Contiene los límites del sistema en cuanto al tamaño de los números de punto flotante. Este archivo de encabezado reemplaza al archivo de encabezado <code><float.h></code> .
<code><climits></code>	Contiene los límites del sistema en cuanto al tamaño de los números enteros. Este archivo de encabezado reemplaza al archivo de encabezado <code><limits.h></code> .
<code><cstdio></code>	Contiene los prototipos de función para las funciones de la biblioteca de entrada/salida estándar de C++ y la información que utilizan. Este archivo de encabezado reemplaza al archivo <code><stdio.h></code> .
<code><locale></code>	Contiene clases y funciones que se utilizan comúnmente en el procesamiento de flujos, para procesar datos en la forma natural para distintos lenguajes (por ejemplo, formatos monetarios, almacenamiento de cadenas, presentación de caracteres, etc.).
<code><limits></code>	Contiene clases para definir los límites de los tipos de datos numéricos en cada plataforma computacional.
<code><utility></code>	Contiene clases y funciones utilizadas por muchos archivos de encabezados de la Biblioteca estándar de C++.

Figura 6.7 | Archivos de encabezado de la Biblioteca estándar de C++. (Parte 3 de 3).

6.7 Ejemplo práctico: generación de números aleatorios

Ahora analizaremos de manera breve una parte divertida de un tipo popular de aplicaciones de la programación: simulación y juegos. En ésta y en la siguiente sección desarrollaremos un programa de juego bien estructurado que incluye varias funciones. El programa utiliza muchas de las instrucciones de control y los conceptos presentados hasta este punto en el libro.

El elemento de azar puede introducirse en las aplicaciones computacionales mediante el uso de la función `rand` de la Biblioteca estándar de C++.

Considere la siguiente instrucción:

```
i = rand();
```

La función `rand` genera un entero sin signo entre 0 y `RAND_MAX` (una constante simbólica definida en el archivo de encabezado `<cstdlib>`). El valor de `RAND_MAX` debe ser por lo menos de 32767: el máximo valor positivo para un entero de dos bytes (16 bits). Para GNU C++, el valor de `RAND_MAX` es 2147483647; para Visual Studio, el valor de `RAND_MAX` es 32767. Si `rand` produce verdaderamente enteros al azar, cada número entre 0 y `RAND_MAX` tiene una *oportunidad* (o *probabilidad*) igual de ser elegido cada vez que se llame a `rand`.

El rango de valores producidos directamente por la función `rand` es a menudo distinto de lo que requiere una aplicación específica. Por ejemplo, un programa que simula el lanzamiento de una moneda sólo requiere 0 para “águila” y 1 para “sol”. Un programa para simular el tiro de un dado de seis lados requeriría enteros aleatorios en el rango de 1 a 6. Un programa que adivine en forma aleatoria el siguiente tipo de nave espacial (de cuatro posibilidades distintas) que volará a lo largo del horizonte en un videojuego requeriría números aleatorios en el rango de 1 a 4.

Tirar un dado de seis lados

Para demostrar la función `rand`, desarrollaremos un programa (figura 6.8) que simula 20 tiros de un dado de seis lados, y que muestra el valor de cada tiro. El prototipo de la función `rand` se encuentra en `<cstdlib>`. Para producir valores enteros en el rango de 0 a 5, usamos el operador módulo (%) con `rand`, como se muestra a continuación:

```
rand() % 6
```

A esto se le conoce como **escalar**. El número 6 se conoce como el **factor de escala**. Después **desplazamos** el rango de números producidos sumando 1 a nuestro resultado anterior. En la figura 6.8 se confirma que los resultados están en el rango de 1 a 6.

```

1 // Fig. 6.8: fig06_08.cpp
2 // Enteros aleatorios desplazados y escalados.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 #include <cstdlib> // contiene el prototipo de función para rand
11 using std::rand;
12
13 int main()
14 {
15     // itera 20 veces
16     for ( int contador = 1; contador <= 20; contador++ )
17     {
18         // elige un número aleatorio de 1 a 6 y lo imprime
19         cout << setw( 10 ) << ( 1 + rand() % 6 );
20
21         // si contador pude dividirse entre 5, empieza una nueva línea de salida
22         if ( contador % 5 == 0 )
23             cout << endl;
24     } // fin de for
25
26     return 0; // indica que terminó correctamente
27 } // fin de main

```

6	6	5	5	6
5	1	1	5	3
6	6	2	4	2
6	2	3	4	1

Figura 6.8 | Enteros desplazados y escalados, producidos por `1 + rand() % 6`.

Tirar un dado de seis lados 6,000,000 de veces

Para mostrar que los números que produce la función `rand` ocurren con una probabilidad aproximadamente igual, la figura 6.9 simula 6,000,000 de tiros de un dado. Cada entero en el rango de 1 a 6 debe aparecer aproximadamente 1,000,000 veces. Esto se confirma en la ventana de resultados, al final de la figura 6.9.

```

1 // Fig. 6.9: fig06_09.cpp
2 // Tiro de un dado de seis lados 6,000,000 veces.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 #include <cstdlib> // contiene el prototipo de la función rand
11 using std::rand;
12
13 int main()
14 {
15     int frecuencia1 = 0; // cuenta de veces que se tiró 1
16     int frecuencia2 = 0; // cuenta de veces que se tiró 2

```

Figura 6.9 | Tiro de un dado de seis lados 6,000,000 veces. (Parte I de 2).

```

17 int frecuencia3 = 0; // cuenta de veces que se tiró 3
18 int frecuencia4 = 0; // cuenta de veces que se tiró 4
19 int frecuencia5 = 0; // cuenta de veces que se tiró 5
20 int frecuencia6 = 0; // cuenta de veces que se tiró 6
21
22 int cara; // almacena el valor que se tiró más recientemente
23
24 // sintetiza los resultados de tirar un dado 6,000,000 veces
25 for ( int tiro = 1; tiro <= 6000000; tiro++ )
26 {
27     cara = 1 + rand() % 6; // número aleatorio del 1 al 6
28
29     // determina el valor del tiro de 1 a 6 e incrementa el contador apropiado
30     switch ( cara )
31     {
32         case 1:
33             ++frecuencia1; // incrementa el contador de 1s
34             break;
35         case 2:
36             ++frecuencia2; // incrementa el contador de 2s
37             break;
38         case 3:
39             ++frecuencia3; // incrementa el contador de 3s
40             break;
41         case 4:
42             ++frecuencia4; // incrementa el contador de 4s
43             break;
44         case 5:
45             ++frecuencia5; // incrementa el contador de 5s
46             break;
47         case 6:
48             ++frecuencia6; // incrementa el contador de 6s
49             break;
50         default: // valor inválido
51             cout << "El programa nunca debio llegar aqui!";
52     } // fin de switch
53 } // fin de for
54
55 cout << "Cara" << setw( 13 ) << "Frecuencia" << endl; // imprime encabezados
56 cout << "    1" << setw( 13 ) << frecuencia1
57     << "\n    2" << setw( 13 ) << frecuencia2
58     << "\n    3" << setw( 13 ) << frecuencia3
59     << "\n    4" << setw( 13 ) << frecuencia4
60     << "\n    5" << setw( 13 ) << frecuencia5
61     << "\n    6" << setw( 13 ) << frecuencia6 << endl;
62 return 0; // indica que terminó correctamente
63 } // fin de main

```

Cara	Frecuencia
1	999702
2	1000823
3	999378
4	998898
5	1000777
6	1000422

Figura 6.9 | Tiro de un dado de seis lados 6,000,000 veces. (Parte 2 de 2).

Como se muestra en los resultados del programa, al escalar y desplazar los valores producidos por `rand`, podemos simular el tiro de un dado de seis lados. Observe que el programa nunca debe llegar al caso `default` (líneas 50 y 51) que se proporciona en la estructura `switch`, ya que la expresión de control del `switch` (`cara`) siempre tiene valores en el rango de 1 a 6; sin embargo, proporcionamos el caso `default` como una cuestión de buena práctica. Una vez que

estudiemos los arreglos en el capítulo 7, le mostraremos cómo reemplazar toda la estructura `switch` de la figura 6.9 de una manera elegante, con una instrucción de una sola línea.



Tip para prevenir errores 6.3

Hay que proporcionar un caso default en una instrucción switch para atrapar errores, ¡incluso si estamos absoluta y positivamente seguros de no tener errores!

Randomización del generador de números aleatorios

Al ejecutar el programa de la figura 6.8 otra vez, se produce lo siguiente:

6	6	5	5	6
5	1	1	5	3
6	6	2	4	2
6	2	3	4	1

Observe que el programa imprime exactamente la misma secuencia de valores que se muestra en la figura 6.8. ¿Cómo pueden ser estos números aleatorios? Irónicamente, esta repetitividad es una característica de la función `rand`. Al depurar un programa de simulación, esta repetitividad es esencial para demostrar que las correcciones al programa funcionan en forma apropiada.

En realidad, la función `rand` genera **números seudoaleatorios**. Si se llama repetidas veces a `rand`, se produce una secuencia de números que parecen ser aleatorios. No obstante, la secuencia se repite a sí misma cada vez que se ejecuta el programa. Una vez que un programa se ha depurado extensivamente, puede condicionarse para producir una secuencia diferente de números aleatorios para cada ejecución. A esto se le conoce como **randomización**, y se logra mediante la función `srand` de la Biblioteca estándar de C++. La función `srand` recibe un argumento entero `unsigned` y **siembra** la función `rand` para que produzca una secuencia distinta de números aleatorios para cada ejecución del programa.

En la figura 6.10 se demuestra el uso de la función `srand`. El programa utiliza el tipo de datos `unsigned`, abreviación de `unsigned int`. Un valor `int` se almacena en al menos dos bytes de memoria (por lo general, cuatro bytes de memoria en los sistemas populares de 32 bits de la actualidad) y puede tener valores positivos y negativos. Una variable de tipo `unsigned int` también se almacena en al menos dos bytes de memoria. Un valor `unsigned int` de dos bytes sólo puede tener valores no negativos en el rango de 0 a 65535. Un valor `unsigned int` de cuatro bytes puede tener sólo valores no negativos en el rango de 0 a 4294967295. La función `srand` recibe un valor `unsigned int` como argumento. El prototipo para la función `srand` se encuentra en el archivo de encabezado `<cstdlib>`.

```

1 // Fig. 6.10: fig06_10.cpp
2 // Randomización de un programa para tirar dados.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 #include <cstdlib> // contiene los prototipos para las funciones srand y rand
12 using std::rand;
13 using std::srand;
14
15 int main()
16 {
17     unsigned seed; // almacena la semilla introducida por el usuario
18
19     cout << "Introduzca la semilla: ";
20     cin >> seed;
21     srand( seed ); // siembra el generador de números aleatorios
22 }
```

Figura 6.10 | Randomización del programa para tirar dados. (Parte 1 de 2).

```

23 // itera 10 veces
24 for ( int contador = 1; contador <= 10; contador++ )
25 {
26     // elige un número aleatorio entre 1 y 6, y lo imprime
27     cout << setw( 10 ) << ( 1 + rand() % 6 );
28
29     // si contador puede dividirse entre 5, empieza una nueva línea de salida
30     if ( contador % 5 == 0 )
31         cout << endl;
32 } // fin de for
33
34 return 0; // indica que terminó correctamente
35 } // fin de main

```

Introduzca la semilla: 67

6	1	4	6	2
1	6	1	6	4

Introduzca la semilla: 432

4	6	3	1	6
3	1	5	4	2

Introduzca la semilla: 67

6	1	4	6	2
1	6	1	6	4

Figura 6.10 | Randomización del programa para tirar dados. (Parte 2 de 2).

Ejecutemos el programa varias veces y observemos los resultados. Observe que el programa produce una secuencia *distinta* de números aleatorios cada vez que se ejecuta, siempre y cuando el usuario introduzca una semilla distinta. Utilizamos la misma semilla en la primera y la tercera ventana de resultados, por lo que se muestra la misma serie de 10 números en cada una de esas ventanas.

Para randomizar sin tener que introducir una semilla cada vez, podemos usar una instrucción como la siguiente:

```
 srand( time( 0 ) );
```

Esto hace que la computadora lea su reloj para obtener el valor para la semilla. La función `time` (con el argumento 0, como se escribe en la instrucción anterior) devuelve la hora actual como el número de segundos transcurridos desde enero 1, 1970, a media noche en Tiempo del Meridiano de Greenwich (GMT). Este valor se convierte en un entero `unsigned` y se utiliza como semilla para el generador de números aleatorios. El prototipo de la función `time` está en `<ctime>`.

Error común de programación 6.7

 Al llamar a la función `srand` más de una vez en un programa, se reinicia la secuencia del número seudoaleatorio y puede afectar la característica aleatoria de los números producidos por `rand`.

Escalamiento y desplazamiento generalizados de números aleatorios

Anteriormente demostramos cómo escribir una sola instrucción para simular cómo tirar un dado de seis lados con la instrucción:

```
cara = 1 + rand() % 6;
```

la cual siempre asigna a la variable `cara` un entero en el rango $1 \leq cara \leq 6$. Observe que la amplitud de este rango (es decir, el número de enteros consecutivos en el rango) es 6, y el número inicial en el rango es 1. Si hacemos referencia a la instrucción anterior, podemos ver que la amplitud del rango se determina con base en el número que se utiliza para escalar `rand` con el operador módulo (es decir, 6), y el número inicial del rango es igual al número (es decir, 1) que se agrega a la expresión `rand % 6`. Podemos generalizar este resultado de la siguiente manera:

```
numero = valorDesplazamiento + rand() % factorEscala;
```

donde *valorDesplazamiento* especifica el primer número en el rango deseado de enteros consecutivos y *factorEscala* es igual al rango deseado de enteros consecutivos. Los ejercicios muestran que es posible elegir enteros al azar, a partir de conjuntos de valores distintos de los rangos de enteros consecutivos.

Error común de programación 6.8



Usar srand en vez de rand para tratar de generar números aleatorios es un error de compilación; la función srand no devuelve un valor.

6.8 Ejemplo práctico: juego de probabilidad, introducción a las enumeraciones

Uno de los juegos de azar más populares es el juego de dados conocido como “craps”, el cual se juega en casinos y callejones por todo el mundo. Las reglas del juego son simples:

Un jugador tira dos dados. Cada dado tiene seis caras, las cuales contienen 1, 2, 3, 4, 5 y 6 puntos negros. Una vez que los dados dejan de moverse, se calcula la suma de los puntos negros en las dos caras superiores. Si la suma es 7 u 11 en el primer tiro, el jugador gana. Si la suma es 2, 3 o 12 en el primer tiro (llamado “craps”), el jugador pierde (es decir, la “casa” gana). Si la suma es 4, 5, 6, 8, 9 o 10 en el primer tiro, esta suma se convierte en el “punto” del jugador. Para ganar, el jugador debe seguir tirando los dados hasta que salga otra vez “su punto”. El jugador pierde si tira un 7 antes de llegar a su punto.

El programa de la figura 6.11 simula el juego de craps.

En las reglas del juego, observe que el jugador debe tirar dos dados en el primer tiro y en todos los tiros subsiguientes. Definimos la función *tirarDado* (líneas 71 a 83) para tirar el dado, calcular e imprimir la suma. La función *tirarDado* está definida sólo una vez, pero se llama desde dos lugares (líneas 27 y 51) en el programa. Lo interesante es que *tirarDados* no recibe argumentos, por lo que hemos indicado una lista de parámetros vacía en el prototipo (línea 14) y en el encabezado de función (línea 71). La función *tirarDado* devuelve la suma de los dos dados, por lo que el tipo de valor de retorno *int* se indica en el prototipo de la función y en el encabezado de la misma.

```

1 // Fig. 6.11: fig06_11.cpp
2 // Simulación del juego de dados "craps".
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // contiene los prototipos para las funciones srand y rand
8 using std::rand;
9 using std::srand;
10
11 #include <ctime> // contiene el prototipo para la función time
12 using std::time;
13
14 int tirarDados(); // tira los dados, calcula y muestra la suma
15
16 int main()
17 {
18     // enumeración con constantes que representa el estado del juego
19     enum Estado { CONTINUAR, GANO, PERDIO }; // todas las constantes en mayúsculas
20
21     int miPunto; // punto si no se gana o pierde en el primer tiro
22     Estado estadoJuego; // puede contener CONTINUAR, GANO o PERDIO
23
24     // randomiza el generador de números aleatorios, usando la hora actual
25     srand( time( 0 ) );
26 }
```

Figura 6.11 | Simulación del juego de “craps”. (Parte I de 3).

```

27     int sumaDeDados = tirarDados(); // primer tiro del dado
28
29     // determina el estado del juego y el punto (si es necesario), con base en el primer tiro
30     switch ( sumaDeDados )
31     {
32         case 7: // gana con 7 en el primer tiro
33         case 11: // gana con 11 en el primer tiro
34             estadoJuego = GANO;
35             break;
36         case 2: // pierde con 2 en el primer tiro
37         case 3: // pierde con 3 en el primer tiro
38         case 12: // pierde con 12 en el primer tiro
39             estadoJuego = PERDIO;
40             break;
41         default: // no ganó ni perdió, por lo que recuerda el punto
42             estadoJuego = CONTINUAR; // el juego no ha terminado
43             miPunto = sumaDeDados; // recuerda el punto
44             cout << "El punto es " << miPunto << endl;
45             break; // opcional al final del switch
46     } // fin de switch
47
48     // mientras el juego no esté completo
49     while ( estadoJuego == CONTINUAR ) // no ganó ni perdió
50     {
51         sumaDeDados = tirarDados(); // tira los dados de nuevo
52
53         // determina el estado del juego
54         if ( sumaDeDados == miPunto ) // gana al hacer un punto
55             estadoJuego = GANO;
56         else
57             if ( sumaDeDados == 7 ) // pierde al tirar 7 antes del punto
58                 estadoJuego = PERDIO;
59     } // fin de while
60
61     // muestra mensaje de que ganó o perdió
62     if ( estadoJuego == GANO )
63         cout << "El jugador gana" << endl;
64     else
65         cout << "El jugador pierde" << endl;
66
67     return 0; // indica que terminó correctamente
68 } // fin de main
69
70 // tira los dados, calcula la suma y muestra los resultados
71 int tirarDados()
72 {
73     // elige valores aleatorios para el dado
74     int dado1 = 1 + rand() % 6; // tiro del primer dado
75     int dado2 = 1 + rand() % 6; // tiro del segundo dado
76
77     int suma = dado1 + dado2; // calcula la suma de valores de los dados
78
79     // muestra los resultados de este tiro
80     cout << "El jugador tiro " << dado1 << " + " << dado2
81     << " = " << suma << endl;
82     return suma; // devuelve la suma de los dados
83 } // fin de la función tirarDados

```

El jugador tiro 2 + 5 = 7
 El jugador gana

Figura 6.11 | Simulación del juego de “craps”. (Parte 2 de 3).

```
El jugador tiro 6 + 6 = 12
El jugador pierde
```

```
El jugador tiro 3 + 3 = 6
El punto es 6
El jugador tiro 5 + 3 = 8
El jugador tiro 4 + 5 = 9
El jugador tiro 2 + 1 = 3
El jugador tiro 1 + 5 = 6
El jugador gana
```

```
El jugador tiro 1 + 3 = 4
El punto es 4
El jugador tiro 4 + 6 = 10
El jugador tiro 2 + 4 = 6
El jugador tiro 6 + 4 = 10
El jugador tiro 2 + 3 = 5
El jugador tiro 2 + 4 = 6
El jugador tiro 1 + 1 = 2
El jugador tiro 4 + 4 = 8
El jugador tiro 4 + 3 = 7
El jugador pierde
```

Figura 6.11 | Simulación del juego de “craps”. (Parte 3 de 3).

El juego es razonablemente complejo. El jugador puede ganar o perder en el primer tiro, o en cualquier tiro subsiguiente. El programa utiliza la variable local `estadoJuego` para llevar la cuenta de esto. La variable `estadoJuego` se declara como del nuevo tipo `Estado`. En la línea 19 se declara un tipo definido por el usuario, llamado `enumeración`. Una enumeración, que se introduce mediante la palabra clave `enum` y va seguida de un `nombre de tipo` (en este caso, `Estado`), es un conjunto de constantes enteras representadas por identificadores. Los valores de estas `constants de enumeración` empiezan en 0, a menos que se especifique lo contrario, y se incrementan en 1. En la enumeración anterior, la constante `CONTINUAR` tiene el valor 0, `GANO` tiene el valor 1 y `PERDIO` tiene el valor 2. Los identificadores en una enumeración deben ser únicos, pero las constantes de enumeración separadas pueden tener el mismo valor entero (en un momento le mostraremos cómo hacer esto).



Buena práctica de programación 6.1

La primera letra de un identificador que se utilice como un nombre de tipo definido por el usuario debe ir en mayúscula.



Buena práctica de programación 6.2

Use sólo letras mayúsculas en los nombres de las constantes de enumeración. Esto hace que resalten y le recuerdan que las constantes de enumeración no son variables.

Las variables del tipo definido por el usuario `Estado` pueden recibir sólo uno de los tres valores declarados en la enumeración. Cuando se gana el juego, el programa establece la variable `estadoJuego` a `GANO` (líneas 34 y 55). Cuando se pierde el juego, el programa establece la variable `estadoJuego` a `PERDIO` (líneas 39 a 58). En caso contrario, el programa establece la variable `estadoJuego` a `CONTINUAR` (línea 42) para indicar que el dado se debe tirar de nuevo.

Otra enumeración popular es

```
enum Meses { ENE = 1, FEB, MAR, ABR, MAY, JUN, JUL, AGO,
SEP, OCT, NOV, DEC };
```

la cual crea el tipo definido por el usuario `Meses`, con constantes de enumeración que representan los meses del año. El primer valor en la enumeración anterior se establece explícitamente en 1, por lo que el resto de los valores se incrementan a partir de 1, lo cual produce los valores del 1 al 12. Cualquier constante de enumeración puede recibir un valor entero en la definición de la enumeración, y cada una de las constantes de enumeración subsiguientes tienen un valor igual a 1 más que la constante anterior en la lista, hasta la siguiente configuración explícita.

Después del primer tiro, si se gana o pierde el juego, el programa omite el cuerpo de la instrucción `while` (líneas 49 a 59) debido a que `estadoJuego` no es igual a `CONTINUAR`. El programa pasa a la instrucción `if...else` en las líneas 62 a 65, que imprime "El jugador gana" si `estadoJuego` es igual a `GANO` y "El jugador pierde" si `estadoJuego` es igual a `PERDIO`.

Después del primer tiro, si el juego no ha terminado, el programa guarda la suma en `miPunto` (línea 43). La ejecución continúa con la instrucción `while`, ya que `estadoJuego` es igual a `CONTINUAR`. Durante cada iteración del `while`, el programa llama a `tirarDados` para producir una nueva suma. Si suma concuerda con `miPunto`, el programa establece `estadoJuego` en `GANO` (línea 55), la prueba del `while` falla, la instrucción `if...else` imprime "El jugador gana" y termina la ejecución. Si suma es igual a 7, el programa establece `estadoJuego` a `PERDIO` (línea 58), falla la prueba del `while`, la instrucción `if...else` imprime "El jugador pierde" y termina la ejecución.

Observe el uso interesante de los diversos mecanismos de control de programas que hemos descrito. El programa de "craps" utiliza dos funciones (`main` y `tirarDados`) y las instrucciones `switch`, `while`, `if...else`, `if...else` anidadas e `if` anidadas. En los ejercicios investigamos varias características interesantes del juego de "craps".



Buena práctica de programación 6.3

El uso de enumeraciones en vez de constantes enteras puede hacer que los programas sean más claros y fáciles de mantener. Puede establecer el valor de una constante de enumeración una vez, en la declaración de la enumeración.



Error común de programación 6.9

Asignar el equivalente entero de una constante de enumeración (en vez de la misma constante de enumeración) a una variable del tipo de enumeración es un error de compilación.



Error común de programación 6.10

Una vez que se ha definido una constante de enumeración, tratar de asignar otro valor a la constante de enumeración es un error de compilación.

6.9 Clases de almacenamiento

Los programas que hemos visto hasta ahora utilizan identificadores para nombres de variables. Los atributos de las variables incluyen su nombre, tipo, tamaño y valor. Este capítulo también utiliza identificadores como nombres para las funciones definidas por el usuario. En la actualidad, cada identificador en un programa tiene otros atributos, incluyendo la clase de almacenamiento, el alcance y la vinculación.

C++ proporciona cinco especificadores de clase de almacenamiento: `auto`, `register`, `extern`, `mutable` y `static`. En esta sección hablaremos sobre los especificadores de clase de almacenamiento `auto`, `register`, `extern` y `static`. El especificador de clase de almacenamiento `mutable` (que veremos con detalle en el capítulo 25, Otros temas) se utiliza exclusivamente con las clases.

Clase de almacenamiento, alcance y vinculación

La clase de almacenamiento de un identificador determina el periodo durante el cual éste existe en la memoria. Algunos identificadores existen brevemente, algunos se crean y destruyen repetidas veces, y otros existen durante toda la ejecución de un programa. Primero hablaremos sobre las clases de almacenamiento `static` y `automatic`.

El alcance de un identificador es la parte en la que se puede hacer referencia a éste en un programa. Se puede hacer referencia a algunos identificadores a lo largo de un programa; otros identificadores sólo se pueden referenciar desde ciertas partes limitadas de un programa. En la sección 6.10 hablaremos sobre el alcance de los identificadores.

La vinculación de un identificador determina si se conoce sólo en el archivo fuente en el que se declara, o en varios archivos fuente que se compilen y después se enlacen. El especificador de clase de almacenamiento de un identificador ayuda a determinar su clase de almacenamiento y su vinculación.

Categorías de clases de almacenamiento

Los especificadores de clases de almacenamiento se pueden dividir en dos clases de almacenamiento: clase de almacenamiento automático y clase de almacenamiento estático. Las palabras clave `auto` y `register` se utilizan para declarar variables de la clase de almacenamiento automático. Dichas variables se crean cuando la ejecución del programa entra en el bloque en el que están definidas, existen mientras el bloque está activo y se destruyen cuando el programa sale del bloque.

Variables locales

Sólo las variables locales de una función pueden ser de clase de almacenamiento automático. Las variables locales de una función y los parámetros generalmente son de clase de almacenamiento automático. El especificador de clase de almacenamiento `auto` declara explícitamente variables de clase de almacenamiento automático. Por ejemplo, la siguiente declaración indica que la variable `double x` es una variable local de clase de almacenamiento automático; sólo existe en el par circundante más cercano de llaves dentro del cuerpo de la función en la cual aparece la definición:

```
auto double x;
```

Las variables locales son de clase de almacenamiento automático de manera predeterminada, por lo que la palabra clave `auto` se utiliza raras veces. Durante el resto de este libro, nos referiremos a las variables de clase de almacenamiento automático simplemente como variables automáticas.



Tip de rendimiento 6.1

El almacenamiento automático es un medio de conservar la memoria, ya que las variables de clase de almacenamiento automático existen en memoria sólo cuando se ejecuta el bloque en el cual están definidas.



Observación de Ingeniería de Software 6.8

El almacenamiento automático es un ejemplo del principio del menor privilegio, el cual es fundamental para la buena ingeniería de software. En el contexto de una aplicación, el principio establece que el código debe recibir sólo el nivel de privilegio y acceso que requiere para realizar su tarea designada, pero no más. ¿Por qué deberíamos tener variables almacenadas en memoria y accesibles cuando no se necesitan?

Variables de registro

Los datos en la versión de lenguaje máquina de un programa se cargan generalmente en los registros, para cálculos y otros tipos de procesamiento.



Tip de rendimiento 6.2

El especificador de clase de almacenamiento `register` se puede colocar antes de la declaración de una variable automática, para sugerir que el compilador debe mantener la variable en uno de los registros de hardware de alta velocidad de la computadora, en vez de hacerlo en memoria. Si las variables de uso intensivo, como los contadores o totales, se mantienen en los registros de hardware, se elimina la sobrecarga de cargar de manera repetitiva las variables de memoria hacia los registros, y almacenar los resultados de vuelta a la memoria.



Error común de programación 6.1 I

El uso de varios especificadores de clase de almacenamiento para un identificador es un error de sintaxis. Sólo se puede aplicar un especificador de clase de almacenamiento a un identificador. Por ejemplo, si incluye `register`, no incluya también `auto`.

El compilador podría ignorar las declaraciones `register`. Por ejemplo, tal vez no haya un número suficiente de registros para que el compilador pueda usarlos. La siguiente definición sugiere que la variable entera `contador` se coloque en uno de los registros de la computadora; sin importar que el compilador haga esto o no, `contador` se inicializa en 1:

```
register int contador = 1;
```

La palabra clave `register` se puede utilizar sólo con variables locales y parámetros de funciones.



Tip de rendimiento 6.3

A menudo, no es necesario usar `register`. Los compiladores optimizadores pueden reconocer las variables de uso frecuente y colocarlas en los registros, sin necesitar una declaración `register`.

Clase de almacenamiento estático

Las palabras clave `extern` y `static` declaran identificadores para variables de la clase de almacenamiento estático y para funciones. Las variables de clase de almacenamiento estático existen a partir del punto en el que el programa empieza a ejecutarse, y dejan de existir cuando termina el programa. El almacenamiento de una variable de clase de almacenamiento estático se asigna cuando el programa empieza su ejecución. Dicha variable se inicializa una vez al encontrar su

declaración. Para las funciones, el nombre de la función existe cuando el programa empieza a ejecutarse, de igual forma que para las otras funciones. Sin embargo, aun y cuando los nombres de las variables y funciones existen desde el inicio de la ejecución del programa, esto no implica que estos identificadores se puedan utilizar a lo largo de todo el programa. La clase de almacenamiento y el alcance (donde se puede usar un nombre) son cuestiones separadas, como veremos en la sección 6.10.

Identificadores con clase de almacenamiento estático

Hay dos tipos de identificadores con clase de almacenamiento estático; los identificadores externos (como las **variables globales** y los nombres de funciones globales) y las variables locales declaradas con el especificador de clase de almacenamiento **static**. Para crear variables globales, se colocan declaraciones de variables fuera de cualquier definición de clase o función. Las variables globales retienen sus valores a lo largo de la ejecución del programa. Las variables y las funciones globales se pueden referenciar mediante cualquier función que siga sus declaraciones o definiciones en el archivo fuente.



Observación de Ingeniería de Software 6.9

Al declarar una variable como global en vez de local, se permite la ocurrencia de efectos secundarios inesperados cuando una función que no requiere acceso a la variable la modifica en forma accidental o premeditada. Esto es otro ejemplo del principio del menor privilegio. En general, con la excepción de los recursos verdaderamente globales como cin y cout, debe evitarse el uso de variables globales, excepto en ciertas situaciones con requerimientos de rendimiento únicos.



Observación de Ingeniería de Software 6.10

Las variables que se utilizan sólo en una función específica deben declararse como locales en esa función, en vez de declararlas como variables globales.

Las variables locales que se declaran con la palabra clave **static** siguen siendo conocidas sólo en la función en la que se declaran, pero a diferencia de las variables automáticas, las variables locales **static** retienen sus valores cuando la función regresa a la función que la llamó. La próxima vez que se hace una llamada a la función, las variables locales **static** contienen los valores que tenían cuando la función se ejecutó por última vez. La siguiente instrucción declara la variable local **cuenta** como **static**, y la inicializa en 1:

```
static int cuenta = 1;
```

Todas las variables numéricas de la clase de almacenamiento estático se inicializan con cero si el programador no las inicializa de manera explícita, pero sin duda es una buena práctica inicializar todas las variables en forma explícita.

Los especificadores de clase de almacenamiento **extern** y **static** tienen un significado especial cuando se aplican de manera explícita a los identificadores externos, como las variables globales y los nombres de funciones globales. En el apéndice E, Temas sobre código heredado de C, hablaremos sobre el uso de **extern** y **static** con identificadores externos y programas con varios archivos de código fuente.

6.10 Reglas de alcance

La porción del programa en la que se puede utilizar un identificador se conoce como su alcance. Por ejemplo, cuando declaramos una variable local en un bloque, sólo se puede referenciar en ese bloque y en los bloques anidados dentro de ese mismo bloque. En esta sección hablaremos sobre cuatro tipos de alcance para un identificador: **alcance de función**, **alcance de archivo**, **alcance de bloque** y **alcance de prototipo de función**. Más adelante veremos otros dos tipos de alcance: **alcance de clase** (capítulo 9) y **alcance de espacio de nombres** (capítulo 25).

Un identificador que se declara fuera de cualquier función o clase tiene alcance de archivo. Dicho identificador se “conoce” en todas las funciones a partir del punto en el que se declara, hasta llegar al final del archivo. Las variables globales, las definiciones de funciones y los prototipos de funciones que se colocan fuera de una función tienen alcance de archivo.

Las **etiquetas** (identificadores seguidos por un punto y coma, como **inicio:**) son los únicos identificadores con alcance de función. Las etiquetas se pueden utilizar en cualquier parte en la función en la que aparecen, pero no se pueden referenciar fuera del cuerpo de la función. Las etiquetas se utilizan en instrucciones **goto** (apéndice E). Las etiquetas son detalles de implementación que las funciones ocultan unas de otras.

Los identificadores que se declaran dentro de un bloque tienen alcance de bloque. El alcance de bloque empieza en la declaración del identificador y termina en la llave derecha de finalización (**}**) del bloque en el que se declara el identificador. Las variables locales tienen alcance de bloque, al igual que los parámetros de las funciones, que también son

variables locales de la función. Cualquier bloque puede contener declaraciones de variables. Cuando los bloques están anidados y un identificador en un bloque exterior tiene el mismo nombre que un identificador en un bloque interior, el identificador del bloque exterior se “oculta” hasta que termine el bloque interior. Mientras se ejecuta el bloque interior, éste ve el valor de su propio identificador local y no el valor del identificador del bloque circundante que tiene el nombre idéntico. Las variables locales declaradas como `static` siguen teniendo alcance de bloque, aun y cuando existan desde el momento en que el programa empieza su ejecución. La duración del almacenamiento no afecta al alcance de un identificador.

Los únicos identificadores con alcance de prototipo de función son los que se utilizan en la lista de parámetros de un prototipo de función. Como se mencionó antes, los prototipos de función no requieren nombres en la lista de parámetros, sólo los tipos. El compilador ignora los nombres que aparecen en la lista de parámetros de un prototipo de función. Los identificadores que se utilizan en un prototipo de función se pueden usar en cualquier otra parte del programa sin ambigüedad. En un solo prototipo, un identificador específico sólo se puede utilizar una vez.

Error común de programación 6.12



El uso accidental del mismo nombre para un identificador en un bloque interior, que se utiliza para un identificador en un bloque exterior, cuando de hecho deseamos que el identificador en el bloque exterior esté activo durante la ejecución del bloque interior, es comúnmente un error lógico.

Buena práctica de programación 6.4



Evite los nombres de variables que ocultan nombres en alcances exteriores. Para lograr esto, hay que evitar el uso de identificadores duplicados en un programa.

El programa de la figura 6.12 demuestra cuestiones sobre el alcance con las variables globales, variables locales automáticas y variables locales `static`.

```

1 // Fig. 6.12: fig06_12.cpp
2 // Un ejemplo sobre el alcance.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void usarLocal(); // prototipo de función
8 void usarLocalStatic(); // prototipo de función
9 void usarGlobal(); // prototipo de función
10
11 int x = 1; // variable global
12
13 int main()
14 {
15     cout << "la x global en main es " << x << endl;
16
17     int x = 5; // variable local para main
18
19     cout << "la x local en el alcance exterior de main es " << x << endl;
20
21     { // empieza nuevo alcance
22         int x = 7; // oculta la x en el alcance exterior y la x global
23
24         cout << "la x local en el alcance interior de main es " << x << endl;
25     } // termina nuevo alcance
26
27     cout << "la x local en el alcance exterior de main es " << x << endl;
28
29     usarLocal(); // usarLocal tiene la x local
30     usarLocalStatic(); // usarLocalStatic tiene la x local estática
31     usarGlobal(); // usarGlobal usa la x global

```

Figura 6.12 | Ejemplo sobre el alcance. (Parte 1 de 2).

```
32     usarLocal(); // usarLocal reinicializa su x local
33     usarLocalStatic(); // la x local estática retiene su valor anterior
34     usarGlobal(); // la x global también retiene su valor anterior
35
36     cout << "\nla x local en main es " << x << endl;
37     return 0; // indica que terminó correctamente
38 } // fin de main
39
40 // usarLocal reinicializa la variable x local durante cada llamada
41 void usarLocal()
42 {
43     int x = 25; // se inicializa cada vez que se llama a usarLocal
44
45     cout << "\nla x local es " << x << " al entrar a usarLocal" << endl;
46     x++;
47     cout << "la x local es " << x << " al salir de usarLocal" << endl;
48 } // fin de la función usarLocal
49
50 // usarLocalStatic inicializa la variable x local estática sólo la
51 // primera vez que se llama a la función; el valor de x se guarda
52 // entre las llamadas a esta función
53 void usarLocalStatic()
54 {
55     static int x = 50; // se inicializa la primera vez que se llama a usarLocalStatic
56
57     cout << "\nla x local estatica es " << x << " al entrar a usarLocalStatic"
58         << endl;
59     x++;
60     cout << "la x local estatica es " << x << " al salir de usarLocalStatic"
61         << endl;
62 } // fin de la función usarLocalStatic
63
64 // usarGlobal modifica la variable global x durante cada llamada
65 void usarGlobal()
66 {
67     cout << "\nla x global es " << x << " al entrar a usarGlobal" << endl;
68     x *= 10;
69     cout << "la x global es " << x << " al salir de usarGlobal" << endl;
70 } // fin de la función usarGlobal
```

```
la x global en main es 1
la x local en el alcance exterior de main es 5
la x local en el alcance interior de main es 7
la x local en el alcance exterior de main es 5

la x local es 25 al entrar a usarLocal
la x local es 26 al salir de usarLocal

la x local estatica es 50 al entrar a usarLocalStatic
la x local estatica es 51 al salir de usarLocalStatic

la x global es 1 al entrar a usarGlobal
la x global es 10 al salir de usarGlobal

la x local es 25 al entrar a usarLocal
la x local es 26 al salir de usarLocal

la x local estatica es 51 al entrar a usarLocalStatic
la x local estatica es 52 al salir de usarLocalStatic

la x global es 10 al entrar a usarGlobal
la x global es 100 al salir de usarGlobal

la x local en main es 5
```

Figura 6.12 | Ejemplo sobre el alcance. (Parte 2 de 2).

En la línea 11 se declara e inicializa la variable global `x` en 1. Esta variable local está oculta en cualquier bloque (o función) que declare una variable llamada `x`. En `main`, la línea 15 muestra el valor de la variable global `x`. En la línea 17 se declara una variable local `x` y se inicializa en 5. En la línea 19 se imprime esta variable para mostrar que la `x` global está oculta en `main`. A continuación, en las líneas 21 a 25 se define un nuevo bloque en `main`, en el cual otra variable local `x` se inicializa con 7 (línea 22). En la línea 24 se imprime esta variable, para mostrar que oculta a `x` en el bloque exterior de `main`. Cuando el bloque termina, la variable `x` que tiene el valor 7 se destruye automáticamente. A continuación, en la línea 27 se imprime la variable local `x` en el bloque exterior de `main`, para mostrar que ya no está oculta.

Para demostrar otros alcances, el programa define tres funciones, cada una de las cuales no recibe argumento y no devuelve nada. La función `usarLocal` (líneas 41 a 48) declara la variable automática `x` (línea 43) y la inicializa en 25. Cuando el programa llama a `usarLocal`, la función imprime la variable, la incrementa y la vuelve a imprimir antes de que la función devuelva el control a la función que la llamó. Cada vez que el programa llama a esta función, ésta vuelve a crear automáticamente la variable `x` y la inicializa en 25.

La función `usarLocalStatic` (líneas 53 a 62) declara la variable `static x` y la inicializa con 50. Las variables locales que se declaran como `static` retienen su valores, aun y cuando están fuera de alcance (es decir, la función en la que se declaran no se está ejecutando). Cuando el programa llama a `usarLocalStatic`, la función imprime `x`, la incrementa y la vuelve a imprimir antes de que la función devuelva el control del programa a la función que la llamó. En la siguiente llamada a esta función, la variable local `static x` contiene el valor 51. La inicialización en la línea 55 ocurre sólo una vez; la primera vez que se llama a `usarLocalStatic`.

La función `usarGlobal` (líneas 65 a 70) no declara ninguna variable. Por lo tanto, cuando hace referencia a la variable `x`, se utiliza la `x` global (línea 11, antes de `main`). Cuando el programa llama a `usarGlobal`, la función imprime la variable global `x`, la multiplica por 10 y la imprime de nuevo, antes de que la función devuelva el control del programa a la función que la llamó. La siguiente vez que el programa llama a `usarGlobal`, se modifica el valor de la variable global, 10. Después de ejecutar las funciones `usarLocal`, `usarLocalStatic` y `usarGlobal` dos veces cada una, el programa imprime la variable local `x` en `main`, de nuevo para mostrar que ninguna de las llamadas a la función modificó el valor de `x` en `main`, debido a que todas las funciones hicieron referencia a las variables en otros alcances.

6.11 La pila de llamadas a funciones y los registros de activación

Para comprender la forma en que C++ realiza las llamadas a las funciones, primero necesitamos considerar una estructura de datos (es decir, colección de elementos de datos relacionados) conocida como **pila**. Piense en una pila como la analogía a una pila de platos. Cuando se coloca un plato en la pila, por lo general se coloca en la parte superior (lo que se conoce como **meter el plato en la pila**). De manera similar, cuando se extrae un plato de la pila, siempre se extrae de la parte superior (lo que se conoce como **sacar el plato de la pila**). Las pilas se denominan **estructuras de datos “último en entrar, primero en salir” (UEPS)**; el último elemento que se mete (inserta) en la pila es el primero que se saca (extrae) de ella.

Uno de los mecanismos más importantes que los estudiantes de ciencias computacionales deben comprender es la **pila de llamadas a funciones** (conocida algunas veces como la **pila de ejecución del programa**). Esta estructura de datos (que trabaja en segundo plano) soporta el mecanismo de llamada a/regreso de las funciones. También soporta la creación, mantenimiento y destrucción de las variables automáticas de cada función a la que se llama. Explicamos el comportamiento “último en entrar, primero en salir (UEPS)” de las pilas con nuestro ejemplo de apilar platos. Como veremos en las figuras 6.14 a 6.16, este comportamiento UEPS es exactamente lo que hace una función cuando regresa a la función que la llamó.

A medida que se hace la llamada a cada función, ésta puede a su vez, llamar a otras funciones, la cual, a su vez, puede llamar a otras funciones; todo ello antes de que regrese alguna de las funciones. En cierto momento, cada función debe regresar el control a la función que la llamó. Por ende, de alguna manera debemos llevar el registro de las direcciones de retorno que requiere cada función para regresar el control a la función que la llamó. La pila de llamadas a funciones es la estructura de datos perfecta para manejar esta información. Cada vez que una función llama a otra función, se mete una entrada en la pila. Esta entrada, conocida como **marco de pila o registro de activación**, contiene la dirección de retorno que necesita la función a la que se llamó para poder regresar a la función que hizo la llamada. También contiene cierta información adicional que veremos en breve. Si la función a la que se llamó regresa, en vez de llamar a otra función antes de regresar, se saca el marco de pila para la llamada a la función, y el control se transfiere a la dirección de retorno en el marco de la pila que se sacó.

La belleza de la pila de llamadas es que cada función a la que se ha llamado siempre encuentra la información que requiere para regresar a la función que la llamó en la parte superior de la pila de llamadas. Y, si una función hace una llamada a otra función, simplemente se mete a la pila de llamadas un marco de pila para esa nueva llamada a una función. Por ende, la dirección de retorno requerida por la función recién llamada para regresar a la función que la llamó se encuentra ahora en la parte superior de la pila.

Los marcos de pila tienen otra responsabilidad importante. La mayoría de las funciones tienen variables automáticas: parámetros y cualquier variable local que declare la función. Las variables automáticas necesitan existir mientras una función se está ejecutando. Necesitan permanecer activas si la función hace llamadas a otras funciones. Pero cuando una función a la que se llamó regresa a la función que la llamó, las variables automáticas de la función a la que se llamó necesitan “desaparecer”. El marco de pila de la función a la que se llamó es un lugar perfecto para reservar la memoria para las variables automáticas de la función a la que se llamó. Ese marco de pila existe mientras la función a la que se llamó esté activa. Cuando esa función regresa (y ya no necesita sus variables automáticas locales) su marco de pila se saca de la pila, y esas variables automáticas ya no son conocidas para el programa.

Desde luego que la cantidad de memoria en una computadora es finita, por lo cual sólo se puede usar cierta cantidad de memoria para almacenar registros de activación en la pila de llamadas a funciones. Si ocurren más llamadas a funciones de las que se puedan guardar sus registros de activación en la pila de llamadas a funciones, se produce un error conocido como **desbordamiento de pila**.

La pila de llamadas a funciones en acción

Así, como hemos visto, la pila de llamadas y los registros de activación soportan el mecanismo de llamadas a/retorno de las funciones, además de la creación y destrucción de variables automáticas. Ahora vamos a considerar cómo la pila de llamadas ofrece soporte para la operación de una función cuadrado llamada por `main` (líneas 11 a 17 de la figura 6.13). Primero, el sistema operativo llama a `main`; esto hace que se meta un registro de activación en la pila (lo cual se muestra en la figura 6.14). El registro de activación indica a `main` cómo debe regresar al sistema operativo (es decir, transferir el control a la dirección de retorno R1) y contiene el espacio para la variable automática de `main` (`a`, que se inicializa en 10).

La función `main` (antes de regresar al sistema operativo) llama ahora a la función `cuadrado` en la línea 15 de la figura 6.13. Esto hace que se meta un marco de pila para `cuadrado` (líneas 20 a 23) en la pila de llamadas a funciones (figura 6.15). Este marco de pila contiene la dirección de retorno que `cuadrado` necesita para regresar a `main` (es decir, `R2`) y la memoria para la variable automática de `cuadrado` (es decir, `x`).

Una vez que `cuadrado` calcula el cuadrado de su argumento, necesita regresar a `main`; y ya no necesita la memoria para su variable automática, `x`. Por ende, se saca un elemento de la pila, con lo cual se proporciona a `cuadrado` la dirección de retorno en `main` (es decir, `R2`) y se pierde la variable automática de `cuadrado`. La figura 6.16 muestra la pila de llamadas a funciones después de sacar el registro de activación de `cuadrado`.

Ahora la función `main` muestra el resultado de llamar a `cuadrado` (línea 15), y después ejecuta la instrucción `return` (línea 16). Esto hace que el registro de activación para `main` se saque de la pila. Esto proporciona a `main` la dirección

```

1 // Fig. 6.13: fig06_13.cpp
2 // función cuadrado utilizada para demostrar la pila
3 // de llamadas a funciones y los registros de activación.
4 #include <iostream>
5 using std::cin;
6 using std::cout;
7 using std::endl;
8
9 int cuadrado( int ); // prototipo para la función cuadrado
10
11 int main()
12 {
13     int a = 10; // valor para cuadrado (variable local automática en main)
14
15     cout << a << " al cuadrado: " << cuadrado( a ) << endl; // muestra a al cuadrado
16     return 0; // indica que terminó correctamente
17 } // fin de main
18
19 // devuelve el cuadrado de un entero
20 int cuadrado( int x ) // x es una variable local
21 {
22     return x * x; // calcula el cuadrado y devuelve el resultado
23 } // fin de la función cuadrado

```

10 al cuadrado: 100

Figura 6.13 | Función `cuadrado` utilizada para demostrar la pila de llamadas a funciones y los registros de activación.

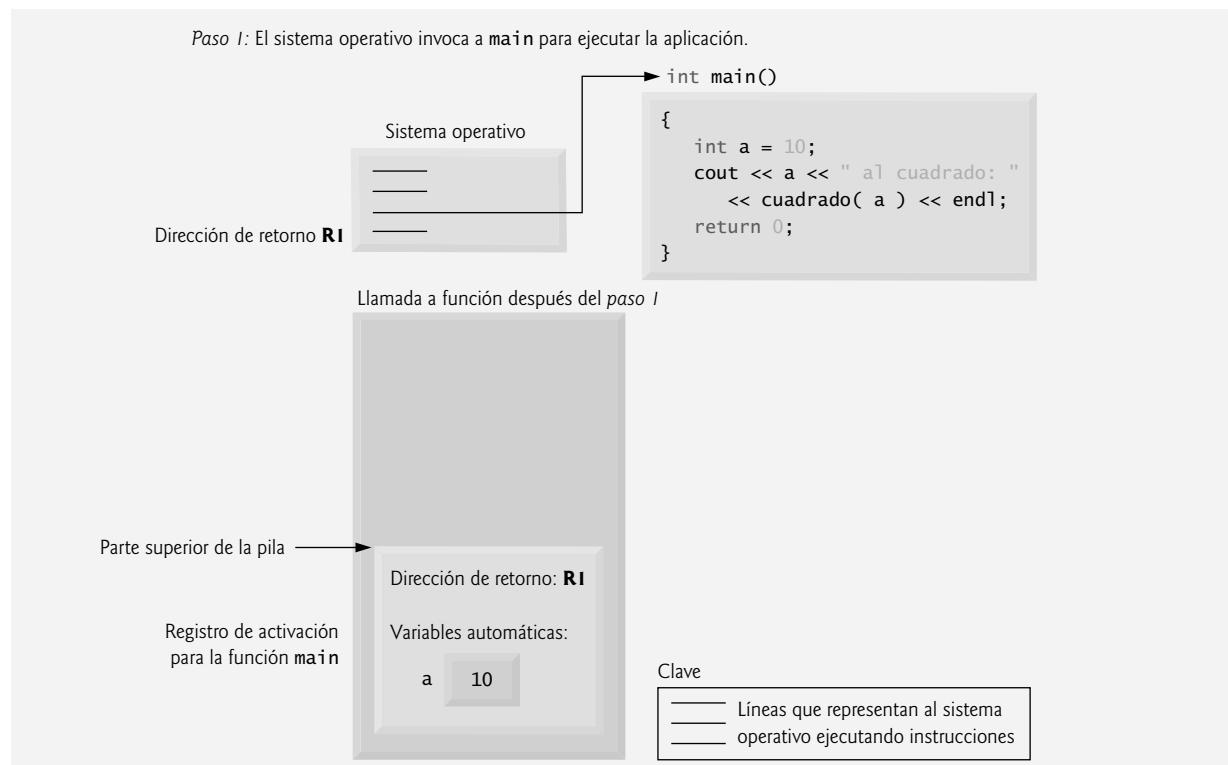


Figura 6.14 | La pila de llamadas a funciones, después de que el sistema operativo invoca a `main` para ejecutar la aplicación.

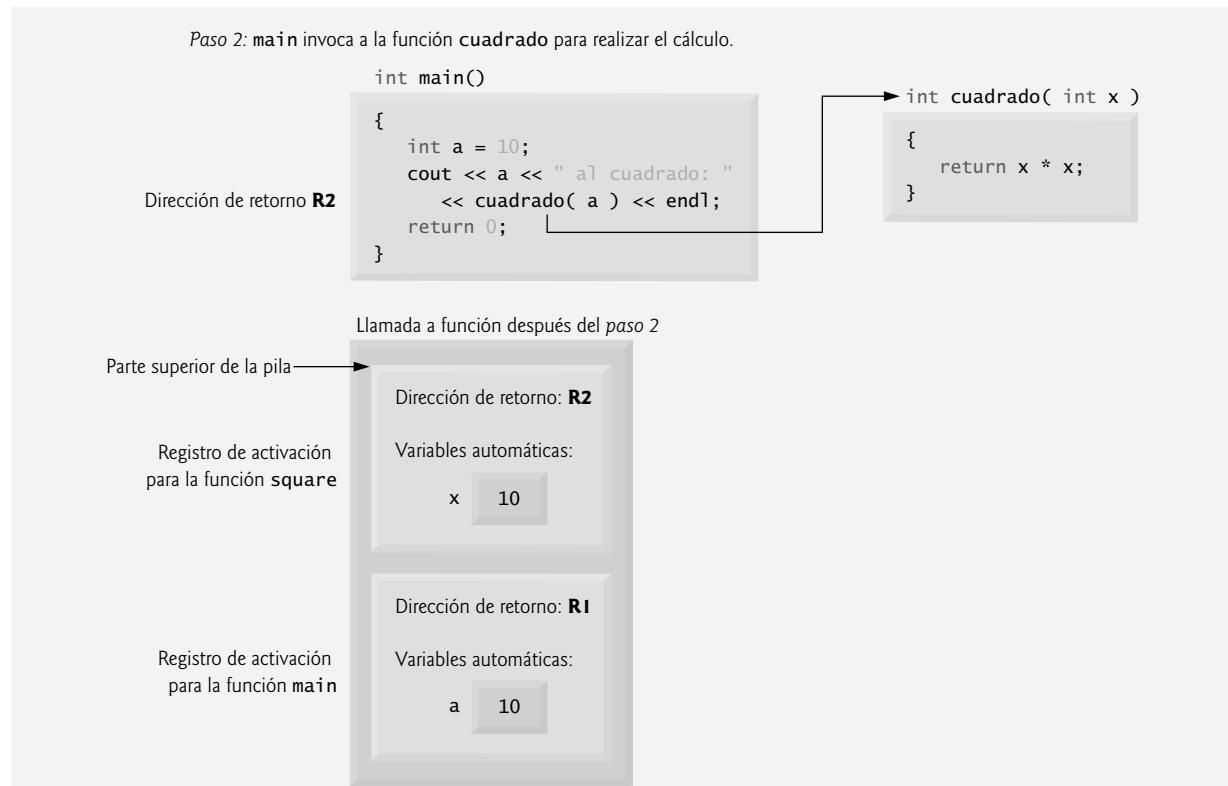


Figura 6.15 | La pila de llamadas a funciones, después de que `main` invoca a la función `cuadrado` para realizar el cálculo.

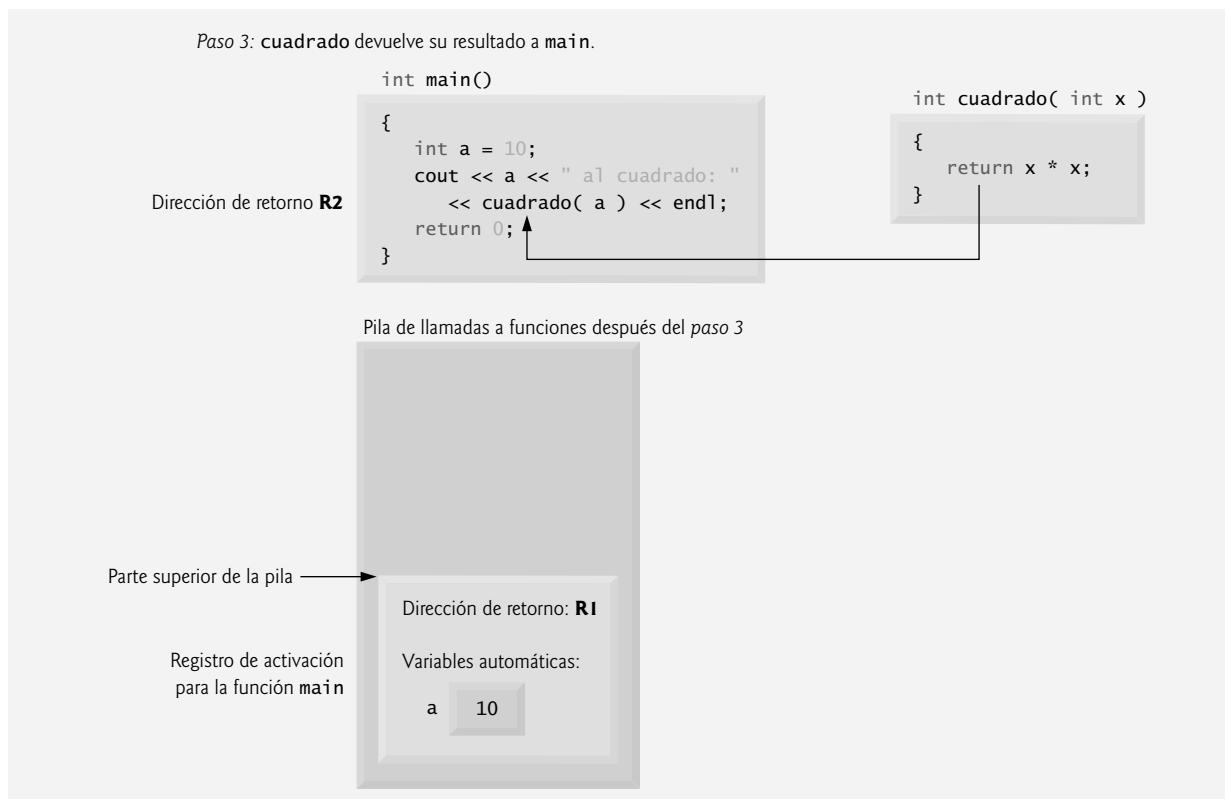


Figura 6.16 | La pila de llamadas a funciones, después de que la función `cuadrado` regresa a `main`.

que requiere para regresar al sistema operativo (es decir, `R1` en la figura 6.14) y hace que la memoria para la variable automática de `main` (es decir, `a`) ya no esté disponible.

Ahora hemos visto qué tan valiosa es la noción de la estructura de datos tipo pila para implementar un mecanismo clave que ofrezca soporte para la ejecución de un programa. Las estructuras de datos tienen muchas aplicaciones importantes en las ciencias computacionales. En el capítulo 20, Estructuras de datos y en el capítulo 22, Biblioteca de plantillas estándar (STL), hablaremos sobre las pilas, colas, listas, árboles y otras estructuras de datos.

6.12 Funciones con listas de parámetros vacías

En C++, una lista de parámetros vacía se especifica mediante `void` o nada entre paréntesis. El prototipo

```
void imprimir();
```

especifica que la función `imprimir` no recibe argumentos y no devuelve un valor. La figura 6.17 demuestra ambas formas de declarar y usar funciones con listas de parámetros vacías.



Tip de portabilidad 6.2

El significado de una lista de parámetros de función vacía en C++ es considerablemente distinto al de C. En C, significa que se deshabilita la comprobación de argumentos (es decir, la llamada a una función puede pasar cualquier argumento que deseé). En C++, significa que la función no recibe argumentos explícitamente. Por ende, los programas en C que utilizan esta característica podrían producir errores de compilación al compilarse en C++.

```

1 // Fig. 6.17: fig06_17.cpp
2 // Funciones que no reciben argumentos.
3 #include <iostream>

```

Figura 6.17 | Funciones que no reciben argumentos. (Parte 1 de 2).

```

4  using std::cout;
5  using std::endl;
6
7  void funcion1(); // función que no recibe argumentos
8  void funcion2( void ); // función que no recibe argumentos
9
10 int main()
11 {
12     funcion1(); // llama a funcion1 sin argumentos
13     funcion2(); // llama a funcion2 sin argumentos
14     return 0; // indica que terminó correctamente
15 } // fin de main
16
17 // funcion1 usa una lista de parámetros vacía para especificar que
18 // la función no recibe argumentos
19 void funcion1()
20 {
21     cout << "funcion1 no recibe argumentos" << endl;
22 } // fin de funcion1
23
24 // funcion2 usa una lista de parámetros vacía para especificar que
25 // la función no reserva argumentos
26 void funcion2( void )
27 {
28     cout << "funcion2 tampoco recibe argumentos" << endl;
29 } // fin de funcion2

```

funcion1 no recibe argumentos
funcion2 tampoco recibe argumentos

Figura 6.17 | Funciones que no reciben argumentos. (Parte 2 de 2).

Error común de programación 6.13



Los programas de C++ no se compilan, a menos que se proporcionen prototipos de función para cada función, o que cada función se defina antes de llamarla.

6.13 Funciones en línea

Es bueno implementar un programa como un conjunto de funciones desde el punto de vista de la ingeniería de software, pero las llamadas a funciones implican una sobrecarga en tiempo de ejecución. C++ cuenta con las **funciones en línea** para ayudar a reducir la sobrecarga de las llamadas a funciones; en especial para las funciones pequeñas. Al colocar el calificador `inline` antes del tipo de valor de retorno de la función en su definición, se “aconseja” al compilador para que genere una copia del código de la función en ese lugar (cuando sea apropiado) para evitar la llamada a una función. La desventaja es que se insertan múltiples copias del código de la función en el programa (lo cual aumenta su tamaño) en vez de que haya una sola copia de la función, a la cual se le pasa el control cada vez que se realiza una llamada. El compilador puede ignorar el calificador `inline`, y por lo general lo hace para todas las funciones, excepto las más pequeñas.

Observación de Ingeniería de Software 6.11



Cualquier modificación a una función `inline` requiere que se vuelvan a compilar todos los clientes de la función. Esto puede ser considerable en ciertas situaciones de desarrollo y mantenimiento de programas.

Buena práctica de programación 6.5



El calificador `inline` debe usarse sólo con funciones pequeñas, de uso frecuente.

Tip de rendimiento 6.4



El uso de funciones `inline` puede reducir el tiempo de ejecución, pero puede incrementar el tamaño del programa.

La figura 6.18 utiliza la función `inline` llamada `cubo` (líneas 11 a 14) para calcular el volumen de un cubo de tamaño `lado`. La palabra clave `const` en la lista de parámetros de la función `cubo` (línea 11) indica al compilador que la función no modifica la variable `lado`. Esto asegura que la función no modifique el valor de `lado` cuando se realice el cálculo. (La palabra clave `const` se describe con detalle en los capítulos 7, 8 y 10.) Observe que la definición completa de la función `cubo` aparece antes de usarla en el programa. Esto se requiere, de manera que el compilador sepa cómo expandir una llamada a la función `cubo` en su código en línea. Por esta razón, las funciones en línea reutilizables se colocan comúnmente en archivos de encabezado, de manera que sus definiciones se puedan incluir en cada archivo de código fuente que las utilice.



Observación de Ingeniería de Software 6.12

El calificador `const` se debe utilizar para hacer valer el principio del menor privilegio. El uso de este principio para diseñar software de manera apropiada puede reducir considerablemente el tiempo de depuración y los efectos secundarios inadecuados, y puede facilitar la modificación y el mantenimiento de un programa.

```

1 // Fig. 6.18: fig06_18.cpp
2 // Uso de una función en línea para calcular el volumen de un cubo.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 // Definición de la función en línea cubo. La definición de la función aparece
9 // antes de llamar a la función, por lo que no se requiere un prototipo de función.
10 // La primera línea de la función actúa como el prototipo.
11 inline double cubo( const double lado )
12 {
13     return lado * lado * lado; // calcula el cubo
14 } // fin de la función cubo
15
16 int main()
17 {
18     double valorLado; // almacena el valor introducido por el usuario
19     cout << "Escriba la longitud del lado de su cubo: ";
20     cin >> valorLado; // lee el valor del usuario
21
22     // calcula el cubo de valorLado y muestra el resultado
23     cout << "El volumen del cubo con un lado de "
24         << valorLado << " es " << cubo( valorLado ) << endl;
25     return 0; // indica finalización exitosa
26 } // end main

```

```

Escriba la longitud del lado de su cubo: 3.5
El volumen del cubo con un lado de 3.5 es 42.875

```

Figura 6.18 | Función `inline` que calcula el volumen de un cubo.

6.14 Referencias y parámetros de referencias

Dos formas de pasar argumentos a las funciones en muchos lenguajes de programación son el **paso por valor** y el **paso por referencia**. Cuando se pasa un argumento por valor, se crea una *copia* del valor del argumento y se pasa (en la pila de llamadas a funciones) a la función que se llamó. Las modificaciones a la copia no afectan al valor de la variable original en la función que hizo la llamada. Esto evita los efectos secundarios accidentales que tanto obstaculizan el desarrollo de sistemas de software correctos y confiables. Cada argumento que se ha pasado en los programas en este capítulo hasta ahora, se ha pasado por valor.



Tip de rendimiento 6.5

Una desventaja del paso por valor es que, si se va a pasar un elemento de datos extenso, el proceso de copiar esos datos puede requerir una cantidad considerable de tiempo de ejecución y espacio en memoria.

Parámetros por referencia

En esta sección presentamos los **parámetros por referencia**: el primero de los dos medios que proporciona C++ para realizar el paso por referencia. Mediante el paso por referencia, la función que hace la llamada proporciona a la función que llamó la habilidad de acceder directamente a los datos de la primera, y de modificar esos datos en caso de que la función que se llamó así lo decidiera.



Tip de rendimiento 6.6

El paso por referencia es bueno por cuestiones de rendimiento, ya que puede eliminar la sobrecarga de copiar grandes cantidades de datos en el paso por valor.



Observación de Ingeniería de Software 6.13

El paso por referencia puede debilitar la seguridad, ya que la función a la que se llamó puede corromper los datos de la función que hizo la llamada.

Más adelante veremos cómo lograr la ventaja de rendimiento que ofrece el paso por referencia, al tiempo que se obtiene la ventaja de ingeniería de software de evitar que la corrupción de los datos de la función que hizo la llamada.

Un parámetro por referencia es un alias para su correspondiente argumento en la llamada a una función. Para indicar que un parámetro de función se pasa por referencia, simplemente hay que colocar un signo & después del tipo del parámetro en el prototipo de la función; use la misma convención al listar el tipo del parámetro en el encabezado de la función. Por ejemplo, la siguiente declaración en el encabezado de una función:

```
int &cuenta
```

si se lee de izquierda a derecha, significa que “cuenta es una referencia a un valor int”. En la llamada a la función, simplemente hay que mencionar la variable por su nombre para pasarla por referencia. Después, al mencionar la variable por el nombre de su parámetro en el cuerpo de la función a la que se llamó, en realidad se refiere a la variable original en la función que hizo la llamada, y esta variable original se puede modificar directamente en la función a la que se llamó. Como siempre, el prototipo de función y el encabezado deben concordar.

Paso de argumentos por valor y por referencia

En la figura 6.19 se compara el paso por valor y el paso por referencia con los parámetros por referencia. Los “estilos” de los argumentos en las llamadas a la función cuadradoPorValor y la función cuadradoPorReferencia son idénticos; ambas variables sólo se mencionan por nombre en las llamadas a las funciones. Sin comprobar los prototipos o las definiciones de las funciones, no es posible deducir sólo de las llamadas si cada función puede modificar sus argumentos. Como los prototipos de función son obligatorios, el compilador no tiene problemas para resolver la ambigüedad.

```

1 // Fig. 6.19: fig06_19.cpp
2 // Comparación entre paso por valor y paso por referencia mediante referencias.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int cuadradoPorValor( int ); // prototipo de función (paso por valor)
8 void cuadradoPorReferencia( int & ); // prototipo de función (paso por referencia)
9
10 int main()
11 {
12     int x = 2; // valor para cuadrado usando cuadradoPorValor
13     int z = 4; // valor para cuadrado usando cuadradoPorReferencia
14
15     // demuestra cuadradoPorValor
16     cout << "x = " << x << " antes de cuadradoPorValor\n";
17     cout << "Valor devuelto por cuadradoPorValor: "
18         << cuadradoPorValor( x ) << endl;
19     cout << "x = " << x << " después de cuadradoPorValor\n" << endl;
20 }
```

Figura 6.19 | Paso de argumentos por valor y por referencia. (Parte 1 de 2).

```

21 // demuestra cuadradoPorReferencia
22 cout << "z = " << z << " antes de cuadradoPorReferencia" << endl;
23 cuadradoPorReferencia( z );
24 cout << "z = " << z << " despues de cuadradoPorReferencia" << endl;
25 return 0; // indica que terminó correctamente
26 } // fin de main
27
28 // cuadradoPorValor multiplica el número por sí mismo, almacena el
29 // resultado en el número y devuelve el nuevo valor del número
30 int cuadradoPorValor( int numero )
31 {
32     return numero *= numero; // no se modificó el argumento de la función que hizo la llamada
33 } // fin de la función cuadradoPorValor
34
35 // cuadradoPorReferencia multiplica a refNumero por sí solo y almacena el resultado
36 // en la variable a la que refNumero hace referencia en la función main
37 void cuadradoPorReferencia( int &refNumero )
38 {
39     refNumero *= refNumero; // se modificó el argumento de la función que hizo la llamada
40 } // fin de la función cuadradoPorReferencia

```

```

x = 2 antes de cuadradoPorValor
Valor devuelto por cuadradoPorValor: 4
x = 2 despues de cuadradoPorValor

z = 4 antes de cuadradoPorReferencia
z = 16 despues de cuadradoPorReferencia

```

Figura 6.19 | Paso de argumentos por valor y por referencia. (Parte 2 de 2).

Error común de programación 6.14



Como los parámetros por referencia se mencionan sólo por su nombre en el cuerpo de la función a la que se llama, podríamos tratar de manera inadvertida los parámetros por referencia como parámetros de paso por valor. Esto puede provocar efectos secundarios inesperados si la función modifica las copias originales de las variables.

El capítulo 8 habla sobre los apuntadores; éstos proporcionan una forma alternativa del paso por referencia, en la cual el estilo de la llamada indica claramente el paso por referencia (y el potencial de modificar los argumentos de la función que hace la llamada).

Tip de rendimiento 6.7



Para pasar objetos extensos, use un parámetro por referencia constante para simular la apariencia y seguridad del paso por valor y evitar la sobrecarga de pasar una copia del objeto extenso.

Observación de Ingeniería de Software 6.14



Muchos programadores no se molestan en declarar parámetros pasados por valor como `const`, aun y cuando la función a la que se llama no debe modificar el argumento que se pasó. La palabra clave `const` en este contexto sólo protegería una copia del argumento original, no al mismo argumento original, el cual cuando se pasa por valor existe la seguridad de que la función a la que se llamó no lo vaya a modificar.

Para especificar una referencia a una constante, hay que colocar el calificador `const` antes del especificador de tipo en la declaración del parámetro.

Observe la colocación del signo `&` en la lista de parámetros de la función `cuadradoPorReferencia` (línea 37, figura 6.19). Algunos programadores de C++ prefieren escribir la forma equivalente `int& refNumero`.

Observación de Ingeniería de Software 6.15



Por cuestión de claridad y rendimiento, muchos programadores de C++ prefieren que los argumentos modificables se pasen a las funciones mediante el uso de apuntadores (que estudiaremos en el capítulo 8), que los argumentos pequeños no modificables se pasen por valor y los argumentos grandes no modificables se pasen a las funciones mediante el uso de referencias a constantes.

Referencias como alias dentro de una función

Las referencias también se pueden usar como alias para otras variables dentro de una función (aunque por lo general se utilizan con las funciones, como se muestra en la figura 6.19). Por ejemplo, el código

```
1 int cuenta = 1; // declara la variable entera cuenta
2 int &cRef = cuenta; // crea cRef como alias para cuenta
3 cRef++; // incrementa cuenta (usando su alias cRef)
```

incrementa la variable `cuenta` mediante el uso de su alias `cRef`. Las variables de referencia deben inicializarse en sus declaraciones (vea las figuras 6.20 y 6.21), y no se pueden reasignar como alias para otras variables. Una vez que se declara una referencia como alias para otra variable, todas las operaciones que supuestamente se realizan en el alias (es decir, la referencia) en realidad se realizan en la variable original. El alias es simplemente otro nombre para la variable original. Al tomar la dirección de una referencia y comparar referencias no se producen errores de sintaxis; en vez de ello, cada operación ocurre realmente en la variable para la cual la referencia es un alias. A menos que sea una referencia a una constante, un argumento por referencia debe ser un *lvalue* (por ejemplo, el nombre de una variable), no una constante o expresión que devuelva un *rvalue* (por ejemplo, el resultado de un cálculo). Consulte la sección 5.9 para ver las definiciones de los términos *lvalue* y *rvalue*.

```
1 // Fig. 6.20: fig06_20.cpp
2 // Las referencias deben inicializarse.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int x = 3;
10    int &y = x; // y hace referencia a (es un alias para) x
11
12    cout << "x = " << x << endl << "y = " << y << endl;
13    y = 7; // en realidad modifica a x
14    cout << "x = " << x << endl << "y = " << y << endl;
15    return 0; // indica que terminó correctamente
16 } // fin de main
```

```
x = 3
y = 3
x = 7
y = 7
```

Figura 6.20 | Inicialización y uso de una referencia.

```
1 // Fig. 6.21: fig06_21.cpp
2 // Las referencias deben inicializarse.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int x = 3;
10    int &y; // Error: y debe inicializarse
11
12    cout << "x = " << x << endl << "y = " << y << endl;
13    y = 7;
14    cout << "x = " << x << endl << "y = " << y << endl;
15    return 0; // indica que terminó correctamente
16 } // fin de main
```

Figura 6.21 | La referencia sin inicializar produce un error de sintaxis. (Parte I de 2).

Mensaje de error del compilador de línea de comandos Borland C++:

```
Error E2304 C:\cpphttp6_ejemplos\cap06\Fig06_21\fig06_21.cpp 10:
Reference variable 'y' must be initialized in function main()
```

Mensaje de error del compilador Microsoft Visual C++:

```
C:\cpphttp6_ejemplos\cap06\Fig06_21\fig06_21.cpp 10: error C2530: 'y' :
references must be initialized
```

Mensaje de error del compilador GNU C++:

```
fig06_21.cpp:10: error: 'y' declared as a reference but not initialized
```

Figura 6.21 | La referencia sin inicializar produce un error de sintaxis. (Parte 2 de 2).

Devolver una referencia de una función

Las funciones pueden devolver referencias, pero esto puede ser peligroso. Al devolver una referencia a una variable declarada en la función que se llamó, la variable debe declararse **static** dentro de esa función. En caso contrario, la referencia se refiere a una variable automática que se descarta cuando termina la función; se dice que dicha variable es “indefinida”, y el comportamiento del programa es impredecible. Las referencias a variables indefinidas se llaman **referencias sueltas**.



Error común de programación 6.15

Si no se inicializa una variable de referencia al declararla se produce un error de compilación, a menos que la declaración forme parte de la lista de parámetros de una función. Los parámetros por referencia se inicializan cuando se hace una llamada a la función en la que se declaran.



Error común de programación 6.16

Tratar de reasignar una referencia antes declarada para que sea un alias de otra variable es un error lógico. El valor de la otra variable simplemente se asigna a la variable para la cual la referencia ya es un alias.



Error común de programación 6.17

Devolver una referencia a una variable automática en una función a la que se ha llamado es un error lógico. Algunos compiladores generan una advertencia cuando esto ocurre.

Mensajes de error para las referencias sin inicializar

El estándar de C++ no especifica los mensajes de error que utilizan los compiladores para indicar errores específicos. Por esta razón, en la figura 6.21 se muestran los mensajes de error producidos por el compilador de línea de comandos Borland C++, el compilador de Microsoft Visual C++ 2005 y el compilador de GNU C++ cuando no se inicializa una referencia.

6.15 Argumentos predeterminados

Para un programa, es algo común el invocar una función repetidas veces con el mismo valor de argumento para un parámetro específico. En tales casos, podemos especificar que dicho parámetro tiene un **argumento predeterminado**; es decir, que tiene un valor predeterminado que debe pasar a ese parámetro. Cuando un programa omite un argumento para un parámetro con un argumento predeterminado en la llamada a una función, el compilador vuelve a escribir la llamada a la función e inserta el valor predeterminado del argumento.

Los argumentos predeterminados deben ser los argumentos de más a la derecha en la lista de parámetros de una función. Al llamar a una función con dos o más argumentos predeterminados, si se omite un argumento que no sea el de más a la derecha en la lista de argumentos, entonces también deben omitirse todos los argumentos que estén a la derecha de ese argumento. Los argumentos predeterminados deben especificarse con la primera ocurrencia del nombre de la función; por lo general, en el prototipo de la función. Si el prototipo se omite debido a que la definición de la función también actúa como el prototipo, entonces los argumentos predeterminados se deben especificar en el encabezado de la función. Los valores predeterminados pueden ser cualquier expresión, incluyendo constantes variables globales o llamadas a funciones. Los argumentos predeterminados también se pueden usar con funciones **inline**.

En la figura 6.22 se demuestra el uso de argumentos predeterminados para calcular el volumen de una caja. El prototipo de función para `volumenCaja` (línea 8) especifica que los tres parámetros reciben valores predeterminados de 1. Observe que proporcionamos nombres de variables en el prototipo de función para mejorar la legibilidad. Como siempre, los nombres de las variables no se requieren en los prototipos de función.



Error común de programación 6.18

Un error de compilación es el especificar argumentos predeterminados tanto en el prototipo como en el encabezado de una función.

La primera llamada a `volumenCaja` (línea 13) especifica que no hay argumentos, con lo cual se utilizan los tres valores predeterminados de 1. En la segunda llamada (línea 17) sólo se pasa un argumento `longitud`, con lo cual se utilizan los valores predeterminados de 1 para los argumentos `anchura` y `altura`. La tercera llamada (línea 21) pasa argumentos sólo para `longitud` y `anchura`, con lo cual se utiliza un valor predeterminado de 1 para el argumento `altura`. La última llamada (línea 25) pasa argumentos para `longitud`, `anchura` y `altura`, con lo cual no utiliza valores predeterminados.

```

1 // Fig. 6.22: fig06_22.cpp
2 // Uso de argumentos predeterminados.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // prototipo de función que especifica argumentos predeterminados
8 int volumenCaja( int longitud = 1, int anchura = 1, int altura = 1 );
9
10 int main()
11 {
12     // sin argumentos--usa valores predeterminados para todas las medidas
13     cout << "El volumen predeterminado de la caja es: " << volumenCaja();
14
15     // especifica la longitud; anchura y altura predeterminadas
16     cout << "\n\nEl volumen de una caja con longitud 10,\n"
17         << "anchura 1 y altura 1 es: " << volumenCaja( 10 );
18
19     // especifica longitud y anchura; altura predeterminada
20     cout << "\n\nEl volumen de una caja con 10,\n"
21         << "anchura 5 y altura 1 " << volumenCaja( 10, 5 );
22
23     // especifica todos los argumentos
24     cout << "\n\nEl volumen de una caja con longitud 10,\n"
25         << "anchura 5 y altura 2 es: " << volumenCaja( 10, 5, 2 )
26         << endl;
27
28     return 0; // indica que terminó correctamente
29 } // fin de main
30
31 // la función volumenCaja calcula el volumen de una caja
32 int volumenCaja( int longitud, int anchura, int altura )
33 {
34     return longitud * anchura * altura;
35 } // fin de la función volumenCaja

```

```

El volumen predeterminado de la caja es: 1
El volumen de una caja con longitud 10,
anchura 1 y altura 1 es: 10
El volumen de una caja con 10,
anchura 5 y altura 1 es: 50
El volumen de una caja con longitud 10,
anchura 5 y altura 2 es: 100

```

Figura 6.22 | Argumentos predeterminados para una función.

Observe que cualquier argumento que se pasa a la función de manera explícita se asigna a los parámetros de la función, de izquierda a derecha. Por lo tanto, cuando `volumenCaja` recibe un argumento, la función asigna el valor de ese argumento a su parámetro `longitud` (es decir, el parámetro de más a la izquierda en la lista de parámetros). Cuando `volumenCaja` recibe dos argumentos, la función asigna el valor de ese argumento con su parámetro `longitud` (es decir, el parámetro de más a la izquierda en la lista de parámetros). Cuando `volumenCaja` recibe dos argumentos, la función asigna los valores de esos argumentos a sus parámetros `longitud` y `anchura` en ese orden. Por último, cuando `volumenCaja` recibe los tres argumentos, la función asigna los valores de esos argumentos a sus parámetros `longitud`, `anchura` y `altura`, respectivamente.



Buena práctica de programación 6.6

El uso de argumentos predeterminados puede simplificar la escritura de las llamadas a funciones. Sin embargo, algunos programadores sienten que es más claro especificar de manera explícita todos los argumentos.



Observación de Ingeniería de Software 6.16

Si los valores predeterminados para una función cambian, se debe volver a compilar todo el código cliente que utilice esa función.



Error común de programación 6.19

Especificar y tratar de usar un argumento predeterminado que no sea el de más a la derecha (mientras no se utilicen al mismo tiempo valores predeterminados para todos los argumentos de más a la derecha) es un error de sintaxis.

6.16 Operador de resolución de ámbito unario

Es posible declarar variables locales y globales con el mismo nombre. C++ proporciona el **operador de resolución de ámbito binario (::)** para acceder a una variable global cuando una variable local con el mismo nombre se encuentra dentro del alcance. El operador de resolución de ámbito unario no se puede utilizar para acceder a una variable local con el mismo nombre en un bloque exterior. Se puede acceder a una variable global directamente sin el operador de resolución de ámbito unario, si el nombre de la variable global no es el mismo que el de una variable local dentro del alcance.

En la figura 6.23 se demuestra el operador de resolución de ámbito unario con variables local y global con el mismo nombre (líneas 7 y 11). Para enfatizar que las versiones local y global de la variable `numero` son distintas, el programa declara una variable de tipo `int` y la otra de tipo `double`.

El uso del operador de resolución de ámbito unario (::) con un nombre de variable dado es opcional cuando la única variable con ese nombre es una variable global.



Error común de programación 6.20

Es un error tratar de usar el operador de resolución de ámbito unario (:) para acceder a una variable no global en un bloque exterior. Si no existe una variable global con ese nombre, se produce un error de compilación. Si existe una variable global con ese nombre, éste es un error lógico, ya que el programa hará referencia a la variable global, cuando la intención era acceder a la variable no global en el bloque exterior.



Buena práctica de programación 6.7

Si se utiliza siempre el operador de resolución de ámbito unario (:) para hacer referencia a las variables globales, los programas serán más fáciles de leer y de comprender, ya que se establece claramente que estamos tratando de acceder a una variable global, en vez de una variable no global.

```

1 // Fig. 6.23: fig06_23.cpp
2 // Uso del operador de resolución de ámbito unario.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int numero = 7; // variable global llamada numero
8

```

Figura 6.23 | Operador de resolución de ámbito unario. (Parte 1 de 2).

```

9 int main()
10 {
11     double numero = 10.5; // variable local llamada numero
12
13     // muestra los valores de las variables local y global
14     cout << "Valor local double de numero = " << numero
15     << "\nValor global int de numero = " << ::numero << endl;
16     return 0; // indica que terminó correctamente
17 } // fin de main

```

```

Valor local double de numero = 10.5
Valor global int de numero = 7

```

Figura 6.23 | Operador de resolución de ámbito unario. (Parte 2 de 2).



Observación de Ingeniería de Software 6.17

Al utilizar siempre el operador de resolución de ámbito unario (:) para hacer referencia a las variables globales, se facilita la modificación de los programas, al reducir el riesgo de conflictos de nombres con variables no globales.



Tip para prevenir errores 6.4

Al utilizar siempre el operador de resolución de ámbito unario (:) para hacer referencia a una variable global, se eliminan los posibles errores lógicos que podrían ocurrir si una variable no global oculta a la variable global.



Tip para prevenir errores 6.5

Evite usar variables con el mismo nombre para distintos propósitos en un programa. Aunque esto se permite en diversas circunstancias, puede producir errores.

6.17 Sobrecarga de funciones

C++ permite definir varias funciones con el mismo nombre, siempre y cuando éstas tengan distintas firmas. A esta capacidad se le conoce como **sobrecarga de funciones**. Cuando se hace una llamada a una función sobrecargada, el compilador de C++ selecciona la función apropiada al examinar el número, tipos y orden de los argumentos en la llamada. La sobrecarga de funciones se utiliza comúnmente para crear varias funciones con el mismo nombre que realicen tareas similares, pero con distintos tipos de datos. Por ejemplo, muchas funciones en la biblioteca de matemáticas están sobre cargadas para distintos tipos de datos numéricos; el estándar de C++ requiere versiones sobre cargadas float, double y long double de las funciones matemáticas de la biblioteca que se describen en la sección 6.3.



Buena práctica de programación 6.8

Al sobre cargar las funciones que realizan tareas estrechamente relacionadas, los programas pueden ser más fáciles de leer y comprender.

Funciones cuadrado sobre cargadas

La figura 6.24 utiliza funciones cuadrado sobre cargadas para calcular el cuadrado de un valor int (líneas 8 a 12) y el cuadrado de un valor double (líneas 15 a 19). En la línea 23 se invoca a la versión int de la función cuadrado, para lo cual se le pasa el valor literal 7. C++ trata a los valores literales numéricos enteros como de tipo int de manera predeterminada. De manera similar, en la línea 25 se invoca a la versión double de la función cuadrado, para lo cual se le pasa el valor literal 7.5, que C++ trata como valor double de manera predeterminada. En cada caso, el compilador elige la función apropiada que va a llamar, con base en el tipo del argumento. Las últimas dos líneas en la ventana de salida confirman que se llamó a la función apropiada en cada caso.

```

1 // Fig. 6.24: fig06_24.cpp
2 // Funciones sobre cargadas.
3 #include <iostream>

```

Figura 6.24 | Funciones cuadrado sobre cargadas. (Parte 1 de 2).

```

4  using std::cout;
5  using std::endl;
6
7  // función cuadrado para valores int
8  int cuadrado( int x )
9  {
10    cout << "el cuadrado del valor int " << x << " es ";
11    return x * x;
12 } // fin de la función cuadrado con argumento int
13
14 // función cuadrado para valores double
15 double cuadrado( double y )
16 {
17    cout << "el cuadrado del valor double " << y << " es ";
18    return y * y;
19 } // fin de la función cuadrado con argumento double
20
21 int main()
22 {
23    cout << cuadrado( 7 ); // llama a la versión int
24    cout << endl;
25    cout << cuadrado( 7.5 ); // llama a la versión double
26    cout << endl;
27    return 0; // indica que terminó correctamente
28 } // fin de main

```

el cuadrado del valor int 7 es 49
el cuadrado del valor double 7.5 es 56.25

Figura 6.24 | Funciones cuadrado sobrecargadas. (Parte 2 de 2).

Cómo diferencia el compilador las funciones sobrecargadas

Las funciones sobrecargadas se diferencian mediante sus firmas. Una firma es una combinación del nombre de una función y los tipos de sus parámetros (en orden). El compilador codifica cada identificador de función con el número y tipos de sus parámetros (lo que algunas veces se conoce como **manipulación de nombres** o **decoración de nombres**) para permitir la **vinculación segura de tipos**. Este tipo de vinculación asegura que se llame a la función sobrecargada apropiada, y que los tipos de los argumentos se conformen a los tipos de los parámetros.

La figura 6.25 se compiló con el compilador de línea de comandos Borland C++ 5.6.4. En vez de mostrar los resultados de la ejecución del programa (como se haría normalmente), mostramos los nombres manipulados de la función que Borland C++ produce en lenguaje ensamblador. Cada nombre manipulado empieza con el carácter @, seguido del nombre de la función. Después, el nombre de la función se separa de la lista de parámetros manipulados mediante \$q. En la lista de parámetros para la función `nada2` (línea 25; vea la cuarta línea de salida), c representa a un valor `char`, i representa a un valor `int`, rf representa a un valor `float &` (es decir, una referencia a un valor `float`) y rd representa a un valor `double &` (es decir, una referencia a un valor `double`). En la lista de parámetros para la función `nada1`, i representa a un valor `int`, f representa a un valor `float`, c representa a un valor `char` y ri representa a un valor `int &`. Las dos funciones `cuadrado` se diferencian en base a sus listas de parámetros; uno especifica d para `double` y el otro especifica i para `int`. Los tipos de valores de retorno de las funciones no se especifican en los nombres manipulados. Las funciones sobrecargadas pueden tener distintos tipos de valores de retorno, pero si es así, también deben tener distintas listas de parámetros. De nuevo, no se pueden tener dos funciones con la misma firma y distintos tipos de valores de retorno. Observe que la manipulación de nombres de funciones es específica para cada compilador. Observe además que la función `main` no está manipulada, ya que no puede sobrecargarse.

```

1 // Fig. 6.25: fig06_25.cpp
2 // Manipulación de nombres.
3
4 // función cuadrado para valores int
5 int cuadrado( int x )

```

Figura 6.25 | Manipulación de nombres para permitir la vinculación segura de tipos. (Parte I de 2).

```

6  {
7      return x * x;
8  } // fin de la función cuadrado
9
10 // función cuadrado para valores double
11 double cuadrado( double y )
12 {
13     return y * y;
14 } // fin de la función cuadrado
15
16 // función que obtiene argumentos de los tipos
17 // int, float, char e int &
18 void nada1( int a, float b, char c, int &d )
19 {
20     // cuerpo vacío de la función
21 } // fin de la función nada1
22
23 // función que recibe argumentos de los tipos
24 // char, int, float & y double &
25 int nada2( char a, int b, float &c, double &d )
26 {
27     return 0;
28 } // fin de la función nada2
29
30 int main()
31 {
32     return 0; // indica que terminó correctamente
33 } // fin de main

```

```

@cuadrado$qi
@cuadrado$qd
@nada1$qifcri
@nada2$qcirfrd
_main

```

Figura 6.25 | Manipulación de nombres para permitir la vinculación segura de tipos. (Parte 2 de 2).

Error común de programación 6.21



Crear funciones sobrecargadas con las listas de parámetros idénticos y distintos tipos de retorno es un error de compilación.

El compilador utiliza sólo las listas de parámetros para diferenciar entre las funciones del mismo nombre. Las funciones sobrecargadas no necesitan tener el mismo número de parámetros. Hay que tener cuidado al sobrecargar funciones con parámetros predeterminados, ya que esto puede provocar ambigüedades.

Error común de programación 6.22



Una función en la que se omiten sus argumentos predeterminados se podría invocar de una manera idéntica a otra función sobrecargada; esto es un error de compilación. Por ejemplo, al tener en un programa tanto una función que no reciba argumentos de manera explícita, como una función con el mismo nombre que contenga todos los argumentos predeterminados, se produce un error de compilación cuando tratamos de usar el nombre de esa función en una llamada en la que no se pasen argumentos. El compilador no sabe cuál versión de la función elegir.

Operadores sobrecargados

En el capítulo 11 hablaremos acerca de cómo sobrecargar operadores, para definir la forma en que deben operar con objetos de tipos de datos definidos por el usuario. (De hecho, hemos estado utilizando muchos operadores sobrecargados hasta este punto, incluyendo el operador de inserción de flujo `<<` y el operador de extracción de flujo `>>`, cada uno de los cuales se sobrecarga para poder mostrar datos de todos los tipos fundamentales. En el capítulo 11 hablaremos más acerca de cómo sobrecargar los operadores `<<` y `>>` para poder manejar objetos de tipos definidos por el usuario.)

En la sección 6.18 se introducen plantillas de funciones para generar de manera automática funciones sobrecargadas que realizan tareas idénticas en distintos tipos de datos.

6.18 Plantillas de funciones

Por lo general, las funciones sobrecargadas se utilizan para realizar operaciones similares que involucren distintos tipos de lógica de programa en distintos tipos de datos. Si la lógica del programa y las operaciones son idénticas para cada tipo de datos, la sobrecarga se puede llevar a cabo de una forma más compacta y conveniente, mediante el uso de **plantillas de funciones**. El programador escribe una sola definición de plantilla de función. Dados los tipos de los argumentos que se proporcionan en las llamadas a esta función, C++ genera de manera automática **especializaciones de plantilla de función** separadas para manejar cada tipo de llamada de manera apropiada. Por ende, al definir una sola plantilla de función, en esencia se define toda una familia de funciones sobrecargadas.

La figura 6.26 contiene la definición de una plantilla de función (líneas 4 a 18) para una función `maximo` que determina el mayor de tres valores. Todas las definiciones de plantillas de función empiezan con la palabra clave `template` (línea 4), seguidas de una **lista de parámetros de plantilla** para la plantilla de función encerrada entre los paréntesis angulares (`< y >`). Antes de cada parámetro en la lista de parámetros (que a menudo se refiere como un **parámetro de tipo formal**) se coloca la palabra clave `typename` o la palabra clave `class` (que son sinónimos). Los parámetros de tipo formal son **receptáculos** para los tipos fundamentales, o los tipos definidos por el usuario. Estos receptáculos se utilizan para especificar los tipos de los parámetros de la función (línea 5), para especificar el tipo de valor de retorno de la función (línea 5) y para declarar variables dentro del cuerpo de la definición de la función (línea 7). Una plantilla de función se define de igual forma que cualquier otra función, pero utiliza los parámetros de tipo formal como receptáculos para los tipos de datos actuales.

```

1 // Fig. 6.26: maximo.h
2 // Definición de la plantilla de función maximo.
3
4 template < class T > // o template< typename T >
5 T maximo( T valor1, T valor2, T valor3 )
6 {
7     T valorMaximo = valor1; // asume que valor1 es maximo
8
9     // determina si valor2 es mayor que valorMaximo
10    if ( valor2 > valorMaximo )
11        valorMaximo = valor2;
12
13    // determina si valor3 es mayor que valorMaximo
14    if ( valor3 > valorMaximo )
15        valorMaximo = valor3;
16
17    return valorMaximo;
18 } // fin de la plantilla de función maximo

```

Figura 6.26 | Archivo de encabezado de la plantilla de la función `maximo`.

La plantilla de función en la figura 6.26 declara un solo parámetro de tipo formal `T` (línea 4) como receptáculo para el tipo de datos a evaluar por la función `maximo`. El nombre de un parámetro de tipo debe ser único en la lista de parámetros de la plantilla para una definición específica de ésta. Cuando el compilador detecta una invocación a `maximo` en el código fuente del programa, el tipo de los datos que se pasan a `maximo` se sustituye por `T` en toda la definición de la plantilla, y C++ crea una función completa para determinar el máximo de tres valores del tipo de datos especificado. Después se compila la función recién creada. Por ende, las plantillas son un medio para generar código.

Error común de programación 6.23

 Si no se coloca la palabra clave `class` o `typename` antes de cada parámetro de tipo formal de una plantilla de función (por ejemplo, escribir `<class S, T>` en vez de `<class S, class T>`), se produce un error de sintaxis.

La figura 6.27 utiliza la plantilla de función `maximo` (líneas 20, 30 y 40) para determinar el mayor de tres valores `int`, tres valores `double` y tres valores `char`, respectivamente.

```

1 // Fig. 6.27: fig06_27.cpp
2 // Programa de prueba de la plantilla de función maximo.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include "maximo.h" // incluye la definición de la plantilla de función maximo
9
10 int main()
11 {
12     // demuestra la función maximo con valores int
13     int int1, int2, int3;
14
15     cout << "Introduzca tres valores enteros: ";
16     cin >> int1 >> int2 >> int3;
17
18     // invoca a la versión int de maximo
19     cout << "El valor int de maximo es: "
20         << maximo( int1, int2, int3 );
21
22     // demuestra la función maximo con valores double
23     double double1, double2, double3;
24
25     cout << "\n\nIntroduzca tres valores double: ";
26     cin >> double1 >> double2 >> double3;
27
28     // invoca a la versión double de maximo
29     cout << "El valor double de maximo es: "
30         << maximo( double1, double2, double3 );
31
32     // demuestra la función maximo con valores char
33     char char1, char2, char3;
34
35     cout << "\n\nIntroduzca tres caracteres: ";
36     cin >> char1 >> char2 >> char3;
37
38     // invoca a la versión char de maximo
39     cout << "El valor char de maximo es: "
40         << maximo( char1, char2, char3 ) << endl;
41     return 0; // indica que terminó correctamente
42 } // fin de main

```

Introduzca tres valores enteros: 1 2 3
 El valor int de maximo es: 3

Introduzca tres valores double: 3.3 2.2 1.1
 El valor double de maximo es: 3.3

Introduzca tres caracteres: A C B
 El valor char de maximo es: C

Figura 6.27 | Demostración de la plantilla de función maximo.

En la figura 6.27, se crean tres funciones como resultado de las llamadas en las líneas 20, 30 y 40: esperar tres valores `int`, tres valores `double` y tres valores `char`, respectivamente. La especialización de plantilla de función que se crea para el tipo `int` reemplaza cada ocurrencia de `T` con `int`, como se muestra a continuación:

```

int maximo( int valor1, int valor2, int valor3 )
{
    int valorMaximo = valor1;

    // determina si valor2 es mayor que valorMaximo
    if ( valor2 > valorMaximo )
        valorMaximo = valor2;

```

```
// determina si valor3 es mayor que valorMaximo
if ( valor3 > valorMaximo )
    valorMaximo = valor3;

return valorMaximo;
} // fin de la plantilla de función maximo
```

6.19 Recursividad

Los programas que hemos visto están estructurados generalmente como funciones que se llaman entre sí, de una manera disciplinada y jerárquica. Para algunos problemas, es conveniente hacer que las funciones se llamen a sí mismas. Una **función recursiva** es una función que se llama a sí misma, ya sea en forma directa o indirecta (a través de otra función). [Nota: aunque muchos compiladores permiten que la función `main` se llame a sí misma, en las secciones 3.6.1, párrafo 3 y 5.2.2, párrafo 9 del documento del estándar de C++ se indica que `main` no debe llamarse dentro de un programa ni en forma recursiva. Su único propósito es servir de punto inicial para la ejecución del programa.] La recursividad es un tema importante que se discute extensamente en los cursos de ciencias computacionales de nivel superior. En esta sección y en la siguiente presentaremos ejemplos simples de recursividad. Este libro contiene un tratamiento exhaustivo de la recursividad. En la figura 6.33 (al final de la sección 6.21) se sintetizan los ejemplos y ejercicios de recursividad que se incluyen en el libro.

Primeramente hablaremos sobre la recursividad en forma conceptual, y después analizaremos dos programas que contienen funciones recursivas. Las metodologías recursivas de solución de problemas tienen varios elementos en común. Se hace una llamada a una función recursiva para resolver un problema. La función en realidad sabe cómo resolver sólo el (los) caso(s) más simple(s), o **caso(s) base**. Si se hace la llamada a la función con un caso base, ésta simplemente devuelve un resultado. Si se hace la llamada a la función con un problema más complejo, la función comúnmente divide el problema en dos piezas conceptuales: una pieza que sabe cómo resolver y otra pieza que no sabe cómo resolver. Para que la recursividad sea factible, esta última pieza debe ser similar al problema original, pero una versión ligeramente más sencilla o simple del mismo. Debido a que este nuevo problema se parece al problema original, la función llama a una nueva copia de sí misma para trabajar en el problema más pequeño; a esto se le conoce como **llamada recursiva**, y también como **paso recursivo**. Por lo general, el paso recursivo incluye la palabra clave `return`, ya que su resultado se combina con la parte del problema que la función supo cómo resolver, para formar un resultado que se pasará de vuelta a la función original que hizo la llamada, que posiblemente sea `main`.

El paso recursivo se ejecuta mientras siga activa la llamada original a la función (es decir, que no haya terminado su ejecución). Este paso recursivo puede producir muchas llamadas recursivas más, a medida que la función divide cada nuevo subproblema en dos piezas conceptuales. Para que la recursividad termine en un momento dado, cada vez que la función se llama a sí misma con una versión más simple del problema original, la secuencia de problemas cada vez más pequeños debe converger en un caso base. En ese punto, la función reconoce el caso base y devuelve un resultado a la copia anterior de la función; después se origina una secuencia de retornos, hasta que la llamada a la función original devuelve el resultado final al método `main`. Todo esto suena bastante extravagante, en comparación con el tipo de solución de problemas “convencionales” que hemos usado hasta este punto. Como ejemplo de estos conceptos en la práctica, vamos a escribir un programa recursivo para realizar un popular cálculo matemático.

El factorial de un entero positivo n , que se escribe como $n!$ (y se pronuncia como “factorial de n ”), viene siendo el producto

$$n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1$$

donde $1!$ es igual a 1 y $0!$ se define como 1. Por ejemplo, $5!$ es el producto $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, que es igual a 120.

El factorial del entero `numero` (donde `numero` es mayor o igual a 0) puede calcularse de manera **iterativa** (sin recursividad), usando una instrucción `for` de la siguiente manera:

```
factorial = 1;
for ( int contador = numero; contador >= 1; contador-- )
    factorial *= contador;
```

Podemos llegar a una definición recursiva de la función factorial, si observamos la siguiente relación algebraica:

$$n! = n \cdot (n - 1)!$$

Por ejemplo, $5!$ es sin duda igual a $5 * 4!$, como se muestra en las siguientes ecuaciones:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$

$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$

$$5! = 5 \cdot (4!)$$

La evaluación de $5!$ procedería como se muestra en la figura 6.28. La figura 6.28(a) muestra cómo procede la sucesión de llamadas recursivas hasta que $1!$ se evalúa como 1, lo cual termina la recursividad. La figura 6.28(b) muestra los valores devueltos de cada llamada recursiva a la función que hizo la llamada, hasta que se calcula y devuelve el valor final.

El programa de la figura 6.29 utiliza la recursividad para calcular e imprimir los factoriales de los enteros del 0 al 10. (En unos momentos explicaremos la elección del tipo de datos `unsigned long`.) La función recursiva `factorial` (líneas 23 a 29) determina primero si la condición de terminación `numero <= 1` (línea 25) es verdadera. Si `numero` es menor o igual que 1, la función `factorial` devuelve 1 (línea 26), ya no es necesaria más recursividad y la función termina. Si `numero` es mayor que 1, en la línea 28 se expresa el problema como el producto de `numero` y una llamada recursiva a `factorial` en la que se evalúa el factorial de `numero - 1`. Observe que `factorial(numero - 1)` es un problema un poco más simple que el cálculo original, `factorial(numero)`.

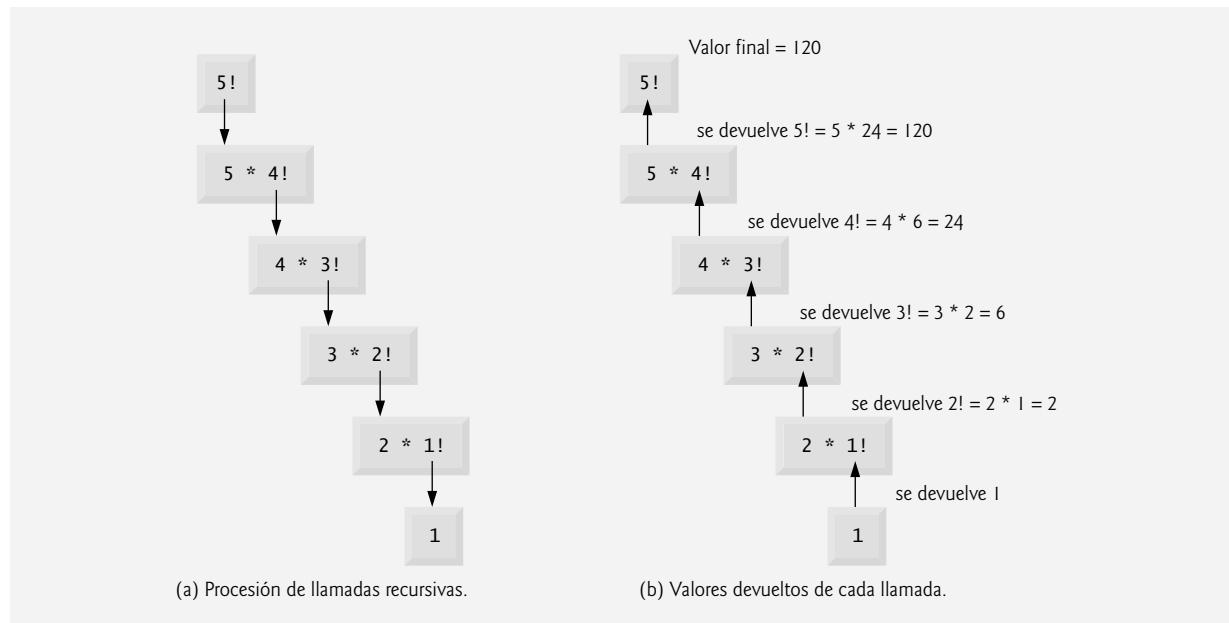


Figura 6.28 | Evaluación recursiva de $5!$.

```

1 // Fig. 6.29: fig06_29.cpp
2 // Demostración de la función recursiva factorial.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 unsigned long factorial( unsigned long ); // prototipo de función
11
12 int main()
13 {
14     // calcula los factoriales del 0 al 10
15     for ( int contador = 0; contador <= 10; contador++ )
16         cout << setw( 2 ) << contador << "!" = " << factorial( contador )
17         << endl;
18
19     return 0; // indica que terminó correctamente
20 } // fin de main
  
```

Figura 6.29 | Demostración de la función recursiva `factorial`. (Parte 1 de 2).

```

21 // definición recursiva de la función factorial
22 unsigned long factorial( unsigned long numero )
23 {
24     if ( numero <= 1 ) // evalúa el caso base
25         return 1; // casos base: 0! = 1 y 1! = 1
26     else // paso recursivo
27         return numero * factorial( numero - 1 );
28 } // fin de la función factorial

```

```

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
0! = 3628800

```

Figura 6.29 | Demostración de la función recursiva `factorial`. (Parte 2 de 2).

La función `factorial` se ha declarado para recibir un parámetro de tipo `unsigned long` y devolver un resultado de tipo `unsigned long`. Ésta es una notación abreviada para `unsigned long int`. El estándar de C++ requiere que una variable de tipo `unsigned long int` sea por lo menos tan grande como un valor `int`. Por lo general, un valor `unsigned long` se almacena en por lo menos cuatro bytes (32 bits); dicha variable puede contener un valor en el rango desde 0 hasta (por lo menos) 4294967295. (El tipo de datos `long int` también se almacena por lo menos en cuatro bytes, y puede contener un valor en por lo menos el rango de -2147483648 a 2147483647.) Como se puede ver en la figura 6.29, los valores de los factoriales aumentan su tamaño rápidamente. Elegimos el tipo de datos `unsigned long` de manera que el programa pueda calcular factoriales mayores que $7!$ en las computadoras con enteros pequeños (como los de dos bytes). Por desgracia, la función `factorial` produce valores extensos con tanta rapidez que, incluso el tipo `unsigned long` no nos ayuda a calcular muchos valores de factorial antes de que se exceda siquiera el valor de una variable `unsigned long`.

Los ejercicios exploran el uso de variables de tipo de datos `double` para calcular los factoriales de números más grandes. Esto apunta a una debilidad en la mayoría de los lenguajes de programación, a saber, que los lenguajes no se extienden fácilmente para manejar los requerimientos únicos de varias aplicaciones. Como veremos cuando hablemos sobre la programación orientada a objetos con más detalle, C++ es un lenguaje extensible que nos permite crear clases que puedan representar enteros arbitrariamente grandes, si lo deseamos. Dichas clases ya están disponibles en bibliotecas de clases populares¹, y trabajaremos en nuestras propias clases similares en los ejercicios 9.14 y 11.14.

Error común de programación 6.24



Si se omite el caso base o se escribe el paso recursivo incorrectamente, de manera que no converja en el caso base, se produce una recursividad “infinita”, con lo cual la memoria se agotará en un momento dado. Esto es analógico para el problema de un ciclo infinito en una solución en una solución iterativa (no recursiva).

6.20 Ejemplo sobre el uso de la recursividad: serie de Fibonacci

La serie de Fibonacci,

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

empieza con 0 y 1, y tiene la propiedad de que cada número subsiguiente de Fibonacci es la suma de los dos números Fibonacci anteriores.

1. Encontrará dichas clases en shoup.net/ntl, cliodhna.cop.uop.edu/~hetrick/c-sources.html y www.trumphurst.com/cpplibs/datapage.phtml?category='intro'.

Esta serie ocurre en la naturaleza y, en específico, describe una forma de espiral. La proporción de números de Fibonacci sucesivos converge en un valor constante de 1.618.... Este número también ocurre con frecuencia en la naturaleza, y se le ha denominado **proporción dorada**, o **media dorada**. Los humanos tienden a descubrir que la media dorada es estéticamente placentera. A menudo, los arquitectos diseñan ventanas, cuartos y edificios cuya longitud y anchura se encuentran en la proporción de la media dorada. A menudo se diseñan tarjetas postales con una proporción de anchura/altura de media dorada.

La serie de Fibonacci se puede definir de manera recursiva como:

$$\begin{aligned} \text{fibonacci}(0) &= 0 \\ \text{fibonacci}(1) &= 1 \\ \text{fibonacci}(n) &= \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2) \end{aligned}$$

El programa de la figura 6.30 calcula el n -ésimo número de Fibonacci en forma recursiva, usando la función **fibonacci**. Observe que los números de Fibonacci tienden a aumentar con bastante rapidez, aunque son más lentos que los factoriales. Por lo tanto, elegimos el tipo de datos **unsigned long** para el tipo del parámetro y el tipo de valor de retorno en la función **fibonacci**. En la figura 6.30 se muestra la ejecución del programa, que muestra los valores de Fibonacci para varios números.

La aplicación empieza con una instrucción **for** que calcula y muestra los valores de Fibonacci para los enteros 0 a 10, y va seguida de tres llamadas para calcular los valores Fibonacci de los enteros 20, 30 y 35 (líneas 18 a 20). Las llamadas a la función **fibonacci** (líneas 15, 18, 19 y 20) de **main** no son llamadas recursivas, pero las llamadas de la línea 30 de **fibonacci** son recursivas. Cada vez que el programa invoca a **fibonacci** (líneas 25 a 31), la función evalúa de inmediato el caso base para determinar si **numero** es igual a 0 o 1 (línea 27). Si esto es verdadero, en la línea 28 se devuelve **numero**. Lo interesante es que, si **numero** es mayor que 1, el paso recursivo (línea 30) genera *dos* llamadas recursivas, cada una para un problema ligeramente más pequeño que la llamada original a **fibonacci**. La figura 6.31 muestra cómo la función **fibonacci** evaluaría **fibonacci(3)**.

```

1 // Fig. 6.30: fig06_30.cpp
2 // Prueba de la función recursiva fibonacci.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 unsigned long fibonacci( unsigned long ); // prototipo de función
9
10 int main()
11 {
12     // calcula los valores de fibonacci del 0 al 10
13     for ( int contador = 0; contador <= 10; contador++ )
14         cout << "fibonacci( " << contador << " ) = "
15         << fibonacci( contador ) << endl;
16
17     // muestra valores de fibonacci mayores
18     cout << "fibonacci( 20 ) = " << fibonacci( 20 ) << endl;
19     cout << "fibonacci( 30 ) = " << fibonacci( 30 ) << endl;
20     cout << "fibonacci( 35 ) = " << fibonacci( 35 ) << endl;
21     return 0; // indica que terminó correctamente
22 } // fin de main
23
24 // método fibonacci recursivo
25 unsigned long fibonacci( unsigned long numero )
26 {
27     if ( ( numero == 0 ) || ( numero == 1 ) ) // casos base
28         return numero;
29     else // paso recursivo
30         return fibonacci( numero - 1 ) + fibonacci( numero - 2 );
31 } // fin de la función fibonacci

```

Figura 6.30 | Demostración de la función **fibonacci**. (Parte I de 2).

```

fibonacci( 0 ) = 0
fibonacci( 1 ) = 1
fibonacci( 2 ) = 1
fibonacci( 3 ) = 2
fibonacci( 4 ) = 3
fibonacci( 5 ) = 5
fibonacci( 6 ) = 8
fibonacci( 7 ) = 13
fibonacci( 8 ) = 21
fibonacci( 9 ) = 34
fibonacci( 10 ) = 55
fibonacci( 20 ) = 6765
fibonacci( 30 ) = 832040
fibonacci( 25 ) = 9227465

```

Figura 6.30 | Demostración de la función fibonacci. (Parte 2 de 2).

Esta figura genera ciertas preguntas interesantes, en cuanto al orden en el que los compiladores de C++ evalúan los operandos de los operadores. Este orden es distinto del orden en el que se aplican los operadores a sus operandos; a saber, el orden que dictan las reglas de la precedencia y asociatividad de los operadores. La figura 6.31 muestra que al evaluar fibonacci(3), se producen dos llamadas recursivas: fibonacci(2) y fibonacci(1). Pero ¿en qué orden se harán estas llamadas?

La mayoría de los programadores simplemente suponen que los operandos se evalúan de izquierda a derecha. C++ no especifica el orden en el que se van a evaluar los operandos de la mayoría de los operadores (incluyendo +). Por lo tanto, no debemos hacer suposiciones en cuanto al orden en el que se evaluarán estas llamadas. De hecho, se podría ejecutar fibonacci(2) primero y después fibonacci(1), o se podrían ejecutar en el orden inverso: fibonacci(1) y luego fibonacci(2). En este programa y en la mayoría de los otros programas, el resultado sería el mismo. Sin embargo, en algunos programas la evaluación de un operando puede tener **efectos secundarios** (cambios en los valores de los datos) que podrían afectar al resultado final de la expresión.

C++ especifica el orden de evaluación de los operandos sólo para cuatro operadores: a saber, `&&`, `||`, el operador coma (,) y `?:`. Los primeros tres son operadores binarios, y se garantiza que sus dos operandos se evaluarán de izquierda a derecha. El último operador es el único operador ternario de C++. Su operando de más a la izquierda siempre se evalúa primero; si se evalúa como un valor distinto de cero (`true`), el operando intermedio se evalúa a continuación y se ignora el último operando; si el operando de más a la izquierda se evalúa como cero (`false`), el tercer operando se evalúa a continuación y se ignora el operando intermedio.

Error común de programación 6.25



La escritura de programas que dependan en el orden de evaluación de los operandos de operadores distintos de `&&`, `||`, `?:` y el operador coma (,) pueden producir errores lógicos.

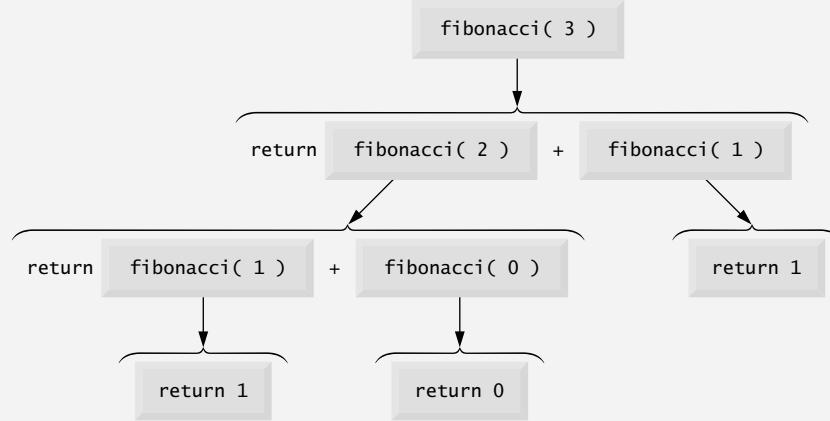


Figura 6.31 | Conjunto de llamadas recursivas a la función fibonacci.



Tip de portabilidad 6.3

Los programas que dependen del orden de evaluación de los operandos de operadores distintos de &&, ||, ?: y el operador coma (,) pueden funcionar de manera distinta en sistemas con distintos compiladores.

Hay que tener cuidado con los programas recursivos, como el que usamos aquí para generar números de Fibonacci. Cada nivel de recursividad en la función `fibonacci` tiene un efecto de duplicación sobre el número de llamadas a funciones; es decir, el número de llamadas recursivas que se requieren para calcular el n -ésimo número de Fibonacci se encuentra en el orden de 2^n . Esto se sale rápidamente de control. Para calcular sólo el 20vo. número de Fibonacci se requiera un orden de 2^{20} , o cerca de un millón de llamadas, para calcular el 30vo. número de Fibonacci se requeriría un orden de 2^{30} , o aproximadamente mil millones de llamadas, y así en lo sucesivo. Los científicos computacionales se refieren a esto como **complejidad exponencial**. Los problemas de esta naturaleza pueden humillar incluso hasta a las computadoras más poderosas del mundo. Las cuestiones relacionadas con la complejidad, tanto en general como en particular, se discuten con detalle en un curso del plan de estudios de ciencias computacionales de nivel superior, al que generalmente se le llama “Algoritmos”.



Tip de rendimiento 6.8

Evite los programas recursivos al estilo Fibonacci que produzcan una “explosión” exponencial de llamadas.

6.21 Comparación entre recursividad e iteración

En las dos secciones anteriores, estudiamos dos funciones que pueden implementarse mediante recursividad e iteración. En esta sección comparamos las dos metodologías, y hablamos acerca de por qué podríamos elegir una metodología sobre la otra en una situación específica.

Tanto la iteración como la recursividad se basan en una instrucción de control: la iteración utiliza una estructura de repetición; la recursividad utiliza una estructura de selección. Tanto la iteración como la recursividad implican la repetición: la iteración utiliza en forma explícita una estructura de repetición; la recursividad logra la repetición a través de llamadas repetidas a una función. La iteración y la recursividad implican una prueba de terminación: la iteración termina cuando falla la condición de continuación de ciclo; la recursividad termina cuando se reconoce un caso base. La iteración mediante la repetición controlada por un contador y la recursividad se acercan gradualmente a la terminación: la iteración modifica un contador hasta que éste asuma un valor que haga que falle la condición de continuación de ciclo; la recursividad produce versiones más simples del problema original hasta que se llega al caso base. Tanto la iteración como la recursividad pueden ocurrir infinitamente: un ciclo infinito ocurre con la iteración si la prueba de continuación de ciclo nunca se vuelve falsa; la recursividad infinita ocurre si el paso recursivo no reduce el problema durante cada llamada recursiva, de forma tal que llegue a converger en el caso base.

Para ilustrar las diferencias entre la iteración y la recursividad, vamos a examinar una solución iterativa para el problema del factorial (figura 6.32). Observe que se utiliza una instrucción de repetición (líneas 28 y 29 de la figura 6.32), en vez de la instrucción de selección de la solución recursiva (líneas 24 a 27 de la figura 6.29). Observe que ambas soluciones usan una prueba de terminación. En la solución recursiva, en la línea 24 se evalúa el caso base. En la solución iterativa, en la línea 28 se evalúa la condición de continuación de ciclo; si la prueba falla, el ciclo termina. Por último, observe que en vez de producir versiones cada vez más pequeñas del problema original, la solución iterativa utiliza un contador que se modifica hasta que la condición de continuación de ciclo se vuelve falsa.

```

1 // Fig. 6.32: fig06_32.cpp
2 // Prueba del método iterativo para el factorial.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 unsigned long factorial( unsigned long ); // prototipo de función

```

Figura 6.32 | Solución iterativa para el factorial. (Parte 1 de 2).

```

11
12 int main()
13 {
14     // calcula los factoriales del 0 al 10
15     for ( int contador = 0; contador <= 10; contador++ )
16         cout << setw( 2 ) << contador << "!" = " << factorial( contador )
17         << endl;
18
19     return 0;
20 } // fin de main
21
22 // método iterativo para el factorial
23 unsigned long factorial( unsigned long numero )
24 {
25     unsigned long resultado = 1;
26
27     // cálculo iterativo del factorial
28     for ( unsigned long i = numero; i >= 1; i-- )
29         resultado *= i;
30
31     return resultado;
32 } // fin de la función factorial

```

```

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
0! = 3628800

```

Figura 6.32 | Solución iterativa para el factorial. (Parte 2 de 2).

La recursividad tiene muchas desventajas. Invoca al mecanismo en forma repetida, y en consecuencia se produce una sobrecarga de las llamadas a la función. Esta repetición puede ser perjudicial, en términos de tiempo del procesador y espacio de la memoria. Cada llamada recursiva crea otra copia de la función (en realidad, sólo las variables de ésta, que se almacenan en el registro de activación); este conjunto de copias puede consumir una cantidad considerable de espacio en memoria. Como la iteración ocurre comúnmente dentro de un método, se evitan las llamadas repetidas a la función y la asignación adicional de memoria. Entonces, ¿por qué elegir la recursividad?



Observación de Ingeniería de Software 6.18

Cualquier problema que se pueda resolver mediante la recursividad, se puede resolver también mediante la iteración (sin recursividad). Por lo general se prefiere un método recursivo a uno iterativo cuando el primero refleja con más naturalidad el problema, y se produce un programa más fácil de entender y de depurar. Otra razón por la que es preferible elegir una solución recursiva es que una iterativa podría no ser aparente.



Tip de rendimiento 6.9

Evite usar la recursividad en situaciones en las que se requiera un alto rendimiento. Las llamadas recursivas requieren tiempo y consumen memoria adicional.



Error común de programación 6.26

Hacer que un método no recursivo se llame a sí mismo por accidente, ya sea en forma directa o indirecta (a través de otra función), es un error lógico.

La mayoría de los libros de texto introducen la recursividad en los últimos capítulos. Nuestra creencia es que la recursividad es un tema lo bastante extenso y complejo como para introducirlo mejor en los primeros capítulos, y esparcir los ejemplos en el resto del libro. En la figura 6.33 se sintetizan los ejemplos de recursividad y los ejercicios que se incluyen en el libro.

Ubicación en el libro	Ejemplos y ejercicios de recursividad
<i>Capítulo 6</i>	
Sección 6.19, fig. 6.29	Función factorial
Sección 6.19, fig. 6.30	Función Fibonacci
Ejercicio 6.40	Elevar un entero a una potencia entera
Ejercicio 6.42	Torre de Hanoi
Ejercicio 6.44	Visualización de la recursividad
Ejercicio 6.45	Máximo común divisor
Ejercicio 6.50 y 6.51	Ejercicio misterioso “¿Qué hace este programa?”
<i>Capítulo 7</i>	
Ejercicio 7.18	Ejercicio misterioso “¿Qué hace este programa?”
Ejercicio 7.21	Ejercicio misterioso “¿Qué hace este programa?”
Ejercicio 7.31	Ordenamiento por selección
Ejercicio 7.32	Determinar si una cadena es un palíndromo
Ejercicio 7.33	Búsqueda lineal
Ejercicio 7.34	Ocho reinas
Ejercicio 7.35	Imprimir un arreglo
Ejercicio 7.36	Imprimir un arreglo en forma inversa
Ejercicio 7.37	Mínimo valor en un arreglo
<i>Capítulo 8</i>	
Ejercicio 8.24	Quicksort
Ejercicio 8.25	Recorrido de laberinto
Ejercicio 8.26	Generación de laberintos al azar
Ejercicio 8.27	Laberintos de cualquier tamaño
<i>Capítulo 19</i>	
Sección 19.3.3, figs. 19.5 y 19.7	Ordenamiento por combinación
Ejercicio 19.8	Búsqueda lineal
Ejercicio 19.9	Búsqueda binaria
Ejercicio 19.10	Quicksort
<i>Capítulo 20</i>	
Sección 20.7, figs. 20.20 a 20.22	Inserción de árboles binarios
Sección 20.7, figs. 20.20 a 20.22	Recorrido preorden de un árbol binario
Sección 20.7, figs. 20.20 a 20.22	Recorrido inorden de un árbol binario
Sección 20.7, figs. 20.20 a 20.22	Recorrido postorden de un árbol binario
Ejercicio 20.20	Imprimir una lista enlazada en forma inversa
Ejercicio 20.21	Buscar en una lista enlazada
Ejercicio 20.22	Eliminación de árboles binarios
Ejercicio 20.23	Búsqueda de árboles binarios
Ejercicio 20.24	Recorrido por orden de nivel de un árbol binario
Ejercicio 20.25	Árbol de impresión

Figura 6.33 | Resumen de los ejemplos y ejercicios de recursividad en el libro.

6.22 (Opcional) Ejemplo práctico de Ingeniería de Software: identificación de las operaciones de las clases en el sistema ATM

En las secciones del Ejemplo práctico de Ingeniería de Software al final de los capítulos 3, 4 y 5, llevamos a cabo los primeros pasos en el diseño orientado a objetos de nuestro sistema ATM. En el capítulo 3 identificamos las clases que necesitaremos implementar, y creamos nuestro primer diagrama de clases. En el capítulo 4 describimos varios atributos de nuestras clases. En el capítulo 5 examinamos los estados de nuestros objetos y modelamos sus transiciones de estado y actividades. En esta sección determinaremos algunas de las operaciones (o comportamientos) de las clases que son necesarios para implementar el sistema ATM.

Identificar las operaciones

Una operación es un servicio que proporcionan los objetos de una clase a los clientes de esa clase. Considere las operaciones de algunos objetos reales. Las operaciones de un radio incluyen el sintonizar su estación y ajustar su volumen (que por lo general lo hace una persona que ajusta los controles del radio). Las operaciones de un auto incluyen acelerar (operación invocada por el conductor cuando oprime el pedal del acelerador), desacelerar (operación invocada por el conductor cuando oprime el pedal del freno o cuando suelta el pedal del acelerador), dar vuelta y cambiar velocidades. Los objetos de software también pueden ofrecer operaciones; por ejemplo, un objeto de gráficos de software podría ofrecer operaciones para dibujar un círculo, dibujar una línea, dibujar un cuadrado, etcétera. Un objeto de software de hoja de cálculo podría ofrecer operaciones tales como imprimir la hoja de cálculo, totalizar los elementos en una fila o columna, y graficar la información de la hoja de cálculo como un gráfico de barras o de pastel.

Podemos derivar muchas de las operaciones de cada clase mediante un análisis de los verbos y las frases verbales clave en la especificación de requerimientos. Después relacionamos cada una de ellas con las clases específicas en nuestro sistema (figura 6.34). Las frases verbales en la figura 6.34 nos ayudan a determinar las operaciones de cada clase.

Modelar las operaciones

Para identificar las operaciones, analizamos las frases verbales que se listan para cada clase en la figura 6.34. La frase “ejecuta transacciones financieras” asociada con la clase ATM implica que esta clase instruye a las transacciones a que se ejecuten. Por lo tanto, cada una de las clases **SolicitudSaldo**, **Retiro** y **Depósito** necesitan una operación para proporcionar este servicio al ATM. Colocamos esta operación (que hemos nombrado **ejecutar**) en el tercer compartimiento de las tres clases de transacciones en el diagrama de clases actualizado de la figura 6.35. Durante una sesión con el ATM, el objeto ATM invocará a la operación **ejecutar** de cada objeto transacción, para indicarle que se ejecute.

Para representar las operaciones (que se implementan como funciones miembro en C++), UML lista el nombre de la operación, seguido de una lista separada por comas de parámetros entre paréntesis, un signo de punto y coma y el tipo de valor de retorno:

nombreOperación(parámetro1, parámetro2, ..., parámetroN) : tipo de valor de retorno

Cada parámetro en la lista separada por comas consiste en un nombre de parámetro, seguido de un signo de punto y coma y del tipo del parámetro:

nombreParámetro : tipoParámetro

Por el momento, no listamos los parámetros de nuestras operaciones; en breve identificaremos y modelaremos los parámetros de algunas de las operaciones. Para algunas de estas operaciones no conocemos todavía los tipos de valores de retorno, por lo que también las omitiremos del diagrama. Estas omisiones son perfectamente normales en este punto. A medida que avancemos en nuestro proceso de diseño e implementación, agregaremos el resto de los tipos de valores de retorno.

Operaciones de la clase BaseDatosBanco y la clase Cuenta

La figura 6.34 lista la frase “autentica a un usuario” enseguida de la clase **BaseDatosBanco**; la base de datos es el objeto que contiene la información necesaria de la cuenta para determinar si el número de cuenta y el NIP introducidos por un usuario concuerdan con los de una cuenta en el banco. Por lo tanto, la clase **BaseDatosBanco** necesita una operación que proporcione un servicio de autenticación al ATM. Colocamos la operación **autenticarUsuario** en el tercer compartimiento de la clase **BaseDatosBanco** (figura 6.35). No obstante, un objeto de la clase **Cuenta** y no de la clase **Base-Datos-Banco** es el que almacena el número de cuenta y el NIP a los que se debe acceder para autenticar a un usuario, por lo que la clase **Cuenta** debe proporcionar un servicio para validar un NIP obtenido como entrada del usuario, y compararlo con un NIP almacenado en un objeto **Cuenta**. Por ende, agregamos una operación **validarNIP** a la clase **Cuenta**. Observe que especificamos un tipo de valor Boolean para las operaciones **autenticarUsuario**

y `validarNIP`.y `validarNIP`. Cada operación devuelve un valor que indica que la operación tuvo éxito al realizar su tarea (es decir, un valor de retorno `true`) o que no tuvo éxito (es decir, un valor de retorno `false`).

Clase	Verbos y frases verbales
ATM	ejecuta transacciones financieras
SolicitudSaldo	[ninguna en el documento de requerimientos]
Retiro	[ninguna en el documento de requerimientos]
Deposito	[ninguna en el documento de requerimientos]
BaseDeDatosBanco	autentica a un usuario, obtiene el saldo de una cuenta, abona un monto de depósito a una cuenta, carga un monto de retiro a una cuenta
Cuenta	obtiene el saldo de una cuenta, abona un monto de depósito a una cuenta, carga un monto de retiro a una cuenta
Pantalla	muestra un mensaje al usuario
Teclado	recibe entrada numérica del usuario
DispensadorEfectivo	dispensa efectivo, indica si contiene suficiente efectivo para satisfacer una solicitud de retiro
RanuraDeposito	recibe un sobre de depósito

Figura 6.34 | Verbos y frases verbales para cada clase en el sistema ATM.

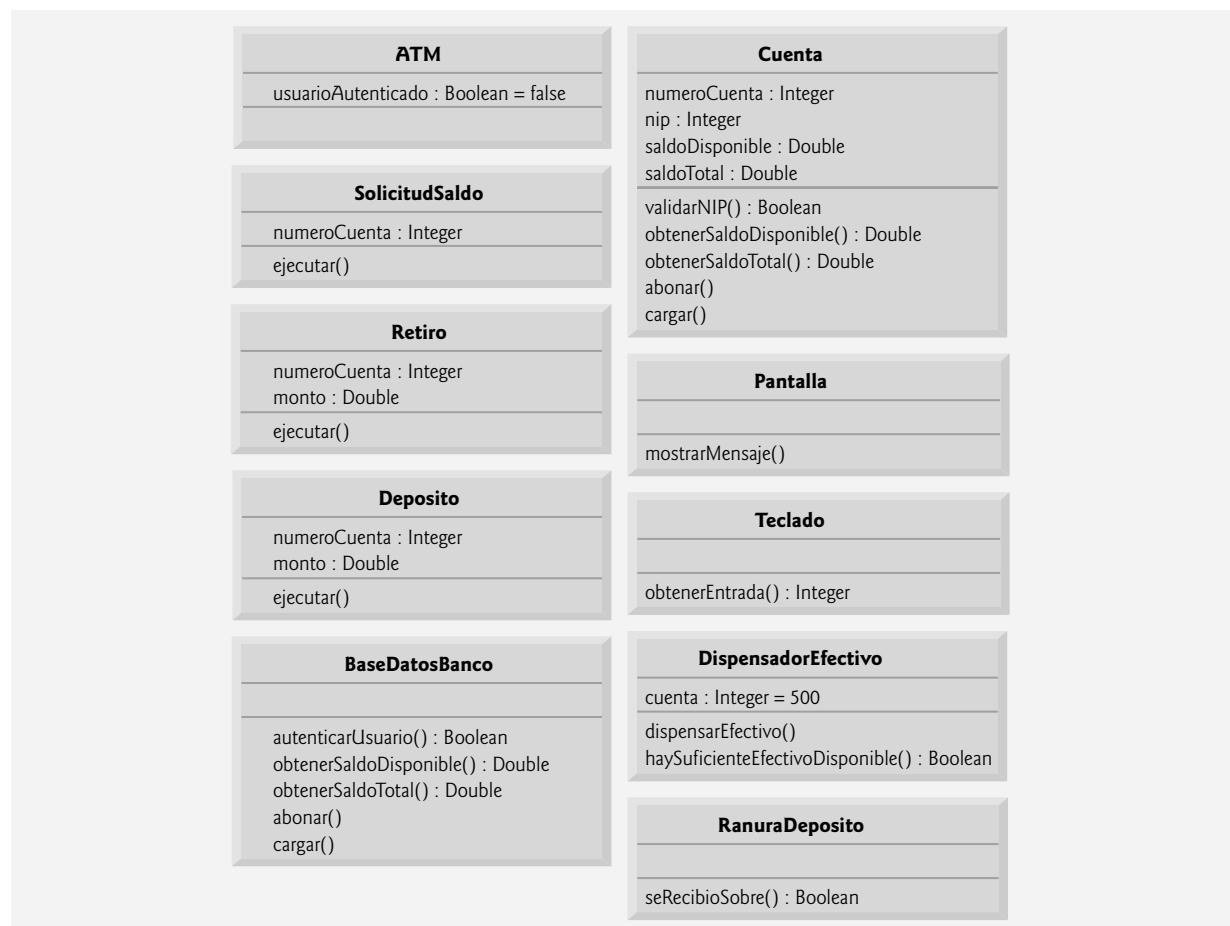


Figura 6.35 | Las clases en el sistema ATM, con atributos y operaciones.

La figura 6.34 lista varias frases verbales adicionales para la clase `BaseDatosBanco`: “obtiene el saldo de una cuenta”, “abona un monto de depósito a una cuenta” y “carga un monto de retiro a una cuenta”. Al igual que “autentica a un usuario”, estas frases restantes se refieren a los servicios que debe proporcionar la base de datos al ATM, ya que la base de datos almacena todos los datos de las cuentas que se utilizan para autenticar a un usuario y realizar transacciones con el ATM. No obstante, los objetos de la clase `Cuenta` son los que en realidad realizan las operaciones a las que se refieren estas frases. Por ello, asignamos una operación tanto a la clase `BaseDatosBanco` como a la clase `Cuenta`, que corresponda con cada una de estas frases. En la sección 3.11 vimos que, como una cuenta de banco contiene información delicada, no permitimos que el ATM acceda a las cuentas en forma directa. La base de datos actúa como un intermediario entre el ATM y los datos de la cuenta, evitando el acceso no autorizado. Como veremos en la sección 7.12, la clase `ATM` invoca las operaciones de la clase `BaseDatosBanco`, cada una de las cuales a su vez invoca a la operación con el mismo nombre en la clase `Cuenta`.

La frase “obtiene el saldo de una cuenta” sugiere que las clases `BaseDatosBanco` y `Cuenta` necesitan una operación `obtenerSaldo`. Sin embargo, recuerde que creamos dos atributos en la clase `Cuenta` para representar un saldo: `saldoDisponible` y `saldoTotal`. Una solicitud de saldo requiere el acceso a estos dos atributos del saldo, de manera que pueda mostrarlos al usuario, pero un retiro sólo requiere verificar el valor de `saldoDisponible`. Para permitir que los objetos en el sistema obtengan cada atributo de saldo en forma individual, agregamos las operaciones `obtenerSaldoDisponible` y `obtenerSaldoTotal` al tercer compartimiento de las clases `BaseDatosBanco` y `Cuenta` (figura 6.35). Especificamos un tipo de valor de retorno `Double` para estas operaciones, debido a que los atributos de los saldos que van a obtener son de tipo `Double`.

Las frases “abona un monto de depósito a una cuenta” y “carga un monto de retiro a una cuenta” indican que las clases `BaseDatosBanco` y `Cuenta` deben realizar operaciones para actualizar una cuenta durante un depósito y un retiro, respectivamente. Por lo tanto, asignamos las operaciones `abonar` y `cargar` a las clases `BaseDatosBanco` y `Cuenta`. Tal vez recuerde que cuando se abona a una cuenta (como en un depósito) se suma un monto sólo al atributo `saldoTotal`. Por otro lado, cuando se carga a una cuenta (como en un retiro) se resta el monto de ambos atributos de saldo. Ocultamos estos detalles de implementación dentro de la clase `Cuenta`. Éste es un buen ejemplo de encapsulamiento y ocultamiento de información.

Si éste fuera un sistema ATM real, las clases `BaseDatosBanco` y `Cuenta` también proporcionarían un conjunto de operaciones para permitir que otro sistema bancario actualizara el saldo de la cuenta de un usuario después de confirmar o rechazar todo, o parte de, un depósito. Por ejemplo, la operación `confirmarMontoDeposito` sumaría un monto al atributo `saldoDisponible`, y haría que los fondos depositados estuvieran disponibles para retirarlos. La operación `rechazarMontoDeposito` restaría un monto al atributo `saldoTotal` para indicar que un monto especificado, que se había depositado recientemente a través del ATM y se había sumado al `saldoTotal`, no se encontró en el sobre de depósito. El banco invocaría esta operación después de determinar que el usuario no incluyó el monto correcto de efectivo o que algún cheque no fue validado (es decir, que “rebotó”). Aunque al agregar estas operaciones nuestro sistema estaría más completo, no las incluiremos en nuestros diagramas de clases ni en nuestra implementación, ya que se encuentran más allá del alcance de este ejemplo práctico.

Operaciones de la clase Pantalla

La clase `Pantalla` “muestra un mensaje al usuario” en diversos momentos durante una sesión con el ATM. Toda la salida visual se produce a través de la pantalla del ATM. La especificación de requerimientos describe muchos tipos de mensajes (por ejemplo, un mensaje de bienvenida, un mensaje de error, un mensaje de agradecimiento) que la pantalla muestra al usuario. La especificación de requerimientos también indica que la pantalla muestra indicadores y menús al usuario. No obstante, un indicador es en realidad sólo un mensaje que describe lo que el usuario debe introducir a continuación, y un menú es en esencia un tipo de indicador que consiste en una serie de mensajes (es decir, las opciones del menú) que se muestran en forma consecutiva. Por lo tanto, en vez de asignar a la clase `Pantalla` una operación individual para mostrar cada tipo de mensaje, indicador y menú, basta con crear una operación que pueda mostrar cualquier mensaje especificado por un parámetro. Colocamos esta operación (`mostrarMensaje`) en el tercer compartimiento de la clase `Pantalla` en nuestro diagrama de clases (figura 6.35). Observe que no nos preocupa el parámetro de esta operación en estos momentos; lo modelaremos más adelante en esta sección.

Operaciones de la clase Teclado

De la frase “recibe entrada numérica del usuario” listada por la clase `Teclado` en la figura 6.34, podemos concluir que la clase `Teclado` debe realizar una operación `obtenerEntrada`. A diferencia del teclado de una computadora, el teclado del ATM sólo contiene los números del 0 al 9, por lo cual especificamos que esta operación devuelve un valor entero. Si recuerda, en la especificación de requerimientos vimos que en distintas situaciones, tal vez se requiera que el usuario introduzca un tipo distinto de número (por ejemplo, un número de cuenta, un NIP, el número de una opción del menú, un monto de depósito como número de centavos). La clase `Teclado` tan sólo obtiene un valor numérico para un cliente de la clase; no determina si el valor cumple con algún criterio específico. Cualquier clase que utilice esta operación debe

verificar que el usuario haya introducido números apropiados y, si no es así, debe mostrar mensajes de error a través de la clase `Pantalla`. [Nota: cuando implementemos el sistema, simularemos el teclado del ATM con el teclado de una computadora y, por cuestión de simpleza, asumiremos que el usuario no va a escribir datos de entrada que no sean números, usando las teclas en el teclado de la computadora que no aparezcan en el teclado del ATM. Más adelante en el libro, aprenderá a examinar las entradas para determinar si son de tipos específicos.]

Operaciones de las clases DispensadorEfectivo y RanuraDepósito

La figura 6.34 lista la frase “dispensa efectivo” para la clase `DispensadorEfectivo`. Por lo tanto, creamos la operación `dispensarEfectivo` y la listamos bajo la clase `DispensadorEfectivo` en la figura 6.35. La clase `DispensadorEfectivo` también “indica si contiene suficiente efectivo para satisfacer una solicitud de retiro”. Para esto incluimos a `haySuficienteEfectivoDisponible`, una operación que devuelve un valor de tipo `Boolean` de UML, en la clase `DispensadorEfectivo`. La figura 6.34 también lista la frase “recibe un sobre de depósito” para la clase `RanuraDepósito`. La ranura de depósito debe indicar si recibió un sobre, por lo que colocamos una operación `seRecibioSobre`, la cual devuelve un valor `Boolean`, en el tercer compartimiento de la clase `RanuraDepósito`. [Nota: es muy probable que una ranura de depósito de hardware real envíe una señal al ATM para indicarle que se recibió un sobre. No obstante, simularemos este comportamiento con una operación en la clase `RanuraDepósito`, que la clase `ATM` pueda invocar para averiguar si la ranura de depósito recibió un sobre.]

Operaciones de la clase ATM

No listamos ninguna operación para la clase `ATM` en este momento. Todavía no sabemos de algún servicio que proporcione la clase `ATM` a otras clases en el sistema. No obstante, cuando implementemos el sistema en código de C++, tal vez emerjan las operaciones de esta clase junto con las operaciones adicionales de las demás clases en el sistema.

Identificar y modelar los parámetros de operación

Hasta ahora no nos hemos preocupado por los parámetros de nuestras operaciones, sólo hemos tratado de obtener una comprensión básica de las operaciones de cada clase. Ahora vamos a dar un vistazo más de cerca a varios parámetros de operación. Para identificar los parámetros de una operación, analizamos qué datos requiere la operación para realizar su tarea asignada.

Considere la operación `autenticarUsuario` de la clase `BaseDatosBanco`. Para autenticar a un usuario, esta operación debe conocer el número de cuenta y el NIP que suministra el usuario. Por lo tanto, especificamos que la operación `autenticarUsuario` debe recibir los parámetros enteros `numeroCuentaUsuario` y `nipUsuario`, que la operación debe comparar con el número de cuenta y el NIP de un objeto `Cuenta` en la base de datos. Vamos a colocar después de estos nombres de parámetros la palabra “usuario”, para evitar confusión entre los nombres de los parámetros de la operación y los nombres de los atributos que pertenecen a la clase `Cuenta`. Listamos estos parámetros en el diagrama de clases de la figura 6.36, el cual modela sólo a la clase `BaseDatosBanco`. [Nota: es perfectamente normal modelar sólo una clase en un diagrama de clases. En este caso lo que más nos preocupa es analizar los parámetros de esta clase específica, por lo que omitimos las demás clases. Más adelante en los diagramas de clase de este ejemplo práctico, donde los parámetros dejarán de ser el centro de nuestra atención, los omitiremos para ahorrar espacio. No obstante, recuerde que las operaciones que se listan en estos diagramas siguen teniendo parámetros.]

Recuerde que para modelar a cada parámetro en una lista de parámetros separados por comas, UML lista el nombre del parámetro, seguido de un signo de dos puntos y el tipo del parámetro (en notación de UML). Así, la figura 6.36 especifica que la operación `autenticarUsuario` recibe dos parámetros: `numeroCuentaUsuario` y `nipUsuario`, ambos de tipo `Integer`. Cuando implementemos el sistema en C++, representaremos estos parámetros con valores `int`.

Las operaciones `obtenerSaldoDisponible`, `obtenerSaldoTotal`, `abonar` y `cargar` de la clase `BaseDatosBanco` también requieren un parámetro `nombreCuentaUsuario` para identificar la cuenta a la cual la base de datos debe aplicar las operaciones, por lo que incluimos estos parámetros en el diagrama de clases de la figura 6.36. Además, las operaciones `abonar` y `cargar` requieren un parámetro `Double` llamado `monto`, para especificar el monto de dinero que se va a abonar o a cargar, respectivamente.

El diagrama de clases de la figura 6.37 modela los parámetros de las operaciones de la clase `Cuenta`. La operación `validarNIP` sólo requiere un parámetro `nipUsuario`, el cual contiene el NIP especificado por el usuario, que se va a comparar con el NIP asociado a la cuenta. Al igual que sus contrapartes en la clase `BaseDatosBanco`, las operaciones `abonar` y `cargar` en la clase `Cuenta` requieren un parámetro `Double` llamado `monto`, el cual indica la cantidad de dinero involucrada en la operación. Las operaciones `obtenerSaldoDisponible` y `obtenerSaldoTotal` en la clase `Cuenta` no requieren datos adicionales para realizar sus tareas. Observe que las operaciones de la clase `Cuenta` no requieren un parámetro de número de cuenta ; cada una de estas operaciones se puede invocar sólo en un objeto `Cuenta` específico, por lo que no es necesario incluir un parámetro para especificar una `Cuenta`.

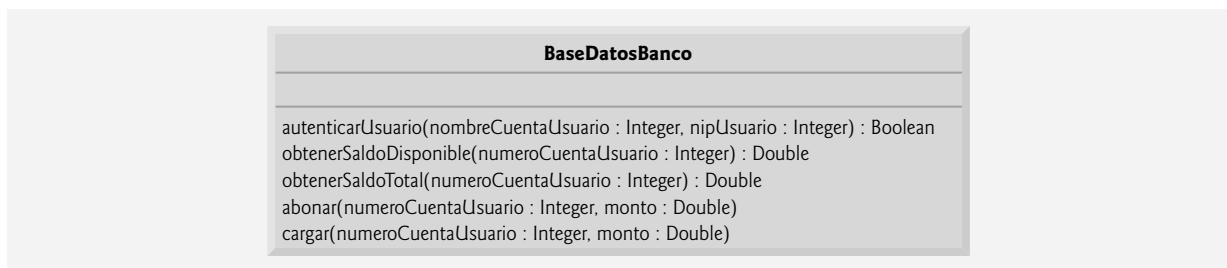


Figura 6.36 | La clase **BaseDatosBanco** con parámetros de operación.

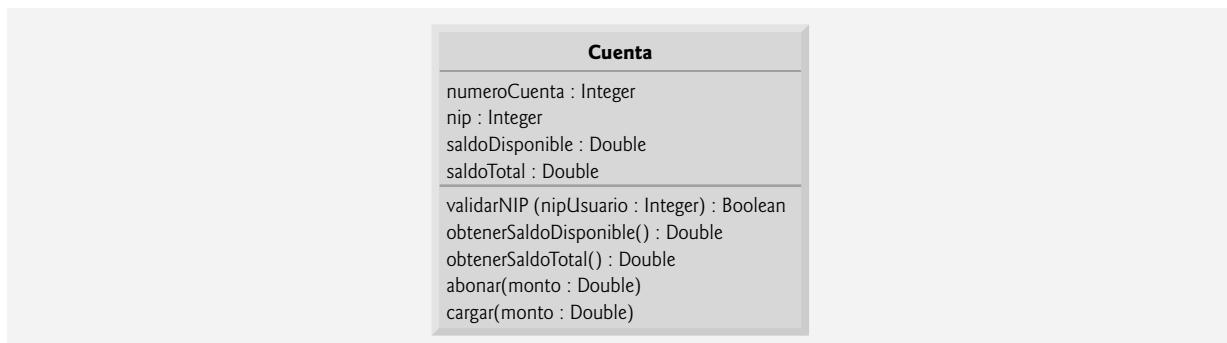


Figura 6.37 | La clase **Cuenta** con parámetros de operación.

La figura 6.38 modela la clase **Pantalla** con un parámetro especificado para la operación **mostrarMensaje**. Esta operación requiere sólo un parámetro **String** llamado **mensaje**, el cual indica el texto que debe mostrarse en pantalla. Recuerde que los tipos de los parámetros que se enlistan en nuestros diagramas de clases están en notación de UML, por lo que el tipo **String** que se enumera en la figura 6.38 se refiere al tipo de UML. Cuando implementemos el sistema en C++, utilizaremos de hecho un objeto **string** de C++ para representar este parámetro.

El diagrama de clases de la figura 6.39 especifica que la operación **dispensarEfectivo** de la clase **DispensadorEfectivo** recibe un parámetro **Double** llamado **monto** para indicar el monto de efectivo (en dólares) que se va a dispensar al usuario. La operación **haySuficienteEfectivoDisponible** también recibe un parámetro **Double** llamado **monto** para indicar el monto de efectivo en cuestión.

Observe que no hablamos sobre los parámetros para la operación **ejecutar** de las clases **SolicitudSaldo**, **Retiro** y **Depósito**, de la operación **obtenerEntrada** de la clase **Teculado** y la operación **seRecibioSobre** de la clase **RanuraDeposito**. En este punto de nuestro proceso de diseño, no podemos determinar si estas operaciones requieren datos adicionales para realizar sus tareas, por lo que dejaremos sus listas de parámetros vacías. A medida que avancemos por el ejemplo práctico, tal vez decidamos agregar parámetros a estas operaciones.

En esta sección hemos determinado muchas de las operaciones que realizan las clases en el sistema ATM. Identificamos los parámetros y los tipos de valores de retorno de algunas operaciones. A medida que continuemos con nuestro proceso de diseño, el número de operaciones que pertenezcan a cada clase puede variar; podríamos descubrir que se necesitan nuevas operaciones o que ciertas operaciones actuales no son necesarias; y podríamos determinar que algunas de las operaciones de nuestras clases necesitan parámetros adicionales y tipos de valores de retorno distintos.



Figura 6.38 | La clase **Pantalla** con parámetros de operación.

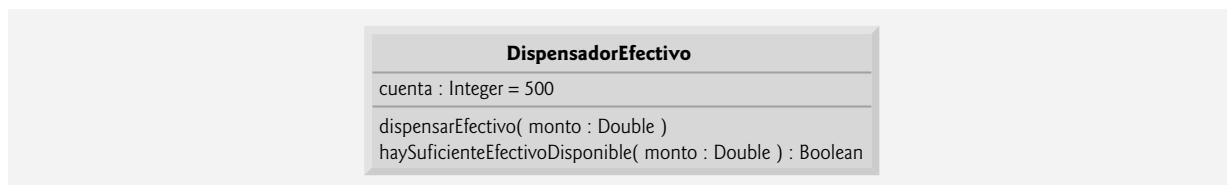


Figura 6.39 | La clase *DispensadorEfectivo* con parámetros de operación.

Ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

- 6.1** ¿Cuál de las siguientes opciones no es un comportamiento?
- leer datos de un archivo.
 - imprimir los resultados.
 - imprimir texto.
 - obtener la entrada del usuario.
- 6.2** Si quisiera agregar al sistema ATM una operación que devuelva el atributo *monto* de la clase *Retiro*, ¿cómo y en dónde especificaría esta operación en el diagrama de clases de la figura 6.35?
- 6.3** Describa el significado del siguiente listado de operaciones, el cual podría aparecer en un diagrama de clases para el diseño orientado a objetos de una calculadora:

```
sumar( x : Integer, y : Integer ) : Integer
```

Respuestas a los ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

- 6.1** c.
- 6.2** Para especificar una operación que obtenga el atributo *monto* de la clase *Retiro*, se debe colocar la siguiente operación en el (tercer) compartimiento de operaciones de la clase *Retiro*:
- ```
obtenerMonto() : Double
```
- 6.3** Ésta es una operación llamada *sumar*, la cual recibe los enteros *x* y *y* como parámetros y devuelve un valor entero.

## 6.23 Repaso

En este capítulo aprendió más acerca de los detalles de las declaraciones de funciones. Las funciones tienen distintas piezas, como el prototipo, la firma, el encabezado y el cuerpo de una función. Aprendió acerca de la coerción de argumentos, o la acción de forzar a que los argumentos sean de los tipos apropiados que se especifiquen mediante las declaraciones de los parámetros de una función. Demostramos cómo usar las funciones *rand* y *srand* para generar conjuntos de números aleatorios que se puedan utilizar para las simulaciones. Aprendió también acerca del alcance de las variables, o la porción de un programa donde se puede utilizar un identificador. Vimos dos formas distintas de pasar argumentos a las funciones: paso por valor y paso por referencia. Para el paso por referencia, las referencias se utilizan como un alias para una variable. Aprendió que varias funciones en una clase se pueden sobrecargar, usando el mismo nombre y distintas firmas. Dichas funciones se pueden utilizar para realizar las mismas tareas (o similares), usando distintos tipos o números de parámetros. Después, demostramos una manera más simple de sobrecargar funciones mediante las plantillas de función, donde una función se define una sola vez, pero se puede usar para varios tipos distintos. Posteriormente le presentamos el concepto de la recursividad, donde una función se llama a sí misma para resolver un problema.

En el capítulo 7 aprenderá a mantener listas y tablas de datos en arreglos. Veremos una implementación basada en arreglo más elegante de la aplicación para tirar dados, y dos versiones mejoradas de nuestro ejemplo práctico *Libro-Calificaciones* que estudiamos en los capítulos 3 al 5, donde utilizaremos arreglos para almacenar las calificaciones introducidas.

## Resumen

### Sección 6.1 Introducción

- La experiencia ha demostrado que la mejor forma de desarrollar y mantener un programa extenso es construirlo a partir de piezas (o componentes) simples y pequeñas. A esta técnica se le conoce como divide y vencerás.

## **Sección 6.2 Componentes de los programas en C++**

- Por lo general, los programas en C++ se escriben mediante la combinación de nuevas funciones y clases que escribimos con funciones “pre-empaquetadas”, y clases disponibles en la Biblioteca estándar de C++.
- Las funciones permiten al programador modularizar un programa, al separar sus tareas en unidades autocontenidoas.
- Las instrucciones en los cuerpos de las funciones se escriben sólo una vez, se pueden reutilizar desde varias ubicaciones en un programa; además están ocultas de las demás funciones.

## **Sección 6.3 Funciones matemáticas de la biblioteca**

- Algunas veces, las funciones no son miembros de una clase. A dichas funciones se les conoce como funciones globales.
- Los prototipos para las funciones globales se colocan en archivos de encabezado, de manera que las funciones globales se puedan reutilizar en cualquier programa que incluya el archivo de encabezado y pueda crear un enlace con el código objeto de la función.

## **Sección 6.4 Definiciones de funciones con varios parámetros**

- El compilador hace referencia al prototipo de función, para comprobar que las llamadas a una función tengan el número y tipos de argumentos correctos, que los tipos de los argumentos estén en el orden correcto y que el valor devuelto por la función se pueda utilizar de manera correcta en la expresión que llamó a la función.
- Hay tres formas de devolver el control al punto en el que se invocó a una función. Si la función no devuelve un resultado, el control regresa cuando el programa llega a la llave derecha de fin de la función, o mediante la ejecución de la instrucción

```
return;
```

Si la función devuelve un resultado, la instrucción

```
return expresión;
```

evalúa *expresión* y devuelve el valor de *expresión* a la función que hizo la llamada.

## **Sección 6.5 Prototipos de funciones y coerción de argumentos**

- Un prototipo de función indica al compilador el nombre de una función, el tipo de datos devuelto por la función, el número de parámetros que la función espera recibir, los tipos de esos parámetros y el orden en el que éstos se esperan.
- La porción de un prototipo de función que incluya el nombre de la función y los tipos de sus argumentos se conoce como la firma de la función, o simplemente firma.
- Una característica importante de los prototipos de función es la coerción de argumentos; es decir, obligar a que los argumentos tengan los tipos especificados por las declaraciones de los parámetros.
- Los valores de los argumentos que no corresponden precisamente a los tipos de los parámetros en el prototipo de función pueden ser convertidos por el compilador al tipo apropiado, según lo especificado por las reglas de promoción de C++. Las reglas de promoción indican cómo realizar conversiones entre tipos sin perder datos.

## **Sección 6.6 Archivos de encabezado de la Biblioteca estándar de C++**

- La Biblioteca estándar de C++ está dividida en muchas porciones, cada una con su propio archivo de encabezado. Los archivos de encabezado también contienen definiciones de varios tipos de clases, funciones y constantes.
- Un archivo de encabezado “instruye” al compilador acerca de cómo interconectarse con los componentes de la biblioteca y los componentes escritos por el usuario.

## **Sección 6.7 Ejemplo práctico: generación de números aleatorios**

- El elemento de azar puede introducirse en las aplicaciones computacionales mediante el uso de la función `rand` de la Biblioteca estándar de C++.
- En realidad, la función `rand` genera números seudoaleatorios. Si se llama repetidas veces a `rand`, se produce una secuencia de números que parecen ser aleatorios. No obstante, la secuencia se repite a sí misma cada vez que se ejecuta el programa.
- Una vez que un programa se ha depurado extensivamente, puede condicionarse para producir una secuencia diferente de números aleatorios para cada ejecución. A esto se le conoce como randomización, y se logra mediante la función `srand` de la Biblioteca estándar de C++.
- La función `srand` recibe un argumento entero `unsigned` y siembra la función `rand` para que produzca una secuencia distinta de números aleatorios para cada ejecución del programa.
- Los números aleatorios en un rango se pueden generar de la siguiente manera:

```
numero = valorDesplazamiento + rand() % factorEscala;
```

donde *valorDesplazamiento* especifica el primer número en el rango deseado de enteros consecutivos y *factorEscala* es igual a la anchura del rango deseado de enteros consecutivos.

### Sección 6.8 Ejemplo práctico: juego de probabilidad, introducción a las enumeraciones

- Una enumeración, que se introduce mediante la palabra clave enum y va seguida de un nombre de tipo, es un conjunto de constantes enteras representadas por identificadores. Los valores de estas constantes de enumeración empiezan en 0, a menos que se especifique lo contrario, y se incrementan en 1.

### Sección 6.9 Clases de almacenamiento

- La clase de almacenamiento de un identificador determina el periodo durante el cual éste existe en la memoria.
- El alcance de un identificador es la parte en la que se puede hacer referencia a éste en un programa.
- La vinculación de un identificador determina si se conoce sólo en el archivo fuente en el que se declara, o en varios archivos fuente que se compilen y después se enlacen.
- Las palabras clave auto y register se utilizan para declarar variables de la clase de almacenamiento automático. Dichas variables se crean cuando la ejecución del programa entra en el bloque en el que están definidas, existen mientras el bloque está activo y se destruyen cuando el programa sale del bloque.
- Sólo las variables locales de una función pueden ser de clase de almacenamiento automático.
- El especificador de clase de almacenamiento auto declara explícitamente variables de clase de almacenamiento automático. Las variables locales son de clase de almacenamiento automático de manera predeterminada, por lo que la palabra clave auto se utiliza raras veces.
- Las palabras clave extern y static declaran identificadores para variables de la clase de almacenamiento estático y para funciones. Las variables de clase de almacenamiento estático existen a partir del punto en el que el programa empieza a ejecutarse, y dejan de existir cuando termina el programa.
- El almacenamiento de una variable de clase de almacenamiento estático se asigna cuando el programa empieza su ejecución. Dicha variable se inicializa una vez al encontrar su declaración. Para las funciones, el nombre de la función existe cuando el programa empieza a ejecutarse, de igual forma que para las otras funciones.
- Hay dos tipos de identificadores con clase de almacenamiento estático; los identificadores externos (como las variables globales y los nombres de funciones globales) y las variables locales declaradas con el especificador de clase de almacenamiento static.
- Para crear variables globales, se colocan declaraciones de variables fuera de cualquier definición de clase o función. Las variables globales retienen sus valores a lo largo de la ejecución del programa. Las variables y las funciones globales se pueden referenciar mediante cualquier función que siga sus declaraciones o definiciones en el archivo fuente.

### Sección 6.10 Reglas de alcance

- Las variables locales que se declaran con la palabra clave static sólo son conocidas en la función en la que están declaradas pero, a diferencia de las variables automáticas, las variables locales static retienen sus valores cuando la función regresa a la función que la llamó. La siguiente vez que se hace una llamada a la función, las variables locales static contienen los valores que tenían cuando la función se ejecutó por última vez.
- Un identificador que se declara fuera de cualquier función o clase tiene alcance de archivo.
- Las etiquetas son los únicos identificadores con alcance de función. Las etiquetas se pueden utilizar en cualquier parte en la función en la que aparecen, pero no se pueden referenciar fuera del cuerpo de la función.
- Los identificadores que se declaran dentro de un bloque tienen alcance de bloque. El alcance de bloque empieza en la declaración del identificador y termina en la llave derecha de finalización (}) del bloque en el que se declara el identificador.
- Los únicos identificadores con alcance de prototipo de función son los que se utilizan en la lista de parámetros de un prototipo de función.

### Sección 6.11 La pila de llamadas a funciones y los registros de activación

- Las pilas se denominan estructuras de datos “último en entrar, primero en salir” (UEPS); el último elemento que se mete (inserta) en la pila es el primero que se saca (extrae) de ella.
- Uno de los mecanismos más importantes que los estudiantes de ciencias computacionales deben comprender es la pila de llamadas a funciones (conocida algunas veces como la pila de ejecución del programa). Esta estructura de datos soporta el mecanismo de llamada a/regreso de las funciones.
- La pila de llamadas a funciones también soporta la creación, mantenimiento y destrucción de las variables automáticas de cada función a la que se llama.
- Cada vez que una función llama a otra función, se mete una entrada en la pila. Esta entrada, conocida como marco de pila o registro de activación, contiene la dirección de retorno que necesita la función a la que se llamó para poder regresar a la función que hizo la llamada, junto con las variables automáticas y parámetros de la llamada a la función.
- El marco de pila existe mientras la función a la que se llamó esté activa. Cuando esa función regresa (y ya no necesita sus variables automáticas locales) su marco de pila se saca de la pila, y esas variables automáticas ya no son conocidas para el programa.

### Sección 6.12 Funciones con listas de parámetros vacías

- En C++, una lista de parámetros vacía se especifica mediante void o nada entre paréntesis.

### **Sección 6.13 Funciones en línea**

- C++ cuenta con las funciones en línea para ayudar a reducir la sobrecarga de las llamadas a funciones; en especial para las funciones pequeñas. Al colocar el calificador `inline` antes del tipo de valor de retorno de la función en su definición, se “aconseja” al compilador para que genere una copia del código de la función en ese lugar para evitar la llamada a una función.

### **Sección 6.14 Referencias y parámetros de referencias**

- Dos formas de pasar argumentos a las funciones en muchos lenguajes de programación son el paso por valor y el paso por referencia.
- Cuando se pasa un argumento por valor, se crea una *copia* del valor del argumento y se pasa (en la pila de llamadas a funciones) a la función que se llamó. Las modificaciones a la copia no afectan al valor de la variable original en la función que hizo la llamada.
- Mediante el paso por referencia, la función que hace la llamada proporciona a la función que llamó la habilidad de acceder directamente a los datos de la primera, y de modificar esos datos en caso de que la función que se llamó así lo decida.
- Un parámetro por referencia es un alias para su correspondiente argumento en la llamada a una función.
- Para indicar que un parámetro de función se pasa por referencia, simplemente hay que colocar un signo & después del tipo del parámetro en el prototipo de la función; use la misma convención al listar el tipo del parámetro en el encabezado de la función.
- Una vez que se declara una referencia como alias para otra variable, todas las operaciones que supuestamente se realizan en el alias (es decir, la referencia) en realidad se realizan en la variable original. El alias es simplemente otro nombre para la variable original.

### **Sección 6.15 Argumentos predeterminados**

- Para un programa, es algo común el invocar una función repetidas veces con el mismo valor de argumento para un parámetro específico. En tales casos, podemos especificar que dicho parámetro tiene un argumento predeterminado; es decir, que tiene un valor predeterminado que debe pasar a ese parámetro.
- Cuando un programa omite un argumento para un parámetro con un argumento predeterminado, el compilador vuelve a escribir la llamada a la función e inserta el valor predeterminado del argumento para pasarlo a la llamada a la función.
- Los argumentos predeterminados deben ser los argumentos de más a la derecha en la lista de parámetros de una función.
- Los argumentos predeterminados deben especificarse con la primera ocurrencia del nombre de la función; por lo general, en el prototipo de la función.

### **Sección 6.16 Operador de resolución de ámbito unario**

- C++ proporciona el operador de resolución de ámbito binario (::) para acceder a una variable global cuando una variable local con el mismo nombre se encuentra dentro del alcance.

### **Sección 6.17 Sobre carga de funciones**

- C++ permite definir varias funciones con el mismo nombre, siempre y cuando éstas tengan diferentes conjuntos de parámetros. A esta capacidad se le conoce como sobre carga de funciones.
- Cuando se hace una llamada a una función sobre cargada, el compilador de C++ selecciona la función apropiada al examinar el número, tipos y orden de los argumentos en la llamada.
- Las funciones sobre cargadas se diferencian mediante sus firmas.
- El compilador codifica cada identificador de función con el número y tipos de sus parámetros para permitir la vinculación segura de tipos. Este tipo de vinculación asegura que se llame a la función sobre cargada apropiada, y que los tipos de los argumentos se conformen a los tipos de los parámetros.

### **Sección 6.18 Plantillas de funciones**

- Por lo general, las funciones sobre cargadas se utilizan para realizar operaciones similares que involucren distintos tipos de lógica de programa en distintos tipos de datos. Si la lógica del programa y las operaciones son idénticas para cada tipo de datos, la sobre carga se puede llevar a cabo de una forma más compacta y conveniente, mediante el uso de plantillas de funciones.
- El programador escribe una sola definición de plantilla de función. Dados los tipos de los argumentos que se proporcionan en las llamadas a esta función, C++ genera de manera automática especializaciones de plantilla de función separadas para manejar cada tipo de llamada de manera apropiada. Por ende, al definir una sola plantilla de función, en esencia se define toda una familia de funciones sobre cargadas.
- Todas las definiciones de plantillas de función empiezan con la palabra clave `template` seguida de una lista de parámetros de plantilla para la plantilla de función encerrada entre los paréntesis angulares (< y >).
- Los parámetros de tipo formal son receptáculos para los tipos fundamentales, o los tipos definidos por el usuario. Estos receptáculos se utilizan para especificar los tipos de los parámetros de la función, para especificar el tipo de valor de retorno de la función y para declarar variables dentro del cuerpo de la definición de la función.

**Sección 6.19 Recursividad**

- Una función recursiva es una función que se llama a sí misma, ya sea en forma directa o indirecta.
- Una función recursiva sabe cómo resolver sólo el (los) caso(s) más simple(s), o caso(s) base. Si se hace la llamada a la función con un caso base, ésta simplemente devuelve un resultado.
- Si se hace la llamada a la función con un problema más complejo, la función comúnmente divide el problema en dos piezas conceptuales: una pieza que sabe cómo resolver y otra pieza que no sabe cómo resolver. Para que la recursividad sea factible, esta última pieza debe ser similar al problema original, pero una versión ligeramente más sencilla o simple del mismo.
- Para que la recursividad termine en un momento dado, cada vez que la función se llama a sí misma con una versión más simple del problema original, la secuencia de problemas cada vez más pequeños debe converger en el caso base.

**Sección 6.20 Ejemplo sobre el uso de la recursividad: serie de Fibonacci**

- La proporción de números de Fibonacci sucesivos converge en un valor constante de 1.618.... Este número también ocurre con frecuencia en la naturaleza, y se le ha denominado proporción dorada, o media dorada.

**Sección 6.21 Comparación entre recursividad e iteración**

- La iteración y la recursividad tienen muchas similitudes: ambas se basan en una instrucción de control, implican la repetición, implican una prueba de terminación, se acercan gradualmente a la terminación y pueden ocurrir infinitamente.
- La recursividad tiene muchas desventajas. Invoca al mecanismo en forma repetida, y en consecuencia se produce una sobrecarga de las llamadas a la función. Esta repetición puede ser perjudicial, en términos de tiempo del procesador y espacio de la memoria. Cada llamada recursiva crea otra copia de la función (en realidad, sólo las variables de ésta); este conjunto de copias puede consumir una cantidad considerable de espacio en memoria.

**Terminología**

|                                                |                                                  |
|------------------------------------------------|--------------------------------------------------|
| & para declarar una referencia                 | enteros desplazados y escalados                  |
| alcance de archivo                             | enum, palabra clave                              |
| alcance de bloque                              | enumeración                                      |
| alcance de clase                               | escalado                                         |
| alcance de espacio de nombres                  | especialización de plantilla de función          |
| alcance de función                             | especificadores de clase de almacenamiento       |
| alcance de prototipo de función                | etiqueta                                         |
| alcance de un identificador                    | evaluación recursiva                             |
| alias                                          | expresión de tipo mixto                          |
| anchura del rango de números aleatorios        | extern, especificador de clase de almacenamiento |
| argumento predeterminado                       | factor de escala                                 |
| argumentos de más a la derecha                 | factorial                                        |
| auto, especificador de clase de almacenamiento | Fibonacci, serie                                 |
| bloque exterior                                | firma                                            |
| bloque interior                                | firma de función                                 |
| bloques anidados                               | fuerza de alcance                                |
| caso(s) base                                   | función de plantilla                             |
| ciclo infinito                                 | función definida por el programador              |
| clase de almacenamiento                        | función definida por el usuario                  |
| clase de almacenamiento automática             | función en línea                                 |
| coerción de argumentos                         | función recursiva                                |
| compilador optimizador                         | funciones “pre-empaquetadas”                     |
| complejidad exponencial                        | global función                                   |
| condición de terminación                       | inicialización de una referencia                 |
| constante de enumeración                       | inline, palabra clave                            |
| converger en un caso base                      | invocar a un método                              |
| declaración de función                         | iteración                                        |
| decoración de nombres                          | límites de los tipos de datos numéricos          |
| definición de función                          | límites de tamaño entero                         |
| definición de plantilla                        | lista de parámetros de plantilla                 |
| desbordamiento de pila                         | llamada recursiva                                |
| desplaza un rango de números                   | llave derecha {} de fin de un bloque             |
| devolver una referencia de una función         | manipulación de nombres                          |
| efecto secundario de una expresión             | marco de pila                                    |

|                                                                 |                                                                  |
|-----------------------------------------------------------------|------------------------------------------------------------------|
| media dorada                                                    | referencia a una constante                                       |
| meter en una pila                                               | referencia a una variable automática                             |
| método “divide y vencerás”                                      | referencia suelta                                                |
| métodos                                                         | <code>register</code> , especificador de clase de almacenamiento |
| modularización de un programa mediante funciones                | registro de activación                                           |
| <code>mutable</code> , especificador de clase de almacenamiento | reglas de promoción                                              |
| nombre de función                                               | repetición de la función <code>rand</code>                       |
| nombre de función manipulado                                    | reutilización de software                                        |
| nombre de tipo (enumeraciones)                                  | sacar de una pila                                                |
| nombre de una variable                                          | secuencia de números aleatorios                                  |
| número aleatorio                                                | semilla                                                          |
| números seudoaleatorios                                         | sobrecarga                                                       |
| operador de resolución de ámbito unario (::)                    | sobrecarga de función                                            |
| parámetro                                                       | sobrecarga de la recursividad                                    |
| parámetro de tipo                                               | sobrecarga de llamadas a funciones                               |
| parámetro de tipo formal                                        | solución iterativa                                               |
| parámetro formal                                                | solución recursiva                                               |
| parámetro por referencia                                        | <code>srand</code> , función                                     |
| paso por referencia                                             | <code>static</code> , clase de almacenamiento                    |
| paso por valor                                                  | <code>static</code> , especificador de clase de almacenamiento   |
| paso recursivo                                                  | <code>static</code> , variable local                             |
| pila                                                            | <code>template</code> , palabra clave                            |
| pila de ejecución del programa                                  | tipo “más alto”                                                  |
| pila de llamadas a funciones                                    | tipo “más bajo”                                                  |
| plantilla de función                                            | tipo de una variable                                             |
| principio del menor privilegio                                  | tipo definido por el usuario                                     |
| procedimiento                                                   | truncar parte fraccionaria de un <code>double</code>             |
| proporción dorada                                               | UEPS (último en entrar, primero en salir)                        |
| prototipo de función                                            | validar la llamada a una función                                 |
| prototipos de función obligatorios                              | valor de desplazamiento                                          |
| prueba de terminación                                           | variable global                                                  |
| <code>rand</code> , función                                     | variable local automática                                        |
| <code>RAND_MAX</code> , constante simbólica                     | Vinculación                                                      |
| randomizar                                                      | vinculación segura para tipos                                    |
| recursividad                                                    | <code>void</code> , tipo de valor de retorno                     |
| recursividad infinita                                           |                                                                  |

## Ejercicios de autoevaluación

6.1 Complete las siguientes oraciones:

- a) En C++, los componentes de un programa se llaman \_\_\_\_\_ y \_\_\_\_\_.
- b) Una función se invoca con un(a) \_\_\_\_\_.
- c) A una variable que se conoce sólo dentro de la función en la que está declarada, se le llama \_\_\_\_\_.
- d) La instrucción \_\_\_\_\_ en una función a la que se llamó puede usarse para pasar el valor de una expresión, de vuelta a la función que hizo la llamada.
- e) La palabra clave \_\_\_\_\_ se utiliza en un encabezado de función para indicar que una función no devuelve ningún valor, o para indicar que esa función no contiene parámetros.
- f) El \_\_\_\_\_ de un identificador es la porción del programa en la que puede usarse.
- g) Las tres formas de regresar el control de una llamada a una función a la función que la llamó son \_\_\_\_\_, \_\_\_\_\_ y \_\_\_\_\_.
- h) Un \_\_\_\_\_ permite al compilador comprobar el número, tipos y orden de los argumentos que se pasan a una función.
- i) La función \_\_\_\_\_ se utiliza para producir números aleatorios.
- j) La función \_\_\_\_\_ se utiliza para establecer la semilla de números aleatorios, para randomizar un programa.
- k) Los especificadores de clase de almacenamiento son `mutable`, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_ y \_\_\_\_\_.
- l) Se asume que las variables declaradas en un bloque o en la lista de parámetros de una función son de la clase de almacenamiento \_\_\_\_\_, a menos que se especifique lo contrario.

- m) El especificador de clase de almacenamiento \_\_\_\_\_ es una recomendación que se hace al compilador para que almacene una variable en uno de los registros de la computadora.
- n) Una variable que se declara fuera de cualquier bloque o función es una variable \_\_\_\_\_.
- o) Para que una variable local en una función retenga su valor entre las llamadas a la función, debe declararse con el especificador de clase de almacenamiento \_\_\_\_\_.
- p) Los seis posibles alcances de un identificador son \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_ y \_\_\_\_\_.
- q) Una función que se llama a sí misma, ya sea en forma directa o indirecta (a través de otra función), es una función \_\_\_\_\_.
- r) Por lo general, una función recursiva tiene dos componentes: uno que proporciona el medio para que termine la recursividad, al evaluar un caso(s) \_\_\_\_\_, y uno que expresa el problema como una llamada recursiva para un problema más simple que el de la llamada original.
- s) Es posible tener varias funciones con el mismo nombre, que operen con distintos tipos o números de argumentos. A esto se le conoce como \_\_\_\_\_ de funciones.
- t) El \_\_\_\_\_ permite acceder a una variable global con el mismo nombre que una variable en el alcance actual.
- u) El calificador \_\_\_\_\_ se usa para declarar variables de sólo lectura.
- v) Una \_\_\_\_\_ de función permite definir una sola función para realizar una tarea en muchos tipos de datos distintos.

**6.2** Para el programa de la figura 6.40, indique el alcance (ya sea de función, de archivo, de bloque o de prototipo de función) de cada uno de los siguientes elementos:

- a) la variable `x` en `main`.
- b) la variable `y` en `cubo`.
- c) la función `cubo`.
- d) la función `main`.
- e) el prototipo de función para `cubo`.
- f) El identificador `y` en el prototipo de función para `cubo`.

```

1 // Ejercicio 6.2: ej06_02.cpp
2 #include <iostream>
3 using std::cout;
4 using std::endl;
5
6 int cubo(int y); // prototipo de función
7
8 int main()
9 {
10 int x;
11
12 for (x = 1; x <= 10; x++) // itera 10 veces
13 cout << cubo(x) << endl; // calcula el cubo de x e imprime los resultados
14
15 return 0; // indica que terminó correctamente
16 } // fin de main
17
18 // definición de la función cubo
19 int cubo(int y)
20 {
21 return y * y * y;
22 } // fin de la función cubo

```

**Figura 6.40** | Programa para el ejercicio 6.2.

**6.3** Escriba un programa que pruebe si los ejemplos de las llamadas a las funciones matemáticas de la biblioteca que se muestran en la figura 6.2 realmente producen los resultados indicados.

**6.4** Proporcione el encabezado para cada una de las siguientes funciones:

- a) La función `hipotenusa`, que toma dos argumentos de punto flotante con doble precisión, llamados `lado1` y `lado2`, y que devuelve un resultado de punto flotante, con doble precisión.
- b) La función `menor`, que toma tres enteros `x`, `y` y `z`, y devuelve un entero.

- c) La función `instrucciones`, que no recibe argumentos y no devuelve ningún valor. [Nota: dichas funciones se utilizan comúnmente para mostrar instrucciones a un usuario.]
- d) La función `intADouble`, que recibe un argumento entero llamado `numero` y devuelve un resultado de punto flotante, con precisión doble.
- 6.5** Proporcione el prototipo de función (sin nombres de parámetros) para cada una de las siguientes situaciones:
- La función descrita en el ejercicio 6.4(a).
  - La función descrita en el ejercicio 6.4(b).
  - La función descrita en el ejercicio 6.4(c).
  - La función descrita en el ejercicio 6.4(d).
- 6.6** Escriba una instrucción para cada uno de los siguientes casos:
- Una variable `int` llamada `cuenta`, que deba mantenerse en un registro. Inicialice `cuenta` en 0.
  - La variable de punto flotante con precisión doble llamada `ultimoVal`, que debe retener su valor entre las llamadas a la función en la que está definida.
- 6.7** Encuentre el error en cada uno de los siguientes segmentos de programas, y explique cómo se puede corregir el error (vea también el ejercicio 6.53):
- ```
int g()
{
    cout << "Dentro de la funcion g" << endl;
    int h()
    {
        cout << "Dentro de la funcion h" << endl;
    }
}
```
 - ```
int suma(int x, int y)
{
 int resultado;
 resultado = x + y;
}
```
  - ```
int suma( int n )
{
    if ( n == 0 )
        return 0;
    else
        n + suma( n - 1 );
}
```
 - ```
void f(double a);
{
 float a;
 cout << a << endl;
}
```
  - ```
void producto()
{
    int a;
    int b;
    int c;
    int resultado;
    cout << "Escribe tres enteros: ";
    cin >> a >> b >> c;
    resultado = a * b * c;
    cout << "El resultado es " <<, resultado );
    return resultado;
}
```
- 6.8** ¿Para qué un prototipo de función podría contener la declaración del tipo de un parámetro, como `double &`?
- 6.9** (Verdadero/Falso) Todos los argumentos a las llamadas a funciones en C++ se pasan por valor.

6.10 Escriba un programa completo que pida al usuario el radio de una esfera, calcule e imprima el volumen de esa esfera. Use una función `inline` llamada `volumenEsfera` que devuelva el resultado de la siguiente expresión: $(4.0 / 3.0) * 3.14159 * \text{pow}(\text{radio}, 3)$.

Respuestas a los ejercicios de autoevaluación

6.1 a) funciones, clases. b) llamada a una función. c) variable local. d) `return`. e) `void`. f) alcance. g) `return; return expresión;` o encontrar la llave derecha de cierre de una función. h) prototipo de función. i) `rand`. j) `srand`. k) `auto`, `register`, `extern`, `static`. l) `auto`. m) `register`. n) `global`. o) `static`. p) alcance de función, alcance de archivo, alcance de bloque, alcance de prototipo de función, alcance de clase, alcance de espacio de nombres. q) recursiva. r) base. s) sobrecarga t) operador de resolución de ámbito unario (`:::`). u) `const`. v) plantilla.

6.2 a) alcance de bloque. b) alcance de bloque. c) alcance de archivo. d) alcance de archivo. e) alcance de archivo. f) alcance de prototipo de función.

6.3 Vea el siguiente programa:

```

1 // Ejercicio 6.3: ej06_03.cpp
2 // Prueba de las funciones matemáticas de la biblioteca.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 #include <cmath>
12 using namespace std;
13
14 int main()
15 {
16     cout << fixed << setprecision( 1 );
17
18     cout << "sqrt(" << 900.0 << ") = " << sqrt( 900.0 )
19         << "\nsqrt(" << 9.0 << ") = " << sqrt( 9.0 );
20     cout << "\nexp(" << 1.0 << ") = " << setprecision( 6 )
21         << exp( 1.0 ) << "\nexp(" << setprecision( 1 ) << 2.0
22         << ") = " << setprecision( 6 ) << exp( 2.0 );
23     cout << "\nlog(" << 2.718282 << ") = " << setprecision( 1 )
24         << log( 2.718282 )
25         << "\nlog(" << setprecision( 6 ) << 7.389056 << ") = "
26         << setprecision( 1 ) << log( 7.389056 );
27     cout << "\nlog10(" << 1.0 << ") = " << log10( 1.0 )
28         << "\nlog10(" << 10.0 << ") = " << log10( 10.0 )
29         << "\nlog10(" << 100.0 << ") = " << log10( 100.0 ) ;
30     cout << "\nfabs(" << 13.5 << ") = " << fabs( 13.5 )
31         << "\nfabs(" << 0.0 << ") = " << fabs( 0.0 )
32         << "\nfabs(" << -13.5 << ") = " << fabs( -13.5 );
33     cout << "\nceil(" << 9.2 << ") = " << ceil( 9.2 )
34         << "\nceil(" << -9.8 << ") = " << ceil( -9.8 );
35     cout << "\nfloor(" << 9.2 << ") = " << floor( 9.2 )
36         << "\nfloor(" << -9.8 << ") = " << floor( -9.8 );
37     cout << "\npow(" << 2.0 << ", " << 7.0 << ") = "
38         << pow( 2.0, 7.0 ) << "\npow(" << 9.0 << ", "
39         << 0.5 << ") = " << pow( 9.0, 0.5 );
40     cout << setprecision(3) << "\nmod("
41         << 13.675 << ", " << 2.333 << ") = "
42         << fmod( 13.675, 2.333 ) << setprecision( 1 );
43     cout << "\nsin(" << 0.0 << ") = " << sin( 0.0 );
44     cout << "\ncos(" << 0.0 << ") = " << cos( 0.0 );
45     cout << "\ntan(" << 0.0 << ") = " << tan( 0.0 ) << endl;
46
47 } // fin de main

```

```

sqrt(900.0) = 30.0
sqrt(9.0) = 3.0
exp(1.0) = 2.718282
exp(2.0) = 7.389056
log(2.718282) = 1.0
log(7.389056) = 2.0
log10(1.0) = 0.0
log10(10.0) = 1.0
log10(100.0) = 2.0
fabs(13.5) = 13.5
fabs(0.0) = 0.0
fabs(-13.5) = 13.5
ceil(9.2) = 10.0
ceil(-9.8) = -9.0
floor(9.2) = 9.0
floor(-9.8) = -10.0
pow(2.0, 7.0) = 128.0
pow(9.0, 0.5) = 3.0
fmod(13.675, 2.333) = 2.010
sin(0.0) = 0.0
cos(0.0) = 1.0
tan(0.0) = 0.0

```

- 6.4**
- a) double hipotenusa(double lado1, double lado2)
 - b) int menor(int x, int y, int z)
 - c) void instrucciones()
 - d) double intADouble(int numero)
- 6.5**
- a) double hipotenusa(double, double);
 - b) int menor(int, int, int);
 - c) void instrucciones();
 - d) double intADouble(int);
- 6.6**
- a) register int cuenta = 0;
 - b) static double ultimoVal;
- 6.7**
- a) *Error:* la función h está definida en la función g.
Corrección: mueva la definición de h fuera de la definición de g.
 - b) *Error:* se supone que la función debe devolver un entero, pero no es así.
Corrección: elimine la variable resultado, y coloque la siguiente instrucción en la función:
`return x + y;`
 - c) *Error:* el resultado de n + suma(n - 1) no se devuelve; suma devuelve un resultado incorrecto.
Corrección: vuelva a escribir la instrucción en la cláusula else de la siguiente manera:
`return n + suma(n - 1);`
 - d) *Errores:* el punto y coma que va después del paréntesis derecho de la lista de parámetros es incorrecto, y el parámetro a no debe volver a definirse en la definición de la función.
Correcciones: elimine el punto y coma que va después del paréntesis derecho de la lista de parámetros, y elimine la declaración float a;.
 - e) *Error:* la función devuelve un valor cuando no debe hacerlo.
Corrección: elimine la instrucción return.
- 6.8** Esto crea un parámetro de referencia de tipo “referencia a double”, el cual permite que la función modifique la variable original en la función que hace la llamada.
- 6.9** Falso. C++ permite el paso por referencia mediante el uso de parámetros por referencia (y apunadores, como veremos en el capítulo 8).
- 6.10** Vea el siguiente programa:

```

1 // Ejercicio 6.10: ej06_10.cpp
2 // Función en línea que calcula el volumen de una esfera.
3 #include <iostream>

```

```

4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  #include <cmath>
9  using std::pow;
10
11 const double PI = 3.14159; // define la constante global PI
12
13 // calcula el volumen de una esfera
14 inline double volumenEsfera( const double radio )
15 {
16     return 4.0 / 3.0 * pow( radio, 3 );
17 } // fin de la función en línea volumenEsfera
18
19 int main()
20 {
21     double valorRadio;
22
23     // pide el radio al usuario
24     cout << "Escriba la longitud del radio de su esfera: ";
25     cin >> valorRadio; // recibe el radio
26
27     // usa valorRadio para calcular el volumen de la esfera y mostrar el resultado
28     cout << "El volumen de la esfera con radio " << valorRadio
29         << " es " << volumenEsfera( valorRadio ) << endl;
30
31     return 0; // indica que terminó correctamente
32 } // fin de main

```

Ejercicios

6.11 Muestre el valor de *x* después de ejecutar cada una de las siguientes instrucciones:

- a) *x* =*fabs*(7.5);
- b) *x* = *floor*(7.5);
- c) *x* = *fabs*(0.0);
- d) *x* = *ceil*(0.0);
- e) *x* = *fabs*(-6.4);
- f) *x* = *ceil*(-6.4);
- g) *x* =*ceil*(-*fabs*(-8 + *floor*(-5.5)));

6.12 Un estacionamiento cobra una cuota mínima de \$2.00 por estacionarse hasta tres horas. El estacionamiento cobra \$0.50 adicionales por cada hora o *fracción* que se pase de tres horas. El cargo máximo para cualquier periodo dado de 24 horas es de \$10.00. Suponga que ningún auto se estaciona durante más de 24 horas a la vez. Escriba un programa que calcule y muestre los cargos por estacionamiento para cada uno de tres clientes que estacionaron su auto ayer en este estacionamiento. Debe introducir las horas de estacionamiento para cada cliente. El programa debe imprimir los resultados en un formato tabular ordenado, debe calcular e imprimir el total de los recibos de ayer. El programa debe utilizar la función *calcularCargos* para determinar el cargo para cada cliente. Sus resultados deben aparecer en el siguiente formato:

Auto	Horas	Cargo
1	1.5	2.00
2	4.0	2.50
3	24.0	10.00
TOTAL	29.50	14.50

6.13 Una aplicación de la función *floor* es redondear un valor al siguiente entero. La instrucción

y = *floor*(*x* + .5);

redondea el número *x* al entero más cercano y asigna el resultado a *y*. Escriba un programa que lea varios números y que utilice la instrucción anterior para redondear cada uno de los números a su entero más cercano. Para cada número procesado, muestre tanto el número original como el redondeado.

6.14 La función `floor` puede utilizarse para redondear un número hasta un lugar decimal específico. La instrucción

```
y = floor( x * 10 + .5 ) / 10;
```

redondea `x` en la posición de las décimas (es decir, la primera posición a la derecha del punto decimal). La instrucción

```
y = floor( x * 100 + 0.5 ) / 100;
```

redondea `x` en la posición de las centésimas (es decir, la segunda posición a la derecha del punto decimal). Escriba un programa que defina cuatro funciones para redondear un número `x` en varias formas:

- `redondearAEntero(numero)`
- `redondearADecimas(numero)`
- `redondearACentesimas(numero)`
- `redondearAMilesimas(numero)`

Para cada valor leído, su programa debe imprimir el valor original, el número redondeado al entero más cercano, el número redondeado a la décima más cercana, el número redondeado a la centésima más cercana y el número redondeado a la milésima más cercana.

6.15 Responda a cada una de las siguientes preguntas:

- ¿Qué significa elegir números “al azar”?
- ¿Por qué es la función `rand` útil para simular juegos al azar?
- ¿Por qué se debe randomizar un programa mediante `srand`? ¿Bajo qué circunstancias es aconsejable no randomizar?
- ¿Por qué a menudo es necesario escalar o desplazar los valores producidos por `rand`?
- ¿Por qué es la simulación computarizada de las situaciones reales una técnica útil?

6.16 Escriba instrucciones que asignen enteros aleatorios a la variable `n` en los siguientes rangos:

- $1 \leq n \leq 2$
- $1 \leq n \leq 100$
- $0 \leq n \leq 9$
- $1000 \leq n \leq 1112$
- $-1 \leq n \leq 1$
- $-3 \leq n \leq 11$

6.17 Para cada uno de los siguientes conjuntos de enteros, escriba una sola instrucción que imprima un número al azar del conjunto:

- 2, 4, 6, 8, 10.
- 3, 5, 7, 9, 11.
- 6, 10, 14, 18, 22.

6.18 Escriba una función llamada `enteroPotencia(base, exponente)` que devuelva el valor de

`base exponente`

Por ejemplo, `enteroPotencia(3, 4) = 3 * 3 * 3 * 3`. Suponga que `exponente` es un entero positivo distinto de cero y que `base` es un entero. La función `enteroPotencia` debe utilizar un ciclo `for` o `while` para controlar el cálculo. No utilice ninguna función de la biblioteca de matemáticas.

6.19 (*Hipotenusa*) Defina una función llamada `hipotenusa` que calcule la longitud de la hipotenusa de un triángulo recto, cuando se proporcionen las longitudes de los otros dos lados. Use esta función en un programa para determinar la longitud de la hipotenusa para cada uno de los triángulos que se muestran a continuación. La función debe recibir dos argumentos `double` y devolver la hipotenusa como `double`.

Triángulo	Lado 1	Lado 2
1	3.0	4.0
2	5.0	12.0
3	8.0	15.0

6.20 Escriba una función llamada `multiple` que determine, para un par de enteros, si el segundo entero es múltiplo del primero. La función debe tomar dos argumentos enteros y devolver `true` si el segundo es múltiplo del primero, y `false` en caso contrario. Use esta función en un programa que reciba como entrada una serie de pares de enteros.

6.21 Escriba un programa que reciba una serie de enteros y los pase, uno a la vez, a una función llamada `esPar` que utilice el operador módulo para determinar si un entero dado es par. La función debe tomar un argumento entero y devolver `true` si el entero es par, y `false` en caso contrario.

6.22 Escriba una función que muestre en el margen izquierdo de la pantalla un cuadrado relleno de asteriscos, cuyo lado se especifique en el parámetro entero `lado`. Por ejemplo, si `lado` es 4, el método debe mostrar lo siguiente:

```
****  
***  
***  
***
```

6.23 Modifique la función creada en el ejercicio 6.22 para formar el cuadrado de cualquier carácter que esté contenido en el parámetro tipo carácter `caracterRelleno`. Por ejemplo, si `lado` es 5 y `caracterRelleno` es "#", el método debe imprimir lo siguiente:

```
####  
####  
####  
####  
####
```

6.24 Use técnicas similares a las desarrolladas en los ejercicios 6.22 y 6.23 para producir un programa que grafique un amplio rango de figuras.

6.25 Escriba segmentos de programas que realicen cada una de las siguientes tareas:

- Calcular la parte entera del cociente, cuando el entero `a` se divide por el entero `b`.
- Calcular el residuo entero cuando el entero `a` se divide entre el entero `b`.
- Utilizar las piezas de los programas desarrollados en las partes (a) y (b) para escribir una función que reciba un entero entre 1 y 32767, y que lo imprima como una serie de dígitos, separando cada par de dígitos por dos espacios. Por ejemplo, el entero 4562 debe imprimirse de la siguiente manera:

```
4 5 6 2
```

6.26 Escriba una función que recibe la hora en forma de tres argumentos enteros (horas, minutos y segundos) y devuelva el número de segundos transcurridos desde la última vez que el reloj "marcó las 12". Use esta función para calcular el monto de tiempo en segundos entre dos horas, ambas de las cuales están dentro de un ciclo de 12 horas del reloj.

6.27 (*Temperaturas en Centígrados y Fahrenheit*) Implemente las siguientes funciones enteras:

- El método `centigrados` que devuelve la equivalencia en grados Centígrados de una temperatura en grados Fahrenheit.
- La función `fahrenheit` que devuelve la equivalencia en grados Fahrenheit de una temperatura en grados Centígrados.
- Utilice estas funciones para escribir un programa que imprima gráficos que muestren los equivalentes en grados Fahrenheit de todas las temperaturas en grados Centígrados, desde 0 hasta 100, y los equivalentes en grados Centígrados de todas las temperaturas en grados Fahrenheit, desde 32 hasta 212. Imprima los resultados en un formato tabular ordenado que minimice el número de líneas de salida, al tiempo que permanezca legible.

6.28 Escriba un programa que reciba tres números de punto flotante de precisión doble, y que los pase a una función que devuelva el número más pequeño.

6.29 (*Números perfectos*) Se dice que un número entero es un *número perfecto* si la suma de sus divisores, incluyendo 1 (pero no el número en sí), es igual al número. Por ejemplo, 6 es un número perfecto ya que $6 = 1 + 2 + 3$. Escriba una función llamada `perfecto` que determine si el parámetro `numero` es un número perfecto. Use esta función en un programa que determine e imprima todos los números perfectos entre 1 y 1000. Imprima los divisores de cada número perfecto para confirmar que el número sea realmente perfecto. Ponga a prueba el poder de su computadora, evaluando números mucho más grandes que 1000.

6.30 (*Números primos*) Se dice que un entero es *primo* si puede dividirse solamente por 1 y por sí mismo. Por ejemplo, 2, 3, 5 y 7 son primos, pero 4, 6, 8 y 9 no.

- Escriba una función que determine si un número es primo.
- Use esta función en un programa que determine e imprima todos los números primos entre 2 y 10,000. ¿Cuántos de estos números hay que probar realmente para asegurarse de encontrar todos los números primos?

- c) Al principio podría pensarse que $n/2$ es el límite superior para evaluar si un número es primo, pero lo máximo que se necesita es ir hasta la raíz cuadrada de n . ¿Por qué? Vuelva a escribir el programa y ejecútelo de ambas formas. Estime la mejora en el rendimiento.

6.31 (*Dígitos inversos*) Escriba una función que reciba un valor entero y devuelva el número con sus dígitos invertidos. Por ejemplo, para el número 7631, la función debe regresar 1367.

6.32 El *máximo común divisor (MCD)* de dos enteros es el entero más grande que puede dividir uniformemente a cada uno de los dos números. Escriba una función llamada `mcd` que devuelva el máximo común divisor de dos enteros.

6.33 Escriba una función llamada `puntosCalidad` que reciba como entrada el promedio de un estudiante y devuelva 4 si el promedio se encuentra entre 90 y 100, 3 si el promedio se encuentra entre 80 y 89, 2 si el promedio se encuentra entre 70 y 79, 1 si el promedio se encuentra entre 60 y 69, 0 si el promedio es menor de 60.

6.34 Escriba un programa que simule el lanzamiento de monedas. Cada vez que se lance la moneda, el programa debe imprimir Cara o Cruz. Deje que el programa lance la moneda 100 veces y cuente el número de veces que aparezca cada uno de los lados de la moneda. Imprima los resultados. El programa debe llamar a un método separado, llamado `tirar`, que no reciba argumentos y devuelva 0 en caso de cara y 1 en caso de cruz. [Nota: si el programa simula en forma realista el lanzamiento de monedas, cada lado de la moneda debe aparecer aproximadamente la mitad del tiempo].

6.35 (*Las computadoras en la educación*) Las computadoras están tomando un papel cada vez más importante en la educación. Escriba un programa que ayude a un estudiante de escuela primaria, para que aprenda a multiplicar. Use la función `rand` para producir dos enteros positivos de un dígito. El programa debe entonces mostrar una pregunta al usuario, como

¿Cuánto es 6 por 7?

El estudiante entonces debe escribir la respuesta. Luego, el programa debe verificar la respuesta del estudiante. Si es correcta, debe imprimir "Muy bien!" y hacer otra pregunta de multiplicación. Si la respuesta es incorrecta, debe imprimir "No. Por favor intenta de nuevo." y deje que el estudiante intente la misma pregunta varias veces, hasta que esté correcta.

6.36 (*Instrucción asistida por computadora*) El uso de las computadoras en la educación se conoce como *instrucción asistida por computadora (CAI)*. Un problema que se desarrolla en los entornos CAI es la fatiga de los estudiantes. Este problema puede eliminarse si se varía el diálogo de la computadora para mantener la atención del estudiante. Modifique el programa del ejercicio 6.35 de manera que los diversos comentarios se impriman para cada respuesta correcta e incorrecta, de la siguiente manera:

Contestaciones a una respuesta correcta:

Muy bien!
Excelente!
Buen trabajo!
Sigue así!

Contestaciones a una respuesta incorrecta:

No. Por favor intenta de nuevo.
Incorrecto. Intenta una vez mas.
No te rindas!
No. Sigue intentando.

Use el generador de números aleatorios para elegir un número entre 1 y 4 que se utilice para seleccionar una contestación apropiada a cada respuesta. Use una instrucción `switch` para emitir las contestaciones.

6.37 Los sistemas de instrucción asistida por computadora más sofisticados supervisan el rendimiento del estudiante durante cierto tiempo. La decisión de empezar un nuevo tema se basa a menudo en el éxito del estudiante con los temas anteriores. Modifique el programa del ejercicio 6.36 para contar el número de respuestas correctas e incorrectas por parte del estudiante. Una vez que el estudiante escribe 10 respuestas, su programa debe calcular el porcentaje de respuestas correctas. Si éste es menor del 75%, el programa deberá imprimir Por favor pida ayuda adicional a su instructor y terminar.

6.38 (*Juego “Adivina el número”*) Escriba una aplicación que juegue a “adivina el número” de la siguiente manera: su programa elige el número a adivinar, seleccionando un entero aleatorio en el rango de 1 a 1000. Después, el programa muestra lo siguiente:

```
Tengo un numero entre 1 y 1000.  
Puedes adivinar mi numero?  
Por favor escribe tu primera respuesta.
```

El jugador escribe su primer intento. El programa responde con uno de los siguientes mensajes:

1. Excelente! Adivinaste el numero!
Te gustaria jugar de nuevo (s/n)?
2. Demasiado bajo. Intenta de nuevo.
3. Demasiado alto. Intenta de nuevo.

Si la respuesta del jugador es incorrecta, su programa deberá iterar hasta que el jugador adivine correctamente. Su programa deberá seguir indicando al jugador los mensajes "Demasiado alto. Intenta de nuevo." o "Demasiado bajo. Intenta de nuevo.", para ayudar a que el jugador "se acerque" a la respuesta correcta.

6.39 Modifique el programa del ejercicio 6.38 para contar el número de intentos que haga el jugador. Si el número es 10 o menos, imprima el mensaje "0 ya sabia usted el secreto, o tuvo suerte!" Si el jugador adivina el número en 10 intentos, imprima el mensaje "Aja! Sabía usted el secreto!" Si el jugador hace más de 10 intentos, imprima el mensaje "Debería haberlo hecho mejor!" ¿Por qué no se deben requerir más de 10 intentos? Bueno, en cada "buen intento", el jugador debe poder eliminar la mitad de los números. Ahora muestre por qué cualquier número de 1 a 1000 puede adivinarse en 10 intentos o menos.

6.40 Escriba una función recursiva llamado `potencia(base, exponente)` que, cuando sea llamada, devuelva `base ^ exponente`

Por ejemplo, $\text{potencia}(3, 4) = 3 * 3 * 3 * 3$. Suponga que `exponente` es un entero mayor o igual que 1. *Sugerencia:* el paso recursivo debe utilizar la relación

$$\text{base}^{\text{exponente}} = \text{base} \cdot \text{base}^{\text{exponente} - 1}$$

y la condición de terminación ocurre cuando `exponente` es igual a 1, ya que

$$\text{base}^1 = \text{base}$$

6.41 (*Serie de Fibonacci*) La serie de Fibonacci,

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \dots$$

empieza con 0 y 1, y tiene la propiedad de que cada número subsiguiente de Fibonacci es la suma de los dos números Fibonacci anteriores. (a) Escriba una función *no recursiva* llamada `fibonacci(n)`, que calcule el n -ésimo número de Fibonacci. (b) Determine el mayor número `int` de Fibonacci que puede imprimirse en su sistema. Modifique el programa de la parte (a) para usar `double` en vez de `int`, para calcular y devolver números de Fibonacci, y utilice este programa modificado para repetir la parte (b).

6.42 (*Torres de Hanoi*) En este capítulo estudiamos funciones que pueden implementarse con facilidad, tanto en forma recursiva como iterativa. En este ejercicio presentamos un problema cuya solución recursiva demuestra la elegancia de la recursividad, y cuya solución iterativa tal vez no sea tan aparente.

Las **Torres de Hanoi** son uno de los problemas clásicos más famosos con los que todo científico computacional en ciernes tiene que lidiar. Cuenta la leyenda que en un templo del Lejano Oriente, los sacerdotes intentan mover una pila de discos dorados, de una aguja de diamante a otra (figura 6.41). La pila inicial tiene 64 discos insertados en una aguja y se ordenan de abajo hacia arriba, de mayor a menor tamaño. Los sacerdotes intentan mover la pila de una aguja a otra, con las restricciones de que sólo se puede mover un disco a la vez, y en ningún momento se puede colocar un disco más grande encima de uno más pequeño. Se cuenta con tres agujas, una de las cuales se utiliza para almacenar discos temporalmente. Se supone que el mundo acabará cuando los sacerdotes completen su tarea, por lo que hay pocos incentivos para que nosotros podamos facilitar sus esfuerzos.

Vamos a suponer que los sacerdotes intentan mover los discos de la aguja 1 a la aguja 3. Deseamos desarrollar un algoritmo que imprima la secuencia precisa de transferencias de los discos de una aguja a otra.

Si tratamos de encontrar una solución iterativa, es probable que terminemos "atados" manejando los discos sin esperanza. En vez de ello, si atacamos este problema teniendo en mente la recursividad, los pasos serán más simples. La acción de mover n discos puede verse en términos de mover sólo $n - 1$ discos (de ahí la recursividad) de la siguiente forma:

- Mover $n - 1$ discos de la aguja 1 a la aguja 2, usando la aguja 3 como un área de almacenamiento temporal.
- Mover el último disco (el más grande) de la aguja 1 a la aguja 3.
- Mover $n - 1$ discos de la aguja 2 a la aguja 3, usando la aguja 1 como área de almacenamiento temporal.

El proceso termina cuando la última tarea implica mover $n = 1$ disco (es decir, el caso base). Esta tarea se logra con sólo mover el disco, sin necesidad de un área de almacenamiento temporal.

Escriba un programa para resolver el problema de las Torres de Hanoi. Use una función recursiva con cuatro parámetros:

- El número de discos a mover.
- La aguja en la que están insertados estos discos en un principio.
- La aguja a la que se va a mover esta pila de discos.
- La aguja que se va a utilizar como área de almacenamiento temporal.

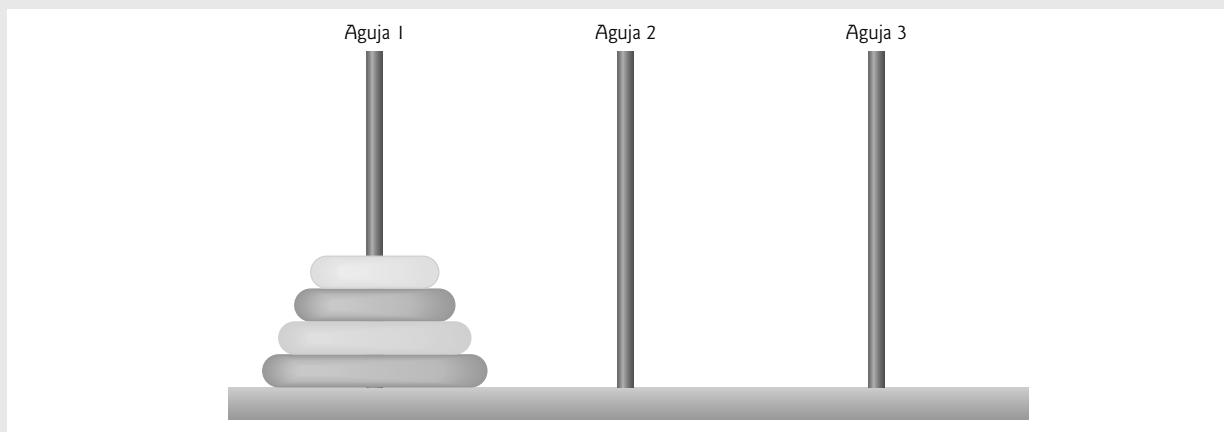


Figura 6.41 | Las Torres de Hanoi para el caso con cuatro discos.

Su programa debe imprimir las instrucciones precisas que requerirá para mover los discos de la aguja inicial a la aguja de destino. Por ejemplo, para mover una pila de tres discos de la aguja 1 a la aguja 3, su programa debe imprimir la siguiente serie de movimientos:

```

1 → 3 (Esto significa mover un disco de la aguja 1 a la aguja 3.)
1 → 2
3 → 2
1 → 3
2 → 1
2 → 3
1 → 3

```

6.43 Cualquier programa que se pueda implementar en forma recursiva se puede implementar en forma iterativa, aunque algunas veces con mayor dificultad y menor claridad. Pruebe a escribir una versión iterativa de las Torres de Hanoi. Si tiene éxito, compare su versión iterativa con la versión recursiva desarrollada en el ejercicio 6.42. Investigue las cuestiones relacionadas con el rendimiento, la claridad y su habilidad de demostrar que los programas estén correctos.

6.44 (*Visualización de la recursividad*) Es interesante observar la recursividad “en acción”. Modifique la función factorial de la figura 6.29 para imprimir su variable local y su parámetro de llamada recursiva. Para cada llamada recursiva, muestre los resultados en una línea separada y agregue un nivel de sangría. Haga su máximo esfuerzo por hacer que los resultados sean claros, interesantes y significativos. Su meta aquí es diseñar e implementar un formato de salida que facilite la comprensión de la recursividad. Tal vez desee agregar ciertas capacidades de visualización a otros ejemplos y ejercicios recursivos a lo largo de este libro.

6.45 (*Mayor común divisor recursivo*) El mayor común divisor de los enteros x y y es el entero más grande que se puede dividir entre x y y de manera uniforme. Escriba una función recursiva llamado `mcd`, que devuelva el mayor común divisor de x y y , definida mediante la recursividad, de la siguiente manera: si y es igual a 0, entonces $\text{mcd}(x, y)$ es x ; en caso contrario, $\text{mcd}(x, y)$ es $\text{mcd}(y, x \% y)$, donde $\%$ es el operador módulo. [Nota: para este algoritmo, x debe ser mayor que y .]

6.46 ¿Se puede llamar a `main` de manera recursiva en su sistema? Escriba un programa que contenga una función `main`. Incluya una variable local `static` llamada `cuenta` e inicialícela con 1. Realice un postincremento e imprima el valor de `cuenta` cada vez que se llame a `main`. Compile su programa. ¿Qué ocurre?

6.47 En los ejercicios 6.35 a 6.37 se desarrolló un programa de instrucción asistida por computadora para enseñar multiplicación a un estudiante de primaria. Este ejercicio sugiere algunas mejoras a ese programa.

- Modifique el programa para permitir al usuario introducir una capacidad de nivel de grado. Un nivel de grado de 1 significa que se deben usar sólo números de un dígito en los problemas, un nivel de grado 2 significa que se deben usar números no mayores de dos dígitos, etcétera.
- Modifique el programa para permitir al usuario elegir el tipo de problemas aritméticos que desea estudiar. Una opción de 1 indica sólo problemas de sumas, 2 indica sólo problemas de restas, 3 indica sólo problemas de multiplicación, 4 indica sólo problemas de división y 5 indica una mezcla aleatoria de problemas de todos estos tipos.

6.48 Escriba una función llamada `distancia` que calcule la distancia entre dos puntos (x_1, y_1) y (x_2, y_2) . Todos los números y valores de retorno deben ser de tipo `double`.

6.49 ¿Qué error tiene el siguiente programa?

```

1 // Ejercicio 6.49: ej06_49.cpp
2 // ¿Qué error tiene este programa?
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6
7 int main()
8 {
9     int c;
10
11    if ( ( c = cin.get() ) != EOF )
12    {
13        main();
14        cout << c;
15    } // fin de if
16
17    return 0; // indica que terminó correctamente
18 } // fin de main

```

6.50 ¿Qué hace el siguiente programa?

```

1 // Ejercicio 6.50: ej06_50.cpp
2 // ¿Qué hace este programa?
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 int misterio( int, int ); // prototipo de función
9
10 int main()
11 {
12     int x, y;
13
14     cout << "Escriba dos enteros: ";
15     cin >> x >> y;
16     cout << "El resultado es " << misterio( x, y ) << endl;
17
18     return 0; // indica que terminó correctamente
19 } // fin de main
20
21 // el parámetro b debe ser un entero positivo para prevenir la recursividad infinita
22 int misterio( int a, int b )
23 {
24     if ( b == 1 ) // caso base
25         return a;
26     else // paso recursivo
27         return a + misterio( a, b - 1 );
28 } // fin de la función misterio

```

6.51 Una vez que determine qué es lo que hace el programa del ejercicio 6.50, modifíquelo para que funcione de manera apropiada, después de eliminar la restricción de que el segundo argumento debe ser positivo.

6.52 Escriba un programa que evalúe todas las funciones matemáticas de la biblioteca en la figura 6.2 que pueda. Ejercite cada una de estas funciones, haciendo que su programa imprima tablas de valores de retornos para una diversidad de valores de los argumentos.

6.53 Encuentre el error en cada uno de los siguientes segmentos de programa, y explique cómo corregirlo:

- float cubo(float); // prototipo de función

```

cubo( float numero ) // definición de función
{
    return numero * numero * numero;
}

```

```

b) register auto int x = 7;
c) int numeroAleatorio = srand();
d) float y = 123.45678;
int x;

x = y;
cout << static_cast< float >( x ) << endl;
e) double cuadrado( double numero )
{
    double numero;
    return numero * numero;
}
f) int suma( int n )
{
    if ( n == 0 )
        return 0;
    else
        return n + suma( n );
}

```

6.54 Modifique el programa Craps de la figura 6.11 para permitir apuestas. Empaque como función la parte del programa que ejecuta un juego de craps. Inicialice la variable `saldoBanco` con 1000 dólares. Pida al jugador que introduzca una apuesta. Use un ciclo `while` para comprobar que esa apuesta sea menor o igual al `saldoBanco` y, si no lo es, haga que el usuario vuelva a introducir la apuesta hasta que se introduzca un valor válido. Después de esto, comience un juego de craps. Si el jugador gana, agregue la apuesta al `saldoBanco` e imprima el nuevo `saldoBanco`. Si el jugador pierde, reste la apuesta al `saldoBanco`, imprima el nuevo `saldoBanco`, compruebe si `saldoBanco` se ha vuelto cero y, de ser así, imprima el mensaje "Lo siento. Se quedo sin fondos!" A medida que el juego progrese, imprima varios mensajes para crear algo de "charla", como "Oh, se esta yendo a la quiebra, verdad?", o "Oh, vamos, arriesguese!", o "La hizo en grande. Ahora es tiempo de cambiar sus fichas por efectivo!".

6.55 Escriba un programa en C++ que pida al usuario el radio de un círculo y después llame a la función `inline areaCírculo` para calcular el área de ese círculo.

6.56 Escriba un programa completo en C++ con las dos funciones alternativas que se especifican a continuación, de las cuales cada una simplemente triplica la variable `cuenta` definida en `main`. Después compare y contraste ambos métodos. Estas dos funciones son:

- la función `triplicarPorValor`, que pasa una copia de `cuenta` por valor, triplica la copia y devuelve el nuevo valor, y
- la función `triplicarPorReferencia`, que pasa `cuenta` por referencia a través de un parámetro por referencia y triplica el valor original de `cuenta` a través de su alias (es decir, el parámetro por referencia).

6.57 ¿Cuál es el propósito del operador de resolución de ámbito unario?

6.58 Escriba un programa que use una plantilla de función llamada `min` para determinar el menor de dos argumentos. Pruebe el programa usando argumentos tipo entero, carácter y número de punto flotante.

6.59 Escriba un programa que utilice una plantilla de función llamada `max` para determinar el mayor de dos argumentos. Pruebe el programa usando argumentos tipo entero, carácter y número de punto flotante.

6.60 Determine si los siguientes segmentos de programa contienen errores. Para cada error, explique cómo puede corregirse. [Nota: para un segmento de programa específico, es posible que no haya errores presentes en el segmento.]

- `template < class A >`
`int suma(int num1, int num2, int num3)`
`{`
 `return num1 + num2 + num3;`
`}`
- `void imprimirResultados(int x, int y)`
`{`
 `cout << "La suma es " << x + y << '\n';`
 `return x + y;`
`}`

- c) `template< A >`
`A producto(A num1, A num2, A num3)`
`{`
`return num1 * num2 * num3;`
`}`
- d) `double cubo(int);`
`int cubo(int);`

7



Arreglos y vectores

*Abora ve, escríbelo
ante ellos en una tabla,
y anótalo en un libro.*

—Isaías 30:8

*Comienza en el principio...
y continúa hasta que llegues
al final; después detente.*

—Lewis Carroll

*Ir más allá es tan malo
como no llegar.*

—Confucio

OBJETIVOS

En este capítulo aprenderá a:

- Utilizar la estructura de datos tipo arreglo para representar un conjunto de elementos de datos relacionados.
- Utilizar arreglos para almacenar, ordenar y buscar datos en listas y tablas de valores.
- Declarar arreglos, inicializarlos y hacer referencia a elementos individuales de los arreglos.
- Aprender a pasar arreglos a las funciones.
- Conocer las técnicas de búsqueda y ordenamiento.
- Aprender a declarar y manipular arreglos multidimensionales.
- Utilizar la plantilla de clase `vector` de la Biblioteca estándar de C++.

- 7.1 Introducción
- 7.2 Arreglos
- 7.3 Declaración y creación de arreglos
- 7.4 Ejemplos acerca del uso de los arreglos
 - 7.4.1 Declaración de un arreglo y uso de un ciclo para inicializar los elementos del arreglo
 - 7.4.2 Inicialización de un arreglo en una declaración mediante una lista inicializadora
 - 7.4.3 Especificación del tamaño de un arreglo con una variable constante y establecimiento de los elementos de un arreglo con cálculos
 - 7.4.4 Suma de los elementos de un arreglo
 - 7.4.5 Uso de gráficos de barra para mostrar los datos de un arreglo en forma gráfica
 - 7.4.6 Uso de los elementos de un arreglo como contadores
 - 7.4.7 Uso de arreglos para sintetizar los resultados de una encuesta
 - 7.4.8 Uso de arreglos tipo carácter para almacenar y manipular cadenas
 - 7.4.9 Arreglos locales estáticos y arreglos locales automáticos
- 7.5 Paso de arreglos a funciones
- 7.6 Ejemplo práctico: la clase `LibroCalificaciones` que usa un arreglo para almacenar las calificaciones
- 7.7 Búsqueda de datos en arreglos mediante la búsqueda lineal
- 7.8 Ordenamiento de arreglos mediante el ordenamiento por inserción
- 7.9 Arreglos multidimensionales
- 7.10 Ejemplo práctico: la clase `LibroCalificaciones` que usa un arreglo bidimensional
- 7.11 Introducción a la plantilla de clase `vector` de la Biblioteca estándar de C++
- 7.12 (Opcional) Ejemplo práctico de Ingeniería de Software: colaboración entre los objetos en el sistema ATM
- 7.13 Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) |
[Ejercicios](#) | [Ejercicios de recursividad](#) | [Ejercicios con `vector`](#)

7.1 Introducción

En este capítulo presentamos el importante tema de las **estructuras de datos**: colecciones de elementos de datos relacionados. Los **arreglos** son estructuras de datos que consisten de elementos de datos relacionados, del mismo tipo. En el capítulo 3 aprendió acerca de las clases. En el capítulo 21 hablaremos sobre la noción de las **estructuras**. Cada estructura y clase puede contener elementos de datos relacionados de tipos posiblemente distintos. Los arreglos, las estructuras y las clases son entidades “estáticas”, en cuanto a que permanecen del mismo tamaño durante la ejecución del programa. [Desde luego que pueden ser de una clase de almacenamiento automático y, por ende, se pueden crear y destruir cada vez que el control del programa entre a (y salga de) los bloques en los que se definen].

Después de hablar acerca de cómo se declaran, se crean y se inicializan los arreglos, presentaremos una serie de ejemplos prácticos que demuestran varias manipulaciones comunes de los arreglos. Después explicaremos cómo las cadenas de caracteres (representadas hasta ahora por objetos `string`) también se pueden representar mediante arreglos de caracteres. Presentaremos un ejemplo de búsqueda en arreglos para encontrar elementos específicos. En este capítulo también introduciremos una de las aplicaciones computacionales más importante: el ordenamiento de datos (es decir, colocar los datos en cierto orden específico). Hay dos secciones de este capítulo en las que se amplía el ejemplo práctico de la clase `LibroCalificaciones` de los capítulos 3 a 6. En especial, utilizaremos los arreglos para permitir que la clase mantenga un conjunto de calificaciones en memoria y analizar las calificaciones que obtuvieron los estudiantes en distintos exámenes en un semestre; dos herramientas que no están presentes en las versiones anteriores de la clase `LibroCalificaciones`. Éstos y otros ejemplos del capítulo demostrarán las formas en las que los arreglos permiten a los programadores organizar y manipular datos.

El estilo de arreglos que utilizaremos en la mayor parte de este capítulo son los arreglos basados en apuntador, estilo C. (En el capítulo 8 estudiaremos los apuntadores.) En la sección final de este capítulo, y en el capítulo 22, Biblioteca de plantillas estándar (STL), veremos los arreglos como objetos completos llamados vectores. Descubriremos que estos arreglos basados en objetos son más seguros y versátiles que los arreglos basados en apuntadores estilo C que veremos en la primera parte de este capítulo.

7.2 Arreglos

Un arreglo es un grupo de ubicaciones de memoria consecutivas, todas ellas del mismo tipo. Para hacer referencia a una ubicación o elemento específico en el arreglo, especificamos su nombre y el **número de posición** del elemento específico en el arreglo.

La figura 7.1 muestra un arreglo de enteros llamado `c`. Este arreglo contiene 12 **elementos**. Para hacer referencia a cualquiera de estos elementos en un programa, se proporciona el nombre del arreglo seguido del número de posición del elemento específico entre corchetes (`[]`). Al número de posición se le conoce más formalmente como el **índice** o **subíndice** (este número especifica el número de elementos a partir del inicio del arreglo). El primer elemento en todo arreglo tiene el **subíndice 0 (cero)** y se conoce algunas veces como el **elemento cero**. Por ende, los elementos del arreglo `c` son `c[0]` (se pronuncia como “`c` sub cero”), `c[1]`, `c[2]` y así en lo sucesivo. El subíndice más alto en el arreglo `c` es 11, el cual es 1 menos que el número de elementos en el arreglo (12). Los nombres de los arreglos siguen las mismas convenciones que los demás nombres de variables; es decir, deben ser identificadores.

Un subíndice debe ser un entero o una expresión entera (usando cualquier tipo integral). Si un programa utiliza una expresión como un subíndice, entonces el programa evalúa la expresión para determinar el subíndice. Por ejemplo, si suponemos que la variable `a` es igual a 5 y que la variable `b` es igual a 6, entonces la instrucción

```
c[ a + b ] += 2;
```

suma 2 al elemento `c[11]` del arreglo. Observe que el nombre del arreglo con subíndice es un *value*: se puede utilizar en el lado izquierdo de una asignación, de igual forma que los nombres de las variables que no son arreglos.

Vamos a examinar el arreglo `c` de la figura 7.1 con más detalle. El **nombre** del arreglo completo es `c`. La manera en que se hace referencia a los 12 elementos de este arreglo es de `c[0]` a `c[11]`. El **valor** de `c[0]` es -45, el valor de `c[1]` es 6, el valor de `c[2]` es 0, el valor de `c[7]` es 62 y el valor de `c[11]` es 78. Para imprimir la suma de los valores contenidos en los primeros tres elementos del arreglo `c`, escribiríamos lo siguiente:

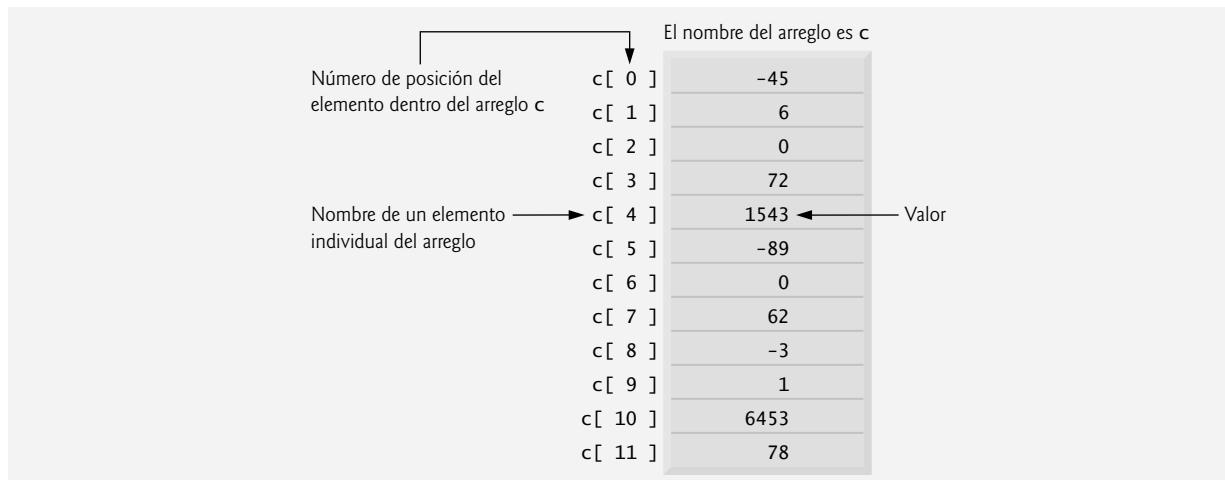
```
cout << c[ 0 ] + c[ 1 ] + c[ 2 ] << endl;
```

Para dividir el valor de `c[6]` entre 2 y asignar el resultado a la variable `x`, escribiríamos lo siguiente:

```
x = c[ 6 ] / 2;
```

Error común de programación 7.1

 Observe la diferencia entre el “séptimo elemento del arreglo” y el “elemento 7 del arreglo”. Los subíndices de los arreglos empiezan en 0, por lo que el “séptimo elemento del arreglo” tiene un subíndice de 6, mientras que el “elemento 7 del arreglo” tiene un subíndice de 7 y es en realidad el octavo elemento del arreglo. Por desgracia, esta distinción genera con frecuencia errores de desplazamiento en 1. Para evitar dichos errores, nos referimos explícitamente a los elementos específicos de un arreglo por medio del nombre del arreglo y el número de subíndice (por ejemplo, `c[6]` o `c[7]`).



Número de posición del elemento dentro del arreglo <code>c</code>	<code>c[0]</code>	-45	El nombre del arreglo es <code>c</code>
	<code>c[1]</code>	6	
	<code>c[2]</code>	0	
	<code>c[3]</code>	72	
Nombre de un elemento individual del arreglo	<code>c[4]</code>	1543	Valor
	<code>c[5]</code>	-89	
	<code>c[6]</code>	0	
	<code>c[7]</code>	62	
	<code>c[8]</code>	-3	
	<code>c[9]</code>	1	
	<code>c[10]</code>	6453	
	<code>c[11]</code>	78	

Figura 7.1 | Un arreglo con 12 elementos.

Los corchetes que se utilizan para encerrar el subíndice de un arreglo son en realidad un operador. Los corchetes tienen el mismo nivel de precedencia que los paréntesis. En la figura 7.2 se muestra la precedencia y asociatividad de los operadores introducidos hasta ahora. Observe que se agregaron los corchetes ([]) a la segunda fila de la figura 7.2. Los operadores se muestran de arriba hacia abajo, en orden descendente de precedencia, con su asociatividad y su tipo.

Operadores	Asociatividad	Tipo
::	izquierda a derecha	resolución de ámbito
O []	izquierda a derecha	Más alta
++ -- static_cast<tipo>(operando)	izquierda a derecha	unario (postfijo)
++ -- + - !	derecha a izquierda	unario (prefijo)
* / %	izquierda a derecha	Multiplicativo
+ -	izquierda a derecha	Aditivo
<< >>	izquierda a derecha	inserción/extracción
< <= > >=	izquierda a derecha	relacional
== !=	izquierda a derecha	igualdad
&&	izquierda a derecha	AND lógico
	izquierda a derecha	OR lógico
? :	derecha a izquierda	condicional
= += -= *= /= %=	derecha a izquierda	asignación
,	izquierda a derecha	Coma

Figura 7.2 | Precedencia y asociatividad de los operadores.

7.3 Declaración y creación de arreglos

Los objetos arreglo ocupan espacio en memoria. Para especificar el tipo de los elementos y el número de elementos requerido por un arreglo, use una declaración de la forma:

```
tipo nombreArreglo[ tamañoArreglo ];
```

El compilador reserva la cantidad apropiada de memoria. (Recuerde que en C++, una declaración que reserva memoria se conoce en forma más apropiada como definición.) El *tamañoArreglo* debe ser una constante entera mayor que cero. Por ejemplo, para indicar al compilador que debe reservar 12 elementos para el arreglo *c* de enteros, use la siguiente declaración:

```
int c[ 12 ]; // c es un arreglo de 12 enteros
```

Se puede reservar memoria para varios arreglos con una sola declaración. La siguiente declaración reserva 100 elementos para el arreglo *b* de enteros y 27 elementos para el arreglo *x* de enteros.

```
int b[ 100 ]; // b es un arreglo de 100 enteros
x[ 27 ]; // x es un arreglo de 27 enteros
```



Buena práctica de programación 7.1

Declaramos un arreglo por cada declaración para facilitar la legibilidad del código, de modificarlo y de hacer comentarios.

Los arreglos se pueden declarar de manera que contengan valores de cualquier tipo de datos que no sea referencia. Por ejemplo, un arreglo de tipo *char* se puede utilizar para almacenar una cadena de caracteres. Hasta ahora, hemos usado objetos *string* para almacenar cadenas de caracteres. En la sección 7.4 se introduce el uso de arreglos de caracteres para almacenar cadenas. En el capítulo 8 hablamos sobre las cadenas de caracteres y su similitud con los arreglos (una relación que C++ heredó de C), y sobre la relación entre los apuntadores y los arreglos.

7.4 Ejemplos acerca del uso de los arreglos

En esta sección presentaremos muchos ejemplos que demuestran cómo declarar arreglos, inicializarlos y realizar manipulaciones comunes en éstos.

7.4.1 Declaración de un arreglo y uso de un ciclo para inicializar los elementos del arreglo

El programa de la figura 7.3 declara el arreglo `n` entero de 10 elementos (línea 12). En las líneas 15 y 16 se utiliza una instrucción `for` para inicializar los elementos del arreglo con cero. Al igual que otras variables automáticas, los arreglos automáticos no se inicializan de manera implícita con cero, aunque los arreglos `static` sí. La primera instrucción de salida (línea 18) muestra los encabezados de columna para las columnas impresas en la instrucción `for` subsiguiente (líneas 21 y 22), la cual imprime el arreglo en formato tabular. Recuerde que `setw` especifica la anchura de campo en la que sólo se va a imprimir el *siguiente* valor.

7.4.2 Inicialización de un arreglo en una declaración mediante una lista inicializadora

Los elementos de un arreglo también se pueden inicializar en la declaración del arreglo, para lo cual colocamos después del nombre del arreglo un signo igual y una lista entre llaves, separada por comas, de **inicializadores**. El programa de la figura 7.4 utiliza una **lista inicializadora** para inicializar un arreglo de enteros con 10 valores (línea 13) y lo imprime en formato tabular (líneas 15 a 19).

Si hay menos inicializadores que elementos en el arreglo, el resto de los elementos del arreglo se inicializan con cero. Por ejemplo, los elementos del arreglo `n` en la figura 7.3 podrían haberse inicializado con la declaración

```
int n[ 10 ] = {};
```

// inicializa los elementos del arreglo n con 0

```

1 // Fig. 7.3: fig07_03.cpp
2 // Inicialización de un arreglo.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 int main()
11 {
12     int n[ 10 ]; // n es un arreglo de 10 enteros
13
14     // inicializa los elementos del arreglo n con 0
15     for ( int i = 0; i < 10; i++ )
16         n[ i ] = 0; // establece el elemento en la ubicación i a 0
17
18     cout << "Elemento" << setw( 13 ) << "Valor" << endl;
19
20     // imprime el valor de cada elemento del arreglo
21     for ( int j = 0; j < 10; j++ )
22         cout << setw( 7 ) << j << setw( 13 ) << n[ j ] << endl;
23
24     return 0; // indica que terminó correctamente
25 }
```

// fin de main

Elemento	Valor
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

Figura 7.3 | Inicialización de los elementos de un arreglo en cero, e impresión del arreglo.

```

1 // Fig. 7.4: fig07_04.cpp
2 // Inicialización de un arreglo en una declaración.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 int main()
11 {
12     // usa la lista inicializadora para inicializar el arreglo n
13     int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
14
15     cout << "Elemento" << setw( 13 ) << "Valor" << endl;
16
17     // imprime el valor de cada elemento del arreglo
18     for ( int i = 0; i < 10; i++ )
19         cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;
20
21     return 0; // indica que terminó correctamente
22 } // fin de main

```

Elemento	Valor
0	32
1	27
2	64
3	18
4	95
5	14
6	90
7	70
8	60
9	37

Figura 7.4 | Inicialización de los elementos de un arreglo en su declaración.

La declaración inicializa de manera implícita los elementos con cero, ya que hay menos inicializadores (ninguno en este caso) que elementos en el arreglo. Esta técnica sólo se puede utilizar en la declaración del arreglo, mientras que la técnica de inicialización que se muestra en la figura 7.3 se puede utilizar de manera repetida durante la ejecución del programa, para “reinicializar” los elementos de un arreglo.

Si el tamaño del arreglo se omite en una declaración con una lista inicializadora, el compilador determina el número de elementos en el arreglo mediante un conteo del número de elementos en la lista inicializadora. Por ejemplo,

```
int n[] = { 1, 2, 3, 4, 5 };
```

crea un arreglo de cinco elementos.

Si se especifican el tamaño del arreglo y una lista inicializadora en la declaración de un arreglo, el número de inicializadores debe ser menor o igual que el tamaño del arreglo. La declaración del arreglo

```
int n[ 5 ] = { 32, 27, 64, 18, 95, 14 };
```

produce un error de compilación, ya que hay seis inicializadores y sólo cinco elementos en el arreglo.

Error común de programación 7.2



Si se proporcionan más inicializadores en una lista inicializadora que los elementos que contiene el arreglo, se produce un error de compilación.

Error común de programación 7.3



Olvidar inicializar los elementos de un arreglo, cuyos elementos deben inicializarse, es un error lógico.

7.4.3 Especificación del tamaño de un arreglo con una variable constante y establecimiento de los elementos de un arreglo con cálculos

En la figura 7.5 se establecen los elementos de un arreglo `s` de 10 elementos con los enteros 2, 4, 6, ..., 20 (líneas 17 y 18), y se imprime el arreglo en formato tabular (líneas 20 a 24). Para generar estos números (línea 18), se multiplica cada valor sucesivo del contador de ciclo por 2, y se le suma 2.

En la línea 13 se utiliza el **calificador const** para declarar lo que se conoce como una **variable constante** llamada `tamanoArreglo` con el valor 10. Las variables constantes deben inicializarse con una expresión constante cuando se declaran y no pueden modificarse de ahí en adelante (como se muestra en las figuras 7.6 y 7.7). Las variables constantes también se conocen como **constants con nombre** o **variables de sólo lectura**.

Error común de programación 7.4



Si no se asigna un valor a una variable constante cuando se declara es un error de compilación.

```

1 // Fig. 7.5: fig07_05.cpp
2 // Establece el arreglo s con los enteros pares del 2 a 20.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 int main()
11 {
12     // la variable constante se puede usar para especificar el tamaño de los arreglos
13     const int tamanoArreglo = 10; // debe inicializarse en la declaración
14
15     int s[ tamanoArreglo ]; // el arreglo s tiene 10 elementos
16
17     for ( int i = 0; i < tamanoArreglo; i++ ) // establece los valores
18         s[ i ] = 2 + 2 * i;
19
20     cout << "Elemento" << setw( 13 ) << "Valor" << endl;
21
22     // imprime el contenido del arreglo s en formato tabular
23     for ( int j = 0; j < tamanoArreglo; j++ )
24         cout << setw( 7 ) << j << setw( 13 ) << s[ j ] << endl;
25
26     return 0; // indica que terminó correctamente
27 } // fin de main

```

Elemento	Valor
0	2
1	4
2	6
3	8
4	10
5	12
6	14
7	16
8	18
9	20

Figura 7.5 | Generación de valores para colocarlos en los elementos de un arreglo.

```

1 // Fig. 7.6: fig07_06.cpp
2 // Uso de una variable constante inicializada en forma apropiada.
3 #include <iostream>

```

Figura 7.6 | Inicialización y uso de una variable constante. (Parte I de 2).

```

4  using std::cout;
5  using std::endl;
6
7  int main()
8  {
9      const int x = 7; // variable constante sin inicializar
10
11     cout << "El valor de la variable constante x es: " << x << endl;
12
13     return 0; // indica que terminó correctamente
14 } // fin de main

```

El valor de la variable constante x es: 7

Figura 7.6 | Inicialización y uso de una variable constante. (Parte 2 de 2).

```

1 // Fig. 7.7: fig07_07.cpp
2 // Una variable const se debe inicializar.
3
4 int main()
5 {
6     const int x; // Error: x debe inicializarse
7
8     x = 7; // Error: no se puede modificar una variable const
9
10    return 0; // indica que terminó correctamente
11 } // fin de main

```

Mensaje de error del compilador de línea de comandos Borland C++:

```

Error E2304 fig07_07.cpp 6: Constant variable 'x' must be initialized
in function main()
Error E2024 fig07_07.cpp 8: Cannot modify a const object in function main()

```

Mensaje de error del compilador Microsoft Visual C++ 2005:

```

C:\cpphttp6_ejemplos\cap07\fig07_07.cpp(6) : error C2734: 'x' : const object
must be initialized if not extern
C:\cpphttp6_ejemplos\cap07\fig07_07.cpp(8) : error C3892: 'x' : you cannot
assign to a variable that is const

```

Mensaje de error del compilador GNU C++:

```

fig07_07.cpp:6: error: uninitialized const 'x'
fig07_07.cpp:8: error: assignment of read-only variable 'x'

```

Figura 7.7 | Las variables const se deben inicializar.



Error común de programación 7.5

Asignar un valor a una variable constante en una instrucción ejecutable es un error de compilación.

En la figura 7.7, observe que los errores de compilación producidos por Borland C++ y Microsoft Visual C++ se refieren a la variable `int x` como un “objeto const”. El estándar ISO/IEC de C++ define a un “objeto” como una “región de almacenamiento”. Al igual que los objetos de las clases, las variables de tipo fundamental también ocupan espacio en memoria, por lo que se conocen comúnmente como “objetos”.

Las variables constantes se pueden colocar en cualquier parte en la que se espera una expresión constante. En la figura 7.5, la variable constante `tamanioArreglo` especifica el tamaño del arreglo `s` en la línea 15.



Error común de programación 7.6

Sólo pueden utilizarse para declarar el tamaño de arreglos automáticos y estáticos. Si no se utiliza una constante para este propósito, se produce un error de compilación.

El uso de variables constantes para especificar tamaños de arreglos hace a los programas más **escalables**. En la figura 7.5, la primera instrucción `for` podría llenar un arreglo de 1000 elementos con sólo modificar el valor de `tamanoArreglo` en su declaración, de 10 a 1000. Si no se hubiera utilizado la variable constante `tamanoArreglo`, tendríamos que modificar las líneas 15, 17 y 23 del programa para escalarlo y que pudiera manejar 1000 elementos en el arreglo. A medida que los programas van creciendo, esta técnica se vuelve más útil para escribir programas más claros y fáciles de modificar.



Observación de Ingeniería de Software 7.1

Definir el tamaño de cada arreglo como una variable constante, en vez de una constante literal, puede hacer a los programas más escalables.



Buena práctica de programación 7.2

Definir el tamaño de un arreglo como una variable constante, en vez de una constante literal, hace a los programas más claros. Esta técnica elimina lo que se conoce como **números mágicos**. Por ejemplo, al mencionar en forma repetida el tamaño 10 en el código para procesar arreglos para un elemento de 10 arreglos, el número 10 recibe un significado artificial y puede ser confuso cuando el programa incluya otros números 10 que no tengan nada que ver con el tamaño del arreglo.

7.4.4 Suma de los elementos de un arreglo

A menudo, los elementos de un arreglo representan una serie de valores para ser utilizados en un cálculo. Por ejemplo, si los elementos de un arreglo representan calificaciones de un examen, tal vez un profesor desea obtener el total de los elementos del arreglo y usar esa suma para calcular el promedio de la clase para el examen. Los ejemplos acerca del uso de la clase `LibroCalificaciones` que se muestran más adelante en el capítulo, a saber las figuras 7.16 a 7.17, y las figuras 7.23 a 7.24, use esta técnica.

El programa en la figura 7.8 suma los valores contenidos en el arreglo `a` de 10 elementos enteros. El programa declara, crea e inicializa el arreglo en la línea 10. La instrucción `for` (líneas 14 y 15) realiza los cálculos. Los valores que se suministran como inicializadores para el arreglo `a` también se podrían haber pedido en el programa al usuario mediante el teclado, o de un archivo en el disco (vea el capítulo 17, Procesamiento de archivos). Por ejemplo, la instrucción `for`

```
for ( int j = 0; j < tamanoArreglo; j++ )
    cin >> a[ j ];
```

lee un valor a la vez del teclado y almacena el valor en el elemento `a[j]`.

```

1 // Fig. 7.8: fig07_08.cpp
2 // Calcula la suma de los elementos del arreglo.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     const int tamanoArreglo = 10; // variable constante que indica el tamaño del arreglo
10    int a[ tamanoArreglo ] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
11    int total = 0;
12
13    // suma el contenido del arreglo a
14    for ( int i = 0; i < tamanoArreglo; i++ )
15        total += a[ i ];
16
17    cout << "Total de elementos del arreglo: " << total << endl;

```

Figura 7.8 | Cálculo de la suma de los elementos de un arreglo. (Parte I de 2).

```

18
19     return 0; // indica que terminó correctamente
20 } // fin de main

```

Total de elementos del arreglo: 849

Figura 7.8 | Cálculo de la suma de los elementos de un arreglo. (Parte 2 de 2).

7.4.5 Uso de gráficos de barra para mostrar los datos de un arreglo en forma gráfica

Muchos programas presentan datos a los usuarios en forma gráfica. Por ejemplo, los valores numéricos se muestran comúnmente como barras en un gráfico de barras. En dicho gráfico, las barras más extensas representan valores numéricos proporcionalmente más grandes. Una manera simple de mostrar datos numéricos en forma gráfica es mediante un gráfico de barras que muestra cada valor numérico como una barra de asteriscos (*).

A menudo, a los profesores les gusta examinar la distribución de calificaciones en un examen. Un profesor podría graficar el número de calificaciones en cada una de varias categorías, para visualizar la distribución de calificaciones. Suponga que las calificaciones fueron 87, 68, 94, 100, 83, 78, 85, 91, 76 y 87. Observe que hubo una calificación de 100, dos calificaciones entre 90 y 99, cuatro calificaciones entre 80 y 89, dos calificaciones entre 70 y 79, una calificación entre 60 y 69, y ninguna calificación menor a 60. Nuestro siguiente programa (figura 7.9) almacena estos datos de distribución de calificaciones en un arreglo de 11 elementos, cada uno de los cuales corresponde a una categoría de calificaciones. Por ejemplo, `n[0]` indica el número de calificaciones en el rango de 0 a 9, `n[7]` indica el número de calificaciones en el rango de 70 a 79 y `n[10]` indica el número de calificaciones de 100. Las dos versiones de la clase `LibroCalificaciones` que se muestran más adelante en este capítulo (figuras 7.16 y 7.17, y figuras 7.23 y 7.24) contienen código que calcula estas frecuencias de calificaciones, con base en un conjunto de calificaciones. Por ahora crearemos el arreglo en forma manual, analizando el conjunto de calificaciones.

```

1 // Fig. 7.9: fig07_09.cpp
2 // Programa para imprimir gráficos de barra.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 int main()
11 {
12     const int tamanoArreglo = 11;
13     int n[ tamanoArreglo ] = { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };
14
15     cout << "Distribucion de calificaciones:" << endl;
16
17     // para cada elemento del arreglo n, imprime una barra del gráfico
18     for ( int i = 0; i < tamanoArreglo; i++ )
19     {
20         // imprime etiquetas de las barras ("0-9:", ..., "90-99:", "100:")
21         if ( i == 0 )
22             cout << " 0-9: ";
23         else if ( i == 10 )
24             cout << " 100: ";
25         else
26             cout << i * 10 << "-" << ( i * 10 ) + 9 << ": ";
27
28         // imprime barra de asteriscos
29         for ( int estrellas = 0; estrellas < n[ i ]; estrellas++ )
30             cout << '*';
31     }

```

Figura 7.9 | Programa para imprimir gráficos de barra. (Parte 1 de 2).

```

32     cout << endl; // inicia una nueva línea de salida
33 } // fin de for externo
34
35 return 0; // indica que terminó correctamente
36 } // fin de main

```

```

0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
100: *

```

Figura 7.9 | Programa para imprimir gráficos de barra. (Parte 2 de 2).

El programa lee los números del arreglo y grafica la información como un gráfico de barras, mostrando cada rango de calificaciones seguido de una barra de asteriscos, los cuales indican el número de calificaciones en ese rango. Para etiquetar cada barra, en las líneas 21 a 26 se imprime un rango de calificaciones (por ejemplo, "70-79: ") con base en el valor actual de la variable contador *i*. La instrucción **for** anidada (líneas 29 y 30) imprime las barras. Observe la condición de continuación de ciclo en la línea 29 (*estrellas < n[i]*). Cada vez que el programa llega al **for** interior, el ciclo cuenta desde 0 hasta *n[i]*, con lo cual usa un valor en el arreglo *n* para determinar el número de asteriscos a mostrar. En este ejemplo, *n[0] - n[5]* contiene ceros, ya que ningún estudiante recibió una calificación menor a 60. Por ende, el programa no muestra asteriscos enseguida de los primeros seis rangos de calificaciones.



Error común de programación 7.7

Aunque es posible utilizar la misma variable de control en una instrucción **for** y en una segunda instrucción **for** anidada en la primera, esto es confuso y puede producir errores lógicos.

7.4.6 Uso de los elementos de un arreglo como contadores

Algunas veces, los programas usan variables contadores para sintetizar datos, como los resultados de una encuesta. En la figura 6.9, utilizamos contadores separados en nuestro programa para tirar dados, para rastrear el número de ocurrencias de cada lado de un dado, a medida que el programa tiraba el dado 6,000,000 veces. En la figura 7.10 se muestra una versión de este programa, en la que se utiliza un arreglo.

```

1 // Fig. 7.10: fig07_10.cpp
2 // Tira un dado de seis lados 6,000,000 veces.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 #include <cstdlib>
11 using std::rand;
12 using std::srand;
13
14 #include <ctime>
15 using std::time;
16

```

Figura 7.10 | Programa para tirar dados que utiliza un arreglo en vez de una instrucción **switch**. (Parte 1 de 2).

```

17 int main()
18 {
19     const int tamanoArreglo = 7; // ignora el elemento cero
20     int frecuencia[ tamanoArreglo ] = {};// inicializa los elementos con 0
21
22     srand( time( 0 ) ); // siembra el generador de números aleatorios
23
24     // tira el dado 6,000,000 de veces; usa el valor del dado como índice de frecuencia
25     for ( int tiro = 1; tiro <= 6000000; tiro++ )
26         frecuencia[ 1 + rand() % 6 ]++;
27
28     cout << "Cara" << setw( 13 ) << "Frecuencia" << endl;
29
30     // imprime el valor de cada elemento del arreglo
31     for ( int cara = 1; cara < tamanoArreglo; cara++ )
32         cout << setw( 4 ) << cara << setw( 13 ) << frecuencia[ cara ]
33             << endl;
34
35     return 0; // indica que terminó correctamente
36 } // fin de main

```

Cara	Frecuencia
1	1001086
2	1000538
3	998953
4	999742
5	999894
6	999787

Figura 7.10 | Programa para tirar dados que utiliza un arreglo en vez de una instrucción `switch`. (Parte 2 de 2).

La figura 7.10 utiliza el arreglo `frecuencia` (línea 20) para contar las ocurrencias de cada lado del dado. *La instrucción individual en la línea 26 de este programa reemplaza a la instrucción switch en las líneas 30 a 52 de la figura 6.9.* En la línea 26 se utiliza un valor aleatorio para determinar cuál elemento de `frecuencia` incrementar durante cada iteración del ciclo. El cálculo en la línea 26 produce un subíndice aleatorio de 1 a 6, por lo que el arreglo `frecuencia` debe ser lo suficientemente grande como para almacenar seis contadores. Sin embargo, usamos un arreglo de siete elementos en el que ignoramos `frecuencia[0]`; es más lógico hacer que la cara del dado 1 incremente a `frecuencia[1]` que a `frecuencia[0]`. Por ende, el valor de cada cara se utiliza como subíndice para el arreglo `frecuencia`. También reemplazamos las líneas 56 a 61 de la figura 6.9, iterando a través del arreglo `frecuencia` para imprimir los resultados (líneas 31 a 33).

7.4.7 Uso de arreglos para sintetizar los resultados de una encuesta

Nuestro siguiente ejemplo (figura 7.11) utiliza arreglos para sintetizar los resultados de los datos recolectados en una encuesta. Considere el siguiente enunciado del problema:

Se pidió a cuarenta estudiantes que calificaran la calidad de la comida en la cafetería estudiantil, en una escala del 1 al 10 (donde 1 significa pésimo y 10 significa excelente). Coloque las 40 respuestas en un arreglo entero y sintetice los resultados de la encuesta.

Ésta es una típica aplicación de procesamiento de arreglos. Deseamos resumir el número de respuestas de cada tipo (es decir, del 1 al 10). El arreglo `respuestas` (líneas 17 a 19) es un arreglo entero de 40 elementos, y contiene las respuestas de los estudiantes a la encuesta. Observe que el arreglo `respuestas` se declara como `const`, ya que sus valores no cambian (y no deben hacerlo). Utilizamos un arreglo de 11 elementos llamado `frecuencia` (línea 22) para contar el número de ocurrencias de cada respuesta. Cada elemento del arreglo se utiliza como un contador para una de las respuestas de la encuesta, y se inicializa con cero. Al igual que en la figura 7.10, ignoramos `frecuencia[0]`.



Observación de Ingeniería de Software 7.2

El calificador `const` se debe utilizar para hacer valer el principio de menor privilegio. Al usar este principio para diseñar software de manera apropiada, se pueden reducir de manera considerable el tiempo de depuración y los efectos secundarios inapropiados, y se pueden facilitar los procesos de modificación y mantenimiento del programa.



Buena práctica de programación 7.3

Hay que esforzarse por mantener la claridad del programa. Algunas veces es conveniente sacrificar el uso más eficiente de la memoria o del tiempo del procesador, para escribir programas más claros.



Tip de rendimiento 7.1

Algunas veces, las consideraciones de rendimiento sobrepasan de manera considerable a las consideraciones de claridad.

La primera instrucción **for** (líneas 26 y 27) recibe las respuestas, una a la vez, del arreglo **respuestas** e incrementa uno de los 10 contadores en el arreglo **frecuencia** (de **frecuencia[1]** a **frecuencia[10]**). La instrucción clave en el ciclo es la línea 27, la cual incrementa el contador de **frecuencia** apropiado, dependiendo del valor de **respuestas[respuesta]**.

```

1 // Fig. 7.11: fig07_11.cpp
2 // Programa de encuesta de estudiantes.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 int main()
11 {
12     // define los tamaños de los arreglos
13     const int tamanoRespuesta = 40; // tamaño del arreglo respuestas
14     const int tamanoFrecuencia = 11; // tamaño del arreglo frecuencia
15
16     // coloca las respuestas de la encuesta en el arreglo respuestas
17     const int respuestas[ tamanoRespuesta ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8,
18         10, 1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7,
19         5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
20
21     // inicializa los contadores de frecuencia con 0
22     int frecuencia[ tamanoFrecuencia ] = {};
23
24     // para cada respuesta, selecciona el elemento de respuestas y usa ese valor
25     // como subíndice de frecuencia para determinar el elemento a incrementar
26     for ( int respuesta = 0; respuesta < tamanoRespuesta; respuesta++ )
27         frecuencia[ respuestas[ respuesta ] ]++;
28
29     cout << "Calificacion" << setw( 17 ) << "Frecuencia" << endl;
30
31     // imprime el valor de cada elemento del arreglo
32     for ( int calificacion = 1; calificacion < tamanoFrecuencia; calificacion++ )
33         cout << setw( 12 ) << calificacion << setw( 17 ) << frecuencia[ calificacion ]
34             << endl;
35
36     return 0; // indica que terminó correctamente
37 } // fin de main

```

Calificación	Frecuencia
1	2
2	2
3	2
4	2
5	5
6	11
7	5
8	7
9	1
10	3

Figura 7.11 | Programa para analizar encuestas.

Vamos a considerar varias iteraciones del ciclo `for`. Cuando la variable de control `respuesta` es 0, el valor de `respuestas[respuesta]` es el valor de `respuestas[0]` (es decir, 1 en la línea 17), por lo que el programa interpreta a `frecuencia[respuestas[respuesta]]++` como

```
frecuencia[ 1 ]++
```

con lo cual se incrementa el valor en el elemento 1 del arreglo. Para evaluar la expresión, empiece con el valor en el conjunto más interno de corchetes (`respuesta`). Una vez que conozca el valor de `respuesta` (que viene siendo el valor de la variable de control de ciclo en la línea 26), insértelo en la expresión y evalúe el siguiente conjunto más externo de corchetes (`respuestas[respuesta]`, que viene siendo un valor seleccionado del arreglo `respuestas` en las líneas 17 a 19). Después utilice el valor resultante como subíndice del arreglo `frecuencia`, para especificar cuál contador se va a incrementar.

Cuando `respuesta` es 1, `respuestas[respuesta]` es el valor de `respuestas[1]` (2), por lo que el programa interpreta a `frecuencia[respuestas[respuesta]]++` como

```
frecuencia[ 2 ]++
```

con lo cual se incrementa el elemento 2 del arreglo.

Cuando `respuesta` es 2, `respuestas[respuesta]` es el valor de `respuestas[2]` (6), por lo que el programa interpreta a `frecuencia[respuestas[respuesta]]++` como

```
frecuencia[ 6 ]++
```

con lo cual se incrementa el elemento 6 del arreglo, y así en lo sucesivo. Sin importar el número de respuestas procesadas en la encuesta, el programa sólo requiere un arreglo de 11 elementos (en el cual se ignora el elemento cero) para resumir los resultados, ya que todos los valores de las respuestas se encuentran entre 1 y 10, y los valores de subíndice para un arreglo de 11 elementos son del 0 al 10.

Si los datos en el arreglo `respuestas` tuvieran valores inválidos como 13, el programa trataría de sumar 1 a `frecuencia[13]`, lo cual se encuentra fuera de los límites del arreglo. *C++ no cuenta con comprobación de límites para evitar que la computadora haga referencia a un elemento que no existe.* Por lo tanto, un programa en ejecución puede “salirse” de cualquier extremo de un arreglo sin advertencia. El programador debe asegurar que todas las referencias a los arreglos permanezcan dentro de los límites del arreglo.

Error común de programación 7.8



Hacer referencia a un elemento fuera de los límites del arreglo es un error lógico en tiempo de ejecución. No es un error de sintaxis.

Tip para prevenir errores 7.1



Al iterar a través de un arreglo, el subíndice del arreglo no debe ser mayor o igual a 0 y siempre debe ser menor que el número total de elementos en el arreglo (uno menos que el tamaño del arreglo). Asegúrese que la condición de terminación de ciclo evite acceder a los elementos fuera de este rango.

Tip de portabilidad 7.1



Los efectos (por lo general, graves) de hacer referencia a los elementos fuera de los límites del arreglo son dependientes del sistema. A menudo, esto produce modificaciones en el valor de una variable no relacionada, o un error fatal que termina la ejecución del programa.

C++ es un lenguaje extensible. La sección 7.11 presenta la plantilla de clase `vector` de la Biblioteca estándar de C++, la cual permite a los programadores realizar muchas operaciones que no están disponibles para los arreglos integrados en C++. Por ejemplo, podremos comparar objetos `vector` directamente y asignar un `vector` a otro. En el capítulo 11, extenderemos aún más el C++, al implementar un arreglo como nuestra propia clase definida por el usuario. Esta nueva definición del arreglo nos permitirá recibir e imprimir arreglos completos mediante `cint` y `cout`, inicializar arreglos al momento de crearlos, evitar el acceso a los elementos del arreglo que estén fuera del rango, y modificar el rango de los subíndices (e incluso hasta el tipo de su subíndice) de manera que el primer elemento de un arreglo no tenga que ser el elemento 0. Incluso, podremos usar subíndices que no sean enteros.

Tip para prevenir errores 7.2



En el capítulo 11 veremos cómo desarrollar una clase que represente a un “arreglo inteligente”, el cual comprueba que todas las referencias a los subíndices se encuentren dentro de los límites, en tiempo de ejecución. El uso de tales tipos de datos inteligentes ayuda a eliminar los errores.

7.4.8 Uso de arreglos tipo carácter para almacenar y manipular cadenas

Hasta este momento, hemos descrito sólo el uso de los arreglos enteros. Sin embargo, los arreglos pueden ser de cualquier tipo. Ahora veremos cómo almacenar cadenas de caracteres en arreglos tipo carácter. Recuerde que, desde el capítulo 3, hemos usado objetos `string` para almacenar cadenas de caracteres, como el nombre del curso en nuestra clase `LibroCalificaciones`. Una cadena como "holá" es en realidad un arreglo de caracteres. Aunque los objetos `string` son convenientes de usar y reducen el potencial de errores, los arreglos de caracteres que representan cadenas tienen varias características únicas, las cuales veremos en esta sección. A medida que el lector avance en su estudio de C++, tal vez encuentre herramientas de C++ con las que se vea forzado a utilizar arreglos de caracteres en vez de objetos `string`. Además, puede encontrarse con casos en los que tenga que actualizar el código existente mediante el uso de arreglos.

Un arreglo de caracteres se puede inicializar mediante el uso de una literal de cadena. Por ejemplo, la declaración

```
char cadena1[] = "primero";
```

inicializa los elementos del arreglo `cadena1` con los caracteres individuales en la literal de cadena "primero". El compilador determina el tamaño del arreglo `cadena1` en la anterior declaración, con base en la longitud de la cadena. Es importante observar que la cadena "primero" contiene cinco caracteres más un carácter especial de terminación de cadena, conocido como **carácter nulo**. Por ende, el arreglo `cadena1` en realidad contiene seis elementos. La constante tipo carácter que representa el carácter nulo es '\0' (barra diagonal inversa seguida de un cero). Todas las cadenas representadas mediante arreglos de caracteres terminan con este carácter. Un arreglo de caracteres que representa a una cadena siempre debe declararse con el tamaño suficiente como para contener el número de caracteres en la cadena, junto con el carácter nulo de terminación.

Los arreglos de caracteres también se pueden inicializar mediante constantes tipo carácter individuales en una lista inicializadora. La declaración anterior es equivalente a la siguiente forma más compleja:

```
char cadena1[] = { 'f', 'i', 'r', 's', 't', '\0' };
```

Observe el uso de comillas sencillas para delinear cada constante tipo carácter. Además, observe que hemos proporcionado de manera explícita el carácter nulo de terminación como el último valor inicializador. Sin él, este arreglo representaría tan sólo un arreglo de caracteres, no una cadena. Como veremos en el capítulo 8, si no se proporciona un carácter nulo de terminación para una cadena, se pueden producir errores lógicos.

Debido a que una cadena es un arreglo de caracteres, podemos acceder directamente a los caracteres individuales en una cadena mediante la notación de subíndices de arreglos. Por ejemplo, `cadena1[0]` es el carácter 'f', `cadena1[3]` es el carácter 's' y `cadena1[5]` es el carácter nulo.

Podemos introducir una cadena directamente en un arreglo de caracteres mediante el teclado, usando `cin` y `>>`. Por ejemplo, la declaración

```
char cadena2[ 20 ];
```

crea un arreglo de caracteres capaz de almacenar una cadena de hasta 19 caracteres y un carácter nulo de terminación. La instrucción

```
cin >> cadena2;
```

lee una cadena del teclado y la coloca en `cadena2`; después adjunta el carácter nulo al final de la cadena introducida por el usuario. Observe que la instrucción anterior sólo proporciona el nombre del arreglo, no da información sobre el tamaño del mismo. Es responsabilidad del programador asegurar que el arreglo en el que se coloque la cadena sea capaz de contener cualquier cadena que el usuario escriba en el teclado. De manera predeterminada, `cin` lee caracteres del teclado hasta encontrar el primer carácter de espacio en blanco; sin importar el tamaño del arreglo. Por lo tanto, al recibir datos con `cin` y `>>` se pueden insertar más allá del final del arreglo (en la sección 8.3 podrá consultar información acerca de cómo evitar que se inserten datos más allá del final de un arreglo `char`).

Error común de programación 7.9

 Si no se proporciona a `cin >>` un arreglo de caracteres lo bastante grande como para almacenar una cadena que se escriba mediante el teclado, se puede producir pérdida de datos en un programa, además de otros errores graves en tiempo de ejecución.

Un arreglo de caracteres que representa una cadena con terminación nula se puede imprimir mediante `cout` y `<<`. La instrucción

```
cout << cadena2;
```

imprime el arreglo cadena2. Observe que, al igual que `cin >>`, `cout <<` no necesita saber qué tan grande es el arreglo de caracteres. Los caracteres de la cadena se imprimen hasta encontrar un carácter nulo de terminación; el carácter nulo no se imprime. [Nota: `cin` y `cout` asumen que los arreglos de caracteres se deben procesar como cadenas terminadas por caracteres nulos; `cin` y `cout` no proporcionan capacidades de procesamiento de entrada y salida similares para otros tipos de arreglos].

La figura 7.12 demuestra cómo inicializar un arreglo de caracteres con una literal de cadena, cómo leer una cadena y colocarla en un arreglo de caracteres, cómo imprimir un arreglo de caracteres como una cadena, y cómo acceder a los caracteres individuales de una cadena.

En las líneas 23 y 24 de la figura 7.12 se utiliza una instrucción `for` para iterar a través del arreglo `cadena1` e imprimir sus caracteres separados por espacios. La condición en la instrucción `for`, `cadena1[i] != '\0'`, es verdadera hasta que el ciclo encuentra el carácter nulo de terminación de la cadena.

7.4.9 Arreglos locales estáticos y arreglos locales automáticos

En el capítulo 6 hablamos sobre el especificador de clase de almacenamiento `static`. Una variable local `static` en la definición de una función existe durante todo el programa, pero sólo puede verse en el cuerpo de la función.



Tip de rendimiento 7.2

Podemos aplicar static a la declaración de un arreglo local, de manera que el arreglo no se cree e inicialice cada vez que el programa llame a la función, y no se destruya cada vez que termine la función en el programa. Esto puede mejorar el rendimiento, en especial cuando se utilizan arreglos extensos.

```

1 // Fig. 7.12: fig07_12.cpp
2 // Cómo tratar los arreglos de caracteres como cadenas.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 int main()
9 {
10     char cadena1[ 20 ]; // reserva 20 caracteres
11     char cadena2[] = "literal de cadena"; // reserva 17 caracteres
12
13     // lee la cadena del usuario y la coloca en el arreglo cadena1
14     cout << "Escriba la cadena \"hola todos\": ";
15     cin >> cadena1; // lee "hola" [el espacio termina la entrada]
16
17     // imprime las cadenas
18     cout << "cadena1 es: " << cadena1 << "\ncadena2 es: " << cadena2;
19
20     cout << "\ncadena1 con espacios entre caracteres es:\n";
21
22     // imprime caracteres hasta llegar al carácter nulo
23     for ( int i = 0; cadena1[ i ] != '\0'; i++ )
24         cout << cadena1[ i ] << ' ';
25
26     cin >> cadena1; // lee "todos"
27     cout << "\ncadena1 es: " << cadena1 << endl;
28
29     return 0; // indica que terminó correctamente
30 } // fin de main

```

```

Escriba la cadena "hola todos": hola todos
cadena1 es: hola
cadena2 es: literal de cadena
cadena1 con espacios entre caracteres es:
h o l a
cadena1 es: todos

```

Figura 7.12 | Arreglos de caracteres procesados como cadenas.

Un programa inicializa los arreglos locales **static** la primera vez que encuentra sus declaraciones. Si el programador no inicializa un arreglo **static** de manera explícita, el compilador inicializa con cero cada elemento de ese arreglo al momento de su creación. Recuerde que C++ no realiza dicha inicialización predeterminada para las variables automáticas.

La figura 7.13 demuestra la función **inicArregloStatic** (líneas 25 a 41) con un arreglo local **static** (línea 28) y la función **inicArregloAutomatico** (líneas 44 a 60) con un arreglo local automático (línea 47).

```
1 // Fig. 7.13: fig07_13.cpp
2 // Los arreglos static se inicializan con cero.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void inicArregloStatic( void ); // prototipo de función
8 void inicArregloAutomatico( void ); // prototipo de función
9
10 int main()
11 {
12     cout << "Primera llamada a cada función:\n";
13     inicArregloStatic();
14     inicArregloAutomatico();
15
16     cout << "\n\nSegunda llamada a cada función:\n";
17     inicArregloStatic();
18     inicArregloAutomatico();
19     cout << endl;
20
21     return 0; // indica que terminó correctamente
22 } // fin de main
23
24 // función para demostrar un arreglo local static
25 void inicArregloStatic( void )
26 {
27     // inicializa los elementos con 0 la primera vez que se llama a la función
28     static int arreglo1[ 3 ]; // arreglo local static
29
30     cout << "\nValores al entrar en inicArregloStatic:\n";
31
32     // imprime el contenido de arreglo1
33     for ( int i = 0; i < 3; i++ )
34         cout << "arreglo1[" << i << "] = " << arreglo1[ i ] << " ";
35
36     cout << "\nValores al salir de inicArregloStatic:\n";
37
38     // modifica e imprime el contenido de arreglo1
39     for ( int j = 0; j < 3; j++ )
40         cout << "arreglo1[" << j << "] = " << ( arreglo1[ j ] += 5 ) << " ";
41 } // fin de la función inicArregloStatic
42
43 // función para demostrar un arreglo local automático
44 void inicArregloAutomatico( void )
45 {
46     // inicializa los elementos cada vez que se llama a la función
47     int arreglo2[ 3 ] = { 1, 2, 3 }; // arreglo local automático
48
49     cout << "\n\nValores al entrar a inicArregloAutomatico:\n";
50
51     // imprime el contenido de arreglo2
52     for ( int i = 0; i < 3; i++ )
```

Figura 7.13 | Inicialización de un arreglo **static** e inicialización de un arreglo automático. (Parte I de 2).

```

53     cout << "arreglo2[" << i << "] = " << arreglo2[ i ] << " ";
54
55     cout << "\nValores al salir de inicArregloAutomatico:\n";
56
57     // modifica e imprime el contenido de arreglo2
58     for ( int j = 0; j < 3; j++ )
59         cout << "arreglo2[" << j << "] = " << ( arreglo2[ j ] += 5 ) << " ";
60 } // fin de la función inicArregloAutomatico

```

Primera llamada a cada función:

Valores al entrar en inicArregloStatic:
`arreglo1[0] = 0 arreglo1[1] = 0 arreglo1[2] = 0`
 Valores al salir de inicArregloStatic:
`arreglo1[0] = 5 arreglo1[1] = 5 arreglo1[2] = 5`

Valores al entrar a inicArregloAutomatico:
`arreglo2[0] = 1 arreglo2[1] = 2 arreglo2[2] = 3`
 Valores al salir de inicArregloAutomatico:
`arreglo2[0] = 6 arreglo2[1] = 7 arreglo2[2] = 8`

Segunda llamada a cada función:

Valores al entrar en inicArregloStatic:
`arreglo1[0] = 5 arreglo1[1] = 5 arreglo1[2] = 5`
 Valores al salir de inicArregloStatic:
`arreglo1[0] = 10 arreglo1[1] = 10 arreglo1[2] = 10`

Valores al entrar a inicArregloAutomatico:
`arreglo2[0] = 1 arreglo2[1] = 2 arreglo2[2] = 3`
 Valores al salir de inicArregloAutomatico:
`arreglo2[0] = 6 arreglo2[1] = 7 arreglo2[2] = 8`

Figura 7.13 | Inicialización de un arreglo static e inicialización de un arreglo automático. (Parte 2 de 2).

La función `inicArregloStatic` se llama dos veces (líneas 13 y 17). El compilador inicializa el arreglo local `static` con cero la primera vez que se hace una llamada a la función. La función imprime el arreglo, suma 5 a cada elemento e imprime el arreglo de nuevo. La segunda vez que se llama a la función, el arreglo `static` contiene los valores modificados que se almacenan durante la primera llamada a la función. La función `inicArregloAutomatico` también se llama dos veces (líneas 14 y 18). Los elementos del arreglo local automático se inicializan (línea 47) con los valores 1, 2 y 3. La función imprime el arreglo, suma 5 a cada elemento e imprime el arreglo de nuevo. La segunda vez que se llama a la función, los elementos del arreglo se reinicializan con 1, 2 y 3. El arreglo tiene una clase de almacenamiento automático, por lo que se vuelve a crear y se reinicializa durante cada llamada a `inicArregloAutomatico`.



Error común de programación 7.10

Suponiendo que los elementos del arreglo local `static` de una función se inicializan cada vez que se llama a la función, se pueden producir errores lógicos en un programa.

7.5 Paso de arreglos a funciones

Para pasar un argumento tipo arreglo a una función, se debe especificar el nombre del arreglo sin corchetes. Por ejemplo, si el arreglo `temperaturasPorHora` se ha declarado como

```
int temperaturasPorhora[ 24 ];
```

la llamada a la función

```
modificarArreglo( temperaturasPorHora, 24 );
```

pasa el arreglo `temperaturasPorHora` y su tamaño a la función `modificarArreglo`. Al pasar un arreglo a una función, por lo general también se pasa el tamaño del arreglo, de manera que la función pueda procesar el número específico de

elementos en el arreglo. En caso contrario, tendríamos que integrar este conocimiento a la misma función que se llamó o, peor aún, colocar el tamaño del arreglo en una variable global. En la sección 7.11, cuando presentemos la plantilla de clase `vector` de la Biblioteca estándar de C++ para representar un tipo más robusto de arreglo, podremos ver que el tamaño de un `vector` está integrado; cada objeto `vector` “conoce” su propio tamaño, el cual se puede obtener mediante una invocación a la función miembro `size` del objeto `vector`. Por ende, al pasar un *objeto vector* a una función, no tenemos que pasar el tamaño del `vector` como argumento.

C++ pasa los arreglos a las funciones por referencia; las funciones llamadas pueden modificar los valores de los elementos en los arreglos originales de la función que hace las llamadas. El valor del nombre del arreglo es la dirección en la memoria de la computadora del primer elemento del arreglo. Como se pasa la dirección inicial del arreglo, la función llamada conoce exactamente dónde se almacena el arreglo en la memoria. Por lo tanto, cuando la función llamada modifica los elementos del arreglo en su cuerpo, está modificando los elementos actuales del arreglo en sus ubicaciones originales en memoria.



Tip de rendimiento 7.3

El paso de arreglos por referencia tiene sentido por cuestiones de rendimiento. Si los arreglos se pasaran por valor, se pasaría una copia de cada elemento. Para los arreglos extensos que se pasan con frecuencia, esto requeriría mucho tiempo y una cantidad considerable de almacenamiento para las copias de los elementos del arreglo.



Observación de Ingeniería de Software 7.3

Es posible pasar un arreglo por valor (mediante el uso de un simple truco que explicaremos en el capítulo 21); sin embargo, esto se hace muy raras veces.

Aunque se pueden pasar arreglos enteros por referencia, los elementos individuales de un arreglo se pasan por valor, de la misma forma que las variables simples. A esas piezas de datos simples e individuales se les llama **escalares** o **cantidades escalares**. Para pasar un elemento de un arreglo a una función, use el nombre con subíndice del elemento del arreglo como argumento en la llamada a la función. En el capítulo 6 le mostramos cómo pasar escalares (es decir, variables individuales y elementos de arreglos) por referencia mediante las referencias. En el capítulo 8 le mostraremos cómo pasar escalares por referencia mediante apuntadores.

Para que una función reciba un arreglo a través de la llamada a una función, la lista de parámetros de la función debe especificar que ésta espera recibir un arreglo. Por ejemplo, el encabezado para la función `modificarArreglo` se podría escribir así:

```
void modificarArreglo( int b[], int tamanioArreglo )
```

esto indica que `modificarArreglo` espera recibir la dirección de un arreglo de enteros en el parámetro `b`, y el número de elementos del arreglo en el parámetro `tamanioArreglo`. El tamaño del arreglo no se requiere en los corchetes del mismo. Si se incluye, el compilador lo ignora; por lo tanto, se pueden pasar arreglos de cualquier tamaño a la función. C++ pasa arreglos a las funciones por referencia; cuando la función llamada usa el nombre del arreglo `b`, se refiere al arreglo actual en la función que la llamó (es decir, el arreglo `temperaturasPorHora` que vimos al principio de esta sección).

Observe la extraña apariencia del prototipo de función para `modificarArreglo`:

```
void modificarArreglo( int [], int );
```

Este prototipo se podría haber escrito así:

```
void modificarArreglo( int unNombreArreglo[], int unNombreVariable );
```

pero, como vimos en el capítulo 3, los compiladores de C++ ignoran los nombres de las variables en los prototipos. Recuerde que el prototipo indica al compilador el número de argumentos y el tipo de cada argumento (en el orden en el que se espera que aparezcan los argumentos).

El programa de la figura 7.14 demuestra la diferencia entre pasar un arreglo completo y pasar un elemento del arreglo. En las líneas 22 y 23 se imprimen los cinco elementos originales del arreglo entero `a`. En la línea 28 se pasa `a` y su tamaño a la función `modificarArreglo` (líneas 45 a 50), la cual multiplica cada uno de los elementos de `a` por 2 (a través del parámetro `b`). Después, en las líneas 32 y 33 se imprime el arreglo `a` de nuevo en `main`. Como se muestra en la salida, es evidente que los elementos de `a` se modifican mediante `modificarArreglo`. A continuación, en la línea 36 se imprime el valor del escalar `a[3]`, y luego en la línea 38 se pasa el elemento `a[3]` a la función `modificarElemento` (líneas 54 a 58), la cual multiplica su parámetro por 2 e imprime el nuevo valor. Observe que, cuando la línea 39 imprime

de nuevo el valor de `a[3]` en `main`, este valor no se ha modificado, ya que los elementos individuales del arreglo se pasan por valor.

```

1 // Fig. 7.14: fig07_14.cpp
2 // Paso de arreglos y elementos individuales de arreglos a funciones.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 void modificarArreglo( int [], int ); // se ve extraño; arreglo y tamaño
11 void modificarElemento( int ); // recibe el valor del elemento del arreglo
12
13 int main()
14 {
15     const int tamanoArreglo = 5; // tamaño del arreglo a
16     int a[ tamanoArreglo ] = { 0, 1, 2, 3, 4 }; // inicializa el arreglo a
17
18     cout << "Efectos de pasar todo el arreglo por referencia:"
19         << "\n\nLos valores del arreglo original son:\n";
20
21     // imprime los elementos originales del arreglo
22     for ( int i = 0; i < tamanoArreglo; i++ )
23         cout << setw( 3 ) << a[ i ];
24
25     cout << endl;
26
27     // pasa el arreglo a a modificarArreglo por referencia
28     modificarArreglo( a, tamanoArreglo );
29     cout << "Los valores del arreglo modificado son:\n";
30
31     // imprime los elementos modificados del arreglo
32     for ( int j = 0; j < tamanoArreglo; j++ )
33         cout << setw( 3 ) << a[ j ];
34
35     cout << "\n\n\nEfectos de pasar el elemento del arreglo por valor:"
36         << "\n\na[3] antes de modificarElemento: " << a[ 3 ] << endl;
37
38     modificarElemento( a[ 3 ] ); // pasa el elemento a[ 3 ] del arreglo por valor
39     cout << "a[3] despues de modificarElemento: " << a[ 3 ] << endl;
40
41     return 0; // indica que terminó correctamente
42 } // fin de main
43
44 // en la función modificarArreglo, "b" apunta al arreglo "a" original en la memoria
45 void modificarArreglo( int b[], int tamanoDeArreglo )
46 {
47     // multiplica cada elemento del arreglo por 2
48     for ( int k = 0; k < tamanoDeArreglo; k++ )
49         b[ k ] *= 2;
50 } // fin de la función modificarArreglo
51
52 // en la función modificarElemento, "e" es una copia local del
53 // elemento a[ 3 ] del arreglo que se pasa de main
54 void modificarElemento( int e )
55 {
56     // multiplica el parámetro por 2
57     cout << "Valor del elemento en modificarElemento: " << ( e *= 2 ) << endl;
58 } // fin de la función modificarElemento

```

Figura 7.14 | Paso de arreglos y elementos individuales de arreglos a funciones. (Parte I de 2).

Efectos de pasar todo el arreglo por referencia:

Los valores del arreglo original son:

0 1 2 3 4

Los valores del arreglo modificado son:

0 2 4 6 8

Efectos de pasar el elemento del arreglo por valor:

a[3] antes de modificarElemento: 6

Valor del elemento en modificarElemento: 12

a[3] después de modificarElemento: 6

Figura 7.14 | Paso de arreglos y elementos individuales de arreglos a funciones. (Parte 2 de 2).

El programador se puede encontrar con situaciones en las que una función no tenga permitido modificar los elementos de un arreglo. C++ cuenta con el calificador de tipos **const**, el cual se puede usar para evitar que la función llamada modifique los valores de un arreglo mediante código en la función que hace la llamada. Cuando una función especifica un parámetro tipo arreglo al que se antepone el calificador **const**, los elementos del arreglo se hacen constantes en el cuerpo de la función, y cualquier intento de modificar un elemento del arreglo en el cuerpo de la función produce un error de compilación. Esto nos permite evitar la modificación accidental de los elementos del arreglo en el cuerpo de la función.

En la figura 7.15 se demuestra el calificador **const**. La función **tratarDeModificarArreglo** (líneas 21 a 26) se define con el parámetro **const int b[]**, el cual especifica que el arreglo **b** es constante y no se puede modificar. Cada uno de los tres intentos de la función por modificar los elementos del arreglo **b** (líneas 23 a 25) produce un error de compilación. Por ejemplo, el compilador Borland C++ produce el error “*Cannot modify a const object.*” [Nota: el estándar de C++ define a un “objeto” como cualquier “región de almacenamiento”, con lo cual incluye variables o elementos de un arreglo de los tipos de datos fundamentales, así como instancias de clases (que hemos denominado objetos)]. Este mensaje indica que es un error usar un objeto **const** (por ejemplo, **b[0]**) como un *lvalue*; no se puede asignar un nuevo valor a un objeto **const** si se coloca en la parte izquierda de un operador de asignación. Observe que los mensajes de error varían de un compilador a otro (como se muestra en la figura 7.15). En el capítulo 10 hablaremos otra vez sobre el calificador **const**.



Error común de programación 7.11

Olvidar que los arreglos en la función que hace la llamada se pasan por referencia y, por ende, se pueden modificar en las funciones llamadas, puede producir errores lógicos.



Observación de Ingeniería de Software 7.4

*Aplicar el calificador de tipo **const** a un parámetro tipo arreglo en la definición de una función, para evitar que el arreglo original se modifique en el cuerpo de la función, es otro ejemplo del principio de menor privilegio. Las funciones no deben recibir la capacidad de modificar un arreglo, a menos que sea absolutamente necesario.*

```

1 // Fig. 7.15: fig07_15.cpp
2 // Demostración del calificador de tipo const.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void tratarDeModificarArreglo( const int [] ); // prototipo de función
8
9 int main()
10 {
11     int a[] = { 10, 20, 30 };
12

```

Figura 7.15 | Calificador de tipo **const** que se aplica a un parámetro tipo arreglo. (Parte 1 de 2).

```

13     tratarDeModificarArreglo( a );
14     cout << a[ 0 ] << ' ' << a[ 1 ] << ' ' << a[ 2 ] << '\n';
15
16     return 0; // indica que terminó correctamente
17 } // fin de main
18
19 // En la función tratarDeModificarArreglo, "b" no se puede usar
20 // para modificar el arreglo original "a" en main.
21 void tratarDeModificarArreglo( const int b[] )
22 {
23     b[ 0 ] /= 2; // error de compilación
24     b[ 1 ] /= 2; // error de compilación
25     b[ 2 ] /= 2; // error de compilación
26 } // fin de la función tratarDeModificarArreglo

```

Mensaje de error del compilador de líneas de comandos Borland C++:

```

Error E2404 fig07_15.cpp 23: Cannot modify a const object
in function tratarDeModificarArreglo(const int * const)
Error E2404 fig07_15.cpp 24: Cannot modify a const object
in function tratarDeModificarArreglo(const int * const)
Error E2404 fig07_15.cpp 25: Cannot modify a const object
in function tratarDeModificarArreglo(const int * const)

```

Mensaje de error del compilador Microsoft Visual C++ 2005:

```

c:\cpphttp6_ejemplos\cap07\fig07_15\fig07_15.cpp(23) : error C3892: 'b' : you
    cannot assign to a variable that is const
c:\cpphttp6_ejemplos\cap07\fig07_15\fig07_15.cpp(24) : error C3892: 'b' : you
    cannot assign to a variable that is const
c:\cpphttp6_ejemplos\cap07\fig07_15\fig07_15.cpp(25) : error C3892: 'b' : you
    cannot assign to a variable that is const

```

Mensaje de error del compilador GNU C++

```

fig07_15.cpp:23: error: assignment of read-only location
fig07_15.cpp:24: error: assignment of read-only location
fig07_15.cpp:25: error: assignment of read-only location

```

Figura 7.15 | Calificador de tipo `const` que se aplica a un parámetro tipo arreglo. (Parte 2 de 2).

7.6 Ejemplo práctico: la clase LibroCalificaciones que usa un arreglo para almacenar las calificaciones

En esta sección desarrollaremos aún más la clase `LibroCalificaciones`, que presentamos en el capítulo 3 y expandimos en los capítulos 4 a 6. Recuerde que esta clase representa un libro de calificaciones utilizado por un instructor para almacenar y analizar un conjunto de calificaciones de estudiantes. Las versiones anteriores de esta clase procesan un conjunto de calificaciones introducidas por el usuario, pero no mantienen los valores de las calificaciones individuales en los miembros de datos de la clase. Por ende, los cálculos repetidos requieren que el usuario vuelva a introducir las calificaciones. Una manera de resolver este problema sería almacenar cada calificación introducida por el usuario en un miembro de datos individual de la clase. Por ejemplo, podríamos crear los miembros de datos `calificacion1`, `calificacion2`, ..., `calificacion10` en la clase `LibroCalificaciones` para almacenar 10 calificaciones de estudiantes. No obstante, el código para totalizar las calificaciones y determinar el promedio de la clase sería voluminoso. En esta sección resolvemos este problema, almacenando las calificaciones en un arreglo.

Almacenar las calificaciones de los estudiantes en un arreglo en la clase `LibroCalificaciones`

La versión de la clase `LibroCalificaciones` (figuras 7.16 y 7.17) que presentamos aquí utiliza un arreglo de enteros para almacenar las calificaciones de varios estudiantes en un solo examen. Esto elimina la necesidad de introducir varias veces el mismo conjunto de calificaciones. El arreglo `calificaciones` se declara como miembro de datos en

la línea 29 de la figura 7.16; por lo tanto, cada objeto `LibroCalificaciones` mantiene su propio conjunto de calificaciones.

```

1 // Fig. 7.16: LibroCalificaciones.h
2 // Definición de la clase LibroCalificaciones que utiliza un arreglo para almacenar
3 // calificaciones de una prueba. Las funciones miembro se definen en LibroCalificaciones.cpp
4
5 #include <string> // el programa usa la clase string de la Biblioteca estándar de C++
6 using std::string;
7
8 // definición de la clase LibroCalificaciones
9 class LibroCalificaciones
10 {
11 public:
12     // constante -- número de estudiantes que tomaron la prueba
13     const static int estudiantes = 10; // observe los datos públicos
14
15     // el constructor inicializa el nombre del curso y el arreglo de calificaciones
16     LibroCalificaciones( string, const int [] );
17
18     void establecerNombreCurso( string ); // función para establecer el nombre del curso
19     string obtenerNombreCurso(); // función para obtener el nombre del curso
20     void mostrarMensaje(); // muestra un mensaje de bienvenida
21     void procesarCalificaciones(); // realiza varias operaciones con los datos de
22     // calificaciones
23     int obtenerMinimo(); // buscar la calificación mínima para la prueba
24     int obtenerMaximo(); // buscar la calificación máxima para la prueba
25     double obtenerPromedio(); // determina la calificación promedio para la prueba
26     void imprimirGraficoBarras(); // imprime gráfico de barras de la distribución de
27     // calificaciones
28     void imprimirCalificaciones(); // imprime el contenido del arreglo calificaciones
29 private:
30     string nombreCurso; // nombre del curso para este libro de calificaciones
31     int calificaciones[ estudiantes ]; // arreglo de calificaciones de estudiantes
32 }; // fin de la clase LibroCalificaciones

```

Figura 7.16 | Definición de la clase `LibroCalificaciones` usando un arreglo para almacenar calificaciones de una prueba.

```

1 // Fig. 7.17: LibroCalificaciones.cpp
2 // Definiciones de funciones miembro para la clase LibroCalificaciones
3 // que utiliza un arreglo para almacenar las calificaciones de una prueba.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8 using std::fixed;
9
10 #include <iomanip>
11 using std::setprecision;
12 using std::setw;
13
14 #include "LibroCalificaciones.h" // definición de la clase LibroCalificaciones
15
16 // el constructor inicializa nombreCurso y el arreglo calificaciones
17 LibroCalificaciones::LibroCalificaciones( string nombre, const int arregloCalificaciones[] )
18 {
19     establecerNombreCurso( nombre ); // inicializa nombreCurso
20
21     // copia calificaciones de arregloCalificaciones al miembro de datos calificaciones

```

Figura 7.17 | Funciones miembro de la clase `LibroCalificaciones` que manipulan un arreglo de calificaciones. (Parte I de 4).

```

22     for ( int calificacion = 0; calificacion < estudiantes; calificacion++ )
23         calificaciones[ calificacion ] = arregloCalificaciones[ calificacion ];
24 } // fin del constructor de LibroCalificaciones
25
26 // función para establecer el nombre del curso
27 void LibroCalificaciones::establecerNombreCurso( string nombre )
28 {
29     nombreCurso = nombre; // almacena el nombre del curso
30 } // fin de la función establecerNombreCurso
31
32 // función para obtener el nombre del curso
33 string LibroCalificaciones::obtenerNombreCurso()
34 {
35     return nombreCurso;
36 } // fin de la función obtenerNombreCurso
37
38 // muestra un mensaje de bienvenida para el usuario de LibroCalificaciones
39 void LibroCalificaciones::mostrarMensaje()
40 {
41     // esta instrucción llama a obtenerNombreCurso para obtener el
42     // nombre del curso que representa este LibroCalificaciones
43     cout << "Bienvenido al libro de calificaciones para\n" << obtenerNombreCurso() << "!"
44     << endl;
45 } // fin de la función mostrarMensaje
46
47 // realiza varias operaciones con los datos
48 void LibroCalificaciones::procesarCalificaciones()
49 {
50     // imprime el arreglo calificaciones
51     imprimirCalificaciones();
52
53     // llama a la función obtenerPromedio para calcular la calificación promedio
54     cout << "\nEl promedio de la clase es " << setprecision( 2 ) << fixed <<
55     obtenerPromedio() << endl;
56
57     // llama a las funciones obtenerMinimo y obtenerMaximo
58     cout << "La calificación más baja es " << obtenerMinimo() << "\nLa calificación más alta es "
59     << obtenerMaximo() << endl;
60
61     // llama a la función imprimirGraficoBarras para imprimir el gráfico de distribución de
62     // calificaciones
63     imprimirGraficoBarras();
64 } // fin de la función procesarCalificaciones
65
66 // busca la calificación mínima
67 int LibroCalificaciones::obtenerMinimo()
68 {
69     int calificacionInf = 100; // asume que la calificación más baja es 100
70
71     // itera a través del arreglo calificaciones
72     for ( int calificacion = 0; calificacion < estudiantes; calificacion++ )
73     {
74         // si la calificación actual es menor que calificacionInf, la asigna a calificacionInf
75         if ( calificaciones[ calificacion ] < calificacionInf )
76             calificacionInf = calificaciones[ calificacion ]; // nueva calificación más baja
77     } // fin de for
78
79     return calificacionInf; // devuelve la calificación más baja
80 } // fin de la función obtenerMinimo
81
82 // busca la calificación máxima
83 int LibroCalificaciones::obtenerMaximo()

```

Figura 7.17 | Funciones miembro de la clase LibroCalificaciones que manipulan un arreglo de calificaciones. (Parte 2 de 4).

```
83  {
84      int calificacionSup = 0; // asume que la calificación más alta es 0
85
86      // itera a través del arreglo calificaciones
87      for ( int calificacion = 0; calificacion < estudiantes; calificacion++ )
88      {
89          // si la calificación actual es mayor que calificacionSup, la asigna a calificacionSup
90          if ( calificaciones[ calificacion ] > calificacionSup )
91              calificacionSup = calificaciones[ calificacion ]; // nueva calificación más alta
92      } // fin de for
93
94      return calificacionSup; // devuelve la calificación más alta
95  } // fin de la función obtenerMaximo
96
97  // determina la calificación promedio para la prueba
98  double LibroCalificaciones::obtenerPromedio()
99  {
100     int total = 0; // inicializa el total
101
102     // suma las calificaciones en el arreglo
103     for ( int calificacion = 0; calificacion < estudiantes; calificacion++ )
104         total += calificaciones[ calificacion ];
105
106     // devuelve el promedio de las calificaciones
107     return static_cast< double >( total ) / estudiantes;
108 } // fin de la función obtenerPromedio
109
110 // imprime gráfico de barras que muestra la distribución de las calificaciones
111 void LibroCalificaciones::imprimirGraficoBarras()
112 {
113     cout << "\nDistribucion de calificaciones:" << endl;
114
115     // almacena la frecuencia de calificaciones en cada rango de 10 calificaciones
116     const int tamanoFrecuencia = 11;
117     int frequency[ tamanoFrecuencia ] = {}; // inicializa elementos con 0
118
119     // para cada calificación, incrementa la frecuencia apropiada
120     for ( int calificacion = 0; calificacion < estudiantes; calificacion++ )
121         frequency[ calificaciones[ calificacion ] / 10 ]++;
122
123     // para cada frecuencia de calificación, imprime barra en el gráfico
124     for ( int cuenta = 0; cuenta < tamanoFrecuencia; cuenta++ )
125     {
126         // imprime etiquetas de las barras ("0-9:", ..., "90-99:", "100:")
127         if ( cuenta == 0 )
128             cout << " 0-9: ";
129         else if ( cuenta == 10 )
130             cout << " 100: ";
131         else
132             cout << cuenta * 10 << "-" << ( cuenta * 10 ) + 9 << ": ";
133
134         // imprime barra de asteriscos
135         for ( int estrellas = 0; estrellas < frequency[ cuenta ]; estrellas++ )
136             cout << '*';
137
138         cout << endl; // empieza una nueva línea de salida
139     } // fin de for exterior
140 } // fin de la función imprimirGraficoBarras
141
142 // imprime el contenido del arreglo calificaciones
143 void LibroCalificaciones::imprimirCalificaciones()
144 {
```

Figura 7.17 | Funciones miembro de la clase `LibroCalificaciones` que manipulan un arreglo de calificaciones. (Parte 3 de 4).

```

145     cout << "\nLas calificaciones son:\n\n";
146
147     // imprime la calificación de cada estudiante
148     for ( int estudiante = 0; estudiante < estudiantes; estudiante++ )
149         cout << "Estudiante " << setw( 2 ) << estudiante + 1 << ":" << setw( 3 )
150             << calificaciones[ estudiante ] << endl;
151 } // fin de la función imprimirCalificaciones

```

Figura 7.17 | Funciones miembro de la clase `LibroCalificaciones` que manipulan un arreglo de calificaciones. (Parte 4 de 4).

Observe que el tamaño del arreglo en la línea 29 de la figura 7.16 se especifica mediante el miembro de datos `public const static` llamado `estudiantes` (declarado en la línea 13). Este miembro de datos es `public`, de manera que sea accesible para los clientes de la clase. Pronto veremos un ejemplo de un programa cliente que utiliza esta constante. Al declarar a `estudiantes` con el calificador `const`, indicamos que este miembro de datos es constante; su valor no se puede modificar antes de inicializarlo. La palabra clave `static` en esta declaración de variable indica que el miembro de datos es compartido por todos los objetos de la clase; todos los objetos `LibroCalificaciones` almacenan calificaciones para el mismo número de estudiantes. En la sección 3.6 vimos que cuando cada objeto de una clase mantiene su propia copia de un atributo, la variable que representa a ese atributo se conoce como miembro de datos; cada objeto (instancia) de la clase tiene una copia separada de la variable en memoria. Hay variables para las que cada objeto de una clase no tiene una copia separada. Éste es el caso con los miembros de datos `static`, que también se conocen como **variables de clase**. Cuando se crean los objetos de una clase que contiene miembros de datos `static`, todos los objetos comparten una copia de los miembros de datos `static` de la clase. Se puede acceder a un miembro de datos `static` dentro de la definición de la clase y de las definiciones de las funciones miembro al igual que con cualquier otro miembro de datos. Como veremos pronto, también se puede acceder a un miembro de datos `public static` desde el exterior de la clase, aun y cuando no existan objetos de la misma; para ello se utiliza el nombre de la clase, seguido del operador de resolución de ámbito binario (`::`) y el nombre del miembro de datos. En el capítulo 10 aprenderá más acerca de los miembros de datos `static`.

El constructor de la clase (declarado en la línea 16 de la figura 7.16 y definido en las líneas 17 a 24 de la figura 7.17) tiene dos parámetros: el nombre del curso y un arreglo de calificaciones. Cuando un programa crea un objeto `LibroCalificaciones` (por ejemplo, en la línea 13 de `fig07_18.cpp`), el programa pasa un arreglo `int` existente al constructor, el cual copia los valores del arreglo en el miembro de datos `calificaciones` (líneas 22 y 23 de la figura 7.17). Los valores de las calificaciones en el arreglo que se pasa podrían haberse recibido de un usuario, o de un archivo en disco (como veremos en el capítulo 17, Procesamiento de archivos). En nuestro programa de prueba, simplemente inicializamos un arreglo con un conjunto de valores de calificaciones (figura 7.18, líneas 10 y 11). Una vez que las calificaciones se almacenan en el miembro de datos `calificaciones` de la clase `LibroCalificaciones`, todas las funciones miembro de la clase pueden acceder a los elementos de `calificaciones` según sea necesario, para realizar varios cálculos.

```

1 // Fig. 7.18: fig07_18.cpp
2 // Crea un objeto LibroCalificaciones usando un arreglo de calificaciones.
3
4 #include "LibroCalificaciones.h" // definición de la clase LibroCalificaciones
5
6 // la función main empieza la ejecución del programa
7 int main()
8 {
9     // arreglo de calificaciones de estudiantes
10    int arregloCalificaciones[ LibroCalificaciones::estudiantes ] =
11        { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
12
13    LibroCalificaciones miLibroCalificaciones(
14        "CS101 Introducción a la programación en C++", arregloCalificaciones );
15    miLibroCalificaciones.mostrarMensaje();
16    miLibroCalificaciones.procesarCalificaciones();
17    return 0;
18 } // fin de main

```

Figura 7.18 | Crea un objeto `LibroCalificaciones` usando un arreglo de calificaciones, y después invoca a la función miembro `procesarCalificaciones` para analizarlas. (Parte 1 de 2).

```
Bienvenido al libro de calificaciones para  
CS101 Introducción a la programación en C++!
```

Las calificaciones son:

```
Estudiante 1: 87  
Estudiante 2: 68  
Estudiante 3: 94  
Estudiante 4: 100  
Estudiante 5: 83  
Estudiante 6: 78  
Estudiante 7: 85  
Estudiante 8: 91  
Estudiante 9: 76  
Estudiante 10: 87
```

El promedio de la clase es 84.90

La calificación más baja es 68

La calificación más alta es 100

Distribución de calificaciones:

```
0-9:  
10-19:  
20-29:  
30-39:  
40-49:  
50-59:  
60-69: *  
70-79: **  
80-89: ****  
90-99: **  
100: *
```

Figura 7.18 | Crea un objeto `LibroCalificaciones` usando un arreglo de calificaciones, y después invoca a la función miembro `procesarCalificaciones` para analizarlas. (Parte 2 de 2).

La función miembro `procesarCalificaciones` (declarada en la línea 21 de la figura 7.16 y definida en las líneas 48 a 63 de la figura 7.17) contiene una serie de llamadas a funciones miembro que produce un reporte en el que se resumen las calificaciones. La línea 51 llama al método `imprimirCalificaciones` para imprimir el contenido del arreglo `calificaciones`. Las líneas 148 a 150 en la función miembro `imprimirCalificaciones` utilizan una instrucción `for` para imprimir la calificación de cada estudiante. Aunque los subíndices de los arreglos empiezan en 0, lo común es que el profesor enumere a los estudiantes empezando desde 1. Por ende, las líneas 149 y 150 imprimen `estudiante + 1` como el número de estudiante para producir las etiquetas "Estudiante 1:", "Estudiante 2:", y así en lo sucesivo.

A continuación, la función miembro `procesarCalificaciones` llama a la función miembro `obtenerPromedio` (líneas 54 y 55) para obtener el promedio de las calificaciones en el arreglo. La función miembro `obtenerPromedio` (declarada en la línea 24 de la figura 7.16 y definida en las líneas 98 a 108 de la figura 7.17) utiliza una instrucción `for` para totalizar los valores en el arreglo `calificaciones` antes de calcular el promedio. Observe que el cálculo del promedio en la línea 107 utiliza el miembro de datos `const static` llamado `estudiantes` para determinar el número de calificaciones que se van a promediar.

Las líneas 58 y 59 en la función miembro `procesarCalificaciones` llaman a las funciones miembro `obtenerMinimo` y `obtenerMaximo` para determinar las calificaciones más baja y más alta de cualquier estudiante en el examen, en forma respectiva. Vamos a examinar la forma en que la función miembro `obtenerMinimo` encuentra la calificación *más baja*. Como la calificación más alta permitida es 100, empezamos por suponer que 100 es la calificación más baja (línea 68). Despues comparamos cada uno de los elementos en el arreglo con la calificación más baja, buscando valores más pequeños. En las líneas 71 a 76 de la función miembro `obtenerMinimo` se itera a través del arreglo, y en las líneas 74 y 75 se compara cada calificación con `calificacionInf`. Si una calificación es menor que `calificacionInf`, a `calificacionInf` se le asigna esa calificación. Cuando se ejecuta la línea 78, `calificacionInf` contiene la calificación más baja en el arreglo. La función miembro `obtenerMaximo` (líneas 82 a 95) funciona de manera similar a la función miembro `obtenerMinimo`.

Por último, la línea 62 en la función miembro `procesarCalificaciones` llama a la función miembro `imprimirGraficoBarras` para imprimir un gráfico de distribución de los datos de las calificaciones, usando una técnica similar a la de la figura 7.9. En ese ejemplo, calculamos en forma manual el número de calificaciones en cada categoría (es decir, 0-9, 10-19, ..., 90-99 y 100), para lo cual simplemente analizamos un conjunto de calificaciones. En este ejemplo, en las líneas 120 y 121 se utiliza una técnica similar a la de las figuras 7.10 y 7.11 para calcular la frecuencia de calificaciones en cada categoría. En la línea 117 se declara y crea el arreglo `frecuencia` de 11 valores `int` para almacenar la frecuencia de calificaciones en cada categoría. Para cada `calificacion` en el arreglo `calificaciones`, en las líneas 120 y 121 se incrementa el elemento apropiado del arreglo `frecuencia`. Para determinar qué elemento se debe incrementar, en la línea 121 se divide la `calificacion` actual entre 10, usando la división entera. Por ejemplo, si `calificacion` es 85, en la línea 121 se incrementa `frecuencia[8]` para actualizar la cuenta de calificaciones en el rango de 80 a 89. Después, en las líneas 123 a 139 se imprime el gráfico de barras (vea la figura 7.18) con base en los valores en el arreglo `frecuencia`. Al igual que en las líneas 29 y 30 de la figura 7.9, en las líneas 135 y 136 de la figura 7.17 se utiliza un valor en el arreglo `frecuencia` para determinar el número de asteriscos a mostrar en cada barra.

Prueba de la clase LibroCalificaciones

El programa de la figura 7.18 crea un objeto de la clase `LibroCalificaciones` (figuras 7.16 y 7.17) mediante el uso del arreglo `int` `arregloCalificaciones` (que se declara y se inicializa en las líneas 10 y 11). Observe que usamos el operador de resolución de ámbito binario (`::`) en la expresión “`LibroCalificaciones::estudiantes`” (línea 10) para acceder a la constante `static` llamada `estudiantes`, de la clase `LibroCalificaciones`. Utilizamos aquí esta constante para crear un arreglo que sea del mismo tamaño que el arreglo `calificaciones` que se almacena como miembro de datos en la clase `LibroCalificaciones`. En las líneas 13 y 14 se pasa el nombre de un curso y `arregloCalificaciones` al constructor de `LibroCalificaciones`. En la línea 15 se imprime un mensaje de bienvenida, y en la línea 16 se invoca la función miembro `procesarCalificaciones` del objeto `LibroCalificaciones`. La salida muestra el resumen de las 10 calificaciones en `miLibroCalificaciones`.

7.7 Búsqueda de datos en arreglos mediante la búsqueda lineal

Es común que un programador trabaje con grandes cantidades de datos almacenados en arreglos. Tal vez sea necesario determinar si un arreglo contiene un valor que concuerde con cierto **valor clave**. Al proceso de buscar un elemento específico de un arreglo se le llama **búsqueda**. En esta sección hablaremos sobre la búsqueda lineal simple. El ejercicio 7.33 al final de este capítulo le pedirá que implemente una versión recursiva de la búsqueda lineal. En el capítulo 19, Búsqueda y ordenamiento, presentaremos la búsqueda binaria, que es más efectiva pero a la vez más eficiente.

Búsqueda lineal

La **búsqueda lineal** (figura 7.19, líneas 37 a 44) compara cada elemento del arreglo con una **clave de búsqueda** (línea 40). Como el arreglo no está en ningún orden específico, existe la misma probabilidad de que se encuentre el valor tanto en el primer elemento como en el último. Por lo tanto, en promedio el programa debe comparar la clave de búsqueda con la mitad de los elementos del arreglo. Para determinar que un valor no se encuentra en el arreglo, el programa debe comparar la clave de búsqueda con cada elemento del arreglo.

El método de búsqueda lineal funciona bien para pequeños arreglos o para arreglos desordenados (es decir, arreglos cuyos elementos no se encuentren en un orden específico). Sin embargo, para arreglos extensos, la búsqueda lineal es ineficiente. Si el arreglo está ordenado (por ejemplo, si sus elementos están en orden ascendente), puede usar la técnica de búsqueda binaria de alta velocidad que aprenderá en el capítulo 19, Búsqueda y ordenamiento.

```

1 // Fig. 7.19: fig07_19.cpp
2 // Búsqueda lineal de un arreglo.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 int busquedaLineal( const int [], int, int ); // prototipo
9
10 int main()

```

Figura 7.19 | Búsqueda lineal de un arreglo. (Parte I de 2).

```

11  {
12      const int tamanoArreglo = 100; // tamaño del arreglo a
13      int a[ tamanoArreglo ]; // crea el arreglo a
14      int claveBusqueda; // valor a localizar en el arreglo a
15
16      for ( int i = 0; i < tamanoArreglo; i++ )
17          a[ i ] = 2 * i; // crea ciertos datos
18
19      cout << "Introduzca la clave de busqueda entera: ";
20      cin >> claveBusqueda;
21
22      // trata de localizar la claveBusqueda en el arreglo a
23      int elemento = busquedaLineal( a, claveBusqueda, tamanoArreglo );
24
25      // muestra los resultados
26      if ( elemento != -1 )
27          cout << "Se encontro el valor en el elemento " << elemento << endl;
28      else
29          cout << "No se encontro el valor" << endl;
30
31      return 0; // indica que terminó correctamente
32 } // fin de main
33
34 // compara la clave con cada elemento del arreglo hasta que lo
35 // encuentra, o hasta llegar al final del arreglo; devuelve el subíndice
36 // del elemento si se encontró la clave, o -1 si no se encontró
37 int busquedaLineal( const int arreglo[], int clave, int tamanoDelArreglo )
38 {
39     for ( int j = 0; j < tamanoDelArreglo; j++ )
40         if ( arreglo[ j ] == clave ) // si se encontró,
41             return j; // devuelve la ubicación de la clave
42
43     return -1; // no se encontró la clave
44 } // fin de la función busquedaLineal

```

Introduzca la clave de busqueda entera: 36
Se encontro el valor en el elemento 18

Introduzca la clave de busqueda entera: 37
No se encontro el valor

Figura 7.19 | Búsqueda lineal de un arreglo. (Parte 2 de 2).

7.8 Ordenamiento de arreglos mediante el ordenamiento por inserción

El **ordenamiento** de datos (es decir, colocar los datos en cierto orden específico, como ascendente o descendente) es una de las aplicaciones computacionales más importantes. Un banco ordena todos los cheques por número de cuenta, de manera que pueda preparar estados de cuenta bancarios individuales al final de cada mes. Las compañías telefónicas ordenan sus directorios telefónicos por apellido paterno y, dentro de ese orden, por primer nombre para facilitar la búsqueda de los números telefónicos. Casi todas las empresas deben ordenar ciertos datos y, en muchos casos, son cantidades masivas. El ordenamiento de datos es un problema intrigante que ha atraído algunos de los esfuerzos de investigación más intensos en el campo de las ciencias computacionales. En este capítulo veremos un esquema de ordenamiento simple. En el capítulo 19, Búsqueda y ordenamiento, investigaremos esquemas más complejos que producen un rendimiento superior, y presentaremos la notación Big O para caracterizar lo duro que debe trabajar cada esquema para realizar su tarea.

Tip de rendimiento 7.4



Algunas veces los algoritmos simples tienen un rendimiento pobre. Su virtud es que son fáciles de escribir, de probar y depurar. En ocasiones se necesitan los algoritmos más complejos para obtener un rendimiento óptimo.

Ordenamiento por inserción

El programa de la figura 7.20 ordena los valores del arreglo datos de 10 elementos en forma ascendente. La técnica que utilizamos se llama **ordenamiento por inserción**; éste es un algoritmo de ordenamiento simple pero inefficiente. La primera iteración de este algoritmo toma el segundo elemento y, si es menor que el primero, lo intercambia (es decir, el

```

1 // Fig. 7.20: fig07_20.cpp
2 // Este programa ordena los valores de un arreglo en forma ascendente.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 int main()
11 {
12     const int tamanoArreglo = 10; // tamaño del arreglo a
13     int datos[ tamanoArreglo ] = { 34, 56, 4, 10, 77, 51, 93, 30, 5, 52 };
14     int insertar; // variable temporal para guardar el elemento a insertar
15
16     cout << "Arreglo desordenado:\n";
17
18     // imprime el arreglo original
19     for ( int i = 0; i < tamanoArreglo; i++ )
20         cout << setw( 4 ) << datos[ i ];
21
22     // ordenamiento por inserción
23     // itera a través de los elementos del arreglo
24     for ( int siguiente = 1; siguiente < tamanoArreglo; siguiente++ )
25     {
26         insertar = datos[ siguiente ]; // almacena el valor en el elemento actual
27
28         int moverElemento = siguiente; // inicializa la ubicación para colocar el elemento
29
30         // busca la ubicación en la que va a colocar el elemento actual
31         while ( ( moverElemento > 0 ) && ( datos[ moverElemento - 1 ] > insertar ) )
32         {
33             // desplaza el elemento una posición a la derecha
34             datos[ moverElemento ] = datos[ moverElemento - 1 ];
35             moverElemento--;
36         } // fin de while
37
38         datos[ moverElemento ] = insertar; // coloca el elemento insertado en el arreglo
39     } // fin de for
40
41     cout << "\nArreglo ordenado:\n";
42
43     // imprime el arreglo ordenado
44     for ( int i = 0; i < tamanoArreglo; i++ )
45         cout << setw( 4 ) << datos[ i ];
46
47     cout << endl;
48     return 0; // indica que terminó correctamente
49 } // fin de main

```

```

Arreglo desordenado:
 34 56 4 10 77 51 93 30 5 52
Arreglo ordenado:
 4 5 10 30 34 51 52 56 77 93

```

Figura 7.20 | Ordenamiento de un arreglo mediante el ordenamiento por inserción.

programa *inserta* el segundo elemento enfrente del primer elemento). La segunda iteración analiza el tercer elemento y lo inserta en la posición correcta respecto a los primeros dos elementos, de manera que los tres elementos estén en orden. En la i -ésima iteración de este algoritmo, los primeros i elementos en el arreglo original estarán ordenados.

En la línea 13 de la figura 7.20 se declara e inicializa el arreglo **datos** con los siguientes valores:

34	56	4	10	77	51	93	30	5	52
----	----	---	----	----	----	----	----	---	----

El programa analiza primero los elementos **datos[0]** y **datos[1]**, cuyos valores son 34 y 56, respectivamente. Estos dos elementos ya se encuentran en orden, por lo que el programa continúa; si estuvieran desordenados, el programa los intercambiaría.

En la segunda iteración, el programa analiza el valor de **datos[2]**, (4). Este valor es menor que 56, por lo que el programa almacena el 4 en una variable temporal y desplaza el 56 un elemento a la derecha. Después el programa comprueba y determina que 4 es menor que 34, por lo que desplaza el 34 un elemento a la derecha. Ahora el programa ha llegado al inicio del arreglo, por lo que coloca el 4 en **datos[0]**. A continuación se muestra el estado actual del arreglo:

4	34	56	10	77	51	93	30	5	52
---	----	----	----	----	----	----	----	---	----

En la tercera iteración, el programa almacena el valor de **datos[3]**, (10), en una variable temporal. Después el programa compara 10 con 56 y desplaza el 56 un elemento a la derecha, ya que es mayor que 10. Luego, el programa compara 10 con 34, y desplaza el 34 un elemento a la derecha. Cuando el programa compara 10 con 4, observa que 10 es mayor que 4 y coloca el 10 en **datos[1]**. Ahora el arreglo tiene el siguiente orden:

4	10	34	56	77	51	93	30	5	52
---	----	----	----	----	----	----	----	---	----

Mediante el uso de este algoritmo, en la i -ésima iteración los primeros i elementos del arreglo original están ordenados. Tal vez no se encuentren en sus posiciones finales, ya que puede haber valores más pequeños en las últimas posiciones del arreglo.

El ordenamiento se realiza mediante la instrucción **for** en las líneas 24 a 39, que itera a través de los elementos del arreglo. En cada iteración, en la línea 26 se almacena temporalmente en la variable **insertar** (declarada en la línea 14) el valor del elemento que se insertará en la porción ordenada del arreglo. En la línea 28 se declara e inicializa la variable **moverElemento**, la cual mantiene la cuenta de la posición en la que se debe insertar el elemento. En las líneas 31 a 36 se itera para localizar la posición correcta en la que se debe insertar el elemento. El ciclo termina cuando el programa llega a la parte frontal del arreglo, o cuando llega a un elemento que sea menor que el valor a insertar. En la línea 34 se desplaza un elemento a la derecha, y en la línea 35 se decrementa la posición en la que se va a insertar el siguiente elemento. Una vez que termina el ciclo **while**, en la línea 38 se inserta el elemento en su lugar. Cuando termina la instrucción **for** de las líneas 24 a 39, los elementos del arreglo están ordenados.

La principal virtud del ordenamiento por inserción es su facilidad para programarse, sin embargo, se ejecuta con lentitud. Esto se vuelve aparente cuando se ordenan arreglos extensos. En los ejercicios investigaremos ciertos algoritmos alternativos para ordenar un arreglo. En el capítulo 19 investigaremos los procesos de ordenamiento y búsqueda con más detalle.

7.9 Arreglos multidimensionales

Los arreglos con dos o más dimensiones se conocen como **arreglos multidimensionales**. Los arreglos de dos dimensiones se utilizan con frecuencia para representar **tablas de valores**, las cuales consisten en información ordenada en **filas** y **columnas**. Para identificar un elemento específico de una tabla, debemos especificar dos subíndices. Por convención, el primero identifica la fila del elemento y el segundo su columna. Los arreglos que requieren dos subíndices para identificar un elemento específico se llaman **arreglos bidimensionales** o **arreglos 2-D**. Los arreglos multidimensionales pueden tener más de dos dimensiones (por ejemplo, los subíndices). La figura 7.21 ilustra un arreglo bidimensional **a**, que contiene tres filas y cuatro columnas (es decir, un arreglo de tres por cuatro). En general, a un arreglo con m filas y n columnas se le llama **arreglo de m por n** .

Cada elemento en el arreglo **a** se identifica en la figura 7.21 mediante una expresión de acceso a un arreglo de la forma **a[i][j]**; donde **a** es el nombre del arreglo, **i** y **j** son los subíndices que identifican en forma única a cada elemento en el arreglo **a**. Observe que los nombres de los elementos en la fila 0 tienen todos un primer subíndice de 0, y los nombres de los elementos en la columna 3 tienen un segundo subíndice de 3.

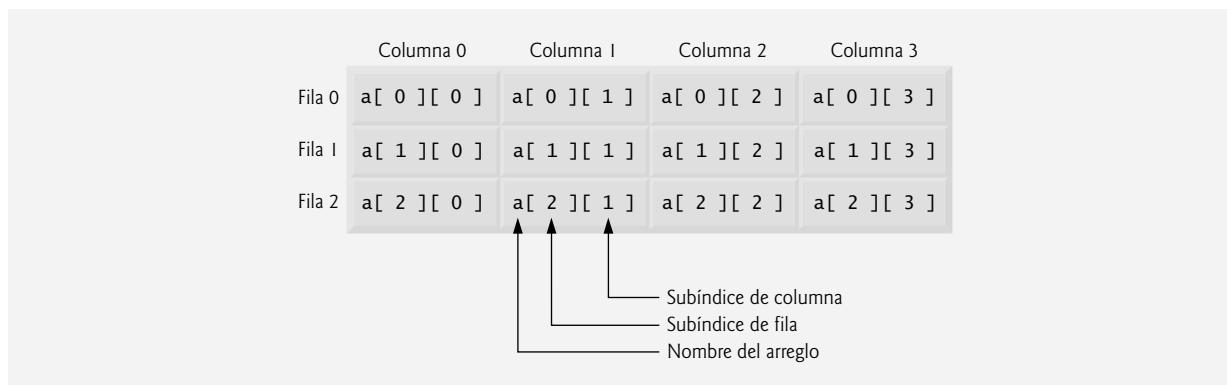


Figura 7.21 | Arreglo bidimensional con tres filas y cuatro columnas.



Error común de programación 7.12

Es un error hacer referencia al elemento `a[x][y]` de un arreglo bidimensional en forma incorrecta como `a[x][y]`. En realidad, `a[x, y]` se trata como `a[y]`, ya que C++ evalúa la expresión `x, y` (que contiene un operador coma) simplemente como `y` (la última de las expresiones separadas por coma).

Un arreglo multidimensional se puede inicializar en su declaración, en forma muy parecida a un arreglo unidimensional. Por ejemplo, un arreglo bidimensional `b` con los valores 1 y 2 en los elementos de su fila 0, y los valores 3 y 4 en los elementos de su fila 1, se podría declarar e inicializar de la siguiente manera:

```
int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
```

Los valores se agrupan por fila entre llaves. Así, 1 y 2 inicializan a `b[0][0]` y `b[0][1]`, respectivamente; 3 y 4 inicializan a `b[1][0]` y `b[1][1]`, respectivamente. Si no hay suficientes inicializadores para cierta fila, el resto de los elementos de esa fila se inicializa con 0. Así, la declaración

```
int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };
```

inicializa `b[0][0]` con 1, `b[0][1]` con 0, `b[1][0]` con 3 y `b[1][1]` con 4.

En la figura 7.22 se demuestra cómo inicializar arreglos bidimensionales en las declaraciones. En las líneas 11 y 13 se declaran tres arreglos, cada uno con dos filas y tres columnas.

```

1 // Fig. 7.22: fig07_22.cpp
2 // Inicialización de arreglos multidimensionales.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void imprimirArreglo( const int [][][ 3 ] ); // prototipo
8
9 int main()
10 {
11     int arreglo1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
12     int arreglo2[ 2 ][ 3 ] = { { 1, 2, 3, 4, 5 } };
13     int arreglo3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };
14
15     cout << "Los valores en arreglo1 por fila son:" << endl;
16     imprimirArreglo( arreglo1 );
17
18     cout << "\nLos valores en arreglo2 por fila son:" << endl;
19     imprimirArreglo( arreglo2 );
20
21     cout << "\nLos valores en arreglo3 por fila son:" << endl;

```

Figura 7.22 | Inicialización de arreglos multidimensionales. (Parte 1 de 2).

```

22     imprimirArreglo( arreglo3 );
23     return 0; // indica que terminó correctamente
24 } // fin de main
25
26 // imprime arreglo con dos filas y tres columnas
27 void imprimirArreglo( const int a[][ 3 ] )
28 {
29     // itera a través de las filas del arreglo
30     for ( int i = 0; i < 2; i++ )
31     {
32         // itera a través de las columnas de la fila actual
33         for ( int j = 0; j < 3; j++ )
34             cout << a[ i ][ j ] << ' ';
35
36         cout << endl; // empieza nueva línea de salida
37     } // fin de for exterior
38 } // fin de la función imprimirArreglo

```

Los valores en arreglo1 por fila son:

1 2 3
4 5 6

Los valores en arreglo2 por fila son:

1 2 3
4 5 0

Los valores en arreglo3 por fila son:

1 2 0
4 0 0

Figura 7.22 | Inicialización de arreglos multidimensionales. (Parte 2 de 2).

La declaración de `arreglo1` (línea 11) proporciona seis inicializadores en dos sublistas. La primera sublista inicializa la fila 0 del arreglo con los valores 1, 2 y 3; y la segunda sublista inicializa la fila 1 del arreglo con los valores 4, 5 y 6. Si las llaves alrededor de cada sublista se eliminan de la lista inicializadora de `arreglo1`, el compilador inicializa los elementos de la fila 0 seguidos de los elementos de la fila 1, lo cual produce el mismo resultado.

La declaración de `arreglo2` (línea 12) proporciona sólo cinco inicializadores. Estos inicializadores se asignan a la fila 0, después a la fila 1. Cualquier elemento que no tenga un inicializador explícito se inicializa con cero, por lo que `arreglo2[1][2]` se inicializa con cero.

La declaración de `arreglo3` (línea 13) proporciona tres inicializadores en dos sublistas. La sublista para la fila 0 inicializa de manera explícita los primeros dos elementos de la fila 0 con 1 y 2; el tercer elemento se inicializa de manera implícita con cero. La sublista para la fila 1 inicializa de manera explícita el primer elemento con 4, e inicializa de manera implícita los últimos dos elementos con cero.

El programa llama a la función `imprimirArreglo` para imprimir cada uno de los elementos del arreglo. Observe que la definición de la función (líneas 27 a 38) especifica el parámetro `const int a[][3]`. Cuando una función recibe un arreglo unidimensional como argumento, los corchetes del arreglo están vacíos en la lista de parámetros de la función. El tamaño de la primera dimensión (es decir, el número de filas) de un arreglo bidimensional no se requiere tampoco, pero sí se requieren todos los tamaños de las dimensiones subsiguientes. El compilador usa estos tamaños para determinar las ubicaciones en memoria de los elementos en los arreglos multidimensionales. Todos los elementos del arreglo se almacenan en forma consecutiva en memoria, sin importar el número de dimensiones. En un arreglo bidimensional, la fila 0 se almacena en memoria junto a la fila 1. En un arreglo bidimensional, cada fila es un arreglo unidimensional. Para localizar un elemento en una fila específica, la función debe saber exactamente cuántos elementos hay en cada fila, para que pueda omitir el número apropiado de ubicaciones en memoria a la hora de acceder a los datos en el arreglo. Por ende, al acceder al elemento `a[1][2]`, la función sabe cómo omitir los tres elementos de la fila 0 en memoria para llegar a la fila 1. Después, la función accede al elemento 2 de esa fila.

Muchas manipulaciones comunes en los arreglos utilizan instrucciones de repetición `for`. Por ejemplo, la siguiente instrucción `for` establece todos los elementos en la fila 2 del arreglo `a` de la figura 7.21 con cero:

```

for ( columna = 0; columna < 4; columna++ )
    a[ 2 ][ columna ] = 0;

```

La instrucción `for` sólo varía el segundo subíndice (es decir, el subíndice de la columna). La instrucción `for` anterior es equivalente a las siguientes instrucciones de asignación:

```
a[ 2 ][ 0 ] = 0;
a[ 2 ][ 1 ] = 0;
a[ 2 ][ 2 ] = 0;
a[ 2 ][ 3 ] = 0;
```

La siguiente instrucción `for` anidada determina el total de todos los elementos en el arreglo `a`:

```
total = 0;
for ( fila = 0; fila < 3; fila++ )
    for ( columna = 0; columna < 4; columna++ )
        total += a[ fila ][ columna ];
```

La instrucción `for` calcula el total de los elementos del arreglo, una fila a la vez. La instrucción `for` exterior empieza estableciendo `fila` (es decir, el subíndice de la fila) en 0, de manera que la instrucción `for` interior pueda calcular el total de los elementos de la fila 0. Después, la instrucción `for` exterior incrementa `fila` a 1, de manera que se pueda obtener el total de los elementos de la fila 1. Luego, la instrucción `for` exterior incrementa `fila` a 2, de manera que pueda obtenerse el total de los elementos de la fila 2. Cuando termina la instrucción `for` anidada, `total` contiene la suma de todos los elementos del arreglo.

7.10 Ejemplo práctico: la clase LibroCalificaciones que usa un arreglo bidimensional

En la sección 7.6 presentamos la clase `LibroCalificaciones` (figuras 7.16 y 7.17), la cual utilizó un arreglo unidimensional para almacenar las calificaciones de los estudiantes en un solo examen. En la mayoría de los cursos, los estudiantes presentan varios exámenes. Es probable que los profesores quieran analizar las calificaciones a lo largo de todo el curso, tanto para un solo estudiante como para la clase en general.

Cómo almacenar las calificaciones de los estudiantes en un arreglo bidimensional en la clase LibroCalificaciones

Las figuras 7.23 y 7.24 contienen una versión de la clase `LibroCalificaciones` que utiliza un arreglo bidimensional llamado `calificaciones`, para almacenar las calificaciones de un número de estudiantes en varios exámenes. Cada fila del arreglo representa las calificaciones de un solo estudiante durante todo el curso, y cada columna representa las calificaciones para la clase completa en uno de los exámenes que presentaron los estudiantes durante el curso. Un programa cliente como `fig07_25.cpp` pasa el arreglo como argumento para el constructor de `LibroCalificaciones`. En este ejemplo, utilizamos un arreglo de diez por tres que contiene diez calificaciones de los estudiantes en tres exámenes.

Cinco funciones miembro (declaradas en las líneas 23 a 27 de la figura 7.23) realizan manipulaciones de arreglos para procesar las calificaciones. Cada una de estas funciones miembro es similar a su contraparte en la versión anterior de la clase `LibroCalificaciones` con un arreglo unidimensional (figuras 7.16 y 7.17). La función miembro `obtenerMinimo` (definida en las líneas 65 a 82 de la figura 7.24) determina la calificación más baja de cualquier estudiante durante el semestre. La función miembro `obtenerMaximo` (definida en las líneas 85 a 102 de la figura 7.24) determina la calificación más alta de cualquier estudiante durante el semestre. La función miembro `obtenerPromedio` (líneas 105 a 115 de la figura 7.24) determina el promedio semestral de un estudiante específico. La función miembro `imprimirGraficoBarras` (líneas 118 a 149 de la figura 7.24) imprime un gráfico de barras de la distribución de todas las calificaciones de los estudiantes durante el semestre. La función miembro `imprimirCalificaciones` (líneas 152 a 177 de la figura 7.24) imprime el arreglo bidimensional en formato tabular, junto con el promedio semestral de cada estudiante.

```
1 // Fig. 7.23: LibroCalificaciones.h
2 // Definición de la clase LibroCalificaciones que utiliza un
3 // arreglo bidimensional para almacenar calificaciones de una prueba.
4 // Las funciones miembro se definen en LibroCalificaciones.cpp
5 #include <string> // el programa usa la clase string de la Biblioteca estándar de C++
6 using std::string;
```

Figura 7.23 | Definición de la clase `LibroCalificaciones` con un arreglo bidimensional para almacenar calificaciones. (Parte 1 de 2).

```

7 // definición de la clase LibroCalificaciones
8 class LibroCalificaciones
9 {
10 public:
11     // constantes
12     const static int estudiantes = 10; // número de estudiantes
13     const static int pruebas = 3; // número de pruebas
14
15     // el constructor inicializa el nombre del curso y el arreglo de calificaciones
16     LibroCalificaciones( string, const int [][] pruebas );
17
18     void establecerNombreCurso( string ); // función para establecer el nombre del curso
19     string obtenerNombreCurso(); // función para obtener el nombre del curso
20     void mostrarMensaje(); // muestra un mensaje de bienvenida
21     void procesarCalificaciones(); // realiza varias operaciones en los datos de las
22     calificaciones
23     int obtenerMinimo(); // encuentra el valor mínimo de calificación
24     int obtenerMaximo(); // encuentra el valor máximo de calificación
25     double obtenerPromedio( const int [], const int ); // obtiene el promedio del estudiante
26     void imprimirGraficoBarras(); // imprime gráfico de barras de la distribución de
27     calificaciones
28     void imprimirCalificaciones(); // imprime el contenido del arreglo calificaciones
29 private:
30     string nombreCurso; // nombre del curso para este libro de calificaciones
31     int calificaciones[ estudiantes ][ pruebas ]; // arreglo bidimensional de calificaciones
32 }; // fin de la clase LibroCalificaciones

```

Figura 7.23 | Definición de la clase `LibroCalificaciones` con un arreglo bidimensional para almacenar calificaciones. (Parte 2 de 2).

```

1 // Fig. 7.24: LibroCalificaciones.cpp
2 // Definiciones de las funciones miembro para la clase LibroCalificaciones
3 // que usa un arreglo bidimensional para almacenar calificaciones.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8 using std::fixed;
9
10 #include <iomanip> // manipuladores de flujo parametrizados
11 using std::setprecision; // establece la precisión numérica de salida
12 using std::setw; // establece la anchura del campo
13
14 // incluye la definición de la clase LibroCalificaciones de LibroCalificaciones.h
15 #include "LibroCalificaciones.h"
16
17 // constructor con dos argumentos que inicializa nombreCurso y el arreglo calificaciones
18 LibroCalificaciones::LibroCalificaciones( string nombre, const int arregloCalificaciones[][]
19     pruebas )
20 {
21     establecerNombreCurso( nombre ); // inicializa nombreCurso
22     // copia calificaciones de arregloCalificaciones a calificaciones
23     for ( int estudiante = 0; estudiante < estudiantes; estudiante++ )
24         for ( int prueba = 0; prueba < pruebas; prueba++ )
25             calificaciones[ estudiante ][ prueba ] = arregloCalificaciones[ estudiante ][ prueba ];
26 }
27 } // fin del constructor de LibroCalificaciones con dos argumentos

```

Figura 7.24 | Definiciones de las funciones miembro de `LibroCalificaciones`, que manipula un arreglo bidimensional de calificaciones. (Parte 1 de 4).

```

28 // función para establecer el nombre del curso
29 void LibroCalificaciones::establecerNombreCurso( string nombre )
30 {
31     nombreCurso = nombre; // almacena el nombre del curso
32 } // fin de la función establecerNombreCurso
33
34 // función para obtener el nombre del curso
35 string LibroCalificaciones::obtenerNombreCurso()
36 {
37     return nombreCurso;
38 } // fin de la función obtenerNombreCurso
39
40 // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
41 void LibroCalificaciones::mostrarMensaje()
42 {
43     // esta instrucción llama a obtenerNombreCurso para obtener el
44     // nombre del curso que representa este LibroCalificaciones
45     cout << "Bienvenido al libro de calificaciones para\n" << obtenerNombreCurso() << "!"
46         << endl;
47 } // fin de la función mostrarMensaje
48
49 // realiza varias operaciones con los datos
50 void LibroCalificaciones::procesarCalificaciones()
51 {
52     // imprime el arreglo calificaciones
53     imprimirCalificaciones();
54
55     // llama a las funciones obtenerMinimo y obtenerMaximo
56     cout << "\nLa calificación más baja en el libro de calificaciones es " << obtenerMinimo()
57         << "\nLa calificación más alta en el libro de calificaciones es " << obtenerMaximo() <<
58             endl;
59
60     // imprime gráfico de distribución de todas las calificaciones en todas las pruebas
61     imprimirGraficoBarras();
62 } // fin de la función procesarCalificaciones
63
64 // encuentra la calificación más baja en todo el libro de calificaciones
65 int LibroCalificaciones::obtenerMinimo()
66 {
67     int calificacionInf = 100; // asume que la calificación más baja es 100
68
69     // itera a través de las filas del arreglo calificaciones
70     for ( int estudiante = 0; estudiante < estudiantes; estudiante++ )
71     {
72         // itera a través de las columnas de la fila actual
73         for ( int prueba = 0; prueba < pruebas; prueba++ )
74         {
75             // si la calificación actual es menor que calificacionInf, la asigna a
76             // calificacionInf
77             if ( calificaciones[ estudiante ][ prueba ] < calificacionInf )
78                 calificacionInf = calificaciones[ estudiante ][ prueba ]; // nueva calificación
79                     mas baja
80         } // fin de for interior
81     } // fin de for exterior
82
83     return calificacionInf; // devuelve la calificación más baja
84 } // fin de la función obtenerMinimo
85
86 // busca la calificación máxima en todo el libro de calificaciones
87 int LibroCalificaciones::obtenerMaximo()

```

Figura 7.24 | Definiciones de las funciones miembro de `LibroCalificaciones`, que manipula un arreglo bidimensional de calificaciones. (Parte 2 de 4).

```
86  {
87      int calificacionSup = 0; // asume que la calificación más alta es 0
88
89      // itera a través de las filas del arreglo calificaciones
90      for ( int estudiante = 0; estudiante < estudiantes; estudiante++ )
91      {
92          // itera a través de las columnas de la fila actual
93          for ( int prueba = 0; prueba < pruebas; prueba++ )
94          {
95              // si la calificación actual es mayor que calificacionSup, la asigna a calificacionSup
96              if ( calificaciones[ estudiante ][ prueba ] > calificacionSup )
97                  calificacionSup = calificaciones[ estudiante ][ prueba ]; // nueva calificación
98                  más alta
99          } // fin de for interior
100     } // fin de for exterior
101
102     return calificacionSup; // devuelve la calificación más alta
103 } // fin de la función obtenerMaximo
104
105 // determina la calificación promedio para el conjunto específico de calificaciones
106 double LibroCalificaciones::obtenerPromedio( const int conjuntoDeCalificaciones[], const int
107 calificaciones )
108 {
109     int total = 0; // initialize total
110
111     // suma las calificaciones en el arreglo
112     for ( int calificacion = 0; calificacion < calificaciones; calificacion++ )
113         total += conjuntoDeCalificaciones[ calificacion ];
114
115     // devuelve el promedio de las calificaciones
116     return static_cast< double >( total ) / calificaciones;
117 } // fin de la función obtenerPromedio
118
119 // imprime gráfico de barras que muestra la distribución de las calificaciones
120 void LibroCalificaciones::imprimirGraficoBarras()
121 {
122     cout << "\nDistribucion general de calificaciones:" << endl;
123
124     // almacena la frecuencia de las calificaciones en cada rango de 10 calificaciones
125     const int tamanoFrecuencia = 11;
126     int frecuencia[ tamanoFrecuencia ] = {};// inicializa elementos con 0
127
128     // para cada calificación, incrementa la frecuencia apropiada
129     for ( int estudiante = 0; estudiante < estudiantes; estudiante++ )
130         for ( int prueba = 0; prueba < pruebas; prueba++ )
131             ++frecuencia[ calificaciones[ estudiante ][ prueba ] / 10 ];
132
133     // para cada frecuencia de calificaciones, imprime la barra en el gráfico
134     for ( int cuenta = 0; cuenta < tamanoFrecuencia; cuenta++ )
135     {
136         // imprime las etiquetas de las barras ("0-9:", ..., "90-99:", "100:")
137         if ( cuenta == 0 )
138             cout << " 0-9: ";
139         else if ( cuenta == 10 )
140             cout << " 100: ";
141         else
142             cout << cuenta * 10 << "-" << ( cuenta * 10 ) + 9 << ": ";
143
144         // imprime barra de asteriscos
145         for ( int stars = 0; stars < frecuencia[ cuenta ]; stars++ )
```

Figura 7.24 | Definiciones de las funciones miembro de LibroCalificaciones, que manipula un arreglo bidimensional de calificaciones. (Parte 3 de 4).

```

145         cout << '*';
146
147     cout << endl; // empieza una nueva línea de salida
148 } // fin de for exterior
149 } // fin de la función imprimirGraficoBarras
150
151 // imprime el contenido del arreglo calificaciones
152 void LibroCalificaciones::imprimirCalificaciones()
153 {
154     cout << "\nLas calificaciones son:\n\n";
155     cout << " "; // alinea los encabezados de las columnas
156
157 // crea un encabezado de columna para cada una de las pruebas
158 for ( int prueba = 0; prueba < pruebas; prueba++ )
159     cout << "Prueba " << prueba + 1 << " ";
160
161 cout << "Promedio" << endl; // encabezado de la columna de promedio de estudiantes
162
163 // crea filas/columnas de texto que representan el arreglo calificaciones
164 for ( int estudiante = 0; estudiante < estudiantes; estudiante++ )
165 {
166     cout << "Estudiante " << setw( 2 ) << estudiante + 1;
167
168     // imprime las calificaciones del estudiante
169     for ( int prueba = 0; prueba < pruebas; prueba++ )
170         cout << setw( 8 ) << calificaciones[ estudiante ][ prueba ];
171
172     // llama a la función miembro obtenerPromedio para calcular el promedio del estudiante;
173     // pasa la fila de calificaciones y el valor de pruebas como argumentos
174     double promedio = obtenerPromedio( calificaciones[ estudiante ], pruebas );
175     cout << setw( 9 ) << setprecision( 2 ) << fixed << promedio << endl;
176 } // fin de for exterior
177 } // fin de la función imprimirCalificaciones

```

Figura 7.24 | Definiciones de las funciones miembro de `LibroCalificaciones`, que manipula un arreglo bidimensional de calificaciones. (Parte 4 de 4).

Cada una de las funciones miembro `obtenerMinimo`, `obtenerMaximo`, `imprimirGraficoBarras` e `imprimirCalificaciones` iteran a través del arreglo `calificaciones` mediante el uso de instrucciones `for` anidadas. Por ejemplo, considere la instrucción `for` anidada en la función miembro `obtenerMinimo` (líneas 70 a 79). La instrucción `for` exterior empieza por establecer `estudiante` (es decir, el subíndice de fila) en 0, de manera que los elementos de la fila 0 pueden compararse con la variable `calificacionInf` en el cuerpo de la instrucción `for` interior. La instrucción `for` interior itera a través de las calificaciones de una fila específica, y compara cada calificación con `calificacionInf`. Si una calificación es menor que `calificacionInf`, a `calificacionInf` se le asigna esa calificación. Después, la instrucción `for` exterior incrementa el subíndice de fila en 1. Los elementos de la fila 1 se comparan con la variable `calificacionInf`. A continuación, la instrucción `for` exterior incrementa el subíndice de fila a 2, y los elementos de la fila 2 se comparan con la variable `calificacionInf`. Esto se repite hasta que se hayan recorrido todas las filas de `calificaciones`. Cuando se completa la ejecución de la instrucción anidada, `calificacionInf` contiene la calificación más baja de todo el arreglo bidimensional. La función miembro `obtenerMaximo` funciona de manera similar a la función miembro `obtenerMinimo`.

La función miembro `imprimirGraficoBarras` en la figura 7.24 es casi idéntica a la de la figura 7.17. Sin embargo, para imprimir la distribución de calificaciones en general durante todo un semestre, la función miembro utiliza una instrucción `for` anidada (líneas 127 a 130) para crear el arreglo unidimensional `frecuencia`, con base en todas las calificaciones en el arreglo bidimensional. El resto del código en cada una de las dos funciones miembro `imprimirGraficoBarras` que muestran el gráfico es idéntico.

La función miembro `imprimirCalificaciones` (líneas 152 a 177) también utiliza instrucciones `for` anidadas para imprimir valores del arreglo `calificaciones`, además del promedio semestral de cada estudiante. La salida en la figura 7.25 muestra el resultado, el cual se asemeja al formato tabular del libro de calificaciones real de un profesor. Las líneas 158 a 159 imprimen los encabezados de columna para cada prueba. Aquí utilizamos una instrucción `for` controlada por contador, para poder identificar cada prueba con un número. De manera similar, la instrucción `for` en las líneas 164 a 176

```

1 // Fig. 7.25: fig07_25.cpp
2 // Crea un objeto LibroCalificaciones usando un arreglo bidimensional de calificaciones.
3
4 #include "LibroCalificaciones.h" // definición de la clase LibroCalificaciones
5
6 // la función main empieza la ejecución del programa
7 int main()
8 {
9     // arreglo bidimensional de calificaciones de estudiantes
10    int arregloCalificaciones[ LibroCalificaciones::estudiantes ][ LibroCalificaciones::pruebas ] =
11        { { 87, 96, 70 },
12          { 68, 87, 90 },
13          { 94, 100, 90 },
14          { 100, 81, 82 },
15          { 83, 65, 85 },
16          { 78, 87, 65 },
17          { 85, 75, 83 },
18          { 91, 94, 100 },
19          { 76, 72, 84 },
20          { 87, 93, 73 } };
21
22    LibroCalificaciones miLibroCalificaciones(
23        "CS101 Introducción a la programación en C++", arregloCalificaciones );
24    miLibroCalificaciones.mostrarMensaje();
25    miLibroCalificaciones.procesarCalificaciones();
26    return 0; // indica que terminó correctamente
27 } // fin de main

```

Bienvenido al libro de calificaciones para
CS101 Introducción a la programación en C++!

Las calificaciones son:

	Prueba 1	Prueba 2	Prueba 3	Promedio
Estudiante 1	87	96	70	84.33
Estudiante 2	68	87	90	81.67
Estudiante 3	94	100	90	94.67
Estudiante 4	100	81	82	87.67
Estudiante 5	83	65	85	77.67
Estudiante 6	78	87	65	76.67
Estudiante 7	85	75	83	81.00
Estudiante 8	91	94	100	95.00
Estudiante 9	76	72	84	77.33
Estudiante 10	87	93	73	84.33

La calificación más baja en el libro de calificaciones es 65
La calificación más alta en el libro de calificaciones es 100

Distribución general de calificaciones:

0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: ***
70-79: *****
80-89: *****
90-99: *****
100: ***

Figura 7.25 | Crea un objeto LibroCalificaciones mediante el uso de un arreglo bidimensional de calificaciones, y después invoca a la función miembro procesarCalificaciones para analizarlas.

imprime primero una etiqueta de fila mediante el uso de una variable contador para identificar a cada estudiante (línea 166). Aunque los subíndices de los arreglos empiezan en 0, observe que en las líneas 159 y 166 se imprimen `prueba + 1` y `estudiante + 1` en forma respectiva, para producir números de prueba y estudiante que empiecen en 1 (vea la figura 7.25). La instrucción `for` interior en las líneas 169 y 170 utiliza la variable contador `estudiante` de la instrucción `for` exterior para iterar a través de una fila específica del arreglo `calificaciones`, e imprime la calificación de la prueba de cada estudiante. Por último, en la línea 174 se obtiene el promedio semestral de cada estudiante, para lo cual se pasa la fila actual de `calificaciones` (es decir, `calificaciones[estudiante]`) a la función miembro `obtenerPromedio`.

La función miembro `obtenerPromedio` (líneas 105 a 115) recibe dos argumentos: un arreglo unidimensional de resultados de la prueba para un estudiante específico, y el número de resultados de la prueba en el arreglo. Cuando la línea 174 llama a `obtenerPromedio`, el primer argumento es `calificaciones[estudiante]`, el cual especifica que debe pasarse una fila específica del arreglo bidimensional `calificaciones` a `obtenerPromedio`. Por ejemplo, con base en el arreglo creado en la figura 7.25, el argumento `calificaciones[1]` representa los tres valores (un arreglo unidimensional de calificaciones) almacenados en la fila 1 del arreglo bidimensional `calificaciones`. Un arreglo bidimensional se puede considerar como un arreglo cuyos elementos son arreglos unidimensionales. La función miembro `obtenerPromedio` calcula la suma de los elementos del arreglo, divide el total entre el número de resultados de la prueba y devuelve el resultado de punto flotante como un valor `double` (línea 114).

Prueba de la clase LibroCalificaciones

El programa de la figura 7.25 crea un objeto de la clase `LibroCalificaciones` (figuras 7.23 y 7.24) mediante el uso del arreglo bidimensional de valores `int` llamado `arregloCalif` (el cual se declara y se inicializa en las líneas 10 a 20). Observe que en la línea 10 se accede a las constantes `static` llamadas `estudiantes` y `pruebas` de la clase `LibroCalificaciones` para indicar el tamaño de cada dimensión del arreglo `arregloCalificaciones`. En las líneas 22 y 23 se pasan el nombre de un curso y `arregloCalificaciones` al constructor de `LibroCalificaciones`. Después, en las líneas 24 y 25 se invocan las funciones miembro `mostrarMensaje` y `procesarCalificaciones` de `miLibroCalificaciones`, para mostrar un mensaje de bienvenida y obtener un informe que sintetice las calificaciones de los estudiantes para el semestre, respectivamente.

7.11 Introducción a la plantilla de clase vector de la Biblioteca estándar de C++

Ahora introduciremos la plantilla de clase `vector` de la Biblioteca estándar de C++, la cual representa un tipo más robusto de arreglo, el cual incluye muchas herramientas adicionales. Como veremos en capítulos posteriores, los arreglos basados en apuntadores estilo C (es decir, el tipo de arreglos presentados hasta ahora) tienen un enorme potencial de errores. Por ejemplo, como dijimos antes, un programa puede “salirse” fácilmente de cualquier extremo de un arreglo, ya que C++ no comprueba si las subscripciones caen fuera del rango de un arreglo. Dos arreglos no pueden compararse con sentido mediante operadores de igualdad o relacionales. Como veremos en el capítulo 8, las variables apuntador (conocidas más comúnmente como apuntadores) contienen direcciones de memoria como valores. Los nombres de arreglos son simplemente apuntadores hacia la ubicación en la que los arreglos empiezan en la memoria y, desde luego que dos arreglos siempre estarán en ubicaciones distintas de memoria. Cuando se pasa un arreglo a una función de propósito general diseñada para manejar arreglos de cualquier tamaño, el tamaño del arreglo debe pasarse como argumento adicional. Lo que es más, un arreglo no se puede asignar a otro con el (los) operador(es) de asignación; los nombres de los arreglos son apuntadores `const` y, como veremos en el capítulo 8, un apuntador constante no se puede utilizar del lado izquierdo de un operador de asignación. Éstas y otras herramientas parecen sin duda destinadas para tratar con arreglos, pero C++ no proporciona dichas herramientas. Sin embargo, la Biblioteca estándar de C++ proporciona la plantilla de clase `vector` para permitir a los programadores crear una alternativa a los arreglos más poderosa y menos propensa a errores. En el capítulo 11, Sobrecarga de operadores: objetos String y Array, presentaremos los medios para implementar dichas herramientas de los arreglos, como las que proporciona `vector`. El lector aprenderá a personalizar los operadores, para usarlos con sus propias clases (una técnica conocida como sobrecarga de operadores).

La plantilla de la clase `vector` está disponible para cualquier diseñador de aplicaciones en C++. Las notaciones que usa el ejemplo con `vector` podrían ser desconocidas para el lector, ya que estos objetos utilizan notación de plantilla. Recuerde que en la sección 6.18 vimos las plantillas de función. En el capítulo 14, hablaremos sobre las plantillas de clase. Por ahora, basta con poder usar la plantilla de clase `vector`, imitando la sintaxis en el ejemplo que mostramos en esta sección. El lector profundizará en su comprensión cuando estudiemos las plantillas de clase en el capítulo 14. El capítulo 22 presenta la plantilla de clase `vector` (y otras clases contenedoras más de C++) con detalle.

El programa de la figura 7.26 demuestra las herramientas proporcionadas por la plantilla de clase `vector` de la Biblioteca estándar de C++ que no están disponibles para los arreglos basados en apuntadores estilo C. La plantilla de

clase estándar `vector` proporciona muchas de las mismas características de la clase `Array` que construimos en el capítulo 11. Sobrecarga de operadores: objetos `String` y `Array`. La plantilla de clase estándar `vector` se define en el encabezado `<vector>` (línea 11) y pertenece al espacio de nombres `std` (línea 12). El capítulo 22 habla sobre la funcionalidad completa de la plantilla de clase estándar `vector`.

En las líneas 19 y 20 se crean dos objetos `vector` que almacenan valores de tipo `int`: `enteros1` contiene siete elementos, y `enteros2` contiene 10 elementos. De manera predeterminada, todos los elementos de cada objeto `vector` se establecen en 0. Observe que los objetos `vector` se definen para almacenar cualquier tipo de datos, al sustituir `int` en `vector< int >` con el tipo de datos apropiado. Esta notación, que especifica el tipo almacenado en el `vector`, es similar a la notación de plantilla que presentamos en la sección 6.18 con las plantillas de función. De nuevo, en el capítulo 14 hablaremos sobre esta sintaxis con detalle.

```
1 // Fig. 7.26: fig07_26.cpp
2 // Demostración de la clase vector de la Biblioteca estándar de C++.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 #include <vector>
12 using std::vector;
13
14 void imprimirVector( const vector< int > & ); // muestra el vector
15 void recibirVector( vector< int > & ); // introduce los valores en el vector
16
17 int main()
18 {
19     vector< int > enteros1( 7 ); // vector de 7 elementos< int >
20     vector< int > enteros2( 10 ); // vector de 10 elementos< int >
21
22     // imprime el tamaño y el contenido de enteros1
23     cout << "El tamano del vector enteros1 es " << enteros1.size()
24         << "\nvector despues de la inicializacion:" << endl;
25     imprimirVector( enteros1 );
26
27     // imprime el tamaño y el contenido de enteros2
28     cout << "\nEl tamano del vector enteros2 es " << enteros2.size()
29         << "\nvector despues de la inicializacion:" << endl;
30     imprimirVector( enteros2 );
31
32     // recibe e imprime enteros1 y enteros2
33     cout << "\nEscriba 17 enteros:" << endl;
34     recibirVector( enteros1 );
35     recibirVector( enteros2 );
36
37     cout << "\nDespues de la entrada, los vectores contienen:\n"
38         << "enteros1:" << endl;
39     imprimirVector( enteros1 );
40     cout << "enteros2:" << endl;
41     imprimirVector( enteros2 );
42
43     // usa el operador de desigualdad (!=) con objetos vector
44     cout << "\nEvaluacion: enteros1 != enteros2" << endl;
45
46     if ( enteros1 != enteros2 )
47         cout << "enteros1 y enteros2 no son iguales" << endl;
48 }
```

Figura 7.26 | Plantilla de clase `vector` de la Biblioteca estándar de C++. (Parte 1 de 3).

```

49 // crea el vector enteros3 usando enteros1 como un
50 // inicializador; imprime el tamaño y el contenido
51 vector< int > enteros3( enteros1 ); // constructor de copia
52
53 cout << "\nEl tamaño del vector enteros3 es " << enteros3.size()
54     << "\nvector después de la inicialización:" << endl;
55 imprimirVector( enteros3 );
56
57 // usa el operador de asignación (=) con objetos vector
58 cout << "\nAsignación de enteros2 a enteros1:" << endl;
59 enteros1 = enteros2; // asigna enteros2 a enteros1
60
61 cout << "enteros1:" << endl;
62 imprimirVector( enteros1 );
63 cout << "enteros2:" << endl;
64 imprimirVector( enteros2 );
65
66 // usa el operador de igualdad (==) con objetos vector
67 cout << "\nEvaluación: enteros1 == enteros2" << endl;
68
69 if ( enteros1 == enteros2 )
70     cout << "enteros1 y enteros2 son iguales" << endl;
71
72 // usa corchetes para crear rvalue
73 cout << "\nenteros1[5] es " << enteros1[ 5 ];
74
75 // usa corchetes para crear lvalue
76 cout << "\n\nAsignación de 1000 a enteros1[5]" << endl;
77 enteros1[ 5 ] = 1000;
78 cout << "enteros1:" << endl;
79 imprimirVector( enteros1 );
80
81 // intenta usar subíndice fuera de rango
82 cout << "\nIntento de asignar 1000 a enteros1.at( 15 )" << endl;
83 enteros1.at( 15 ) = 1000; // ERROR: fuera de rango
84 return 0;
85 } // fin de main
86
87 // imprime el contenido del vector
88 void imprimirVector( const vector< int > &arreglo )
89 {
90     size_t i; // declara la variable de control
91
92     for ( i = 0; i < arreglo.size(); i++ )
93     {
94         cout << setw( 12 ) << arreglo[ i ];
95
96         if ( ( i + 1 ) % 4 == 0 ) // 4 números por fila de resultados
97             cout << endl;
98     } // fin de for
99
100    if ( i % 4 != 0 )
101        cout << endl;
102 } // fin de la función imprimirVector
103
104 // recibe el contenido del vector
105 void recibirVector( vector< int > &arreglo )
106 {
107     for ( size_t i = 0; i < arreglo.size(); i++ )
108         cin >> arreglo[ i ];
109 } // fin de la función recibirVector

```

Figura 7.26 | Plantilla de clase `vector` de la Biblioteca estándar de C++. (Parte 2 de 3).

```

El tamaño del vector enteros1 es 7
vector después de la inicialización:
    0      0      0      0
    0      0      0

El tamaño del vector enteros2 es 10
vector después de la inicialización:
    0      0      0      0
    0      0      0      0
    0      0

Escriba 17 enteros:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

Después de la entrada, los vectores contienen:
enteros1:
    1      2      3      4
    5      6      7

enteros2:
    8      9      10     11
    12     13     14     15
    16     17

Evaluación: enteros1 != enteros2
enteros1 y enteros2 no son iguales

El tamaño del vector enteros3 es 7
vector después de la inicialización:
    1      2      3      4
    5      6      7

Asignación de enteros2 a enteros1:
enteros1:
    8      9      10     11
    12     13     14     15
    16     17

enteros2:
    8      9      10     11
    12     13     14     15
    16     17

Evaluación: enteros1 == enteros2
enteros1 y enteros2 son iguales

enteros1[5] es 13

Asignación de 1000 a enteros1[5]
enteros1:
    8      9      10     11
    12     1000    14     15
    16     17

Intento de asignar 1000 a enteros1.at( 15 )
terminación anormal del programa

```

Figura 7.26 | Plantilla de clase `vector` de la Biblioteca estándar de C++. (Parte 3 de 3).

En la línea 23 se utiliza la función miembro `size` de `vector` para obtener el tamaño (es decir, el número de elementos) de `enteros1`. En la línea 25 se pasa `enteros1` a la función `imprimirVector` (líneas 88 a 102), la cual usa corchetes (`[]`, línea 94) para obtener el valor en cada elemento del `vector` para imprimirla. Observe la semejanza de esta notación a la que se utilizó para acceder al valor del elemento de un arreglo. En las líneas 28 y 30 se realizan las mismas tareas para `enteros2`.

La función miembro `size` de la plantilla de clase `vector` devuelve el número de elementos en un `vector` como un valor de tipo `size_t` (el cual representa al tipo `unsigned int` en muchos sistemas). Como resultado, en la línea 90 se declara la variable de control `i` como del tipo `size_t`, también. En algunos compiladores, declarar a `i` como `int` hace que el compilador genere un mensaje de advertencia, ya que la condición de continuación de ciclo (línea 92) compararía a un valor `signed` (por ejemplo, `int i`) y un valor `unsigned` (es decir, un valor de tipo `size_t` devuelto por la función `size`).

En las líneas 34 y 35 se pasan los objetos `enteros1` y `enteros2` a la función `recibirVector` (líneas 105 a 109) para que el usuario introduzca los valores de los elementos de cada `vector`. La función utiliza corchetes (`[]`) para formar `lvalues` que se utilizan para almacenar los valores de entrada en cada elemento `vector`.

En la línea 46 se demuestra que los objetos `vector` se pueden comparar entre sí mediante el operador `!=`. Si el contenido de dos objetos `vector` no es igual, el operador devuelve `true`; en caso contrario devuelve `false`.

La plantilla de clase `vector` de la Biblioteca estándar de C++ nos permite crear un nuevo objeto `vector` que se inicializa con el contenido de un `vector` existente. En la línea 51 se crea un objeto `vector` llamado `enteros3` y se inicializa con una copia de `enteros1`. Esto invoca al constructor de copia de `vector` para realizar la operación de copia. En el capítulo 11 aprenderá con detalle acerca de los constructores de copia. En las líneas 53 a 55 se imprime el tamaño y el contenido de `enteros3` para demostrar que se inicializó en forma correcta.

En la línea 59 se asigna `enteros2` a `enteros1`, lo cual demuestra que el operador de asignación (`=`) se puede usar con objetos `vector`. En las líneas 61 a 64 se imprime el contenido de ambos objetos para mostrar que ahora contienen valores idénticos. Después, en la línea 69 se compara `enteros2` con `enteros1` mediante el operador de igualdad (`==`) para determinar si el contenido de los dos objetos es el mismo después de la asignación en la línea 59 (lo cual es cierto).

En las líneas 73 y 77 se demuestra que un programa puede usar corchetes (`[]`) para obtener un elemento de `vector` como `rvalue` y `lvalue`, respectivamente. En la sección 5.9 vimos que un `rvalue` no se puede modificar, pero un `lvalue` sí. Como es el caso con los arreglos basados en apuntador estilo C, C++ no realiza comprobación de límites cuando se accede a elementos de un objeto `vector` mediante el uso de corchetes. Por lo tanto, el programador debe asegurar que las operaciones en las que se utilizan los corchetes (`[]`) no traten accidentalmente de manipular elementos fuera de los límites del `vector`. Sin embargo, la plantilla de clase estándar `vector` cuenta con la capacidad de comprobar los límites en su función miembro `at`, la cual “lanza una excepción” (vea el capítulo 16, Manejo de excepciones) si su argumento es un subíndice inválido. De manera predeterminada, esto hace que un programa de C++ termine su ejecución. Si el subíndice es válido, la función `at` devuelve el elemento en la ubicación especificada como un `lvalue` modificable, o como un `lvalue` no modificable, dependiendo del contexto (no `const` o `const`) en el que aparezca la llamada. En la línea 83 se demuestra una llamada a una función `at` con un subíndice inválido. La salida resultante varía según el compilador que se utilice.

En esta sección demostramos el uso de la plantilla de clase `vector` de la Biblioteca estándar de C++, una clase robusta y reutilizable que puede reemplazar a los arreglos basados en apuntador estilo C. En el capítulo 11 veremos que `vector` logra muchas de sus capacidades al “sobrecargar” los operadores integrados de C++, y aprenderá a personalizar los operadores para usarlos en sus propias clases de manera similar. Por ejemplo, crearemos una clase `Array` que, al igual que la plantilla de clase `vector`, se mejora a partir de las capacidades básicas de los arreglos. Nuestra clase `Array` también proporciona características adicionales, como la habilidad de recibir e imprimir arreglos completos mediante los operadores `>>` y `<<`, respectivamente.

7.12 (Opcional) Ejemplo práctico de Ingeniería de Software: colaboración entre los objetos en el sistema ATM

En esta sección nos concentraremos en las colaboraciones (interacciones) entre los objetos en nuestro sistema ATM. Cuando dos objetos se comunican entre sí para realizar una tarea, se dice que **colaboran** (para ello, un objeto invoca a las operaciones del otro). Una **colaboración** consiste en que un objeto de una clase envía un **mensaje** a un objeto de otra clase. En `++`, los mensajes se envían mediante llamadas a funciones miembro.

En la sección 6.22 determinamos muchas de las operaciones de las clases en nuestro sistema. En esta sección, nos concentraremos en los mensajes que invocan a esas operaciones. Para identificar las colaboraciones en el sistema, regresaremos a la especificación de requerimientos de la sección 2.8. Recuerde que este documento especifica el rango de actividades que ocurren durante una sesión con el ATM (por ejemplo, autenticar a un usuario, realizar transacciones). Los pasos utilizados para describir cómo debe realizar el sistema cada una de estas tareas son nuestra primera indicación de las colaboraciones en nuestro sistema. A medida que avancemos por esta sección y las siguientes secciones del Ejemplo práctico de Ingeniería de Software que quedan en el libro, tal vez descubriremos colaboraciones adicionales.

Identificar las colaboraciones en un sistema

Para identificar las colaboraciones en el sistema, leeremos con cuidado las secciones de la especificación de requerimientos que especifican lo que debe hacer el ATM para autenticar un usuario, y para realizar cada tipo de transacción. Para cada acción o paso descrito, decidimos qué objetos en nuestro sistema deben interactuar para lograr el resultado deseado. Identificamos un objeto como el emisor (el objeto que envía el mensaje) y otro como el receptor (el objeto que ofrece la operación a los clientes de la clase). Después seleccionamos una de las operaciones del objeto receptor (identificadas en la sección 6.22) que el objeto emisor debe invocar para producir el comportamiento apropiado. Por ejemplo, el ATM muestra un mensaje de bienvenida cuando está inactivo. Sabemos que un objeto de la clase `Pantalla` muestra un men-

saje al usuario a través de su operación `mostrarMensaje`. Por ende, decidimos que el sistema puede mostrar un mensaje de bienvenida si empleamos una colaboración entre el ATM y la Pantalla, donde el ATM envía un mensaje `mostrarMensaje` a la Pantalla mediante la invocación de la operación `mostrarMensaje` de la clase Pantalla. [Nota: para evitar repetir la frase “un objeto de la clase...”, nos referiremos a cada objeto sólo utilizando su nombre de clase, precedido por un artículo (“un”, “una”, “el” o “la”); por ejemplo, “el ATM” hace referencia a un objeto de la clase ATM.]

La figura 7.27 lista las colaboraciones que pueden derivarse de la especificación de requerimientos. Para cada objeto emisor, listamos las colaboraciones en el orden en el que se describen en la especificación de requerimientos. Listamos cada colaboración en la que se involucra un emisor único, un mensaje y un receptor sólo una vez, aun cuando la colaboración puede ocurrir varias veces durante una sesión con el ATM. Por ejemplo, la primera fila en la figura 7.27 indica que el objeto ATM colabora con el objeto Pantalla cada vez que el ATM necesita mostrar un mensaje al usuario.

Vamos a considerar las colaboraciones en la figura 7.27. Antes de permitir que un usuario realice transacciones, el ATM debe pedirle que introduzca un número de cuenta y que después introduzca un NIP. Para realizar cada una de estas tareas, envía un mensaje `mostrarMensaje` a la Pantalla. Ambas acciones se refieren a la misma colaboración entre el ATM y la Pantalla, que ya se listan en la figura 7.27. El ATM obtiene la entrada en respuesta a un indicador, mediante el envío de un mensaje `obtenerEntrada` al Teclado. A continuación, el ATM debe determinar si el número de cuenta especificado por el usuario y el NIP concuerdan con los de una cuenta en la base de datos. Para ello envía un mensaje `autenticarUsuario` a la BaseDatosBanco. Recuerde que `BaseDatosBanco` no puede autenticar a un usuario en forma directa; sólo la Cuenta del usuario (es decir, la Cuenta que contiene el número de cuenta especificado por el usuario) puede acceder al NIP registrado del usuario para autenticarlo. Por lo tanto, la figura 7.27 lista una colaboración en la que `BaseDatosBanco` envía un mensaje `validarNIP` a una Cuenta.

Una vez autenticado el usuario, el ATM muestra el menú principal enviando una serie de mensajes `mostrarMensaje` a la Pantalla y obtiene la entrada que contiene una selección de menú; para ello envía un mensaje `obtenerEntrada` al Teclado. Ya hemos tomado en cuenta estas colaboraciones. Una vez que el usuario selecciona un tipo de transacción a realizar, el ATM ejecuta la transacción enviando un mensaje `ejecutar` a un objeto de la clase de transacción apropiada (es decir, un objeto `SolicitudSaldo`, `Retiro` o `Deposito`). Por ejemplo, si el usuario elige realizar una solicitud de saldo, el ATM envía un mensaje `ejecutar` a un objeto `SolicitudSaldo`.

Un objeto de la clase...	envía el mensaje...	a un objeto de la clase...
ATM	<code>mostrarMensaje</code> <code>obtenerEntrada</code> <code>autenticarUsuario</code> <code>ejecutar</code> <code>ejecutar</code> <code>ejecutar</code>	Pantalla Teclado BaseDatosBanco SolicitudSaldo Retiro Deposito
SolicitudSaldo	<code>obtenerSaldoDisponible</code> <code>obtenerSaldoTotal</code> <code>mostrarMensaje</code>	BaseDatosBanco BaseDatosBanco Pantalla
Retiro	<code>mostrarMensaje</code> <code>obtenerEntrada</code> <code>obtenerSaldoDisponible</code> <code>haySuficienteEfectivoDisponible</code> <code>cargar</code> <code>dispensarEfectivo</code>	Pantalla Teclado BaseDatosBanco DispensadorEfectivo BaseDatosBanco DispensadorEfectivo
Deposito	<code>mostrarMensaje</code> <code>obtenerEntrada</code> <code>seRecibioSobreDeposito</code> <code>abonar</code>	Pantalla Teclado RanuraDeposito BaseDatosBanco
BaseDatosBanco	<code>validarNIP</code> <code>obtenerSaldoDisponible</code> <code>obtenerSaldoTotal</code> <code>cargar</code> <code>abonar</code>	Cuenta Cuenta Cuenta Cuenta Cuenta

Figura 7.27 | Colaboraciones en el sistema ATM.

Un análisis más a fondo de la especificación de requerimientos revela las colaboraciones involucradas en la ejecución de cada tipo de transacción. Un objeto `SolicitudSaldo` extrae la cantidad de dinero disponible en la cuenta del usuario al enviar un mensaje `obtenerSaldoDisponible` al objeto `BaseDatosBanco`, el cual responde enviando un mensaje `obtenerSaldoDisponible` a la Cuenta del usuario. De manera similar, el objeto `SolicitudSaldo` extrae la cantidad de dinero depositado al enviar un mensaje `obtenerSaldoTotal` al objeto `BaseDatosBanco`, el cual envía el mismo mensaje a la Cuenta del usuario. Para mostrar en pantalla ambas cantidades del saldo del usuario al mismo tiempo, el objeto `SolicitudSaldo` envía un mensaje `mostrarMensaje` a la Pantalla.

Un objeto `Retiro` envía una serie de mensajes `mostrarMensaje` a la Pantalla para mostrar un menú de montos estándar de retiro (es decir, \$20, \$40, \$60, \$100, \$200). El objeto `Retiro` envía un mensaje `obtenerEntrada` al Teclado para obtener la selección del menú elegida por el usuario, y después determina si el monto de retiro solicitado es menor o igual al saldo de la cuenta del usuario. Para obtener el monto de dinero disponible en la cuenta del usuario, el objeto `Retiro` envía un mensaje `obtenerSaldoDisponible` al objeto `BaseDatosBanco`. Después el objeto `Retiro` evalúa si el dispensador contiene suficiente efectivo, enviando un mensaje `haySuficienteEfectivoDisponible` al Dispensador-Efectivo. Un objeto `Retiro` envía un mensaje `cargar` al objeto `BaseDatosBanco` para reducir el saldo de la cuenta del usuario. El objeto `BaseDatosBanco` envía a su vez el mismo mensaje al objeto Cuenta apropiado. Recuerde que al hacer un cargo a una Cuenta se reduce tanto el `saldoTotal` como el `saldoDisponible`. Para dispensar la cantidad solicitada de efectivo, el objeto `Retiro` envía un mensaje `dispensarEfectivo` al objeto Dispensador-Efectivo. Por último, el objeto `Retiro` envía un mensaje `mostrarMensaje` a la Pantalla, instruyendo al usuario para que tome el efectivo.

Para responder a un mensaje `ejecutar`, un objeto `Depósito` primero envía un mensaje `mostrarMensaje` a la Pantalla para pedir al usuario que introduzca un monto a depositar. El objeto `Depósito` envía un mensaje `obtenerEntrada` al Teclado para obtener la entrada del usuario. Después, el objeto `Depósito` envía un mensaje `mostrarMensaje` a la Pantalla para pedir al usuario que inserte un sobre de depósito. Para determinar si la ranura de depósito recibió un sobre de depósito entrante, el objeto `Depósito` envía un mensaje `seRecibioSobreDepósito` al objeto `RanuraDepósito`. El objeto `Depósito` actualiza la cuenta del usuario enviando un mensaje `abonar` al objeto `BaseDatosBanco`, el cual a su vez envía un mensaje `abonar` al objeto Cuenta del usuario. Recuerde que al abonar a una Cuenta se incrementa el `saldoTotal`, pero no el `saldoDisponible`.

Diagramas de interacción

Ahora que identificamos un conjunto de posibles colaboraciones entre los objetos en nuestro sistema ATM, vamos a modelar en forma gráfica estas interacciones mediante el uso de UML. UML cuenta con varios tipos de **diagramas de interacción**, que para modelar el comportamiento de un sistema modelan la forma en que los objetos interactúan entre sí. El **diagrama de comunicación** enfatiza cuáles objetos participan en las colaboraciones. [Nota: los diagramas de comunicación se llamaban **diagramas de colaboración** en versiones anteriores de UML.] Al igual que el diagrama de comunicación, el **diagrama de secuencia** muestra las colaboraciones entre los objetos, pero enfatiza *cuándo* se deben enviar los mensajes entre los objetos *a través del tiempo*.

Diagramas de comunicación

La figura 7.28 muestra un diagrama de comunicación que modela la forma en que el ATM ejecuta una `SolicitudSaldo`. Los objetos se modelan en UML como rectángulos que contienen nombres de la forma `nombreObjeto : NombreClase`. En este ejemplo, que involucra sólo a un objeto de cada tipo, descartamos el nombre del objeto y enlistamos sólo un signo de dos puntos (:) seguido del nombre de la clase. [Nota: se recomienda especificar el nombre de cada objeto en un diagrama de comunicación cuando se modelan varios objetos del mismo tipo.] Los objetos que se comunican se conectan con líneas sólidas y los mensajes se pasan entre los objetos a lo largo de estas líneas, en la dirección mostrada por las flechas. El nombre del mensaje, que aparece enseguida de la flecha, es el nombre de una operación (es decir, una función miembro) que pertenece al objeto receptor; considere el nombre como un “servicio” que el objeto receptor proporciona a los objetos emisores (sus “clientes”).

La flecha rellena en la figura 7.28 representa un mensaje (o **llamada síncrona**) en UML y una llamada a una función en C++. Esta flecha indica que el flujo de control va desde el objeto emisor (el ATM) hasta el objeto receptor (una `SolicitudSaldo`). Como ésta es una llamada síncrona, el objeto emisor no puede enviar otro mensaje, ni hacer cualquier otra cosa, hasta que el objeto receptor procese el mensaje y devuelva el control al objeto emisor; el emisor sólo espera. Por ejemplo, en la figura 7.28 el objeto ATM llama a la función miembro `ejecutar` de un objeto `SolicitudSaldo` y no puede enviar otro mensaje sino hasta que `ejecutar` termine y devuelva el control al objeto ATM. [Nota: si ésta fuera una **llamada asíncrona**, representada por una flecha, el objeto emisor no tendría que esperar a que el objeto receptor devolviera el control; continuaría enviando mensajes adicionales inmediatamente después de la llamada asíncrona. Con frecuencia, dichas llamadas se pueden implementar en C++ mediante el uso de bibliotecas específicas para cada plataforma, que el compilador proporciona. Dichas técnicas están más allá del alcance de este libro.]

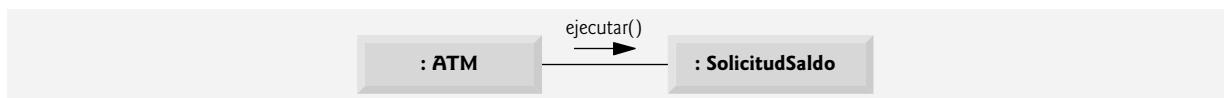


Figura 7.28 | Diagrama de comunicación del ATM, ejecutando una solicitud de saldo.

Secuencia de mensajes en un diagrama de comunicación

La figura 7.29 muestra un diagrama de comunicación que modela las interacciones entre los objetos en el sistema, cuando se ejecuta un objeto de la clase `SolicitudSaldo`. Asumimos que el atributo `numeroCuenta` del objeto contiene el número de cuenta del usuario actual. Las colaboraciones en la figura 7.29 empiezan después de que el objeto ATM envía un mensaje `ejecutar` a un objeto `SolicitudSaldo` (es decir, la interacción modelada en la figura 7.28). El número a la izquierda del nombre de un mensaje indica el orden en el que éste se pasa. La secuencia de mensajes en un diagrama de comunicación progresan en orden numérico, de menor a mayor. En este diagrama, la numeración comienza con el mensaje 1 y termina con el mensaje 3. El objeto `SolicitudSaldo` envía primero un mensaje `obtenerSaldoDisponible` al objeto `BaseDatosBanco` (mensaje 1), después envía un mensaje `obtenerSaldoTotal` al objeto `BaseDatosBanco` (mensaje 2). Dentro de los paréntesis que van después del nombre de un mensaje, podemos especificar una lista separada por comas de los nombres de los parámetros que se envían con el mensaje (es decir, los argumentos en la llamada a una función en C++); el objeto `SolicitudSaldo` pasa el atributo `numeroCuenta` con sus mensajes al objeto `BaseDatosBanco` para indicar de cuál objeto `Cuenta` se va a obtener la información del saldo. En la figura 6.37 vimos que las operaciones `obtenerSaldoDisponible` y `obtenerSaldoTotal` de la clase `BaseDatosBanco` requieren cada una de ellas un parámetro para identificar una cuenta. El objeto `SolicitudSaldo` muestra a continuación el `saldoDisponible` y el `saldoTotal` al usuario; para ello pasa un mensaje `mostrarMensaje` a la `Pantalla` (mensaje 3) que incluye un parámetro, el cual indica el `mensaje` a mostrar.

La figura 7.29 modela dos mensajes adicionales que se pasan del objeto `BaseDatosBanco` a un objeto `Cuenta` (mensaje 1.1 y mensaje 2.1). Para proveer al ATM los dos saldos de la Cuenta del usuario (según lo solicitado por los mensajes 1 y 2), el objeto `BaseDatosBanco` debe pasar un mensaje `obtenerSaldoDisponible` y un mensaje `obtenerSaldoTotal` a la `Cuenta` del usuario. Los mensajes que se pasan dentro del manejo de otro mensaje se llaman **mensajes anidados**. UML recomienda utilizar un esquema de numeración decimal para indicar mensajes anidados. Por ejemplo, el mensaje 1.1 es el primer mensaje anidado en el mensaje 1; el objeto `BaseDatosBanco` pasa un mensaje `obtenerSaldoDisponible` durante el procesamiento de `BaseDatosBanco` de un mensaje con el mismo nombre. [Nota: si el objeto `BaseDatosBanco` necesita pasar un segundo mensaje anidado mientras procesa el mensaje 1, el segundo mensaje se numera como 1.2.] Un mensaje puede pasarse sólo cuando se han pasado ya todos los mensajes anidados del mensaje anterior; por ejemplo, el objeto `SolicitudSaldo` pasa el mensaje 3 sólo hasta que se han pasado los mensajes 2 y 2.1, en ese orden.

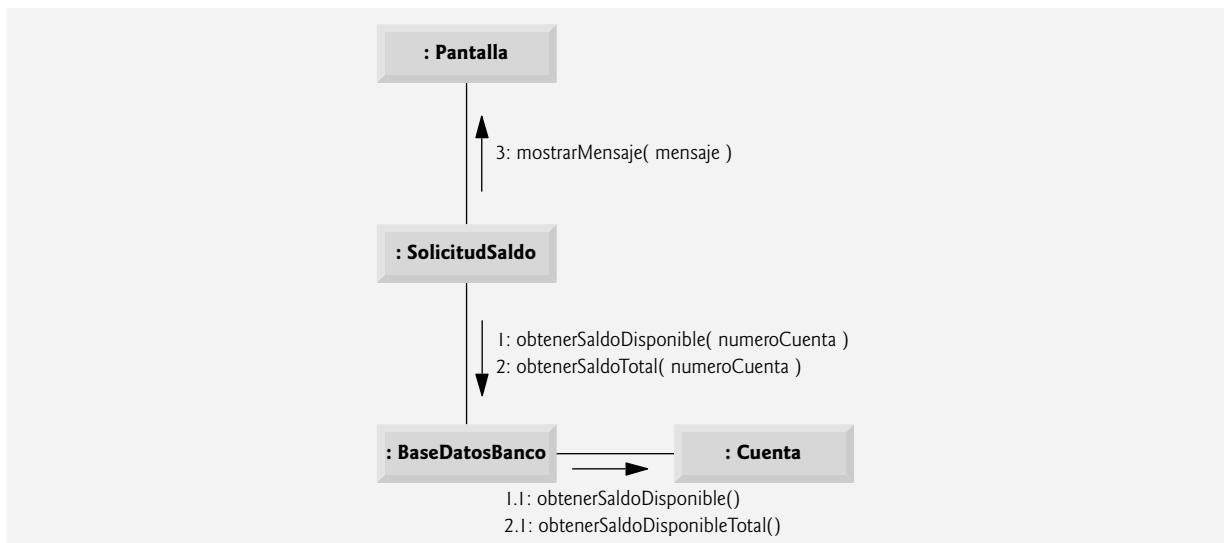


Figura 7.29 | Diagrama de comunicación para ejecutar una solicitud de saldo.

El esquema de numeración anidado que se utiliza en los diagramas de comunicación ayuda a aclarar con precisión cuándo y en qué contexto se pasa cada mensaje. Por ejemplo, si numeramos los cinco mensajes de la figura 7.29 usando un esquema de numeración plano (es decir, 1, 2, 3, 4, 5), podría ser posible que alguien que viera el diagrama no pudiera determinar que el objeto `BaseDatosBanco` pasa el mensaje `obtenerSaldoDisponible` (mensaje 1.1) a una `Cuenta` durante el procesamiento del mensaje 1 por parte del objeto `BaseDatosBanco`, en vez de hacerlo después de completar el procesamiento del mensaje 1. Los números decimales anidados hacen ver que el segundo mensaje `obtenerSaldoDisponible` (mensaje 1.1) se pasa a una `Cuenta` dentro del manejo del primer mensaje `obtenerSaldoDisponible` (mensaje 1) por parte del objeto `BaseDatosBanco`.

Diagramas de secuencia

Los diagramas de comunicación enfatizan los participantes en las colaboraciones, pero modelan su sincronización de una forma bastante extraña. Un diagrama de secuencia ayuda a modelar la sincronización de las colaboraciones con más claridad. La figura 7.30 muestra un diagrama de secuencia que modela la secuencia de las interacciones que ocurren cuando se ejecuta un `Retiro`. La línea punteada que se extiende hacia abajo desde el rectángulo de un objeto es la **línea de vida** de ese objeto, la cual representa la evolución en el tiempo. Por lo general, las acciones ocurren a lo largo de la línea de vida de un objeto, en orden cronológico de arriba hacia abajo; una acción cerca de la parte superior ocurre antes que una cerca de la parte inferior.

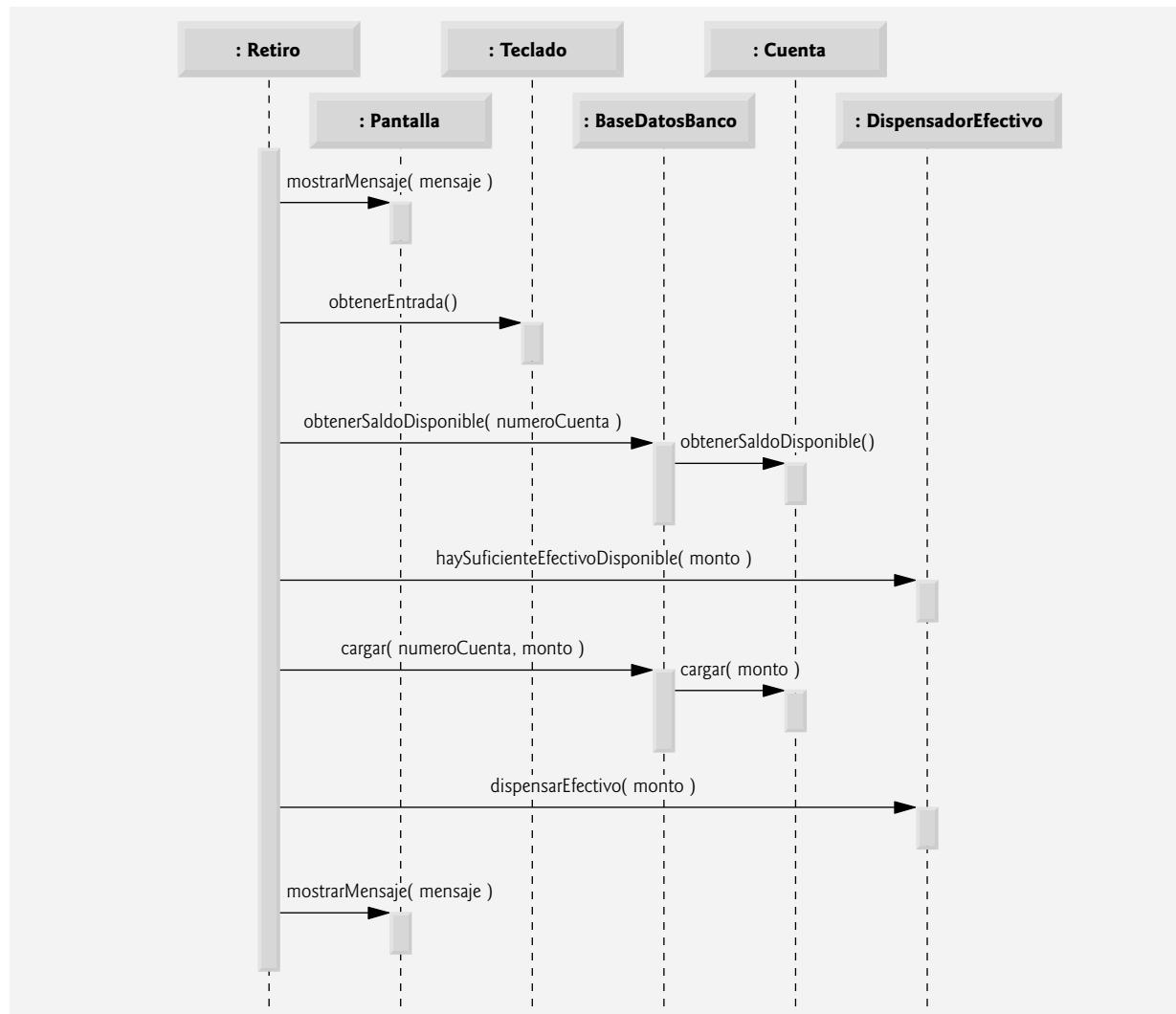


Figura 7.30 | Diagrama de secuencia que modela la ejecución de un `Retiro`.

El paso de mensajes en los diagramas de secuencia es similar al paso de mensajes en los diagramas de comunicación. Una flecha con punta rellena, que se extiende desde el objeto emisor hasta el objeto receptor, representa un mensaje entre dos objetos. La punta de flecha apunta a una activación en la línea de vida del objeto receptor. Una **activación**, que se muestra como un rectángulo vertical delgado, indica que se está ejecutando un objeto. Cuando un objeto devuelve el control, un mensaje de retorno (representado como una línea punteada con una punta de flecha) se extiende desde la activación del objeto que devuelve el control hasta la activación del objeto que envió originalmente el mensaje. Para eliminar el desorden, omitimos las flechas de los mensajes de retorno; UML permite esta práctica para que los diagramas sean más legibles. Al igual que los diagramas de comunicación, los diagramas de secuencia pueden indicar parámetros de mensaje entre los paréntesis que van después del nombre de un mensaje.

La secuencia de mensajes de la figura 7.30 empieza cuando un objeto **Retiro** pide al usuario que seleccione un monto de retiro; para ello envía un mensaje **mostrarMensaje** a la **Pantalla**. Después el objeto **Retiro** envía un mensaje **obtenerEntrada** al **Teclado**, el cual obtiene los datos de entrada del usuario. En el diagrama de actividad de la figura 5.28 ya hemos modelado la lógica de control involucrada en un objeto **Retiro**, por lo que no mostraremos esta lógica en el diagrama de secuencia de la figura 7.30. En lugar de ello modelaremos el escenario para el mejor caso, en el cual el saldo de la cuenta del usuario es mayor o igual al monto de retiro seleccionado, y el dispensador de efectivo contiene un monto de efectivo suficiente como para satisfacer la solicitud. Para obtener información acerca de cómo modelar la lógica de control en un diagrama de secuencia, consulte los recursos Web y las lecturas recomendadas que se listan al final de la sección 2.8.

Después de obtener un monto de retiro, el objeto **Retiro** envía un mensaje **obtenerSaldoDisponible** al objeto **BaseDatosBanco**, el cual a su vez envía un mensaje **obtenerSaldoDisponible** a la **Cuenta** del usuario. Suponiendo que la cuenta del usuario tiene suficiente dinero disponible para permitir la transacción, el objeto **Retiro** envía a continuación un mensaje **haySuficienteEfectivoDisponible** al objeto **DispensadorEfectivo**. Suponiendo que hay suficiente efectivo disponible, el objeto **Retiro** reduce el saldo de la cuenta del usuario (tanto el **saldoTotal** como el **saldoDisponible**) enviando un mensaje **cargar** a la **Cuenta** del usuario. Por último, el objeto **Retiro** envía un mensaje **dispensarEfectivo** al **DispensadorEfectivo** y un mensaje **mostrarMensaje** a la **Pantalla**, indicando al usuario que quite el efectivo de la máquina.

Hemos identificado las colaboraciones entre los objetos en el sistema ATM, y modelamos algunas de estas colaboraciones usando los diagramas de interacción de UML: los diagramas de comunicación y los diagramas de secuencia. En la siguiente sección del Ejemplo práctico de Ingeniería de Software (sección 9.11), mejoraremos la estructura de nuestro modelo para completar un diseño orientado a objetos preliminar, y después empezaremos a implementar el sistema ATM.

Ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

- 7.1 Una _____ consiste en que un objeto de una clase envía un mensaje a un objeto de otra clase.
- asociación.
 - agregación.
 - colaboración.
 - composición.
- 7.2 ¿Cuál forma de diagrama de interacción es la que enfatiza *qué* colaboraciones se llevan a cabo? ¿Cuál forma enfatiza *cuándo* ocurren las interacciones?
- 7.3 Cree un diagrama de secuencia para modelar las interacciones entre los objetos del sistema ATM, que ocurran cuando se ejecute un **Depósito** con éxito, y explique la secuencia de los mensajes modelados por el diagrama.

Respuestas a los ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

- 7.1 c.
- 7.2 Los diagramas de comunicación enfatizan *qué* colaboraciones se llevan a cabo. Los diagramas de secuencia enfatizan *cuándo* ocurren las colaboraciones.
- 7.3 La figura 7.31 presenta un diagrama de secuencia que modela las interacciones entre objetos en el sistema ATM, las cuales ocurren cuando un **Depósito** se ejecuta con éxito. La figura 7.31 indica que un **Depósito** primero envía un mensaje **mostrarMensaje** a la **Pantalla**, para pedir al usuario que introduzca un monto de depósito. A continuación, el **Depósito** envía un mensaje **obtenerEntrada** al **Teclado** para recibir la entrada del usuario. Después, el **Depósito** pide al usuario que inserte un sobre de depósito; para ello envía un mensaje **mostrarMensaje** a la **Pantalla**. Luego, el **Depósito** envía un mensaje **seRecibioSobreDeposito** al objeto **RanuraDeposito** para confirmar que el ATM haya recibido el sobre de depósito. Por últi-

mo, el objeto `Deposito` incrementa el atributo `saldoTotal` (pero no el atributo `saldoDisponible`) de la `Cuenta` del usuario, enviando un mensaje `abonar` al objeto `BaseDatosBanco`. El objeto `BaseDatosBanco` responde enviando el mismo mensaje a la `Cuenta` del usuario.

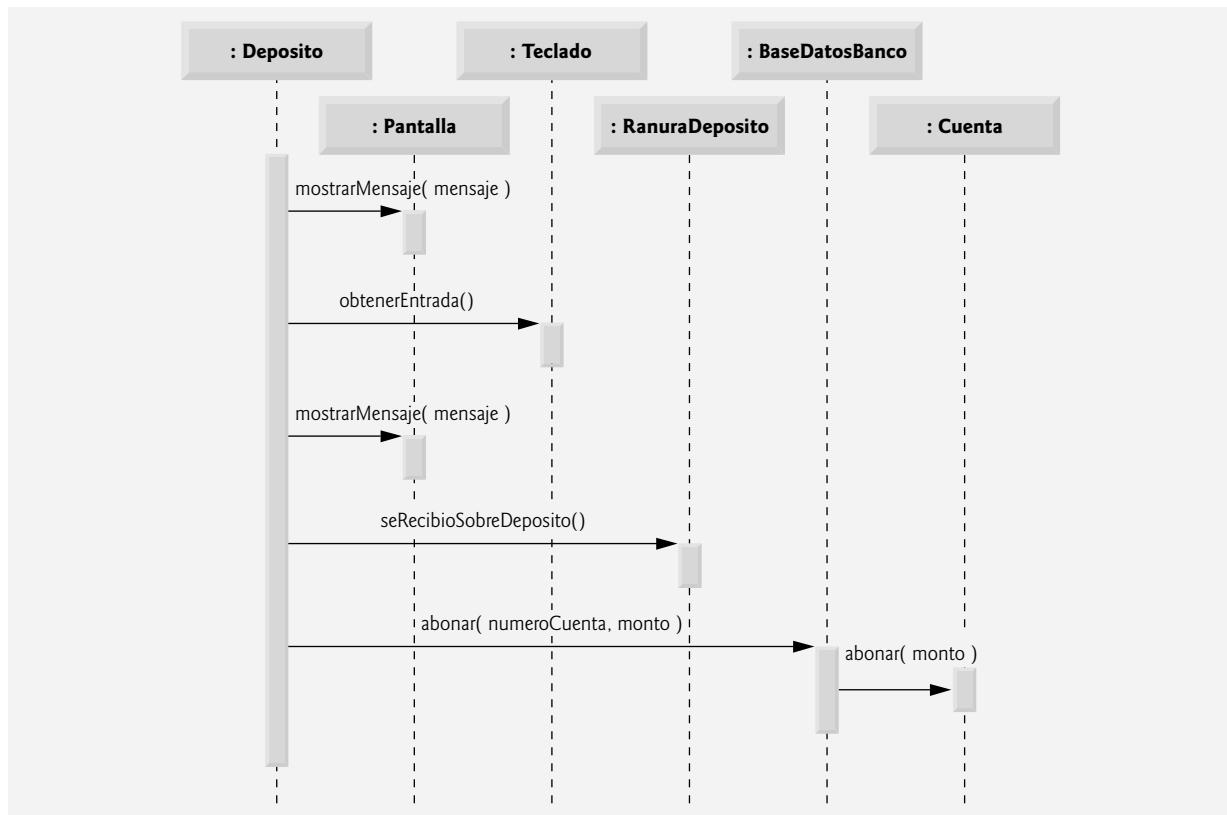


Figura 7.31 | Diagrama de secuencia que modela la ejecución de un `Deposito`.

7.13 Repaso

En este capítulo empezó nuestra introducción a las estructuras de datos, explorando el uso de los arreglos y objetos `vector` para almacenar datos y obtenerlos de listas y tablas de valores. Los ejemplos de este capítulo demostraron cómo declarar un arreglo, inicializarlo y hacer referencia a los elementos individuales de un arreglo. También le mostramos cómo pasar arreglos a las funciones, y cómo usar el calificador `const` para hacer valer el principio del menor privilegio. En los ejemplos del capítulo también presentamos las técnicas básicas de ordenamiento y búsqueda. Aprendió a declarar y manipular arreglos multidimensionales. Por último, demostramos las herramientas de la plantilla de clase `vector` de la Biblioteca estándar de C++, la cual proporciona una alternativa más completa a los arreglos.

Continuaremos con nuestra cobertura de las estructuras de datos en el capítulo 14, Plantillas, donde crearemos una plantilla de clase de pila y en el capítulo 20, Estructuras de datos, que introduce otras estructuras de datos dinámicas, como listas, colas, pilas y árboles, que pueden crecer y reducirse a medida que se ejecutan los programas. En el capítulo 22, Biblioteca de plantillas estándar (STL), se introducen varias de las estructuras de datos predefinidas de la Biblioteca estándar de C++, que los programadores pueden usar en lugar de crear sus propias estructuras de datos. En el capítulo 22 se presenta la funcionalidad completa de la plantilla de clase `vector` y se describen muchas estructuras de datos adicionales, incluyendo `list` y `deque`, que son estructuras de datos tipo arreglo, las cuales pueden crecer y reducirse en respuesta a la variación en los requerimientos de almacenamiento de un programa.

Ya le hemos presentado los conceptos básicos de las clases, los objetos, las instrucciones de control, las funciones y los arreglos. En el capítulo 8 presentaremos una de las herramientas más poderosas de C++: el apuntador. Los apuntadores llevan la cuenta de la ubicación donde se almacenan los datos y las funciones en memoria, lo cual nos permite manipular esos elementos en formas interesantes. Después de introducir los conceptos básicos de los apuntadores, analizaremos con detalle las estrechas relaciones entre los arreglos, los apuntadores y las cadenas.

Resumen

Sección 7.1 Introducción

- Las estructuras de datos son colecciones de elementos de datos relacionados. Los arreglos son estructuras de datos que consisten en elementos de datos relacionados del mismo tipo. Los arreglos son entidades “estáticas”, en cuanto a que permanecen del mismo tamaño a lo largo de la ejecución del programa (desde luego que pueden ser de una clase de almacenamiento automático y, por ende, se pueden crear y destruir cada vez que se entra y sale de los bloques en los que están definidos).

Sección 7.2 Arreglos

- Un arreglo es un grupo consecutivo de ubicaciones de memoria que comparten el mismo tipo.
- Para hacer referencia a una ubicación específica de un elemento en un arreglo, especificamos el nombre del arreglo y el número de posición del elemento específico en el arreglo.
- Para hacer referencia a cualquiera de los elementos de un arreglo, un programa proporciona el nombre del arreglo seguido del número de posición del elemento específico entre corchetes ([]). El número de posición se llama más formalmente como subíndice o índice (este número especifica el número de elementos a partir del inicio del arreglo).
- El primer elemento en cada arreglo tiene el subíndice cero, y algunas veces se le llama el elemento cero.
- Un subíndice debe ser un entero o una expresión entera (que utilice cualquier tipo integral).
- Los corchetes que se utilizan para encerrar el subíndice de un arreglo son un operador en C++. Los corchetes tienen el mismo nivel de precedencia que los paréntesis.

Sección 7.3 Declaración y creación de arreglos

- Los arreglos ocupan espacio en memoria. El programador especifica el tipo de cada elemento y el número de elementos requeridos de la siguiente manera:

`tipo nombreArreglo[tamañoArreglo];`

y el compilador reserva el monto de memoria apropiado.

- Los arreglos se pueden declarar de manera que contengan cualquier tipo de datos. Por ejemplo, un arreglo de tipo char se puede utilizar para almacenar una cadena de caracteres.

Sección 7.4 Ejemplos acerca del uso de arreglos

- Los elementos de un arreglo se pueden inicializar en la declaración de arreglos, seguida del nombre del arreglo con un signo de igual y una lista inicializadora: una lista separada por comas (encerrada entre llaves) de inicializadores constantes. Al inicializar un arreglo con una lista inicializadora, si hay menos inicializadores que elementos en el arreglo, el resto de los elementos se inicializa con cero.
- Si se omite el tamaño del arreglo de una declaración con una lista inicializadora, el compilador determina el número de elementos en el arreglo, para lo cual cuenta el número de elementos en la lista inicializadora.
- Si se especifica el tamaño del arreglo y una lista inicializadora en la declaración de un arreglo, el número de inicializadores debe ser menor o igual al tamaño del arreglo. Si se proporcionan más inicializadores en una lista inicializadora de un arreglo que elementos en el arreglo, se produce un error de compilación.
- Las constantes se deben inicializar con una expresión constante al declararse, y no pueden modificarse en lo sucesivo. Las constantes se pueden colocar en cualquier parte en la que se espere una expresión constante.
- C++ no tiene comprobación de límites de arreglos para evitar que la computadora haga referencia a un elemento que no existe. Por ende, un programa en ejecución se puede “salir” de cualquier extremo de un arreglo sin advertencia. El programador debe asegurar que todas las referencias a los arreglos permanezcan dentro de los límites del arreglo.
- Un arreglo de caracteres se puede inicializar mediante el uso de una literal de cadena. El compilador determina el tamaño de un arreglo de caracteres con base en la longitud de la cadena *más* un carácter especial de terminación de cadena, conocido como carácter nulo (el cual se representa mediante la constante '\0').
- Todas las cadenas representadas por arreglos de caracteres terminan con el carácter nulo. Un arreglo de caracteres que represente a una cadena siempre se debe declarar con la suficiente longitud como para que pueda contener el número de caracteres en la cadena, junto con el carácter nulo de terminación.
- Los arreglos de caracteres también se pueden inicializar mediante constantes tipo carácter individuales en una lista inicializadora.
- Para acceder a los caracteres individuales en una cadena, se utiliza la notación de subíndices de arreglos.
- Una cadena se puede introducir directamente en un arreglo de caracteres mediante el teclado, usando `cin` y `>>`.
- Un arreglo de caracteres que represente a una cadena con terminación nula se puede imprimir con `cout` y `<<`.
- Una variable local `static` en la definición de una función existe durante el tiempo que se ejecute el programa, pero sólo es visible en el cuerpo de la función.
- Un programa inicializa los arreglos locales `static` la primera vez que encuentra sus declaraciones. Si el programador no inicializa en forma explícita un arreglo `static`, el compilador inicializa con cero cada elemento de ese arreglo a la hora de crearlo.

Sección 7.5 Paso de arreglos a funciones

- Para pasar un argumento tipo arreglo a una función, se debe especificar el nombre del arreglo sin corchetes. Para pasar un elemento de un arreglo a una función, se utiliza el nombre con subíndice del elemento del arreglo como un argumento en la llamada a la función.
- Los arreglos se pasan a las funciones por referencia; las funciones llamadas pueden modificar los valores de los elementos en los arreglos originales de la función que hace las llamadas. El valor del nombre del arreglo es la dirección en la memoria de la computadora del primer elemento del arreglo. Como se pasa la dirección inicial del arreglo, la función llamada conoce exactamente dónde se almacena el arreglo en la memoria.
- Los elementos individuales de un arreglo se pasan por valor, de la misma forma que las variables simples. A esas piezas de datos simples e individuales se les llama escalares o cantidades escalares.
- Para recibir un argumento tipo arreglo, la lista de parámetros de una función debe especificar que ésta espera recibir un arreglo. El tamaño del arreglo no se requiere en los corchetes del mismo.
- C++ cuenta con el calificador de tipos `const`, el cual se puede usar para evitar que la función llamada modifique los valores de un arreglo mediante código en la función que hace la llamada. Cuando a un parámetro tipo arreglo se le antepone el calificador `const`, los elementos del arreglo se hacen constantes en el cuerpo de la función, y cualquier intento de modificar un elemento del arreglo en el cuerpo de la función produce un error de compilación.

Sección 7.6 Ejemplo práctico: la clase `LibroCalificaciones` que usa un arreglo para almacenar las calificaciones

- Las variables de clase (miembros de datos `static`) son compartidas por todos los objetos de la clase en la que se declaran las variables.
- Se puede acceder a un miembro de datos `static` dentro de la definición de la clase y de las definiciones de las funciones miembro al igual que con cualquier otro miembro de datos.
- También se puede acceder a un miembro de datos `public static` desde el exterior de la clase, aun y cuando no existan objetos de la misma; para ello se utiliza el nombre de la clase, seguido del operador de resolución de ámbito binario (`::`) y el nombre del miembro de datos.

Sección 7.7 Búsqueda de datos en arreglos mediante la búsqueda lineal

- La búsqueda lineal compara cada elemento de un arreglo con una clave de búsqueda. Como el arreglo no está en ningún orden específico, existe la misma probabilidad de que se encuentre el valor tanto en el primer elemento como en el último. Por lo tanto, en promedio el programa debe comparar la clave de búsqueda con la mitad de los elementos del arreglo. Para determinar que un valor no se encuentra en el arreglo, el programa debe comparar la clave de búsqueda con cada elemento del arreglo. Este método de búsqueda lineal funciona bien para arreglos pequeños, y es aceptable para los arreglos desordenados.

Sección 7.8 Ordenamiento de arreglos mediante el ordenamiento por inserción

- Un arreglo se puede ordenar mediante el ordenamiento por inserción. La primera iteración de este algoritmo toma el segundo elemento y, si es menor que el primero, lo intercambia (es decir, el programa *inserta* el segundo elemento enfrente del primer elemento). La segunda iteración analiza el tercer elemento y lo inserta en la posición correcta respecto a los primeros dos elementos, de manera que los tres elementos estén en orden. En la i -ésima iteración de este algoritmo, los primeros i elementos en el arreglo original estarán ordenados. Para arreglos pequeños el ordenamiento por inserción es aceptable, pero para arreglos grandes es ineficiente en comparación con otros algoritmos de ordenamiento más sofisticados.

Sección 7.9 Arreglos multidimensionales

- Los arreglos multidimensionales de dos dimensiones se utilizan con frecuencia para representar tablas de valores, las cuales consisten en información ordenada en filas y columnas.
- Los arreglos que requieren dos subíndices para identificar a un elemento específico se llaman arreglos bidimensionales. Un arreglo con m filas y n columnas se llama arreglo de m por n .

Sección 7.11 Introducción a la plantilla de clase `vector` de la Biblioteca estándar de C++

- La plantilla de clase `vector` de la Biblioteca estándar de C++ representa una alternativa más completa a los arreglos, ya que cuenta con muchas capacidades que no se proporcionan para los arreglos basados en apuntador estilo C.
- De manera predeterminada, todos los elementos de un objeto `vector` entero se establecen en 0.
- Un `vector` se puede definir de manera que almacene cualquier tipo de datos, mediante el uso de una declaración como la siguiente:

```
vector< tipo > nombre( tamaño );
```

- La función miembro `size` de la plantilla de clase `vector` devuelve el número de elementos en el `vector` en el que se invoca.
- Para acceder al valor de un elemento de un `vector` (o para modificarlo), se utilizan corchetes (`[]`).
- Los objetos de la plantilla de clase estándar `vector` se pueden comparar de manera directa con los operadores de igualdad (`==`) y desigualdad (`!=`). El operador de asignación (`=`) también se puede usar con objetos `vector`.

- Un *lvalue* no modifiable es una expresión que identifica a un objeto en memoria (como un elemento en un *vector*), pero no se puede utilizar para modificar ese objeto. Un *lvalue* modifiable también identifica a un objeto en memoria, pero se puede usar para modificar el objeto.
- La plantilla de clase estándar *vector* proporciona la comprobación de límites mediante su función miembro *at*, que “lanza una excepción” si su argumento es un subíndice inválido. De manera predeterminada, esto hace que un programa en C++ termine su ejecución.

Terminología

<i>a[i]</i>	formato tabular
<i>a[i][j]</i>	índice
arreglo	inicializador
arreglo 2-D	inicializar un arreglo
arreglo bidimensional	lista inicializadora
arreglo de <i>m</i> por <i>n</i>	lista inicializadora de arreglos
arreglo multidimensional	<i>lvalue</i> modifiable
arreglo unidimensional	<i>lvalue</i> no modifiable
<i>at</i> , función miembro de <i>vector</i>	nombre de un arreglo
buscar en un arreglo	número de posición
búsqueda lineal de un arreglo	número mágico
cadena representada por un arreglo de caracteres	ordenamiento por inserción
cantidad escalar	ordenar un arreglo
carácter nulo ('\\0')	paso de arreglos a funciones
clave de búsqueda	paso por referencia
columna de un arreglo bidimensional	“salirse” de un arreglo
comprobación de límites	<i>size</i> , función miembro de un <i>vector</i>
<i>const</i> , calificador de tipo	<i>static</i> , miembro de datos
constante con nombre	subíndice
corchetes, []	subíndice cero
declarar un arreglo	subíndice de columna
elemento cero	subíndice de fila
elemento de un arreglo	tabla de valores
error de desplazamiento por uno	valor clave
escalabilidad	valor de un elemento
escalar	variable constante
estructura de datos	variables de sólo lectura
fila de un arreglo bidimensional	<i>vector</i> (plantilla de clase de la Biblioteca estándar de C++)

Ejercicios de autoevaluación

7.1 Complete las siguientes oraciones:

- Las listas y tablas de valores pueden guardarse en _____ o _____.
- Los elementos de un arreglo están relacionados por el hecho de que tienen el mismo _____ y _____.
- El número utilizado para referirse a un elemento específico de un arreglo se conoce como el _____ de ese elemento.
- Un(a) _____ debe usarse para declarar el tamaño de un arreglo, ya que hace al programa más escalable.
- Al proceso de colocar los elementos de un arreglo en orden se le conoce como _____ el arreglo.
- Al proceso de determinar si un arreglo contiene un valor clave específico se le conoce como _____ el arreglo.
- Un arreglo que utiliza dos subíndices se conoce como un arreglo _____.

7.2 Conteste con *verdadero* o *falso* cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

- Un arreglo puede guardar muchos tipos distintos de valores.
- El subíndice de un arreglo debe ser generalmente de tipo *float*.
- Si hay menos inicializadores en una lista inicializadora que el número de elementos en el arreglo, el resto de los elementos se inicializa con el último valor en la lista inicializadora.
- Es un error si una lista inicializadora contiene más inicializadores que elementos en el arreglo.
- Un elemento individual de un arreglo que se pasa a una función y se modifica ahí mismo, contendrá el valor modificado cuando el método llamado termine su ejecución.

- 7.3** Escriba una o más instrucciones que realicen las siguientes tareas para un arreglo llamado **fracciones**:
- Defina una variable constante entera llamada **tamanioArreglo** que se inicialice con 10.
 - Declare un arreglo con **tamanioArreglo** elementos de tipo **double**, e inicialice los elementos con 0.
 - Nombre el cuarto elemento del arreglo.
 - Haga referencia al elemento 4 del arreglo.
 - Asigne el valor 1.667 al elemento 9 del arreglo.
 - Asigne el valor 3.333 al séptimo elemento del arreglo.
 - Imprima los elementos 6 y 9 del arreglo con dos dígitos de precisión a la derecha del punto decimal, y muestre los resultados que aparecen realmente en la pantalla.
 - Imprima todos los elementos del arreglo usando una instrucción **for**. Defina la variable entera **i** como variable de control para el ciclo. Muestre la salida.
- 7.4** Responda a las siguientes preguntas en relación con un arreglo llamado **tabla**:
- Declare el arreglo como un arreglo entero con tres filas y tres columnas. Suponga que se ha declarado la variable **tamanioArreglo** con el valor de 3.
 - ¿Cuántos elementos contiene el arreglo?
 - Utilice una instrucción **for** para inicializar cada elemento del arreglo con la suma de sus subíndices. Suponga que se declaran las variables enteras **i** y **j** como variables de control.
 - Escriba un segmento de programa para imprimir los valores de cada elemento del arreglo **tabla** en formato tabular, con tres filas y tres columnas. Suponga que el arreglo se inicializó con la siguiente declaración:
- ```
int tabla[tamanioArreglo][tamanioArreglo] = { { 1, 8 }, { 2, 4, 6 }, { 5 } };
```
- y las variables enteras **i** y **j** se declaran como variables de control. Muestre los resultados.
- 7.5** Encuentre y corrija el error en cada uno de los siguientes fragmentos de programa:
- #include <iostream>;
  - tamanioArreglo** = 10; // **tamanioArreglo** se declaró como **const**
  - Suponga que int **b[ 10 ]** = {};
  - for ( int **i** = 0; **i** <= 10; **i**++ )  
    **b[ i ]** = 1;
  - Suponga que int **a[ 2 ][ 2 ]** = { { 1, 2 }, { 3, 4 } };  
**a[ 1, 1 ]** = 5;

## Respuestas a los ejercicios de autoevaluación

- 7.1** a) arreglos, objetos vector. b) nombre de arreglo, tipo. c) subíndice o índice. d) variable constante. e) ordenamiento. f) búsqueda. g) bidimensional.
- 7.2** a) Falso. Un arreglo sólo puede guardar valores del mismo tipo.  
 b) Falso. El subíndice de un arreglo debe ser un entero o una expresión entera.  
 c) Falso. El resto de los elementos se inicializa con cero.  
 d) Verdadero.  
 e) Falso. Los elementos individuales de un arreglo se pasan por valor. Si se pasa el arreglo completo a una función, entonces cualquier modificación se reflejará en el original.
- 7.3** a) **const int tamanioArreglo** = 10;  
 b) **double fracciones[ tamanioArreglo ]** = { 0.0 };  
 c) **fracciones[ 3 ]**  
 d) **fracciones[ 4 ]**  
 e) **fracciones[ 9 ]** = 1.667;  
 f) **fracciones[ 6 ]** = 3.333;  
 g) **cout << fixed << setprecision( 2 );**  
     **cout << fracciones[ 6 ] << ' ' << fracciones[ 9 ] << endl;**  
     *Salida: 3.33 1.67.*  
 h) **for ( int **i** = 0; **i** < tamanioArreglo; **i**++ )**  
     **cout << "fracciones[" << **i** << "] = " << fracciones[ **i** ] << endl;**  
     *Salida:*  
     **fracciones[ 0 ]** = 0.0  
     **fracciones[ 1 ]** = 0.0

```

fracciones[2] = 0.0
fracciones[3] = 0.0
fracciones[4] = 0.0
fracciones[5] = 0.0
fracciones[6] = 3.333
fracciones[7] = 0.0
fracciones[8] = 0.0
fracciones[9] = 1.667

```

- 7.4 a) int tabla[ tamanoArreglo ][ tamanoArreglo ];

b) Nueve.

c) for ( i = 0; i < tamanoArreglo; i++ )

```

 for (j = 0; y < tamanoArreglo; j++)
 tabla[i][j] = i + j;

```

d) cout << " [0] [1] [2]" << endl;

```

for (int i = 0; i < tamanoArreglo; i++) {
 cout << '[' << i << "] ";

 for (int j = 0; j < tamanoArreglo; j++)
 cout << setw(3) << tabla[i][j] << " ";
 cout << endl;
}

```

*Salida:*

|     |     |     |
|-----|-----|-----|
| [0] | [1] | [2] |
| [0] | 1   | 8   |
| [1] | 2   | 4   |
| [2] | 5   | 0   |

- 7.5 a) *Error:* punto y coma al final de la directiva del preprocesador #include.

*Corrección:* elimina el punto y coma.

- b) *Error:* asignar un valor a una variable constante que utiliza una instrucción de asignación.

*Corrección:* inicializa la variable constante en una declaración const int tamanoArreglo.

- c) *Error:* hacer referencia a un elemento del arreglo fuera de sus límites (b[10]).

*Corrección:* cambie el valor final de la variable de control a 9.

- d) *Error:* el subíndice del arreglo se escribió en forma incorrecta.

*Corrección:* cambie la instrucción por a[ 1 ][ 1 ] = 5;.

## Ejercicios

- 7.6 Complete las siguientes oraciones:

a) Los nombres de los cuatro elementos de un arreglo p (int p[4];) son \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_ y \_\_\_\_\_.

b) Al proceso de nombrar un arreglo, declarar su tipo y especificar el número de elementos se le conoce como \_\_\_\_\_ el arreglo.

c) Por convención, el primer subíndice en un arreglo bidimensional identifica el (la) \_\_\_\_\_ de un elemento y el segundo índice identifica el (la) \_\_\_\_\_ del elemento.

d) Un arreglo de m por n contiene \_\_\_\_\_ filas, \_\_\_\_\_ columnas y \_\_\_\_\_ elementos.

e) El nombre del elemento en la fila 3 y la columna 5 del arreglo d es \_\_\_\_\_.

- 7.7 Conteste con *verdadero* o *falso* cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

a) Para referirnos a una ubicación o elemento específico dentro de un arreglo, especificamos el nombre del arreglo y el valor del elemento específico.

b) La definición de un arreglo reserva espacio para el mismo.

c) Para indicar que deben reservarse 100 ubicaciones para el arreglo entero p, el programador escribe la declaración

p[ 100 ];

d) Hay que usar una instrucción for para inicializar con cero los elementos de un arreglo de 15 elementos.

e) Hay que usar instrucciones for anidadas para sumar el total de los elementos de un arreglo bidimensional.

- 7.8** Escriba instrucciones en C++ que realicen cada una de las siguientes tareas:
- Mostrar el valor del elemento 6 del arreglo de caracteres *f*.
  - Recibir un valor y colocarlo en el elemento 4 de un arreglo de punto flotante unidimensional llamado *b*.
  - Inicializar con 8 cada uno de los 5 elementos del arreglo entero unidimensional *g*.
  - Sumar el total e imprimir los elementos del arreglo *c* de punto flotante con 100 elementos.
  - Copiar el arreglo *a* en la primera parte del arreglo *b*. Suponga que se declara *double a[ 11 ], b[ 34 ]*;
  - Determinar e imprimir los valores menor y mayor contenidos en el arreglo *w* con 99 elementos de punto flotante.
- 7.9** Considere un arreglo entero *t* de 2 por 3.
- Escriba una declaración para *t*.
  - ¿Cuántas filas tiene *t*?
  - ¿Cuántas columnas tiene *t*?
  - ¿Cuántos elementos tiene *t*?
  - Escriba los nombres de todos los elementos en la fila 1 de *t*.
  - Escriba los nombres de todos los elementos en la columna 2 de *t*.
  - Escriba una sola instrucción que asigne cero al elemento de *t* en la primera fila y la segunda columna.
  - Escriba una serie de instrucciones que inicialice cada elemento de *t* con cero. No utilice un ciclo.
  - Escriba una instrucción *for* anidada que inicialice cada elemento de *t* con cero.
  - Escriba una instrucción que reciba como entrada los valores para los elementos de *t* mediante la terminal.
  - Escriba una serie de instrucciones que determine e imprima el valor más pequeño en el arreglo *t*.
  - Escriba una instrucción que muestre los elementos en la fila 0 de *t*.
  - Escriba una instrucción que totalice los elementos de la columna 3 de *t*.
  - Escriba una serie de instrucciones para imprimir el contenido de *t* en formato tabular ordenado. Enliste los subíndices de columna como encabezados a lo largo de la parte superior, y enliste los subíndices de fila a la izquierda de cada fila.
- 7.10** Utilice un arreglo unidimensional para resolver el siguiente problema. Una compañía paga a sus vendedores por comisión. Los vendedores reciben \$200 por semana más 9% de sus ventas totales de esa semana. Por ejemplo, un vendedor que acumule \$5000 en ventas en una semana, recibirá \$200 más 9% de \$5000, o un total de \$650. Escriba un programa (utilizando un arreglo de contadores) que determine cuántos vendedores recibieron salarios en cada uno de los siguientes rangos (suponga que el salario de cada vendedor se trunca a una cantidad entera):
- \$200-299
  - \$300-399
  - \$400-499
  - \$500-599
  - \$600-699
  - \$700-799
  - \$800-899
  - \$900-999
  - \$1000 en adelante
- 7.11** (*Ordenamiento de burbuja*) En el *ordenamiento de burbuja*, los valores más pequeños van “subiendo como burbujas” gradualmente, hasta llegar a la parte superior del arreglo (es decir, hacia el primer elemento) como las burbujas de aire que se elevan en el agua, mientras que los valores más grandes se hunden en el fondo. Esta técnica realiza varias pasadas a través del arreglo. En cada pasada compara pares sucesivos de elementos. Si un par se encuentra en orden ascendente (o los valores son idénticos), el ordenamiento de burbuja deja los valores como están. Si un par se encuentra en orden descendente, el ordenamiento de burbuja intercambia sus valores en el arreglo. Escriba un programa que ordene un arreglo de 10 enteros mediante el uso del ordenamiento de burbuja.
- 7.12** El ordenamiento de burbuja descrito en el ejercicio 7.11 es inefficiente para grandes arreglos. Realice las siguientes modificaciones simples para mejorar el rendimiento del ordenamiento de burbuja:
- Después de la primera pasada, se garantiza que el número más grande estará en el elemento con la numeración más alta del arreglo; después de la segunda pasada, los dos números más altos estarán “acomodados”, y así en lo sucesivo. En lugar de realizar nueve comparaciones en cada pasada, modifique el ordenamiento de burbuja para que realice ocho comparaciones en la segunda pasada, siete en la tercera, y así en lo sucesivo.
  - Los datos en el arreglo tal vez se encuentren ya en el orden apropiado, o casi apropiado, así que ¿para qué realizar nueve pasadas, si basta con menos? Modifique el ordenamiento para comprobar al final de cada pasada si se han realizado intercambios. Si no se ha realizado ninguno, los datos ya deben estar en el orden apropiado, por lo que el programa debe terminar. Si se han realizado intercambios, por lo menos se necesita una pasada más.

**7.13** Escriba instrucciones individuales que realicen las siguientes operaciones con arreglos unidimensionales:

- Inicializar con cero los 10 elementos del arreglo `cuentas` de tipo entero.
- Sumar uno a cada uno de los 15 elementos del arreglo `bono` de tipo entero.
- Leer 12 valores para el arreglo `doble` llamado `temperaturasMensuales` mediante el teclado.
- Imprimir los primeros 5 valores del arreglo entero `mejoresPuntuaciones` en formato de columnas.

**7.14** Encuentre el (los) error(es) en cada una de las siguientes instrucciones:

- a) Asuma que: `char str[ 5 ];`

```
cin >> str; // el usuario escribe "hola"
```

- b) Asuma que: `int a[ 3 ];`

```
cout << a[1] << " " << a[2] << " " << a[3] << endl;
```

- c) `double f[ 3 ] = { 1.1, 10.01, 100.001, 1000.0001 };`

- d) Asuma que: `double d[ 2 ][ 10 ];`

```
d[1, 9] = 2.345;
```

**7.15** Use un arreglo unidimensional para resolver el siguiente problema. Recibir como entrada 20 números, cada uno de los cuales debe estar entre 10 y 100, inclusive. A medida que se lea cada número, validararlo y almacenarlo en el arreglo, sólo si no es un duplicado de un número ya leído. Después de leer todos los valores, mostrar sólo los valores únicos que el usuario introdujo. Prepárese para el “peor caso”, en el que los 20 números son diferentes. Use el arreglo más pequeño que sea posible para resolver este problema.

**7.16** Etiquete los elementos del arreglo bidimensional `ventas` de 3 por 5, para indicar el orden en el que se establecen en cero, mediante el siguiente fragmento de programa:

```
for (fila = 0; fila < 3; fila++)
 for (columna = 0; columna < 5; columna++)
 ventas[fila][col] = 0;
```

**7.17** Escriba un programa para simular el tiro de dos dados. El programa debe utilizar `rand` para tirar el primer dado, y de nuevo para tirar el segundo dado. Después debe calcularse la suma de los dos valores. [Nota: cada dado puede mostrar un valor entero del 1 al 6, por lo que la suma de los valores variará del 2 al 12, siendo 7 la suma más frecuente, mientras que 2 y 12 serán las sumas menos frecuentes]. En la figura 7.32 se muestran las 36 posibles combinaciones de los dos dados. Su programa debe tirar los dados 36,000 veces. Utilice un arreglo unidimensional para registrar el número de veces que apareza cada una de las posibles sumas. Imprima los resultados en formato tabular. Determine además si los totales son razonables (es decir, hay seis formas de tirar un 7, por lo que aproximadamente una sexta parte de los tiros deben ser 7).

|   | 1 | 2 | 3 | 4  | 5  | 6  |
|---|---|---|---|----|----|----|
| 1 | 2 | 3 | 4 | 5  | 6  | 7  |
| 2 | 3 | 4 | 5 | 6  | 7  | 8  |
| 3 | 4 | 5 | 6 | 7  | 8  | 9  |
| 4 | 5 | 6 | 7 | 8  | 9  | 10 |
| 5 | 6 | 7 | 8 | 9  | 10 | 11 |
| 6 | 7 | 8 | 9 | 10 | 11 | 12 |

**Figura 7.32** | Los 36 posibles resultados de tirar dos dados.

**7.18** ¿Qué hace el siguiente programa?

```
1 // Ej. 7.18: ej07_18.cpp
2 // ¿Qué hace este programa?
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int queEsEsto(int [], int); // prototipo de función
```

```

8
9 int main()
10 {
11 const int tamanoArreglo = 10;
12 int a[tamanoArreglo] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
13
14 int resultado = queEsEsto(a, tamanoArreglo);
15
16 cout << "El resultado es " << resultado << endl;
17 return 0; // indica que terminó correctamente
18 } // fin de main
19
20 // ¿Qué hace esta función?
21 int queEsEsto(int b[], int tamano)
22 {
23 if (tamano == 1) // caso base
24 return b[0];
25 else // paso recursivo
26 return b[tamano - 1] + queEsEsto(b, tamano - 1);
27 } // fin de la función queEsEsto

```

**7.19** Modifique el programa de la figura 6.11 para ejecutar 1000 juegos de craps. El programa debe mantener un registro de las estadísticas y responder a las siguientes preguntas:

- a) ¿Cuántos juegos se ganan en el primer tiro, en el segundo tiro, ..., en el vigésimo tiro y después de éste?
- b) ¿Cuántos juegos se pierden en el primer tiro, en el segundo tiro, ..., en el vigésimo tiro y después de éste?
- c) ¿Cuáles son las probabilidades de ganar en craps? [Nota: con el tiempo descubrirá que craps es uno de los juegos de casino más justos. ¿Qué cree usted que significa esto?]
- d) ¿Cuál es la duración promedio de un juego de craps?
- e) ¿Las probabilidades de ganar mejoran con la duración del juego?

**7.20** (*Sistema de reservaciones de una aerolínea*) Una pequeña aerolínea acaba de comprar una computadora para su nuevo sistema de reservaciones automatizado. Se le ha pedido a usted que desarrolle el nuevo sistema. Usted va a escribir una aplicación para asignar asientos en cada vuelo del único avión de la aerolínea (capacidad: 10 asientos).

Su programa debe mostrar el siguiente menú de alternativas: **Por favor escriba 1 para "Primera Clase" y Por favor escriba 2 para "Económico".** Si la persona escribe 1, su programa debe asignarle un asiento en la sección de primera clase (asientos 1 a 5). Si la persona escribe 2, su programa debe asignarle un asiento en la sección económica (asientos 6 a 10). Su programa deberá entonces imprimir un pase de abordaje, indicando el número de asiento de la persona y si se encuentra en la sección de primera clase o económica del avión.

Use un arreglo unidimensional para representar la tabla de asientos del avión. Inicialice todos los elementos del arreglo con 0 para indicar que todos los asientos están vacíos. A medida que se asigne cada asiento, establezca los elementos correspondientes del arreglo en 1 para indicar que ese asiento ya no está disponible.

Después de que su programa nunca deberá asignar un asiento que ya haya sido asignado. Cuando esté llena la sección de primera clase, su programa deberá preguntar a la persona si acepta ser colocada en la sección económica (y viceversa). Si la persona acepta, haga la asignación de asiento apropiada. Si no acepta, imprima el mensaje "**El próximo vuelo sale en 3 horas**".

**7.21** ¿Qué hace el siguiente programa?

```

1 // Ej. 7.21: ej07_21.cpp
2 // ¿Qué hace este programa?
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void unaFuncion(int [], int, int); // prototipo de función
8
9 int main()
10 {
11 const int tamanoArreglo = 10;
12 int a[tamanoArreglo] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

```

```

13
14 cout << "Los valores en el arreglo son:" << endl;
15 unaFuncion(a, 0, tamanoArreglo);
16 cout << endl;
17 return 0; // indica que terminó correctamente
18 } // fin de main
19
20 // ¿Qué hace esta función?
21 void unaFuncion(int b[], int actual, int tamano)
22 {
23 if (actual < tamano)
24 {
25 unaFuncion(b, actual + 1, tamano);
26 cout << b[actual] << " ";
27 } // fin de if
28 } // fin de la función unaFuncion

```

**7.22** Use un arreglo bidimensional para resolver el siguiente problema: una compañía tiene cuatro vendedores (1 a 4) que venden cinco productos distintos (1 a 5). Una vez al día, cada vendedor pasa una nota por cada tipo de producto vendido. Cada nota contiene lo siguiente:

- a) El número del vendedor.
- b) El número del producto.
- c) El valor total en dólares de ese producto vendido en ese día.

Así, cada vendedor pasa entre 0 y 5 notas de venta por día. Suponga que está disponible la información sobre todas las notas del mes pasado. Escriba un programa que lea toda esta información para las ventas del último mes y que resuma las ventas totales por vendedor, por producto. Todos los totales deben guardarse en el arreglo bidimensional *ventas*. Después de procesar toda la información del mes pasado, muestre los resultados en formato tabular, donde cada columna represente a un vendedor específico y cada fila represente a un producto. Saque el total de cada fila para obtener las ventas totales de cada producto durante el último mes. Saque el total de cada columna para obtener las ventas totales de cada producto durante el último mes. Su impresión tabular debe incluir estos totales cruzados a la derecha de las filas totalizadas, y en la parte inferior de las columnas totalizadas.

**7.23** (*Gráficos de tortuga*) El lenguaje Logo, popular entre los niños de escuelas primarias, hizo famoso el concepto de los *gráficos de tortuga*. Imagine a una tortuga mecánica que camina por todo el cuarto, bajo el control de un programa en C++. La tortuga sostiene una pluma en una de dos posiciones, arriba o abajo. Mientras la pluma está abajo, la tortuga va trazando figuras a medida que se va moviendo, y mientras la pluma está arriba, la tortuga se mueve alrededor libremente, sin trazar nada. En este problema usted simulará la operación de la tortuga y creará un bloc de dibujo computarizado.

Utilice un arreglo de 20 por 20 llamado *piso*, que se inicialice con ceros. Lea los comandos de un arreglo que los contenga. Lleve el registro de la posición actual de la tortuga en todo momento, y si la pluma se encuentra arriba o abajo. Suponga que la tortuga siempre empieza en la posición (0, 0) del piso, con su pluma hacia arriba. El conjunto de comandos de la tortuga que su aplicación debe procesar se muestra en la figura 7.33.

Suponga que la tortuga se encuentra en algún lado cerca del centro del piso. El siguiente “programa” dibuja e imprime un cuadrado de 12 por 12, dejando la pluma en posición levantada:

```

2
5,12
3
5,12
3
5,12
3
5,12
1
6
9

```

A medida que la tortuga se vaya desplazando con la pluma hacia abajo, asigne 1 a los elementos apropiados del arreglo *piso*. Cuando se dé el comando 6 (imprimir), siempre que haya un 1 en el arreglo muestre un asterisco o cualquier carácter que usted elija. Siempre que haya un 0, muestre un carácter en blanco. Escriba un programa para implementar las herramientas de gráficos de tortuga que describimos aquí. Escriba varios programas de gráficos de tortuga para dibujar figuras interesantes. Agregue otros comandos para incrementar el poder de su lenguaje de gráficos de tortuga.

| Comando | Significado                                                     |
|---------|-----------------------------------------------------------------|
| 1       | Pluma arriba                                                    |
| 2       | Pluma abajo                                                     |
| 3       | Voltear a la derecha                                            |
| 4       | Voltear a la izquierda                                          |
| 5,10    | Avanzar hacia adelante 10 espacios (o un número distinto de 10) |
| 6       | Imprimir el arreglo de 20 por 20                                |
| 9       | Fin de los datos (centinela)                                    |

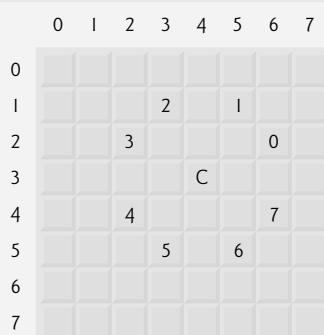
**Figura 7.33** | Comandos de gráficos de tortuga.

**7.24 (Paseo del caballo)** Uno de los enigmas más interesantes para los entusiastas del ajedrez es el problema del Paseo del caballo. La pregunta es: ¿puede la pieza de ajedrez, conocida como caballo, moverse alrededor de un tablero de ajedrez vacío y tocar cada una de las 64 posiciones una y sólo una vez? A continuación estudiaremos detalladamente este intrigante problema.

El caballo realiza solamente movimientos en forma de L (dos espacios en una dirección y un espacio en una dirección perpendicular). Por lo tanto, como se muestra en la figura 7.34, desde una posición cerca del centro de un tablero de ajedrez vacío, el caballo puede hacer ocho movimientos distintos (numerados del 0 al 7).

- Dibuja un tablero de ajedrez de 8 por 8 en una hoja de papel, e intente realizar un Paseo del caballo en forma manual. Ponga un 1 en la posición inicial, un 2 en la segunda posición, un 3 en la tercera, etc. Antes de empezar el paseo, estime qué tan lejos podrá avanzar, recordando que un paseo completo consta de 64 movimientos. ¿Qué tan lejos llegó? ¿Estuvo esto cerca de su estimación?
- Ahora desarrollaremos un programa para mover el caballo alrededor de un tablero de ajedrez. El tablero estará representado por un arreglo bidimensional llamado `tablero`, de ocho por ocho. Cada posición se inicializará con cero. Describiremos cada uno de los ocho posibles movimientos en términos de sus componentes horizontales y verticales. Por ejemplo, un movimiento de tipo 0, como se muestra en la figura 7.34, consiste en mover dos posiciones horizontalmente a la derecha y una posición verticalmente hacia arriba. Un movimiento de tipo 2 consiste en mover una posición horizontalmente a la izquierda y dos posiciones verticalmente hacia arriba. Los movimientos horizontal a la izquierda y vertical hacia arriba se indican con números negativos. Los ocho movimientos pueden describirse mediante dos arreglos unidimensionales llamados `horizontal` y `vertical`, de la siguiente manera:

```
horizontal[0] = 2 vertical[0] = -1
horizontal[1] = 1 vertical[1] = -2
horizontal[2] = -1 vertical[2] = -2
horizontal[3] = -2 vertical[3] = -1
horizontal[4] = -2 vertical[4] = 1
horizontal[5] = -1 vertical[5] = 2
horizontal[6] = 1 vertical[6] = 2
horizontal[7] = 2 vertical[7] = 1
```

**Figura 7.34** | Los ocho posibles movimientos del caballo.

Haga que las variables `filaActual` y `columnaActual` indiquen la fila y columna, respectivamente, de la posición actual del caballo. Para hacer un movimiento de tipo `numeroMovimiento`, donde `numeroMovimiento` puede estar entre 0 y 7, su programa debe utilizar las instrucciones

```
filaActual += vertical[numeroMovimiento];
columnaActual += horizontal[numeroMovimiento];
```

Utilice un contador que varíe de 1 a 64. Registre la última cuenta en cada posición a la que se mueva el caballo. Evalúe cada movimiento potencial para ver si el caballo ya visitó esa posición y, desde luego, pruebe cada movimiento potencial para asegurarse que el caballo no se salga del tablero de ajedrez. Ahora escriba un programa para desplazar el caballo por el tablero. Ejecute el programa. ¿Cuántos movimientos hizo el caballo?

- c) Después de intentar escribir y ejecutar un programa de Paseo del caballo, probablemente haya desarrollado algunas ideas valiosas. Utilizaremos estas ideas para desarrollar una **heurística** (o estrategia) para mover el caballo. La heurística no garantiza el éxito, pero una heurística cuidadosamente desarrollada mejora considerablemente la probabilidad de tener éxito. Probablemente usted ya observó que las posiciones externas son más difíciles que las posiciones cercanas al centro del tablero. De hecho, las posiciones más difíciles o inaccesibles son las cuatro esquinas.

La intuición sugiere que usted debe intentar mover primero el caballo a las posiciones más problemáticas y dejar pendientes aquellas a las que sea más fácil llegar, de manera que cuando el tablero se congestionne cerca del final del paseo, habrá una mayor probabilidad de éxito.

Podríamos desarrollar una “heurística de accesibilidad” clasificando cada una de las posiciones de acuerdo a qué tan accesibles son y luego mover siempre el caballo (usando los movimientos en L del caballo) a la posición más inaccesible. Etiquetaremos un arreglo bidimensional llamado `accesibilidad` con números que indiquen desde cuántas posiciones es accesible una posición determinada. En un tablero de ajedrez en blanco, cada una de las 16 posiciones más cercanas al centro se clasifican con 8; cada posición en la esquina se clasifica con 2; y las demás posiciones tienen números de accesibilidad 3, 4 o 6, de la siguiente manera:

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 4 | 4 | 4 | 4 | 3 | 2 |
| 3 | 4 | 6 | 6 | 6 | 6 | 4 | 3 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 4 | 6 | 8 | 8 | 8 | 8 | 6 | 4 |
| 3 | 4 | 6 | 6 | 6 | 6 | 4 | 3 |
| 2 | 3 | 4 | 4 | 4 | 3 | 2 | 2 |

Escriba una nueva versión del Paseo del caballo, utilizando la heurística de accesibilidad. El caballo deberá moverse siempre a la posición con el número de accesibilidad más bajo. En caso de un empate, el caballo podrá moverse a cualquiera de las posiciones empatadas. Por lo tanto, el paseo puede empezar en cualquiera de las cuatro esquinas. [Nota: al ir moviendo el caballo alrededor del tablero, su aplicación deberá reducir los números de accesibilidad a medida que se vayan ocupando más posiciones. De esta manera y en cualquier momento dado durante el paseo, el número de accesibilidad de cada una de las posiciones disponibles seguirá siendo igual al número preciso de posiciones desde las que se puede llegar a esa posición]. Ejecute esta versión de su programa. ¿Logró completar el paseo? Ahora modifique el programa para realizar 64 paseos, donde cada uno empiece desde una posición distinta en el tablero. ¿Cuántos paseos completos logró realizar?

- d) Escriba una versión del programa del Paseo del caballo que, al encontrarse con un empate entre dos o más posiciones, decida qué posición elegir buscando más adelante aquellas posiciones que se puedan alcanzar desde las posiciones “empatadas”. Su aplicación debe mover el caballo a la posición empatada para la cual el siguiente movimiento lo lleve a una posición con el número de accesibilidad más bajo.

**7.25 (Paseo del caballo: métodos de fuerza bruta)** En el ejercicio 7.24 desarrollamos una solución al problema del Paseo del caballo. El método utilizado, llamado “heurística de accesibilidad”, genera muchas soluciones y se ejecuta con eficiencia.

A medida que se incremente de manera continua la potencia de las computadoras, seremos capaces de resolver más problemas con menos potencia y algoritmos relativamente menos sofisticados. A éste le podemos llamar el método de la “fuerza bruta” para resolver problemas.

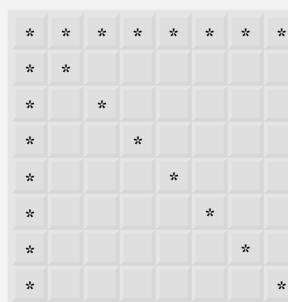
- Utilice la generación de números aleatorios para permitir que el caballo se desplace a lo largo del tablero (mediante sus movimientos legítimos en L) en forma aleatoria. Su programa debe ejecutar un paseo e imprimir el tablero final. ¿Qué tan lejos llegó el caballo?
- La mayoría de las veces, el programa anterior produce un paseo relativamente corto. Ahora modifique su aplicación para intentar 1000 paseos. Use un arreglo unidimensional para llevar el registro del número de paseos de cada longitud. Cuando su programa termine de intentar los 1000 paseos, deberá imprimir esta información en un formato tabular ordenado. ¿Cuál fue el mejor resultado?

- c) Es muy probable que el programa anterior le haya brindado algunos paseos “respetables”, pero no completos. Ahora deje que su aplicación se ejecute hasta que produzca un paseo completo. [Precaución: esta versión del programa podría ejecutarse durante horas en una computadora poderosa]. Una vez más, mantenga una tabla del número de paseos de cada longitud e imprímala cuando se encuentre el primer paseo completo. ¿Cuántos paseos intentó su programa antes de producir uno completo? ¿Cuánto tiempo se tomó?
- d) Compare la versión de la fuerza bruta del Paseo del caballo con la versión heurística de accesibilidad. ¿Cuál requirió un estudio más cuidadoso del problema? ¿Qué algoritmo fue más difícil de desarrollar? ¿Cuál requirió más poder de cómputo? ¿Podríamos tener la certeza (por adelantado) de obtener un paseo completo mediante el método de la heurística de accesibilidad? ¿Podríamos tener la certeza (por adelantado) de obtener un paseo completo mediante el método de la fuerza bruta? Argumente las ventajas y desventajas de solucionar el problema mediante la fuerza bruta en general.

**7.26** (*Ocho reinas*) Otro enigma para los entusiastas del ajedrez es el problema de las Ocho reinas, el cual pregunta lo siguiente: ¿es posible colocar ocho reinas en un tablero de ajedrez vacío, de tal manera que ninguna reina “ataque” a cualquier otra (es decir, que no haya dos reinas en la misma fila, en la misma columna o a lo largo de la misma diagonal)? Use la idea desarrollada en el ejercicio 7.24 para formular una heurística para resolver el problema de las Ocho reinas. Ejecute su programa. [Sugerencia: es posible asignar un valor a cada una de las posiciones en el tablero de ajedrez, para indicar cuántas posiciones de un tablero vacío se “eliminan” si una reina se coloca en esa posición. A cada una de las esquinas se le asignaría el valor 22, como se demuestra en la figura 7.35]. Una vez que estos “números de eliminación” se coloquen en las 64 posiciones, una heurística apropiada podría ser la siguiente: coloque la siguiente reina en la posición con el número de eliminación más pequeño. ¿Por qué esta estrategia es intuitivamente atractiva?]

**7.27** (*Ocho reinas: métodos de fuerza bruta*) En este ejercicio usted desarrollará varios métodos de fuerza bruta para resolver el problema de las Ocho reinas que presentamos en el ejercicio 7.26.

- a) Utilice la técnica de la fuerza bruta aleatoria desarrollada en el ejercicio 7.25, para resolver el problema de las Ocho reinas.
- b) Utilice una técnica exhaustiva (es decir, pruebe todas las combinaciones posibles de las ocho reinas en el tablero).
- c) ¿Por qué el método de la fuerza bruta exhaustiva podría no ser apropiado para resolver el problema del Paseo del caballo?
- d) Compare y contraste el método de la fuerza bruta aleatoria con el de la fuerza bruta exhaustiva.



**Figura 7.35** | Las 22 posiciones eliminadas al colocar una reina en la esquina superior izquierda.

**7.28** (*Paseo del caballo: prueba del paseo cerrado*) En el Paseo del caballo se lleva a cabo un paseo completo cuando el caballo hace 64 movimientos, en los que toca cada esquina del tablero una sola vez. Un paseo cerrado ocurre cuando el movimiento 64 se encuentra a un movimiento de distancia de la posición en la que el caballo empezó el paseo. Modifique el programa del Paseo del caballo que escribió en el ejercicio 7.24 para probar si el paseo ha sido completo, y si se trató de un paseo cerrado.

**7.29** (*La criba de Eratóstenes*) Un entero primo es cualquier entero mayor que 1, divisible sólo por sí mismo y por el número 1. La Criba de Eratóstenes es un método para encontrar números primos, el cual opera de la siguiente manera:

- a) Cree un arreglo con todos los elementos inicializados en 1 (verdadero). Los elementos del arreglo con subíndices primos permanecerán como 1. Cualquier otro elemento del arreglo eventualmente cambiará a cero. En este ejercicio, ignoraremos los elementos 0 y 1.
- b) Empezando con el subíndice 2 del arreglo, cada vez que se encuentre un elemento del arreglo cuyo valor sea 1, itere a través del resto del arreglo y asigne cero a todo elemento cuyo subíndice sea múltiplo del subíndice del elemento que tiene el valor 1. Para el subíndice 2 del arreglo, todos los elementos más allá del elemento 2 en el arreglo que

tengan subíndices múltiplos de 2 (los índices 4, 6, 8, 10, etcétera) se establecerán en cero; para el subíndice 3 del arreglo, todos los elementos más allá del elemento 3 en el arreglo que sean múltiplos de 3 (los índices 6, 9, 12, 15, etcétera) se establecerán en cero; y así sucesivamente.

Cuando este proceso termine, los elementos del arreglo que aún sean uno indicarán que el subíndice es un número primo. Estos subíndices pueden entonces imprimirse. Escriba un programa que utilice un arreglo de 1000 elementos para determinar e imprimir los números primos entre 2 y 999. Ignore el elemento 0 del arreglo.

**7.30 (Ordenamiento de cubeta)** Un **ordenamiento de cubeta** comienza con un arreglo unidimensional de enteros positivos que se deben ordenar, y un arreglo bidimensional de enteros, en el que las filas están indexadas de 0 a 9 y las columnas de 0 a  $n - 1$ , donde  $n$  es el número de valores a ordenar. Cada fila del arreglo bidimensional se conoce como una cubeta. Escriba una función llamada **ordenamientoCubeta**, que reciba un arreglo de enteros y el tamaño del arreglo como argumentos, y que además opere de la siguiente manera:

- Coloque cada valor del arreglo unidimensional en una fila del arreglo de cubeta, con base en el dígito de las unidades del valor. Por ejemplo, el número 97 se coloca en la fila 7, el 3 se coloca en la fila 3 y el 100 se coloca en la fila 0. A este procedimiento se le llama “pasada de distribución”.
- Itere a través del arreglo de cubeta fila por fila, y copie los valores de vuelta al arreglo original. A este procedimiento se le llama “pasada de recopilación”. El nuevo orden de los valores anteriores en el arreglo unidimensional es 100, 3 y 97.
- Repita este proceso para cada posición de dígito subsiguiente (decenas, centenas, miles, etc.).

En la segunda pasada se coloca el 100 en la fila 0, el 3 en la fila 0 (ya que 3 no tiene dígito de decenas) y el 97 en la fila 9. Después de la pasada de recopilación, el orden de los valores en el arreglo unidimensional es 100, 3 y 97. En la tercera pasada (dígito de las centenas), el 100 se coloca en la fila 1, el 3 en la fila 0 y el 97 en la fila 0 (después del 3). Después de esta última pasada de recopilación, el arreglo original se encuentra en orden.

Observe que el arreglo bidimensional de cubetas es 10 veces el tamaño del arreglo entero que se está ordenando. Esta técnica de ordenamiento proporciona un mejor rendimiento que el ordenamiento por inserción, pero requiere mucha más memoria. El ordenamiento por inserción requiere espacio sólo para un elemento adicional de datos. Éste es un ejemplo de la concesión entre espacio y tiempo: el ordenamiento de cubeta utiliza más memoria que el ordenamiento por inserción, pero su rendimiento es mejor. Esta versión del ordenamiento de cubeta requiere copiar todos los datos de vuelta al arreglo original en cada pasada. Otra posibilidad es crear un segundo arreglo de cubeta bidimensional, e intercambiar en forma repetida los datos entre los dos arreglos de cubeta.

## Ejercicios de recursividad

**7.31 (Ordenamiento por selección)** Un **ordenamiento por selección** busca el elemento más pequeño en un arreglo. Después, el elemento más pequeño se intercambia con el primer elemento del arreglo. El proceso se repite para el subarreglo que empieza con el segundo elemento del arreglo. Cada pasada del arreglo ocasiona que un elemento se coloque en su posición apropiada. Este ordenamiento tiene un rendimiento comparable al del ordenamiento por inserción; para un arreglo de  $n$  elementos deben realizarse  $n - 1$  pasadas, y para cada subarreglo, se deben realizar  $n - 1$  comparaciones para encontrar el valor más pequeño. Cuando el subarreglo que se está procesando contiene un elemento, el arreglo está ordenado. Escriba la función recursiva **ordenamientoSelección** para realizar este algoritmo.

**7.32 (Palíndromos)** Un palíndromo es una cadena que se escribe de la misma forma tanto al derecho como al revés. Algunos ejemplos de palíndromos son “radar”, “reconocer” y (si se ignoran los espacios) “anita lava la tina”. Escriba una función recursiva llamada **probarPalindromo**, que devuelva **true** si la cadena almacenada en el arreglo es un palíndromo, y **false** en caso contrario. El método debe ignorar espacios y puntuación en la cadena.

**7.33 (Búsqueda lineal)** Modifique el programa de la figura 7.19 para utilizar la función recursiva **busquedaLinealRecursiva** para realizar una búsqueda lineal en el arreglo. La función debe recibir un arreglo entero y el tamaño del arreglo como argumentos. Si se encuentra la clave de búsqueda, se devuelve el subíndice del arreglo; en caso contrario, se devuelve -1.

**7.34 (Ocho reinas)** Modifique el programa de las Ocho reinas que creó en el ejercicio 7.26 para resolver el problema en forma recursiva.

**7.35 (Imprimir un arreglo)** Escriba una función recursiva llamada **imprimirArreglo** que reciba un arreglo, un subíndice inicial y un subíndice final como argumentos, y que no devuelva nada. La función deberá dejar de procesar y deberá regresar cuando el subíndice inicial sea igual al subíndice final.

**7.36 (Imprimir una cadena en forma inversa)** Escriba una función recursiva llamada **cadenaInversa**, que reciba un arreglo de caracteres que contenga una cadena y un subíndice inicial como argumentos, imprima la cadena en forma inversa y no devuelva nada. La función deberá dejar de procesar y deberá regresar al encontrar la cadena nula de terminación.

**7.37 (Buscar el valor mínimo en un arreglo)** Escriba una función recursiva llamada **minimoRecursivo** que reciba un arreglo de enteros, un subíndice inicial y un subíndice final como argumentos, y que devuelva el elemento más pequeño del arreglo. La función deberá dejar de procesar y deberá regresar al encontrar la cadena nula de terminación.

## Ejercicios con vector

- 7.38** Use un **vector** de enteros para resolver el problema descrito en el ejercicio 7.10.
- 7.39** Modifique el programa para tirar dados que creó en el ejercicio 7.17, de manera que utilice un **vector** para almacenar el número de veces que aparece cada posible suma de los dos lados.
- 7.40** (*Buscar el valor mínimo en un vector*) Modifique su solución al ejercicio 7.37 para buscar el valor mínimo en un **vector**, en lugar de hacerlo en un arreglo.

# 8



*Recibimos direcciones para ocultar nuestro paradero.*

—Saki (H. H. Munro)

*Averigua la dirección mediante la indirección.*

—William Shakespeare

*Muchas cosas, teniendo referencia completa a un consentimiento, pueden trabajar en forma contraria.*

—William Shakespeare

*¡Descubrirá que es una muy buena práctica verificar siempre sus referencias, señor!*

—Dr. Routh

## Apuntadores y cadenas basadas en apuntadores

### OBJETIVOS

En este capítulo aprenderá a:

- Distinguir qué son los apuntadores.
- Conocer las similitudes y diferencias entre los apuntadores y las referencias, y cuándo utilizar cada uno de estos elementos.
- Utilizar apuntadores para pasar argumentos a las funciones por referencia.
- Utilizar cadenas basadas en apuntador estilo C.
- Conocer las estrechas relaciones entre los apuntadores, los arreglos y las cadenas estilo C.
- Utilizar apuntadores a funciones.
- Declarar y usar arreglos de cadenas estilo C.

- 8.1 Introducción**
- 8.2 Declaraciones e inicialización de variables apuntadores**
- 8.3 Operadores de apuntadores**
- 8.4 Paso de argumentos a funciones por referencia mediante apuntadores**
- 8.5 Uso de `const` con apuntadores**
- 8.6 Ordenamiento por selección mediante el uso del paso por referencia**
- 8.7 Operador `sizeof`**
- 8.8 Expresiones y aritmética de apuntadores**
- 8.9 Relación entre apuntadores y arreglos**
- 8.10 Arreglos de apuntadores**
- 8.11 Ejemplo práctico: simulación para barajar y repartir cartas**
- 8.12 Apuntadores a funciones**
- 8.13 Introducción al procesamiento de cadenas basadas en apuntador**
  - 8.13.1 Fundamentos de caracteres y cadenas basadas en apuntador**
  - 8.13.2 Funciones para manipular cadenas de la biblioteca para manejo de cadenas**
- 8.14 Repaso**

[Resumen](#) | 
 [Terminología](#) | 
 [Ejercicios de autoevaluación](#) | 
 [Respuestas a los ejercicios de autoevaluación](#) | 
 [Ejercicios](#)  
 Sección especial: construya su propia computadora | 
 Más ejercicios de apuntadores | 
 Ejercicios de manipulación de cadenas |  
 Sección especial: ejercicios avanzados de manipulación de cadenas | 
 Un proyecto desafiante sobre manipulación de cadenas

## 8.1 Introducción

En este capítulo hablaremos sobre una de las características más poderosas del lenguaje de programación C++: el apuntador. En el capítulo 6 vimos que se pueden utilizar referencias para realizar el paso por referencia. Los apuntadores también permiten el paso por referencia, y se pueden utilizar para crear y manipular estructuras dinámicas de datos (es decir, estructuras de datos que pueden crecer y reducirse), como listas enlazadas, colas, pilas y árboles. En este capítulo explicaremos los conceptos básicos sobre los apuntadores y reforzaremos la estrecha relación entre arreglos y apuntadores. La forma de ver los arreglos como apuntadores se deriva del lenguaje de programación C. Como vimos en el capítulo 7, la clase `vector` de la Biblioteca estándar de C++ proporciona una implementación de los arreglos como objetos completos.

De manera similar, C++ en realidad ofrece dos tipos de cadenas: objetos de la clase `string` (que hemos estado usando desde el capítulo 3) y cadenas `char *` basadas en apuntador, estilo C. Este capítulo sobre apuntadores describe las cadenas `char *` para que el lector amplíe su conocimiento sobre los apuntadores. De hecho, las cadenas con terminación nula que presentamos en la sección 7.4 y utilizamos en la figura 7.12 son cadenas `char *` basadas en apuntador. En este capítulo también se incluye una gran colección de ejercicios de procesamiento de cadenas que utilizan cadenas `char *`. Las cadenas `char *` basadas en apuntador estilo C se utilizan ampliamente en sistemas de C y C++ heredados. Por lo tanto, si usted trabaja con sistemas de C o C++ heredados, tal vez tenga que manipular estas cadenas `char *` basadas en apuntador.

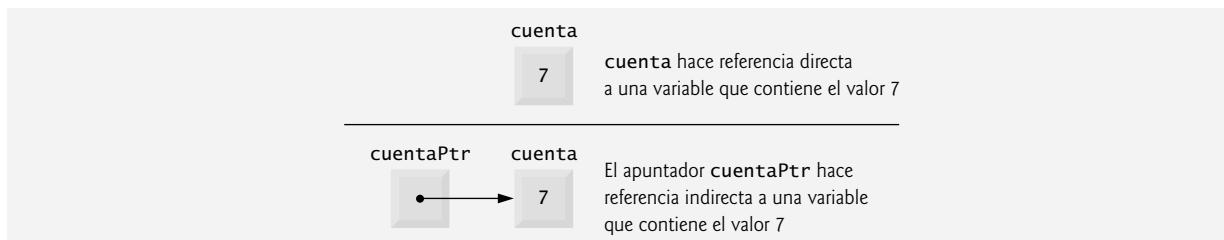
En el capítulo 13, Programación orientada a objetos: polimorfismo, examinaremos el uso de los apuntadores con las clases. En ese capítulo veremos que lo que se denomina “procesamiento polimórfico” de la programación orientada a objetos se lleva a cabo mediante apuntadores y referencias. En el capítulo 20, Estructuras de datos, presentaremos ejemplos acerca de cómo crear y utilizar estructuras dinámicas de datos que se implementan con apuntadores.

## 8.2 Declaraciones e inicialización de variables apuntadores

Las variables apuntadores contienen direcciones de memoria como sus valores. Por lo general, una variable contiene directamente un valor específico. Sin embargo, un apuntador contiene la dirección de memoria de una variable que, a su vez, contiene un valor específico. En este sentido, el nombre de una variable **hace referencia directa a un valor**, y un apuntador **hace referencia indirecta a un valor** (figura 8.1). Al proceso de hacer referencia a un valor a través de un apuntador se le conoce comúnmente como **indirección**. Observe que, por lo general, los diagramas representan un apuntador en forma de una flecha que parte de la variable que contiene una dirección, hasta la variable ubicada en esa dirección de memoria.

Al igual que las demás variables, los apuntadores se deben declarar antes de poder usarlos. Por ejemplo, para el apuntador en la figura 8.1, la declaración

```
int *cuentaPtr, cuenta;
```



**Figura 8.1** | Referencia directa e indirecta a una variable.

declara a la variable `cuentaPtr` como de tipo `int *` (es decir, un apuntador a un valor `int`) y se lee así: “`cuentaPtr` es un apuntador a un valor `int`” o “`cuentaPtr` apunta a un objeto de tipo `int`”. Además, la variable `cuenta` en la declaración anterior se declara como un `int`, y no como un apuntador a un `int`. El `*` en la declaración se aplica sólo a `cuentaPtr`. A cada variable que se declara como apuntador se le debe anteponer un asterisco (`*`). Por ejemplo, la declaración

```
double *xPtr, *yPtr;
```

indica que tanto `xPtr` como `yPtr` son apuntadores a valores `double`. Cuando el `*` aparece en una declaración, no es un operador; en lugar de ello, indica que la variable que se está declarando es un apuntador. Los apuntadores se pueden declarar de manera que apunten a objetos de cualquier tipo de datos.

### Error común de programación 8.1



*Asumir que el `*` que se utiliza para declarar a un apuntador se distribuye a todos los nombres de variables en la lista separada por comas de variables de una declaración puede provocar errores. Cada apuntador se debe declarar con el `*` antepuesto al nombre (ya sea con o sin un espacio entre ellos; el compilador ignora el espacio). Si declaramos sólo una variable por cada declaración, evitamos estos tipos de errores y mejoramos la legibilidad de los programas.*

### Buena práctica de programación 8.1



*Aunque no es un requerimiento, incluir las letras `Ptr` en los nombres de las variables apuntadores deja claro que estas variables son apuntadores, y que deben tratarse de manera acorde.*

Los apuntadores se deben inicializar ya sea cuando se declaran, o en una asignación. Un apuntador se debe inicializar con 0, `NULL` o una dirección del tipo correspondiente. Un apuntador con el valor 0 o `NULL` no apunta a nada, y se conoce como **apuntador nulo**. La constante simbólica `NULL` se define en el archivo de encabezado `<iostream>` (y en varios archivos de encabezado más de la biblioteca estándar) para representar el valor 0. Inicializar un apuntador con `NULL` es equivalente a inicializar un apuntador con 0, pero en C++, se utiliza 0 por convención. Cuando se asigna 0, se convierte en un apuntador del tipo apropiado. El valor 0 es el único valor entero que puede asignarse directamente a una variable apuntador sin tener que convertir primero el entero en un tipo apuntador. En la sección 8.3 veremos cómo asignar una dirección numérica de una variable a un apuntador.

### Tip para prevenir errores 8.1



*Inicialice los apuntadores para evitar que apunten hacia áreas desconocidas o no inicializadas de la memoria.*

## 8.3 Operadores de apuntadores

El **operador dirección (&)** es un operador unario que obtiene la dirección de memoria de su operando. Por ejemplo, teniendo en cuenta las siguientes declaraciones:

```
int y = 5; // declara la variable y
int *yPtr; // declara la variable apuntador yPtr
```

la instrucción

```
yPtr = &y; // asigna la dirección de y a yPtr
```

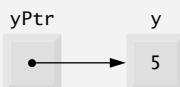
asigna la dirección de la variable `y` a la variable apuntador `yPtr`. Entonces, se dice que la variable `yPtr` “apunta a” `y`. Ahora, `yPtr` hace referencia indirecta al valor de la variable `y`. Observe que el uso del signo `&` en la instrucción anterior no es el mismo que el uso del `&` en la declaración de una variable de referencia, a la cual siempre se le antepone el nombre de un tipo de datos. Al declarar una referencia, el `&` forma parte del tipo. En una expresión como `&y`, el `&` es un operador.

La figura 8.2 muestra una representación esquemática de la memoria después de la anterior asignación. La “relación de señalamiento” se indica mediante el dibujo de una flecha desde el cuadro que representa al apuntador `yPtr` en la memoria, hasta el cuadro que representa a la variable `y` en la memoria.

La figura 8.3 muestra otra representación del apuntador en memoria, suponiendo que la variable entera `y` se almacena en la ubicación de memoria 600000 y que la variable apuntador `yPtr` se almacena en la ubicación de memoria 500000. El operando del operador dirección debe ser un *lvalue* (es decir, algo a lo que se pueda asignar un valor, como el nombre de una variable o una referencia); el operador dirección no se puede aplicar a constantes o expresiones que no den como resultado referencias.

El operador `*`, que se conoce comúnmente como el **operador de indirección** u **operador de desreferencia**, devuelve un sinónimo (es decir, un alias o sobrenombre) para el objeto al que apunta su operando apuntador. Por ejemplo (haciendo referencia otra vez a la figura 8.2), la instrucción

```
cout << *yPtr << endl;
```



**Figura 8.2** | Representación gráfica de un apuntador que apunta a una variable en memoria.



**Figura 8.3** | Representación de `y` y `yPtr` en memoria.

imprime el valor de la variable `y` (en este caso, 5), al igual que la instrucción

```
cout << y << endl;
```

Al proceso de utilizar el `*` de esta manera, se le conoce como **desreferenciar un apuntador**. Observe que un apuntador desreferenciado también se puede usar en el lado izquierdo de una instrucción de asignación, como en

```
*yPtr = 9;
```

lo cual asignaría 9 a `y` en la figura 8.3. El apuntador desreferenciado también se puede utilizar para recibir un valor de entrada, como en

```
cin >> *yPtr;
```

lo cual coloca el valor de entrada en `y`. El apuntador desreferenciado es un *lvalue*.



### Error común de programación 8.2

Al desreferenciar un apuntador que no se haya inicializado apropiadamente, o que no se haya asignado para apuntar a una ubicación específica en memoria, se podría producir un error fatal en tiempo de ejecución, o se podrían modificar datos de manera accidental y permitir que el programa se ejecutara por completo, lo que posiblemente produzca resultados incorrectos.



### Error común de programación 8.3

Tratar de desreferenciar una variable que no sea apuntador es un error de compilación.



### Error común de programación 8.4

Desreferenciar un apuntador nulo es a menudo un error fatal en tiempo de ejecución.

El programa de la figura 8.4 demuestra los operadores `&` y `*` para apuntadores. Las ubicaciones de memoria se imprimen mediante `<<` en este ejemplo como enteros hexadecimales (base 16). (En el apéndice D, Sistemas numéricos, podrá obtener más información acerca de los enteros hexadecimales). Observe que las direcciones de memoria hexadecimales que imprime este programa son dependientes del compilador y del sistema operativo, por lo que tal vez usted obtenga distintos resultados cuando ejecute el programa.

```

1 // Fig. 8.4: fig08_04.cpp
2 // Los operadores & y * de los apuntadores.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 int a; // a es un entero
10 int *aPtr; // aPtr es un int * -- apuntador a un entero
11
12 a = 7; // se asigna 7 a la variable a
13 aPtr = &a; // se asigna la dirección de a a aPtr
14
15 cout << "La dirección de a es " << &a
16 << "\nEl valor de aPtr es " << aPtr;
17 cout << "\n\nEl valor de a es " << a
18 << "\nEl valor de *aPtr es " << *aPtr;
19 cout << "\n\nDemostración de que * y & son inversos "
20 << "uno del otro.\n&aPtr = " << &aPtr
21 << "\n\n*aPtr = " << *aPtr << endl;
22 return 0; // indica que terminó correctamente
23 } // fin de main

```

```

La dirección de a es 0012FF60
El valor de aPtr es 0012FF60

El valor de a es 7
El valor de *aPtr es 7

Demostración de que * y & son inversos uno del otro.
&aPtr = 0012FF60
*aPtr = 0012FF60

```

**Figura 8.4** | Los operadores & y \* de los apuntadores.



### Tip de portabilidad 8.1

*El formato en el que se imprime un apuntador es dependiente del compilador. Algunos imprimen los valores de apuntadores como enteros hexadecimales, algunos utilizan enteros decimales y algunos usan otros formatos.*

Observe que la dirección de a (línea 15) y el valor de aPtr (línea 16) son idénticos en la salida, lo cual confirma que la dirección de a se asigna sin duda a la variable apuntador aPtr. Los operadores & y \* son los inversos uno del otro; cuando ambos se aplican en forma consecutiva a aPtr en cualquier orden, pueden “cancelar uno al otro” y se imprime el mismo resultado (el valor en aPtr).

La figura 8.5 lista la precedencia y asociatividad de los operadores presentados hasta ahora. Observe que el operador dirección (&) y el operador desreferencia (\*) son operadores unarios en el tercer nivel de precedencia en la tabla.

| Operadores                        | Asociatividad       | Tipo                 |
|-----------------------------------|---------------------|----------------------|
| O []                              | izquierda a derecha | mayor                |
| ++ -- static_cast<tipo>(operando) | izquierda a derecha | unario (postfijo)    |
| ++ -- + - ! & *                   | derecha a izquierda | unario (prefijo)     |
| * / %                             | izquierda a derecha | multiplicativa       |
| + -                               | izquierda a derecha | aditiva              |
| << >>                             | izquierda a derecha | inserción/extracción |
| < <= > >=                         | izquierda a derecha | relacional           |
| == !=                             | izquierda a derecha | Igualdad             |

**Figura 8.5** | Precedencia y asociatividad de los operadores. (Parte I de 2).

| Operadores                     | Asociatividad       | Tipo        |
|--------------------------------|---------------------|-------------|
| &&                             | izquierda a derecha | AND lógico  |
|                                | izquierda a derecha | OR lógico   |
| :=                             | derecha a izquierda | condicional |
| =    +=    -=    *=    /=    % | derecha a izquierda | asignación  |
| ,                              | izquierda a derecha | coma        |

Figura 8.5 | Precedencia y asociatividad de los operadores. (Parte 2 de 2).

## 8.4 Paso de argumentos a funciones por referencia mediante apuntadores

En C++ hay tres formas de pasar argumentos a una función: **paso por valor**, **paso por referencia con argumentos tipo referencia** y **paso por referencia con argumentos tipo apuntador**. En el capítulo 6 comparamos y contrastamos el paso por valor y el paso por referencia con argumentos tipo referencia. En esta sección explicaremos el paso por referencia con argumentos tipo apuntador.

Como vimos en el capítulo 6, se puede utilizar `return` para devolver un valor de una función llamada a la función que la llamó (o se puede devolver el control de una función llamada sin devolver un valor). También vimos que se pueden pasar argumentos a una función mediante el uso de argumentos tipo referencia. Dichos argumentos permiten a la función llamada modificar los valores originales de los argumentos en la función que hizo la llamada. Los argumentos tipo referencia también permiten a los programas pasar objetos de datos grandes a una función, y evitar la sobrecarga de pasar los objetos por valor (que, desde luego, requiere de la creación de una copia del objeto). Al igual que las referencias, los apuntadores también se pueden usar para modificar una o más variables en la función que hace la llamada, o pasar apuntadores a objetos de datos grandes para evitar la sobrecarga de pasar los objetos por valor.

En C++, los programadores pueden usar apuntadores y el operador indirección (\*) para realizar el paso por referencia (en forma idéntica al paso por referencia en los programas en C, ya que éste no tiene referencias). Cuando se llama a una función con un argumento que se debe modificar, se pasa la dirección del argumento. Por lo general, para realizar esto se aplica el operador dirección (&) al nombre de la variable cuyo valor se va a modificar.

Como vimos en el capítulo 7, los arreglos no se pasan usando el operador &, ya que el nombre del arreglo es la ubicación inicial en memoria del mismo (es decir, el nombre de un arreglo ya es un apuntador). El nombre de un arreglo, `nombreArreglo`, es equivalente a `&nombreArreglo[ 0 ]`. Cuando se pasa la dirección de una variable a una función, el operador indirección (\*) se puede usar en la función para formar un sinónimo para el nombre de la variable; esto a su vez se puede utilizar para modificar el valor de la variable en esa ubicación en la memoria de la función que hizo la llamada.

Las figuras 8.6 y 8.7 presentan dos versiones de una función que eleva un entero al cubo: `cuboPorValor` y `cuboPorReferencia`. La figura 8.6 pasa la variable `numero` por valor a la función `cuboPorValor` (línea 15). La función `cuboPorValor` (líneas 21 a 24) eleva su argumento al cubo y pasa el nuevo valor de vuelta a `main`, usando una instrucción `return` (línea 23). El nuevo valor se asigna a `numero` (línea 15) en `main`. Observe que la función que llama tiene

```

1 // Fig. 8.6: fig08_06.cpp
2 // Uso del paso por valor para elevar al cubo el valor de una variable.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int cuboPorValor(int); // prototipo
8
9 int main()
10 {
11 int numero = 5;
12
13 cout << "El valor original de numero es " << numero;
14
15 numero = cuboPorValor(numero); // pasa el numero por valor a cuboPorValor

```

Figura 8.6 | Uso del paso por valor para elevar al cubo el valor de una variable. (Parte 1 de 2).

```

16 cout << "\nEl nuevo valor de numero es " << numero << endl;
17 return 0; // indica que terminó correctamente
18 } // fin de main
19
20 // calcula y devuelve el cubo del argumento entero
21 int cuboPorValor(int n)
22 {
23 return n * n * n; // eleva al cubo la variable local n y devuelve el resultado
24 } // fin de la función cuboPorValor

```

El valor original de numero es 5  
 El nuevo valor de numero es 125

**Figura 8.6** | Uso del paso por valor para elevar al cubo el valor de una variable. (Parte 2 de 2).

la oportunidad de examinar el resultado de la llamada a la función antes de modificar el valor de la variable `numero`. Por ejemplo, en este programa podríamos haber almacenado el resultado de `cuboPorValor` en otra variable, para después examinar su valor y asignar el resultado a `numero`, sólo después de determinar que el valor devuelto era razonable.

En la figura 8.7 se pasa la variable `numero` a la función `cuboPorReferencia` mediante el uso del paso por referencia con un argumento tipo apuntador (línea 16); la dirección de `numero` se pasa a la función. La función `cuboPorReferencia` (líneas 23 a 26) especifica el parámetro `nPtr` (un apuntador a `int`) para recibir su argumento. La función desreferencia el apuntador y eleva al cubo el valor al que apunta `nPtr` (línea 25). Esto modifica directamente el valor de `numero` en `main`.

### Error común de programación 8.5



*Si no se desreferencia un apuntador cuando es necesario hacerlo, para obtener el valor al que apunta, se produce un error.*

Una función que recibe una dirección como argumento debe definir un parámetro tipo apuntador para recibir la dirección. Por ejemplo, el encabezado para la función `cuboPorReferencia` (línea 23) especifica que `cuboPorReferencia` debe recibir la dirección de una variable `int` (es decir, un apuntador a un `int`) como argumento, debe almacenar la dirección en forma local en `nPtr` y no debe devolver un valor.

```

1 // Fig. 8.7: fig08_07.cpp
2 // Uso del paso por referencia con un argumento apuntador para elevar
3 // al cubo el valor de una variable.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 void cuboPorReferencia(int *); // prototipo
9
10 int main()
11 {
12 int numero = 5;
13
14 cout << "El valor original de numero es " << numero;
15
16 cuboPorReferencia(&numero); // pasa la dirección de numero a cuboPorReferencia
17
18 cout << "\nEl nuevo valor de numero es " << numero << endl;
19 return 0; // indica que terminó correctamente
20 } // fin de main
21
22 // calcula el cubo de *nPtr; modifica la variable numero en main
23 void cuboPorReferencia(int *nPtr)
24 {
25 *nPtr = *nPtr * *nPtr * *nPtr; // eleva *nPtr al cubo
26 } // fin de la función cuboPorReferencia

```

**Figura 8.7** | Uso del paso por referencia con un argumento tipo apuntador para elevar al cubo el valor de una variable. (Parte 1 de 2).

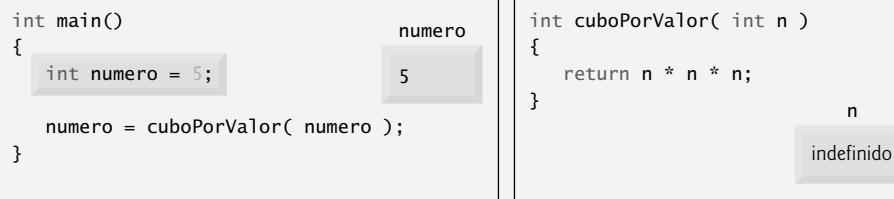
```
El valor original de numero es 5
El nuevo valor de numero es 125
```

**Figura 8.7** | Uso del paso por referencia con un argumento tipo apuntador para elevar al cubo el valor de una variable. (Parte 2 de 2).

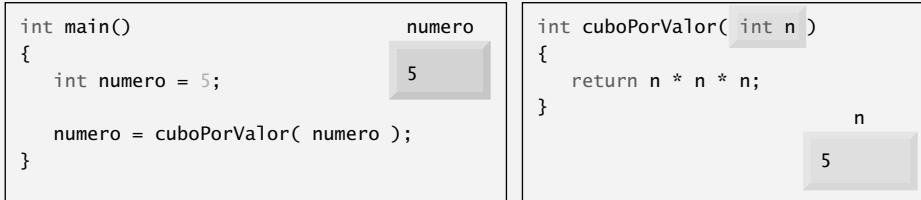
El prototipo de función para `cuboPorReferencia` (línea 8) contiene `int *` entre paréntesis. Al igual que con otros tipos de variables, no es necesario incluir los nombres de los parámetros tipo apuntador en los prototipos de función. El compilador ignora los nombres de los parámetros incluidos para fines de documentación.

Las figuras 8.8 y 8.9 analizan gráficamente la ejecución de los programas de las figuras 8.6 y 8.7, respectivamente.

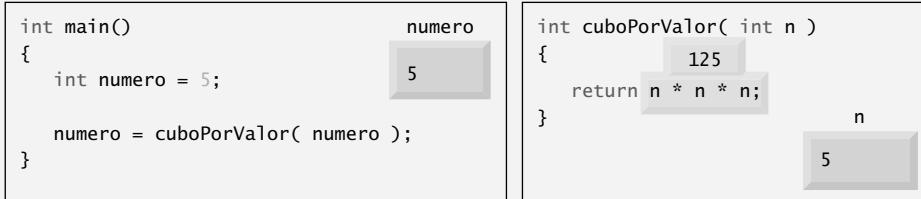
Paso 1: antes de `main` llame a `cuboPorValor`:



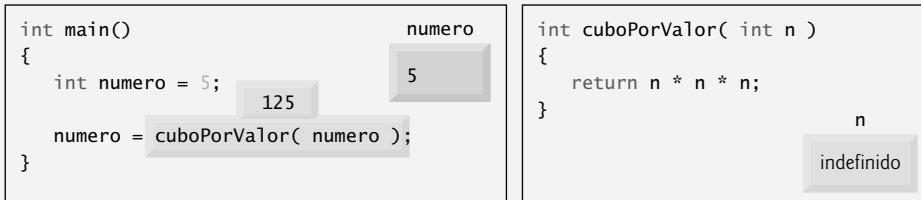
Paso 2: después de que `cuboPorValor` recibe la llamada:



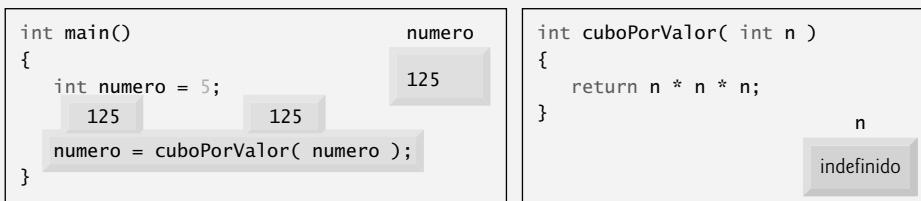
Paso 3: después de que `cuboPorValor` eleva al cubo el parámetro `n` y antes de que `cuboPorValor` regrese a `main`:



Paso 4: después de que `cuboPorValor` regresa a `main` y antes de asignar el resultado a `numero`:



Paso 5: después de que `main` completa la asignación para `numero`:



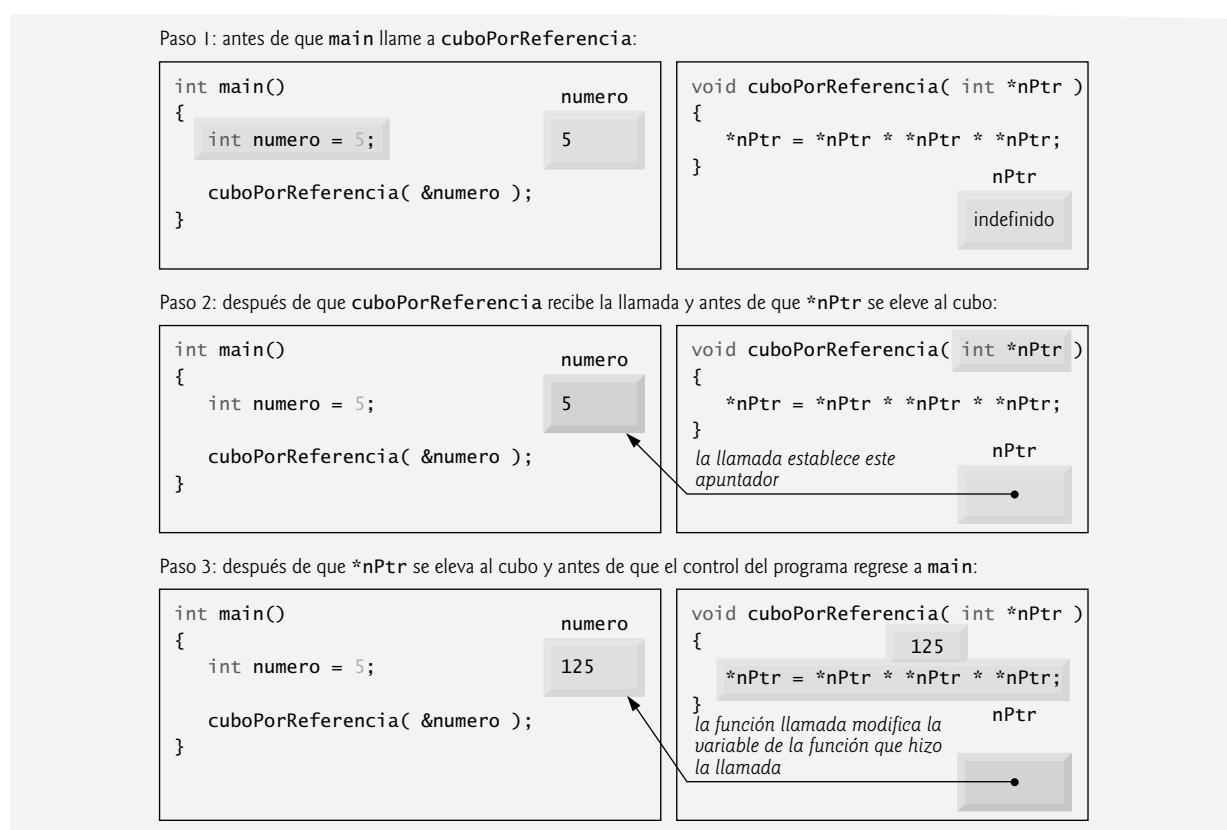
**Figura 8.8** | Análisis del paso por valor del programa de la figura 8.6.



## Observación de Ingeniería de Software 8.1

Use el paso por valor para pasar argumentos a una función, a menos que la función que hace la llamada requiera de manera explícita que la función llamada modifique directamente el valor de la variable que sirve de argumento en la función que hace la llamada. Éste es otro ejemplo del principio del menor privilegio.

En el encabezado de función y en el prototipo para una función que espera un arreglo unidimensional como argumento, se puede usar la notación de apuntador en la lista de parámetros de `cuboPorReferencia`. El compilador no distingue una función que recibe un apuntador de una función que recibe un arreglo unidimensional. Desde luego, esto significa que la función debe “saber” cuando está recibiendo un arreglo, o simplemente una variable que se está pasando por referencia. Cuando el compilador encuentra el parámetro de una función para un arreglo unidimensional de la forma `int b[ ]`, convierte el parámetro a la notación de apuntador `int *b` (lo cual se pronuncia como “`b` es un apuntador a un entero”). Ambas formas de declarar un parámetro de función como arreglo unidimensional son intercambiables.



**Figura 8.9** | Análisis del paso por referencia (con un argumento tipo apuntador) del programa de la figura 8.7.

## 8.5 Uso de `const` con apuntadores

Recuerde que el calificador `const` nos permite informar al compilador que el valor de una variable específica no se debe modificar.

A través de los años, se escribió una gran base de código heredado en las primeras versiones de C en las que no se utilizó `const`, ya que no estaba disponible. Por esta razón, existen grandes oportunidades de mejorar en la ingeniería de software del código de C antiguo (también conocido como “heredado”). Además, muchos programadores que utilizan actualmente ANSI C y C++ no utilizan `const` en sus programas, ya que empezaron a programar en las primeras versiones de C. Estos programadores se están perdiendo muchas oportunidades para la buena ingeniería de software.

Existen muchas posibilidades para usar (o no usar) `const` con los parámetros de funciones. ¿Cómo elegir la más apropiada de estas posibilidades? Hay que dejar que el principio del menor privilegio sea nuestro guía. Siempre debemos otorgar a una función el suficiente acceso a los datos en sus parámetros para que pueda realizar su tarea especificada,

pero nada más. En esta sección veremos cómo combinar `const` con las declaraciones de apuntadores para hacer valer el principio del menor privilegio.

En el capítulo 6 explicamos que, cuando se llama a una función mediante el paso por valor, se crea una copia del argumento (o argumentos) en la llamada a la función y se pasa a la función. Si la copia se modifica en la función, el valor original se mantiene en la función que hizo la llamada sin cambios. En muchos casos, un valor que se pasa a una función se modifica de manera que ésta pueda realizar su tarea. Sin embargo, en ciertos casos, el valor no se debe alterar en la función a la que se llamó, aun y cuando ésta manipule sólo una copia del valor original.

Por ejemplo, considere una función que recibe un arreglo unidimensional y su tamaño como argumentos, y por consiguiente imprime el arreglo. Dicha función debería iterar a través del arreglo e imprimir cada elemento por separado. El tamaño del arreglo se utiliza en el cuerpo de la función para determinar el subíndice más alto del arreglo, de manera que el ciclo pueda terminar cuando se complete la impresión. El tamaño del arreglo no cambia en el cuerpo de la función, por lo que debe declararse como `const`. Desde luego, como el arreglo sólo se va a imprimir, también debe declararse como `const`. En especial, esto es importante ya que un arreglo completo *siempre* se pasa por referencia, y podría ser fácil modificarlo en la función a la que se llama.



### Observación de Ingeniería de Software 8.2

*Si un valor no cambia (o no debe cambiar) en el cuerpo de una función que lo recibe, el parámetro se debe declarar `const` para asegurar que no se modifique por accidente.*

Si hay un intento por modificar un valor `const` se genera una advertencia o un error, dependiendo del compilador específico.



### Tip para prevenir errores 8.2

*Antes de usar una función, compruebe su prototipo para determinar los parámetros que puede modificar.*

Hay cuatro maneras de pasar un apuntador a una función: un apuntador no constante a datos no constantes (figura 8.10), un apuntador no constante a datos constantes (figuras 8.11 y 8.12), un apuntador constante a datos no constantes (figura 8.13) y un apuntador constante a datos constantes (figura 8.14). Cada combinación proporciona un nivel distinto de privilegios de acceso.

#### Apuntador no constante a datos no constantes

El mayor nivel de acceso se otorga mediante un **apuntador no constante a datos no constantes**; los datos se pueden modificar a través del apuntador desreferenciado, y el apuntador se puede modificar para que apunte a otros datos. La declaración para dicho apuntador no incluye `const`. El apuntador se puede utilizar para recibir una cadena con terminación nula en una función que modifique el valor del apuntador que va a procesar (y posiblemente modificar) cada carácter en la cadena. En la sección 7.4 vimos que se puede colocar una cadena con terminación nula en un arreglo de caracteres que contenga los caracteres de la cadena, junto con un carácter nulo que indique dónde termina la cadena.

En la figura 8.10, la función `convertirAMayusculas` (líneas 25 a 34) declara el parámetro `sPtr` (línea 25) como un apuntador no constante a datos no constantes (de nuevo, no se utiliza `const`). La función procesa un carácter a la vez de la cadena con terminación nula almacenada en el arreglo de caracteres `frase` (líneas 27 a 33). Tenga en cuenta que el nombre de un arreglo de caracteres es en realidad equivalente a un apuntador `const` al primer carácter del arreglo, por lo que es posible pasar `frase` como argumento para `convertirAMayusculas`. La función `islower` (línea 29) recibe un argumento tipo carácter y devuelve verdadero si éste es una letra minúscula, y falso en caso contrario. Los caracteres en el rango de '`a`' a '`z`' se convierten a sus correspondientes letras mayúsculas mediante la función `toupper` (línea 30); los demás caracteres no se modifican. La función `toupper` recibe un carácter como argumento. Si éste es una letra

```

1 // Fig. 8.10: fig08_10.cpp
2 // Conversión de una cadena a mayúsculas, usando un
3 // apuntador no constante a datos no constantes.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
```

**Figura 8.10** | Conversión de una cadena a letras mayúsculas, usando un apuntador no constante a datos no constantes. (Parte 1 de 2).

```

7 // Fig. 8.10: fig08_10.cpp
8 // Conversión de una cadena a letras mayúsculas, usando un apuntador no constante a datos no constantes.
9 // (Parte 1 de 2).
10
11 // Prototipos para islower y toupper
12 #include <cctype> // prototipos para islower y toupper
13 using std::islower;
14 using std::toupper;
15
16 void convertirAMayusculas(char *);
17
18 int main()
19 {
20 char frase[] = "caracteres y $32.98";
21
22 cout << "La frase antes de la conversion es: " << frase;
23 convertirAMayusculas(frase);
24 cout << "\nLa frase despues de la conversion es: " << frase << endl;
25 return 0; // indica que terminó correctamente
26 } // fin de main
27
28 // convierte la cadena a letras mayúsculas
29 void convertirAMayusculas(char *sPtr)
30 {
31 while (*sPtr != '\0') // itera mientras el carácter actual no sea '\0'
32 {
33 if (islower(*sPtr)) // si el carácter es minúscula,
34 *sPtr = toupper(*sPtr); // lo convierte a mayúscula
35
36 sPtr++; // mueve sPtr al siguiente carácter en la cadena
37 } // fin de while
38 } // fin de la función convertirAMayusculas

```

```

La frase antes de la conversion es: caracteres y $32.98
La frase despues de la conversion es: CARACTERES Y $32.98

```

**Figura 8.10** | Conversión de una cadena a letras mayúsculas, usando un apuntador no constante a datos no constantes. (Parte 2 de 2).

minúscula, se devuelve la correspondiente letra mayúscula; en caso contrario, se devuelve el carácter original. Las funciones `toupper` e `islower` son parte de la biblioteca de manejo de caracteres `<cctype>` (vea el capítulo 21, Bits, caracteres, cadenas estilo C y estructuras). Después de procesar un carácter, en la línea 32 se incrementa `sPtr` en 1 (esto no sería posible si `sPtr` se declarara `const`). Cuando se aplica el operador `++` a un apuntador que apunta a un arreglo, la dirección de memoria almacenada en el apuntador se modifica para apuntar al siguiente elemento del arreglo (en este caso, el siguiente carácter en la cadena). Sumar uno a un apuntador es una operación válida en la **aritmética de apuntadores**, la cual se cubre con detalle en las secciones 8.8 y 8.9.

#### *Apuntador no constante a datos constantes*

Un **apuntador no constante a datos constantes** es un apuntador que se puede modificar para apuntar a cualquier elemento de datos del tipo apropiado, pero los datos a los que apunta no se pueden modificar a través de ese apuntador. Dicho apuntador podría usarse para recibir un argumento tipo arreglo para una función que procese cada elemento del arreglo, pero no se debe permitir que modifique los datos. Por ejemplo, la función `imprimirCaracteres` (líneas 22 a 26 de la figura 8.11) declara el parámetro `sPtr` (línea 22) para que sea del tipo `const char *`, de manera que pueda

```

1 // Fig. 8.11: fig08_11.cpp
2 // Impresión de una cadena, un carácter a la vez, usando
3 // un apuntador no constante a datos constantes.
4 #include <iostream>
5 using std::cout;
6 using std::endl;

```

**Figura 8.11** | Impresión de una cadena, un carácter a la vez, usando un apuntador no constante a datos constantes. (Parte 1 de 2).

```

7 void imprimirCaracteres(const char *); // imprime usando apuntador a datos const
8
9
10 int main()
11 {
12 const char frase[] = "imprimir caracteres de una cadena";
13
14 cout << "La cadena es:\n";
15 imprimirCaracteres(frase); // imprime los caracteres en frase
16 cout << endl;
17 return 0; // indica que terminó correctamente
18 } // fin de main
19
20 // sPtr se puede modificar, pero no puede modificar el carácter al cual
21 // apunta; es decir, sPtr es un apuntador de "sólo lectura"
22 void imprimirCaracteres(const char *sPtr)
23 {
24 for (; *sPtr != '\0'; sPtr++) // no hay inicialización
25 cout << *sPtr; // muestra el carácter sin modificación
26 } // fin de la función imprimirCaracteres

```

La cadena es:  
Imprimir caracteres de una cadena

**Figura 8.11** | Impresión de una cadena, un carácter a la vez, usando un apuntador no constante a datos constantes. (Parte 2 de 2).

recibir una cadena basada en apuntador, con terminación nula. La declaración se lee de derecha a izquierda como “sPtr es un apuntador a una constante tipo carácter”. El cuerpo de la función utiliza una instrucción `for` (líneas 24 y 25) para imprimir cada carácter en la cadena, hasta encontrar el carácter nulo. Una vez que se imprime cada carácter, el apuntador `sPtr` se incrementa para que apunte al siguiente carácter en la cadena (esto funciona debido a que el apuntador no es `const`). La función `main` crea el arreglo `char` llamado `frase` para pasarlo a `imprimirCaracteres`. De nuevo, podemos pasar el arreglo `frase` a `imprimirCaracteres` ya que el nombre del arreglo es en realidad un apuntador al primer carácter en el arreglo.

En la figura 8.12 se demuestran los mensajes de error de compilación que se producen al tratar de compilar una función que reciba un apuntador no constante a datos constantes, y después se trata de usar un apuntador para modificar los datos. [Nota: recuerde que los mensajes de error pueden variar de un compilador a otro].

```

1 // Fig. 8.12: fig08_12.cpp
2 // Intento de modificar datos a través de un
3 // apuntador no constante a datos constantes.
4
5 void f(const int *); // prototipo
6
7 int main()
8 {
9 int y;
10
11 f(&y); // f intenta una modificación ilegal
12 return 0; // indica que terminó correctamente
13 } // fin de main
14
15 // xPtr no puede modificar el valor de la variable constante a la cual apunta
16 void f(const int *xPtr)
17 {
18 *xPtr = 100; // error: no se puede modificar un objeto const
19 } // fin de la función f

```

**Figura 8.12** | Intento de modificar datos a través de un apuntador no constante a datos constantes. (Parte I de 2).

Mensaje de error del compilador de línea de comandos Borland C++:

```
Error E2024 fig08_12.cpp 18:
 Cannot modify a const object in function f(const int *)
```

Mensaje de error del compilador Microsoft Visual C++:

```
c:\cpphtp6_ejemplos\cap08\Fig08_12\fig08_12.cpp(18) :
error C3892: 'xPtr' : you cannot assign to a variable that is const
```

Mensaje de error del compilador GNU C++:

```
fig08_12.cpp: In function 'void f(const int*)':
fig08_12.cpp: 18: error: assignment of read-only location
```

**Figura 8.12** | Intento de modificar datos a través de un apuntador no constante a datos constantes. (Parte 2 de 2).

Como sabemos, los arreglos son tipos de datos agregados que almacenan elementos de datos relacionados del mismo tipo bajo un nombre. Cuando se llama a una función con un arreglo como argumento, el arreglo se pasa a la función por referencia. Sin embargo, los objetos siempre se pasan por valor: se pasa una copia del objeto completo. Para ello se requiere la sobrecarga en tiempo de ejecución de crear una copia de cada elemento de datos en el objeto y almacenarla en la pila de llamadas a funciones. Cuando se debe pasar un objeto a una función, podemos usar un apuntador a datos constantes (o una referencia a datos constantes) para obtener el rendimiento del paso por referencia y la protección del paso por valor. Cuando se pasa un apuntador a un objeto, sólo se debe crear una copia de la dirección del objeto; el objeto en sí no se copia. En un equipo con direcciones de cuatro bytes, se crea una copia de cuatro bytes de memoria, en vez de una copia de un objeto posiblemente grande.

### Tip de rendimiento 8.1



*Si no es necesario modificarlos por la función a la que se llamó, pase objetos grandes utilizando apuntadores a datos constantes o referencias a datos constantes, para obtener los beneficios de rendimiento del paso por referencia.*

### Observación de Ingeniería de Software 8.3



*Hay que pasar objetos grandes usando apuntadores a datos constantes, o referencias a datos constantes, para obtener la seguridad del paso por valor.*

#### Apuntador constante a datos no constantes

Un **apuntador constante a datos no constantes** es un apuntador que siempre apunta a la misma ubicación de memoria; los datos en esa ubicación se pueden modificar a través del apuntador. Un ejemplo de dicho apuntador es el nombre de un arreglo, el cual es un apuntador constante al principio del arreglo. Todos los datos en el arreglo se pueden utilizar y modificar mediante el uso del nombre del arreglo y del subíndice del mismo. Un apuntador constante a datos no constantes se pueden utilizar para recibir un arreglo como argumento a una función que acceda a los elementos de un arreglo, usando la notación de subíndices. Los apuntadores que se declaran como **const** se deben inicializar a la hora de declararse. (Si el apuntador es un parámetro de función, se inicializa con un apuntador que se pasa a la función). El programa de la figura 8.13 trata de modificar un apuntador constante. En la línea 11 se declara el apuntador **ptr** de tipo **int \* const**. La declaración en la figura se lee de derecha a izquierda como “**ptr** es un apuntador constante a un entero no constante”. El apuntador se inicializa con la dirección de la variable entera **x**. En la línea 14 se hace un intento por asignar la dirección de **y** a **ptr**, pero el compilador genera un mensaje de error. Observe que no ocurre ningún error cuando en la línea 13 se

```

1 // Fig. 8.13: fig08_13.cpp
2 // Intento de modificar un apuntador constante a datos no constantes.
3
4 int main()
5 {
6 int x, y;
7 }
```

**Figura 8.13** | Intento de modificar un apuntador constante a datos no constantes. (Parte 1 de 2).

```

8 // ptr es un apuntador constante a un entero que se puede
9 // modificar a través de ptr, pero ptr siempre apunta a la
10 // misma ubicación de memoria.
11 int * const ptr = &x; // el apuntador const se debe inicializar
12
13 *ptr = 7; // se permite: *ptr no es const
14 ptr = &y; // error: ptr es const; no se puede asignar a una nueva dirección
15 return 0; // indica que terminó correctamente
16 } // fin de main

```

Mensaje de error del compilador de línea de comandos Borland C++:

```
Error E2024 fig08_13.cpp 14: Cannot modify a const object in function main()
```

Mensaje de error del compilador Microsoft Visual C++:

```
c:\cpphttp6_ejemplos\cap08\Fig08_13\fig08_13.cpp(14) : error C3892: 'ptr' :
you cannot assign to a variable that is const
```

Mensaje de error del compilador GNU C++:

```
fig08_13.cpp: In function 'int main()':
fig08_13.cpp: 14: error: assignment of read-only variable 'ptr'
```

**Figura 8.13** | Intento de modificar un apuntador constante a datos no constantes. (Parte 2 de 2).

asigna el valor 7 a \*ptr; el valor no constante al que apunta ptr se puede modificar mediante el uso de ptr desreferenciado, aun y cuando el mismo ptr se haya declarado como const.

### Error común de programación 8.6



*Si no se inicializa un apuntador que se declara const, se produce un error de compilación.*

#### Apuntador constante a datos constantes

La menor cantidad de privilegio de acceso se otorga mediante un **apuntador constante a datos constantes**. Dicho apuntador siempre apunta a la misma ubicación de memoria, y los datos en esa ubicación de memoria no se pueden modificar mediante el uso del apuntador. Así es como se debe pasar un arreglo a una función que sólo lea el arreglo, usando la notación de subíndices de arreglos, y no se modifica el arreglo. El programa de la figura 8.14 declara la variable apuntador ptr de tipo const int \* const (línea 14). Esta declaración se lee de derecha a izquierda como “ptr es un apuntador constante a una constante entera”. La figura muestra los mensajes de error que se generan cuando se hace un intento de modificar los datos a los que ptr apunta (línea 18), y cuando se hace un intento de modificar la dirección almacenada en la variable apuntador (línea 19). Observe que no ocurren errores cuando el programa trata de desreferenciar a ptr, o cuando el programa trata de imprimir el valor al que ptr apunta (línea 16), ya que ni el apuntador ni los datos a los que apunta se están modificando en la instrucción.

```

1 // Fig. 8.14: fig08_14.cpp
2 // Intento de modificar un apuntador constante a datos constantes.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 int x = 5, y;
10
11 // ptr es un apuntador constante a un entero constante.
12 // ptr siempre apunta a la misma ubicación; el entero
13 // en esa ubicación no se puede modificar.
14 const int *const ptr = &x;

```

**Figura 8.14** | Intento de modificar un apuntador constante a datos constantes. (Parte 1 de 2).

```

15 cout << *ptr << endl;
16
17 *ptr = 7; // error: *ptr es const; no se puede asignar un nuevo valor
18 ptr = &y; // error: ptr es const; no se puede asignar una nueva dirección
19
20 return 0; // indica que terminó correctamente
21 } // fin de main

```

Mensaje de error del compilador de líneas de comandos Borland C++:

```
Error E2024 fig08_14.cpp 18: Cannot modify a const object in function main()
Error E2024 fig08_14.cpp 19: Cannot modify a const object in function main()
```

Mensaje de error del compilador Microsoft Visual C++:

```
c:\cpphtp6_ejemplos\cap08\Fig08_14\fig08_14.cpp(18) : error C3892: 'ptr' :
you cannot assign to a variable that is const
c:\cpphtp6_ejemplos\cap08\Fig08_14\fig08_14.cpp(19) : error C3892: 'ptr' :
you cannot assign to a variable that is const
```

Mensaje de error del compilador GNU C++:

```
fig08_13.cpp: In function 'int main()':
fig08_13.cpp: 14: error: assignment of read-only variable 'ptr'
```

**Figura 8.14** | Intento de modificar un apuntador constante a datos constantes. (Parte 2 de 2).

## 8.6 Ordenamiento por selección mediante el uso del paso por referencia

En esta sección definiremos un programa de ordenamiento para demostrar el paso de arreglos y elementos individuales de arreglos por referencia. Utilizaremos el algoritmo de **ordenamiento por selección**, el cual es un algoritmo de ordenamiento fácil de programar, pero por desgracia ineficiente. En la primera iteración del algoritmo se selecciona el elemento más pequeño en el arreglo y se intercambia con el primer elemento. En la segunda iteración se selecciona el segundo elemento más pequeño (que es el elemento más pequeño del elemento restante) y se intercambia con el segundo elemento. El algoritmo continúa hasta que en la última iteración se selecciona el segundo elemento más grande y se intercambia con el penúltimo subíndice, dejando el elemento más grande en el último subíndice. Después de la  $i$ -ésima iteración, los  $i$  elementos más pequeños del arreglo se ordenarán en forma ascendente en los primeros  $i$  elementos del arreglo.

Como ejemplo, considere el arreglo

```
34 56 4 10 77 51 93 30 5 52
```

Un programa que implementa el ordenamiento por selección primero determina el valor más pequeño (4) en el arreglo, el cual está contenido en el elemento 2. El programa intercambia el 4 con el valor en el elemento 0 (34), y el arreglo queda así:

```
4 56 34 10 77 51 93 30 5 52
```

[Nota: usamos negrita para resaltar los valores que se intercambiaron]. Después, el programa determina el valor más pequeño de los elementos restantes (todos los elementos, excepto 4), que es 5, y está contenido en el elemento 8. El programa intercambia el 5 con el 56 en el elemento 1, y el arreglo queda así:

```
4 5 34 10 77 51 93 30 56 52
```

En la tercera iteración, el programa determina el siguiente valor más pequeño (10) y lo intercambia con el valor en el elemento 2 (34).

```
4 5 10 34 77 51 93 30 56 52
```

El proceso continúa hasta que el arreglo queda completamente ordenado.

```
4 5 10 30 34 51 52 56 77 93
```

Observe que después de la primera iteración, el elemento más pequeño se encuentra en la primera posición. Después de la segunda iteración, los dos elementos más pequeños están ordenados en las primeras dos posiciones. Después de la tercera iteración, los tres elementos más pequeños están ordenados en las primeras tres posiciones.

En la figura 8.15 se implementa el ordenamiento por selección mediante el uso de dos funciones: `ordenamientoSelección` e `intercambiar`. La función `ordenamientoSelección` (líneas 36 a 53) ordena el arreglo. En la línea 38 se declara la variable `menor`, la cual almacenará el subíndice del elemento más pequeño en el arreglo restante. En las líneas 41 a 52 se itera `tamanio - 1` veces. En la línea 43 se establece el subíndice del elemento más pequeño al subíndice actual. En las líneas 46 a 49 se itera a través del resto de los elementos en el arreglo. Para cada uno de estos elementos, en la línea 48 se compara su valor con el valor del elemento más pequeño. Si el elemento actual es menor que el elemento más pequeño, en la línea 49 se asigna el subíndice del elemento actual a `menor`. Cuando este ciclo termine, `menor` contendrá el subíndice del elemento más pequeño en el arreglo restante. En la línea 51 se hace una llamada a la función `intercambiar` (líneas 57 a 62) para colocar el elemento más pequeño restante en la siguiente posición en el arreglo (es decir, se intercambian los elementos del arreglo `arreglo[i]` y `arreglo[menor]`).

Ahora analicemos más de cerca la función `intercambiar`. Recuerde que C++ implementa el ocultamiento de información entre funciones, por lo que `intercambiar` no tiene acceso a los elementos individuales del arreglo en `ordenamientoSelección`. Debido a que `ordenamientoSelección` *desea* que `intercambiar` tenga acceso a los elementos del arreglo que se van a intercambiar, `ordenamientoSelección` pasa cada uno de estos elementos a `intercambiar` por

```

1 // Fig. 8.15: fig08_15.cpp
2 // ordenamiento por selección con paso por referencia. Este programa coloca valores en
3 // un arreglo, los ordena en forma ascendente e imprime el arreglo resultante.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 void ordenamientoSelección(int * const, const int); // prototipo
12 void intercambiar(int * const, int * const); // prototipo
13
14 int main()
15 {
16 const int tamanioArreglo = 10;
17 int a[tamanioArreglo] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
18
19 cout << "Elementos de datos en el orden original\n";
20
21 for (int i = 0; i < tamanioArreglo; i++)
22 cout << setw(4) << a[i];
23
24 ordenamientoSelección(a, tamanioArreglo); // ordena el arreglo
25
26 cout << "\nElementos de datos en orden ascendente\n";
27
28 for (int j = 0; j < tamanioArreglo; j++)
29 cout << setw(4) << a[j];
30
31 cout << endl;
32 return 0; // indica que terminó correctamente
33 } // fin de main
34
35 // función para ordenar un arreglo
36 void ordenamientoSelección(int * const arreglo, const int tamanio)
37 {
38 int menor; // subíndice del elemento más pequeño
39
40 // itera a través de tamanio - 1 elementos
41 for (int i = 0; i < tamanio - 1; i++)
42 {
43 menor = i; // primer subíndice del resto del arreglo
44

```

**Figura 8.15** | Ordenamiento por selección mediante el paso por referencia. (Parte 1 de 2).

```

45 // itera hasta encontrar el subíndice del elemento más pequeño
46 for (int subindice = i + 1; subindice < tamano; subindice++)
47
48 if (arreglo[subindice] < arreglo[menor])
49 menor = subindice;
50
51 intercambiar(&arreglo[i], &arreglo[menor]);
52 } // fin de if
53 } // fin de la función ordenamientoSelección
54
55 // intercambia los valores en las ubicaciones de memoria
56 // a las que apuntan elemento1Ptr y elemento2Ptr
57 void intercambiar(int * const elemento1Ptr, int * const elemento2Ptr)
58 {
59 int contenido = *elemento1Ptr;
60 *elemento1Ptr = *elemento2Ptr;
61 *elemento2Ptr = contenido;
62 } // fin de la función intercambiar

```

```

Elementos de datos en el orden original
2 6 4 8 10 12 89 68 45 37
Elementos de datos en orden ascendente
2 4 6 8 10 12 37 45 68 89

```

**Figura 8.15** | Ordenamiento por selección mediante el uso del paso por referencia. (Parte 2 de 2).

referencia: la dirección de cada elemento del arreglo se pasa de manera explícita. Aunque los arreglos completos se pasan por referencia, los elementos individuales de un arreglo son escalares y por lo general se pasan por valor. Por lo tanto, `ordenamientoSelección` utiliza el operador dirección (&) en cada elemento del arreglo en la llamada a `intercambiar` (línea 51) para llevar a cabo el paso por referencia. La función `intercambiar` (líneas 57 a 62) recibe a `&arreglo[ i ]` en la variable apuntador `elemento1Ptr`. El ocultamiento de información evita que `intercambiar` “conozca” el nombre `arreglo[ i ]`, pero `intercambiar` puede usar `*elemento1Ptr` como sinónimo de `arreglo[ i ]`. Por ende, cuando `intercambiar` hace referencia a `*elemento1Ptr`, en realidad está haciendo referencia a `arreglo[ i ]` en `ordenamientoSelección`. De manera similar, cuando `intercambiar` hace referencia a `*elemento1Ptr`, en realidad está haciendo referencia a `arreglo[ menor ]` en `ordenamientoSelección`.

Aun cuando `intercambiar` no puede usar las instrucciones

```

contenido = arreglo[i];
arreglo[i] = arreglo[menor];
arreglo[menor] = contenido;

```

precisamente se obtiene el mismo efecto mediante:

```

int contenido = *elemento1Ptr;
*elemento1Ptr = *elemento2Ptr;
*elemento2Ptr = contenido;

```

en la función `intercambiar` de la figura 8.15.

Hay que observar varias características de la función `ordenamientoSelección`. El encabezado de función (línea 36) declara a `arreglo` como `int * const arreglo`, en vez de `int arreglo[]`, para indicar que la función recibe un arreglo unidimensional como argumento. Tanto el apuntador como el parámetro `tamano` del parámetro `arreglo` se declaran `const` para hacer valer el principio del menor privilegio. Aunque el parámetro `tamano` recibe una copia de un valor en `main` y, aunque se modifique la copia, no se puede modificar el valor en `main`, `ordenamientoSelección` no necesita alterar a `tamano` para llevar a cabo esta tarea; el tamaño del arreglo permanece fijo durante la ejecución de `ordenamientoSelección`. Por lo tanto, `tamano` se declara `const` para asegurar que no se modifique. Si el tamaño del arreglo se modificara durante el proceso de ordenamiento, el algoritmo de ordenamiento no funcionaría de manera apropiada.

Observe que la función `ordenamientoSelección` recibe el tamaño del arreglo como parámetro, ya que la función debe tener esa información para ordenar el arreglo. Cuando se pasa un arreglo a una función, ésta sólo recibe la dirección de memoria del primer elemento del arreglo; el tamaño del mismo se debe pasar a la función por separado.

Al definir la función `ordenamientoSelección` para recibir el tamaño del arreglo como parámetro, permitimos que la función sea utilizada por cualquier programa que ordene arreglos `int` unidimensionales de tamaño arbitrario. El tamaño

del arreglo se podría haber programado directamente en la función, pero esto restringiría a la función de manera que sólo pudiera procesar un arreglo de un tamaño específico, y se reduciría la reutilización de la función; sólo los programas que procesen arreglos `int` unidimensionales del tamaño específico “fijado” a la función podrían utilizarla.



### Observación de Ingeniería de Software 8.4

*Al pasar un arreglo a una función, también se debe pasar el tamaño del arreglo (en vez de integrar en la función el conocimiento del tamaño del arreglo); esto hace a la función más reutilizable.*

## 8.7 Operador `sizeof`

C++ proporciona el operador unario `sizeof` para determinar el tamaño de un arreglo (o de cualquier otro tipo de datos, variable o constante) en bytes durante la compilación de un programa. Cuando se aplica al nombre de un arreglo, como en la figura 8.16 (línea 14), el operador `sizeof` devuelve el número total de bytes en el arreglo como un valor de tipo `size_t` (un tipo entero sin signo que es por lo menos tan grande como un `unsigned int`). Observe que esto es distinto del valor `size` de un `vector<int>`, por ejemplo, que representa el número de elementos enteros en el `vector`. La computadora que utilizamos para compilar este programa almacena variables de tipo `double` en 8 bytes de memoria, y `arreglo` se declara de forma que tenga 20 elementos (línea 12), por lo que `arreglo` usa 160 bytes en memoria. Cuando se aplica a un parámetro apuntador (línea 24) en una función que recibe un arreglo como argumento, el operador `sizeof` devuelve el tamaño del apuntador en bytes (4 en el sistema que utilizamos); no el tamaño del arreglo.



### Error común de programación 8.7

*Al utilizar el operador `sizeof` en una función para buscar el tamaño en bytes de un parámetro tipo arreglo, se obtiene el tamaño en bytes de un apuntador, no el tamaño en bytes del arreglo.*

[Nota: cuando se utiliza el compilador Borland C++ para compilar la figura 8.16, éste genera el mensaje de advertencia “Parameter ‘ptr’ is never used in function getSize(double \*)”. Esta advertencia ocurre debido a que `sizeof` es en realidad un operador en tiempo de compilación; por ende, la variable `ptr` no se utiliza en el cuerpo de la función en tiempo de ejecución. Muchos compiladores emiten advertencias como ésta para hacernos saber que

```

1 // Fig. 8.16: fig08_16.cpp
2 // Al aplicar el operador sizeof al nombre de un arreglo
3 // se devuelve el número de bytes en el arreglo.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 size_t getSize(double *); // prototipo
9
10 int main()
11 {
12 double arreglo[20]; // 20 valores double; ocupa 160 bytes en nuestro sistema
13
14 cout << "El numero de bytes en el arreglo es " << sizeof(arreglo);
15
16 cout << "\nEl numero de bytes devueltos por getSize es "
17 << getSize(arreglo) << endl;
18 return 0; // indica que terminó correctamente
19 } // fin de main
20
21 // devuelve el tamaño de ptr
22 size_t getSize(double *ptr)
23 {
24 return sizeof(ptr);
25 } // fin de la función getSize

```

```

El numero de bytes en el arreglo es 160
El numero de bytes devueltos por getSize es 4

```

Figura 8.16 | Al aplicar el operador `sizeof` al nombre de un arreglo, se devuelve el número de bytes en el mismo.

una variable no se está utilizando, de manera que el programador la elimine del código, o modifique el mismo para usar la variable en forma apropiada. En la figura 8.17 se generan mensajes similares con diversos compiladores].

El número de elementos en un arreglo también se puede determinar mediante el uso de los resultados de dos operaciones `sizeof`. Por ejemplo, considere la declaración del siguiente arreglo:

```
double arregloReal[22];
```

Si las variables de tipo `double` se almacenan en ocho bytes de memoria, el arreglo `arregloReal` contiene un total de 176 bytes. Para determinar el número de elementos en el arreglo, podemos utilizar la siguiente expresión (que se evalúa en tiempo de compilación):

```
sizeof arregloReal / sizeof(double) // calcula el número de elementos
```

La expresión determina el número de bytes en el arreglo `arregloReal` (176) y divide ese valor por el número de bytes utilizados en memoria para almacenar un valor `double` (8); el resultado es el número de elementos en `arregloReal` (22).

```

1 // Fig. 8.17: fig08_17.cpp
2 // Demostración del operador sizeof.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 char c; // variable de tipo char
10 short s; // variable de tipo short
11 int i; // variable de tipo int
12 long l; // variable de tipo long
13 float f; // variable de tipo float
14 double d; // variable de tipo double
15 long double ld; // variable de tipo long double
16 int arreglo[20]; // arreglo de int
17 int *ptr = arreglo; // variable de tipo int *
18
19 cout << "sizeof c = " << sizeof(c
20 << "\nsizeof(char) = " << sizeof(char)
21 << "\nsizeof s = " << sizeof(s
22 << "\nsizeof(short) = " << sizeof(short)
23 << "\nsizeof i = " << sizeof(i
24 << "\nsizeof(int) = " << sizeof(int)
25 << "\nsizeof l = " << sizeof(l
26 << "\nsizeof(long) = " << sizeof(long)
27 << "\nsizeof f = " << sizeof(f
28 << "\nsizeof(float) = " << sizeof(float)
29 << "\nsizeof d = " << sizeof(d
30 << "\nsizeof(double) = " << sizeof(double)
31 << "\nsizeof ld = " << sizeof(ld
32 << "\nsizeof(long double) = " << sizeof(long double)
33 << "\nsizeof arreglo = " << sizeof(arreglo
34 << "\nsizeof ptr = " << sizeof(ptr << endl;
35 return 0; // indica que terminó correctamente
36 } // fin de main

```

|                                  |                                      |
|----------------------------------|--------------------------------------|
| <code>sizeof c = 1</code>        | <code>sizeof(char) = 1</code>        |
| <code>sizeof s = 2</code>        | <code>sizeof(short) = 2</code>       |
| <code>sizeof i = 4</code>        | <code>sizeof(int) = 4</code>         |
| <code>sizeof l = 4</code>        | <code>sizeof(long) = 4</code>        |
| <code>sizeof f = 4</code>        | <code>sizeof(float) = 4</code>       |
| <code>sizeof d = 8</code>        | <code>sizeof(double) = 8</code>      |
| <code>sizeof ld = 8</code>       | <code>sizeof(long double) = 8</code> |
| <code>sizeof arreglo = 80</code> |                                      |
| <code>sizeof ptr = 4</code>      |                                      |

Figura 8.17 | Uso del operador `sizeof` para determinar tamaños estándar de los tipos de datos.

### Cómo determinar el tamaño de los tipos fundamentales, un arreglo y un apuntador

En la figura 8.17 se utiliza `sizeof` para calcular el número de bytes utilizados para almacenar la mayoría de los tipos de datos estándar. Observe que, en los resultados, los tipos `double` y `long double` tienen el mismo tamaño. Los tipos pueden tener distintos tamaños, con base en la plataforma que ejecuta el programa. Por ejemplo, en otro sistema, `double` y `long double` pueden ser de distintos tamaños.



#### Tip de portabilidad 8.2

*El número de bytes utilizados para almacenar un tipo de datos específico puede variar de un sistema a otro. Al escribir programas que dependan de los tamaños de los tipos de datos, y que se vayan a ejecutar en varios sistemas computacionales, use `sizeof` para determinar el número de bytes utilizados para almacenar los tipos de datos.*

El operador `sizeof` se puede aplicar a cualquier expresión o nombre de tipo. Cuando se aplica `sizeof` al nombre de una variable (que no sea el nombre de un arreglo) u otra expresión, se devuelve el número de bytes utilizados para almacenar el tipo específico del valor de la expresión. Observe que los paréntesis utilizados con `sizeof` sólo son requeridos si se suministra el nombre de un tipo (por ejemplo, `int`) como su operando. Los paréntesis utilizados con `sizeof` no son requeridos cuando el operando de `sizeof` es una expresión. Recuerde que `sizeof` es un operador, no una función, y que tiene su efecto en tiempo de compilación, no en tiempo de ejecución.



#### Error común de programación 8.8

*Omitir los paréntesis en una operación `sizeof` cuando el operando es el nombre de un tipo es un error de compilación.*



#### Tip de rendimiento 8.2

*Como `sizeof` es un operador unario en tiempo de compilación, no un operador en tiempo de ejecución, el uso de `sizeof` no tiene un impacto negativo sobre el rendimiento de la ejecución.*



#### Tip para prevenir errores 8.3

*Para evitar los errores asociados al omitir los paréntesis alrededor del operando del operador `sizeof`, muchos programadores incluyen paréntesis alrededor de cada operando de `sizeof`.*

## 8.8 Expresiones y aritmética de apuntadores

Los apuntadores son operadores válidos en las expresiones aritméticas, de asignación y de comparación. Sin embargo, no todos los operadores que se utilizan normalmente en estas expresiones son válidos con las variables apuntadores. En esta sección se describen los operadores que pueden tener apuntadores como operandos, y cómo se utilizan estos operadores con los apuntadores.

Se pueden realizar varias operaciones aritméticas con los apuntadores. Un apuntador se puede incrementar (`++`) o decrementar (`--`), se puede sumar un entero a un apuntador (`+ o +=`), se puede restar un entero de un apuntador (`- o -=`), o se puede restar un apuntador de otro apuntador del mismo tipo.

Suponga que se ha declarado el arreglo `int v[ 5 ]`, y que su primer elemento se encuentra en la ubicación 3000. Suponga que se ha inicializado el apuntador `vPtr` para que apunte a `v[ 0 ]` (es decir, el valor de `vPtr` es 3000). En la figura 8.18 se muestra un diagrama de esta situación para un equipo con enteros de cuatro bytes.

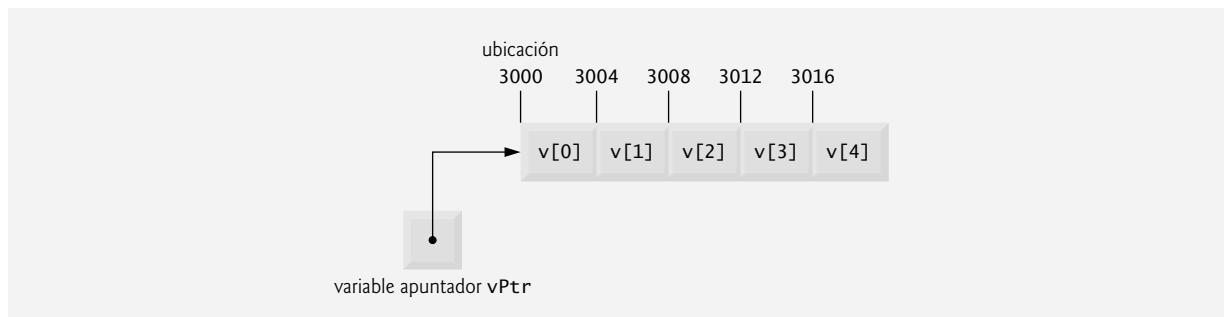


Figura 8.18 | El arreglo `v` y una variable apuntador `int *vPtr` que apunta a `v`.

Observe que `vPtr` se puede inicializar para que apunte al arreglo `v` con cualquiera de las siguientes instrucciones (debido a que el nombre de un arreglo es equivalente a la dirección de su primer elemento):

```
int *vPtr = v;
int *vPtr = &v[0];
```

### Tip de portabilidad 8.3

 La mayoría de las computadoras de la actualidad tienen enteros de dos o de cuatro bytes. Algunos de los equipos más recientes utilizan enteros de ocho bytes. Debido a que los resultados de la aritmética de apuntadores dependen del tamaño de los objetos a los que apunta un apuntador, la aritmética de apuntadores es dependiente del equipo.

En la aritmética convencional, la suma `3000 + 2` produce el valor `3002`. Por lo general éste no es el caso con la aritmética de apuntadores. Cuando se suma (o se resta) un entero a un apuntador, éste no simplemente se incrementa o decrementa debido a ese entero, sino por ese entero multiplicado por el tamaño del objeto al que el apuntador hace referencia. El número de bytes depende del tipo de datos del objeto. Por ejemplo, la instrucción

```
vPtr += 2;
```

produciría `3008` (`3000 + 2 * 4`), suponiendo que un `int` se almacena en cuatro bytes de memoria. En el arreglo `v`, `vPtr` apuntaría ahora a `v[ 2 ]` (figura 8.19). Si se almacena un entero en dos bytes de memoria, entonces el cálculo anterior produciría la ubicación de memoria `3004` (`3000 + 2 * 2`). Si los elementos del arreglo fueran de un tipo de datos distinto, la anterior instrucción incrementaría el apuntador en base al doble del número de bytes que se requieren para almacenar un objeto de ese tipo de datos. Al realizar aritmética de apuntadores con un arreglo de caracteres, los resultados serán consistentes con la aritmética regular, ya que cada carácter es de un byte.

Si `vPtr` se hubiera incrementado a `3016`, lo cual apunta a `v[ 4 ]`, la instrucción

```
vPtr -= 4;
```

establecería a `vPtr` de vuelta en `3000`; el inicio del arreglo. Si un apuntador se va a incrementar o decrementar por uno, se pueden utilizar los operadores de incremento (`++`) y decremento (`--`). Cada una de las instrucciones

```
++vPtr;
vPtr++;
```

incrementa el apuntador para que apunte al siguiente elemento del arreglo. Cada una de las instrucciones

```
--vPtr;
vPtr--;
```

decrementa el apuntador para que apunte al elemento anterior del arreglo.

Las variables apuntador que apuntan al mismo arreglo se pueden restar entre sí. Por ejemplo, si `vPtr` contiene la dirección `3000` y `v2Ptr` contiene la dirección `3008`, la instrucción

```
x = v2Ptr - vPtr;
```

asignaría a `x` el número de elementos del arreglo de `vPtr` a `v2Ptr`; en este caso, `2`. La aritmética de apuntadores no tiene significado, a menos que se realice en un apuntador que apunte a un arreglo. No podemos asumir que dos variables del mismo tipo se almacenan en forma contigua en la memoria, a menos que sean elementos adyacentes de un arreglo.

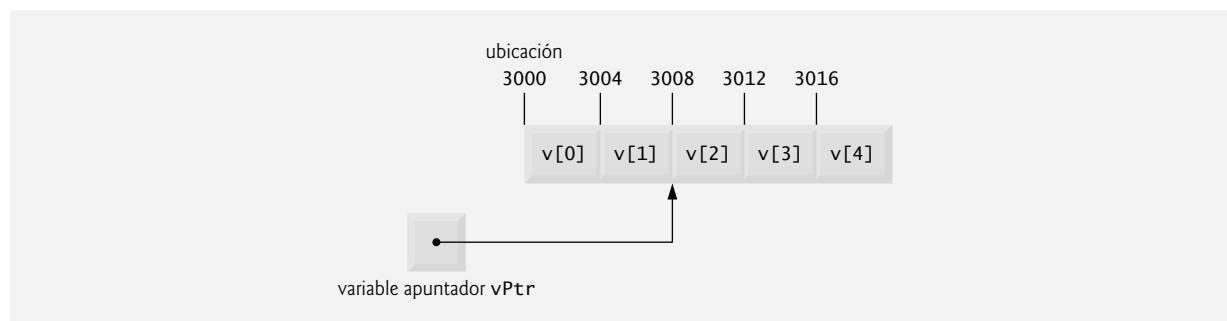


Figura 8.19 | El apuntador `vPtr` después de la aritmética de apuntadores.



### Error común de programación 8.9

*El uso de la aritmética de apuntadores en un apuntador que no haga referencia a un arreglo de valores es un error lógico.*



### Error común de programación 8.10

*Restar o comparar dos apuntadores que no hagan referencia a los elementos del mismo arreglo es un error lógico.*



### Error común de programación 8.11

*El uso de la aritmética de apuntadores para incrementar o decrementar un apuntador, de tal forma que éste haga referencia a un elemento fuera de los límites del arreglo, es por lo general un error lógico.*

Un apuntador se puede asignar a otro apuntador, si ambos apuntadores son del mismo tipo. En caso contrario, se debe usar un operador de conversión de tipos para convertir el valor del apuntador, que está a la derecha de la asignación, al tipo del apuntador que está a la izquierda de la asignación. La excepción a esta regla es el apuntador a `void` (es decir, `void *`), el cual es un apuntador genérico capaz de representar cualquier tipo de apuntador. Todos los tipos de apuntadores se pueden asignar a un apuntador de tipo `void *` sin necesidad de conversión de tipos. Sin embargo, un apuntador de tipo `void *` no se puede asignar directamente a un apuntador de otro tipo; el apuntador de tipo `void *` debe convertirse primero en el tipo apropiado de apuntador.



### Observación de Ingeniería de Software 8.5

*Los argumentos tipo apuntadores no constantes se pueden pasar a los parámetros tipo apuntadores constantes. Esto es útil cuando el cuerpo de un programa utiliza un apuntador no constante para acceder a los datos, pero no desea que los datos se modifiquen mediante una función a la que se llama en el cuerpo del programa.*

No se puede desreferenciar un apuntador `void *`. Por ejemplo, el compilador “sabe” que un apuntador a `int` hace referencia a cuatro bytes de memoria en un equipo con enteros de cuatro bytes, pero un apuntador a `void` simplemente contiene una dirección de memoria para un tipo de datos desconocido; el compilador no conoce el número preciso de bytes a los que hace referencia el apuntador y el tipo de los datos. El compilador debe conocer el tipo de datos para determinar el número de bytes a desreferenciar para un apuntador específico; para un apuntador a `void`, este número de bytes no se puede determinar en base al tipo.



### Error común de programación 8.12

*Asignar un apuntador de un tipo a un apuntador de otro (que no sea `void *`), sin convertir el primer apuntador al tipo del segundo apuntador, es un error de compilación.*



### Error común de programación 8.13

*Todas las operaciones en un operador `void *` son errores de compilación, excepto la comparación de apuntadores `void *` con otros apuntadores, la conversión de apuntadores `void *` a tipos de apuntadores válidos, y la asignación de direcciones a apuntadores `void *`.*

Los apuntadores se pueden comparar mediante el uso de los operadores de igualdad y relacionales. Las comparaciones en las que se utilizan operadores relacionales no tienen significado, a menos que los apuntadores apunten a miembros del mismo arreglo. Las comparaciones con apuntadores comparan las direcciones almacenadas en los apuntadores. Una comparación de dos apuntadores que apunten al mismo arreglo podría mostrar, por ejemplo, que un apuntador apunta a un elemento de mayor numeración del arreglo que el otro apuntador. Un uso común de la comparación de apuntadores es determinar si un apuntador es 0 (es decir, si el apuntador es nulo; que no apunta a nada).

## 8.9 Relación entre apuntadores y arreglos

Los arreglos y los apuntadores están estrechamente relacionados en C++ y se pueden utilizar de manera casi intercambiable. El nombre de un arreglo se puede considerar como un apuntador constante. Los apuntadores se pueden utilizar para realizar cualquier operación en la que se involucren los subíndices de arreglos.

Suponga las siguientes declaraciones:

```
int b[5]; // crea el arreglo int b de 5 elementos
int *bPtr; // crea el apuntador int bPtr;
```

Como el nombre del arreglo (sin subíndice) es un apuntador (constante) al primer elemento del arreglo, podemos establecer `bPtr` a la dirección del primer elemento en el arreglo `b` con la instrucción

```
bPtr = b; // asigna la dirección del arreglo b a bPtr
```

Esto es equivalente a asignar la dirección del primer elemento del arreglo, como se muestra a continuación:

```
bPtr = &b[0]; // también asigna la dirección del arreglo b a bPtr
```

El elemento `b[ 3 ]` del arreglo se puede referenciar de manera alternativa con la siguiente expresión de apuntador:

```
*(bPtr + 3)
```

El 3 en la expresión anterior es el **desplazamiento** para el apuntador. Cuando el apuntador apunta al inicio de un arreglo, el desplazamiento indica a cuál elemento del arreglo se debe hacer referencia, y el valor del desplazamiento es idéntico al subíndice del mismo. La notación anterior se conoce como **notación apuntador/desplazamiento**. Los paréntesis son necesarios, debido a que la precedencia de `*` es mayor que la precedencia de `+`. Sin los paréntesis, la expresión anterior sumaría 3 al valor de `*bPtr` (es decir, 3 se sumaría a `b[ 0 ]`, suponiendo que `bPtr` apunta al inicio del arreglo). Así como se puede hacer referencia al elemento del arreglo con una expresión de apuntador, la dirección

```
&b[3]
```

se puede escribir con la expresión de apuntador

```
bPtr + 3
```

El nombre del arreglo (que es `const` de manera implícita) se puede tratar como apuntador y se puede utilizar en la aritmética de apuntadores. Por ejemplo, la expresión

```
*(b + 3)
```

también se refiere al elemento `b[ 3 ]` del arreglo. En general, todas las expresiones de arreglos con subíndice se pueden escribir con un apuntador y un desplazamiento. En este caso, la notación apuntador/desplazamiento se utilizó con el nombre del arreglo como un apuntador. Observe que la expresión anterior no modifica el nombre del arreglo en ninguna forma; `b` sigue apuntando al primer elemento en el arreglo.

Los apuntadores pueden usar subíndices de la misma forma que los arreglos. Por ejemplo, la expresión

```
bPtr[1]
```

hace referencia al elemento `b[ 1 ]` del arreglo; esta expresión utiliza la **notación apuntador/subíndice**.

Recuerde que el nombre de un arreglo es un apuntador constante; siempre apunta al inicio del arreglo. Por ende, la expresión

```
b += 3
```

produce un error de compilación, ya que trata de modificar el valor del nombre del arreglo (una constante) con la aritmética de apuntadores.

## Error común de programación 8.14



Aunque los nombres de los arreglos son apuntadores al inicio del arreglo, y los apuntadores se pueden modificar en las expresiones aritméticas, los nombres de los arreglos no se pueden modificar en las expresiones aritméticas, ya que los nombres de los arreglos son apuntadores constantes.

## Buena práctica de programación 8.2



Por cuestión de claridad, usamos la notación de arreglos en vez de la notación de apuntadores al manipular arreglos.

En la figura 8.20 se utilizan las cuatro notaciones descritas en esta sección para hacer referencia a los elementos de un arreglo (notación de subíndice de arreglo, notación apuntador/desplazamiento con el nombre de un arreglo como apuntador, notación de subíndice de apuntador y notación apuntador/desplazamiento con un apuntador) para realizar la misma tarea, a saber, imprimir los cuatro elementos del arreglo entero `b`.

Para ilustrar mejor la capacidad de intercambio de los arreglos y apuntadores, veamos las dos funciones de copia de cadenas (`copia1` y `copia2`) en el programa de la figura 8.21. Ambas funciones copian una cadena en un arreglo de caracteres. Después de una comparación de los prototipos de función para `copia1` y `copia2`, las funciones parecen idénticas (debido a que los arreglos y apuntadores pueden intercambiarse). Estas funciones realizan la misma tarea, pero se implementan de manera distinta.

```

1 // Fig. 8.20: fig08_20.cpp
2 // Uso de notaciones de subíndice y apuntador con arreglos.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 int b[] = { 10, 20, 30, 40 }; // crea el arreglo b de 4 elementos
10 int *bPtr = b; // establece bPtr para que apunte al arreglo b
11
12 // imprime el arreglo b usando la notación de subíndice de arreglo
13 cout << "Se imprime el arreglo b con:\n\nNotacion de subindice de arreglo\n";
14
15 for (int i = 0; i < 4; i++)
16 cout << "b[" << i << "] = " << b[i] << '\n';
17
18 // imprime el arreglo b usando el nombre del arreglo y la notación apuntador/desplazamiento
19 cout << "\nNotacion apuntador/desplazamiento en donde "
20 << "el apuntador es el nombre del arreglo\n";
21
22 for (int desplazamiento1 = 0; desplazamiento1 < 4; desplazamiento1++)
23 cout << "*(" << b + " << desplazamiento1 << ") = " << *(b + desplazamiento1) << '\n';
24
25 // imprime el arreglo b usando bPtr y la notación de subíndice de arreglo
26 cout << "\nNotacion de subindice de apuntador\n";
27
28 for (int j = 0; j < 4; j++)
29 cout << "bPtr[" << j << "] = " << bPtr[j] << '\n';
30
31 cout << "\nNotacion apuntador/desplazamiento\n";
32
33 // imprime el arreglo b usando bPtr y la notación apuntador/desplazamiento
34 for (int desplazamiento2 = 0; desplazamiento2 < 4; desplazamiento2++)
35 cout << "*(" << bPtr + " << desplazamiento2 << ") = "
36 << *(bPtr + desplazamiento2) << '\n';
37
38 return 0; // indica que terminó correctamente
39 } // fin de main

```

Se imprime el arreglo b con:

Notacion de subindice de arreglo

```

b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

```

Notacion apuntador/desplazamiento en donde el apuntador es el nombre del arreglo

```

*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

```

Notacion de subindice de apuntador

```

bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

```

Notacion apuntador/desplazamiento

```

*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40

```

**Figura 8.20** | Referencia a los elementos de un arreglo con el nombre del arreglo y apuntadores.

```

1 // Fig. 8.21: fig08_21.cpp
2 // Copia de una cadena usando notación de arreglos y notación de apuntadores.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void copia1(char *, const char *); // prototipo
8 void copia2(char *, const char *); // prototipo
9
10 int main()
11 {
12 char cadena1[10];
13 char *cadena2 = "Hola";
14 char cadena3[10];
15 char cadena4[] = "Hasta luego";
16
17 copia1(cadena1, cadena2); // copia cadena2 a cadena1
18 cout << "cadena1 = " << cadena1 << endl;
19
20 copia2(cadena3, cadena4); // copia cadena4 a cadena3
21 cout << "cadena3 = " << cadena3 << endl;
22 return 0; // indica que terminó correctamente
23 } // fin de main
24
25 // copia s2 a s1 usando notación de arreglos
26 void copia1(char * s1, const char * s2)
27 {
28 // la copia ocurre en el encabezado del for
29 for (int i = 0; (s1[i] = s2[i]) != '\0'; i++)
30 ; // no hace nada en el cuerpo
31 } // fin de la función copia1
32
33 // copia s2 a s1 usando notación de apuntador
34 void copia2(char *s1, const char *s2)
35 {
36 // la copia ocurre en el encabezado del for
37 for (; (*s1 = *s2) != '\0'; s1++, s2++)
38 ; // no hace nada en el cuerpo
39 } // fin de la función copia2

```

```

cadena1 = Hola
cadena3 = Hasta luego

```

**Figura 8.21** | Copia de cadenas usando la notación de arreglos y la notación de apuntadores.

La función `copia1` (líneas 26 a 31) utiliza la notación de subíndice de arreglo para copiar la cadena en `s2` al arreglo de caracteres `s1`. La función declara una variable contador entera `i` para usarla como el subíndice del arreglo. El encabezado de la instrucción `for` (línea 29) realiza toda la operación de copia; su cuerpo es la instrucción vacía. El encabezado especifica que `i` se inicializa con cero y se incrementa en uno durante cada iteración del ciclo. La condición en la instrucción `for`, `( s1[ i ] = s2[ i ] ) != '\0'`, realiza la operación de copia carácter por carácter, de `s2` a `s1`. Cuando se encuentra el carácter nulo en `s2`, se asigna a `s1` y el ciclo termina, ya que el carácter nulo es igual a `'\0'`. Recuerde que el valor de una instrucción de asignación es el valor asignado a su operando izquierdo.

La función `copia2` (líneas 34 a 39) utiliza apuntadores y aritmética de apuntadores para copiar la cadena en `s2` al arreglo de caracteres `s1`. De nuevo, el encabezado de la instrucción `for` (línea 37) realiza toda la operación de copia. El encabezado no incluye ninguna inicialización de variables. Al igual que en la función `copia1`, la condición `( *s1 = *s2 ) != '\0'` realiza la operación de copia. El apuntador `s2` es desreferenciado y el carácter resultante se asigna al apuntador `s1` desreferenciado. Después de la asignación en la condición el ciclo incrementa ambos apuntadores, por lo que apuntan al siguiente elemento del arreglo `s1` y al siguiente carácter de la cadena `s2`, respectivamente. Cuando el ciclo encuentra el carácter nulo en `s2`, éste se asigna al apuntador `s1` desreferenciado y el ciclo termina. Observe que la “porción de incremento” de esta instrucción `for` tiene dos expresiones de incremento separadas por un operador coma.

El primer argumento para `copia1` y `copia2` debe ser un arreglo lo bastante grande como para contener la cadena en el segundo argumento. En cualquier otro caso, un error podría ocurrir cuando hay un intento por escribir en una ubicación de memoria más allá de los límites del arreglo (recuerde que al utilizar arreglos basados en apuntador, no hay comprobación de límites “integrada”). Observe además que el segundo parámetro de cada función se declara como `const char *` (un apuntador a una constante tipo carácter; es decir, una cadena constante). En ambas funciones, el segundo argumento se copia al primer argumento; los caracteres se copian del segundo argumento uno a la vez, pero nunca se modifican. Por lo tanto, el segundo parámetro se declara de manera que apunte a un valor constante para hacer valer el principio del menor privilegio; ninguna de las funciones necesita modificar el segundo argumento, por lo que a ninguna se le permite modificarlo.

## 8.10 Arreglos de apuntadores

Los arreglos pueden contener apuntadores. Un uso común de dicha estructura de datos es formar un arreglo de cadenas basadas en apuntador, lo cual se conoce simplemente como un **arreglo de cadenas**. Cada entrada en el arreglo es una cadena, pero en C++ una cadena es en esencia un apuntador a su primer carácter, por lo que cada entrada en un arreglo de cadenas es simplemente un apuntador al primer carácter de una cadena. Considere la declaración del arreglo de cadenas `palo`, que podría ser útil para representar un mazo de cartas:

```
const char *palo[4] =
{ "Corazones", "Diamantes", "Bastos", "Espadas" };
```

La porción `palo[4]` de la declaración indica un arreglo de cuatro elementos. La porción `const char *` de la declaración indica que cada elemento del arreglo `palo` es de tipo “apuntador a datos constantes `char`”. Los cuatro valores a colocar en el arreglo son “Corazones”, “Diamantes”, “Bastos” y “Espadas”. Cada uno se almacena en memoria como una cadena de caracteres con terminación nula, que es un carácter más grande que el número de caracteres entre comillas. Las cuatro cadenas tienen siete, nueve, seis y siete caracteres de largo (incluyendo sus caracteres nulos de terminación), respectivamente. Aunque parece ser que estas cadenas se colocan en el arreglo `palo`, sólo los apuntadores se guardan en el arreglo, como se muestra en la figura 8.22. Cada apuntador apunta al primer carácter de su cadena correspondiente. Por ende, aun y cuando el arreglo `palo` tiene un tamaño fijo, proporciona acceso a cadenas de caracteres de cualquier longitud. Esta flexibilidad es un ejemplo de las poderosas herramientas de estructuración de datos de C++.

Las cadenas de `palo` podrían colocarse en un arreglo bidimensional, en el cual cada fila representa un palo y cada columna representa una de las letras de un nombre de palo. Dicha estructura de datos debe tener un número fijo de columnas por fila, y ese número debe ser tan grande como la cadena más grande. Por lo tanto, se desperdicia una cantidad considerable de memoria al almacenar un extenso número de cadenas, de las cuales la mayoría son más cortas que la cadena más larga. En la siguiente sección utilizaremos arreglos de cadenas para ayudar a representar un mazo de cartas.

Los arreglos de cadenas se utilizan comúnmente con **argumentos de línea de comandos** que se pasan a la función `main` cuando un programa empieza a ejecutarse. Dichos argumentos van después del nombre del programa cuando éste se ejecuta desde la línea de comandos. Un uso común de los argumentos de línea de comandos es pasar opciones a un programa. Por ejemplo, desde la línea de comandos en una computadora con Windows, el usuario puede escribir

```
dir /p
```

para listar el contenido del directorio actual y hacer una pausa después de cada pantalla de información. Cuando se ejecuta el comando `dir`, la opción `/p` se pasa a `dir` como un argumento de línea de comandos. Dichos argumentos se colocan en un arreglo de cadenas que recibe `main` como un argumento. En el apéndice E, Temas sobre código heredado de C, hablaremos sobre los argumentos de línea de comandos.

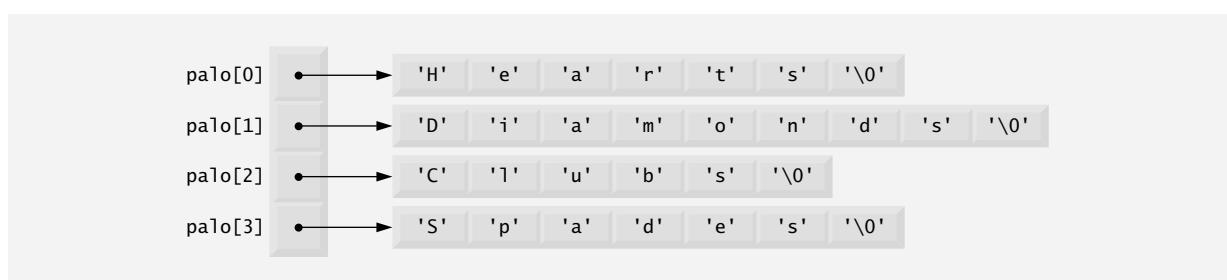


Figura 8.22 | Representación gráfica del arreglo `palo`.

## 8.11 Ejemplo práctico: simulación para barajar y repartir cartas

En esta sección utilizamos la generación de números aleatorios para desarrollar un programa de simulación para barajar y repartir cartas. Así, podemos usar el programa como base para implementar programas que jueguen ciertos juegos de cartas específicos. Para revelar algunos ligeros problemas de rendimiento, hemos utilizado intencionalmente algoritmos subóptimos para barajar y repartir. En los ejercicios desarrollaremos algoritmos más eficientes.

Usando la metodología de mejora paso a paso, de arriba hacia abajo, desarrollaremos un programa para barajar un mazo de 52 cartas de juego, y después repartiremos cada una de las 52 cartas. La metodología de arriba hacia abajo es especialmente útil para atacar problemas más grandes y complejos de los que hemos visto en los capítulos anteriores.

Vamos a utilizar un arreglo bidimensional de 4 por 13 llamado `mazo` para representar el mazo de cartas (figura 8.23). Las filas corresponden a los palos: la fila 0 corresponde a los corazones, la fila 1 a los diamantes, la fila 2 a los bastos y la fila 3 a las espadas. Las columnas corresponden a los valores de las caras de las cartas: las columnas 0 a 9 corresponden a las caras del as al 10, respectivamente, y las columnas 10 a 12 corresponden a la sota, reina y rey, respectivamente. Vamos a cargar el arreglo de cadenas `palos` con cadenas de caracteres que representen los cuatro palos (como en la figura 8.22) y el arreglo de cadenas `cara` con cadenas de caracteres que representen los 13 valores de cartas.

Este mazo simulado de cartas se puede barajar de la siguiente forma. Primero, el arreglo `mazo` de 52 elementos se inicializa con ceros. Después, se elige una `fila` (0 a 3) y una `columna` (0 a 12) al azar. El número 1 se inserta en el elemento `mazo[fila][columna]` del arreglo para indicar que esta carta va a ser la primera que se reparta del mazo barajado. Este proceso continúa, insertando los números 2, 3, ..., 52 al azar en el arreglo `mazo` para indicar cuáles cartas se van a colocar en la posición segunda, tercera, ..., y 52 en el mazo barajado. A medida que el arreglo `mazo` empieza a llenarse con números de cartas, es posible que una carta se seleccione dos veces (es decir, que `mazo[fila][columna]` sea distinto de cero cuando se seleccione). Esta selección simplemente se ignora, y se repiten otras combinaciones de `fila` y `columna` al azar hasta encontrar una carta no seleccionada. En un momento dado, los números del 1 al 52 ocuparán las 52 posiciones del arreglo `mazo`. En este punto, el mazo de cartas se baraja por completo.

Este algoritmo para barajar se podría ejecutar durante un período indefinidamente extenso de tiempo, si las cartas que ya se han barajado se seleccionan repetidas veces al azar. Este fenómeno se conoce como **aplazamiento indefinido** (también conocido como **inanición**). En los ejercicios veremos un algoritmo para barajar más rápido, que también elimina la posibilidad del aplazamiento indefinido.

## **Tip de rendimiento 8.3**



*Algunas veces, los algoritmos que emergen en forma “natural” pueden contener ligeros problemas de rendimiento, como el aplazamiento indefinido. Busque algoritmos que eviten el aplazamiento indefinido.*

Para repartir la primera carta, buscamos en el arreglo el elemento `mazo[fila][columna]` que concuerde con 1. Esto se logra con las instrucciones `for` anidadas que varían `fila` de 0 a 3, y `columna` de 0 a 12. ¿A cuál carta corresponde esa ranura del arreglo? El arreglo `palo` se ha precargado con los cuatro palos, por lo que para obtener el palo, imprimimos la cadena de caracteres `palo[fila]`. De manera similar, para obtener el valor de la cara de la carta, imprimimos la cadena de caracteres `cara[columna]`. También imprimimos la cadena de caracteres " de ". Al imprimir esta

|           | As | Dos | Tres | Cuatro | Cinco | Seis | Siete | Ocho | Nueve | Diez | Sota | Reina | Rey |
|-----------|----|-----|------|--------|-------|------|-------|------|-------|------|------|-------|-----|
| Corazones | 0  | 1   | 2    | 3      | 4     | 5    | 6     | 7    | 8     | 9    | 10   | 11    | 12  |
| Diamantes | 1  |     |      |        |       |      |       |      |       |      |      |       |     |
| Bastos    | 2  |     |      |        |       |      |       |      |       |      |      |       |     |
| Espadas   | 3  |     |      |        |       |      |       |      |       |      |      |       |     |

El mazo [2][12] representa al Rey de Bastos

**Figura 8.23** | Representación de un mazo de cartas mediante un arreglo bidimensional.

información en el orden apropiado, podemos imprimir cada carta en la forma "Rey de Bastos", "As de Diamantes", y así en lo sucesivo.

Vamos a continuar con el proceso de mejoramiento paso a paso, de arriba hacia abajo. La parte superior es simplemente

*Barajar y repartir 52 cartas*

Nuestra primera mejora produce

*Iniciarizar el arreglo palo  
Iniciarizar el arreglo cara  
Iniciarizar el arreglo mazo  
Barajar el mazo  
Repartir 52 cartas*

"Barajar el mazo" puede expandirse de la siguiente manera:

*Para cada una de las 52 cartas*

*Colocar el número de la carta en una posición desocupada del mazo, seleccionada al azar*

"Repartir 52 cartas" se puede expandir de la siguiente manera:

*Para cada una de las 52 cartas*

*Buscar el número de carta en el arreglo mazo e imprimir la cara y el palo de la carta*

Al incorporar estas expansiones, se produce nuestro segundo mejoramiento:

*Iniciarizar el arreglo palo  
Iniciarizar el arreglo cara  
Iniciarizar el arreglo mazo*

*Para cada una de las 52 cartas*

*Colocar el número de carta en una posición desocupada del mazo, seleccionada al azar*

*Para cada una de las 52 cartas*

*Buscar el número de carta en el arreglo mazo e imprimir la cara y el palo de la carta*

"Colocar el número de carta en una posición desocupada del mazo" se puede expandir de la siguiente manera:

*Elegir la posición en el mazo al azar*

*Mientras que la posición elegida del mazo ya se haya elegido antes*

*Elegir posición del mazo al azar*

*Colocar el número de carta en la posición elegida en el mazo*

"Buscar el número de carta en el arreglo mazo e imprimir la cara y el palo de la carta" se puede expandir de la siguiente manera:

*Para cada posición del arreglo mazo*

*Si la posición contiene el número de carta*

*Imprimir la cara y el palo de la carta*

Al incorporar estas expansiones, se produce nuestro tercer mejoramiento (figura 8.24).

Con esto completamos el proceso de refinamiento. La figuras 8.25 a 8.27 contienen el programa para barajar y repartir cartas, junto con una ejecución de ejemplo. En las líneas 61 a 67 de la función `repartir` (figura 8.26) se implementan las líneas 1 a 2 de la figura 8.24. El constructor (líneas 22 a 35 de la figura 8.6) implementa las líneas 1 a 3 de la figura 8.24. La función `barajar` (líneas 38 a 55 de la figura 8.26) implementa las líneas 5 a 11 de la figura 8.24. La función `repartir` (líneas 58 a 88 de la figura 8.26) implementan las líneas 13 a 16 de la figura 8.24. Observe el formato de salida utilizado en la función `repartir` (líneas 81 a 83 de la figura 8.26).

```

1 Inicializar el arreglo palo
2 Inicializar el arreglo cara
3 Inicializar el arreglo mazo
4
5 Para cada una de las 52 cartas
6 Elegir posición del mazo al azar
7
8 Mientras la posición en el mazo ya se haya elegido
9 Elegir posición del mazo al azar
10
11 Colocar el número de carta en la posición elegida en el mazo
12
13 Para cada una de las 52 cartas
14
15 Para cada posición del arreglo mazo
16
17 Si la posición contiene el número deseado de la carta
18 Imprimir la cara y el palo de la carta

```

**Figura 8.24** | Algoritmo en pseudocódigo para el programa de barajar y repartir cartas.

```

1 // Fig. 8.25: MazoDeCartas.h
2 // Definición de la clase MazoDeCartas que
3 // representa un mazo de cartas de juego.
4
5 // definición de la clase MazoDeCartas
6 class MazoDeCartas
7 {
8 public:
9 MazoDeCartas(); // el constructor inicializa el mazo
10 void barajar(); // baraja las cartas en el mazo
11 void repartir(); // reparte las cartas en el mazo
12 private:
13 int mazo[4][13]; // representa el mazo de cartas
14 } // fin de la clase MazoDeCartas

```

**Figura 8.25** | Archivo de encabezado de MazoDeCartas.

```

1 // Fig. 8.26: MazoDeCartas.cpp
2 // Definiciones de las funciones miembro para la clase MazoDeCartas
3 // que simula cómo barajar y repartir un mazo de cartas de juego.
4 #include <iostream>
5 using std::cout;
6 using std::left;
7 using std::right;
8
9 #include <iomanip>
10 using std::setw;
11
12 #include <cstdlib> // prototipos para rand y srand
13 using std::rand;
14 using std::srand;
15
16 #include <ctime> // prototipo para el tiempo
17 using std::time;
18

```

**Figura 8.26** | Definiciones de las funciones miembro para barajar y repartir. (Parte I de 3).

```

19 #include "MazoDeCartas.h" // definición de la clase MazoDeCartas
20
21 // el constructor predeterminado de MazoDeCartas inicializa el mazo
22 MazoDeCartas::MazoDeCartas()
23 {
24 // itera a través de las filas del mazo
25 for (int fila = 0; fila <= 3; fila++)
26 {
27 // itera a través de las columnas del mazo para la fila actual
28 for (int columna = 0; columna <= 12; columna++)
29 {
30 mazo[fila][columna] = 0; // inicializa la posición del mazo en 0
31 } // fin de for interior
32 } // fin de for exterior
33
34 srand(time(0)); // siembra el generador de números aleatorios
35 } // fin del constructor predeterminado de MazoDeCartas
36
37 // baraja las cartas en el mazo
38 void MazoDeCartas::barajar()
39 {
40 int fila; // representa el valor del palo de la carta
41 int columna; // representa el valor de la cara de la carta
42
43 // para cada una de las 52 cartas, selecciona una posición del mazo al azar
44 for (int carta = 1; carta <= 52; carta++)
45 {
46 do // selecciona una nueva posición aleatoria hasta encontrar una desocupada
47 {
48 fila = rand() % 4; // selecciona al azar la fila (0 a 3)
49 columna = rand() % 13; // selecciona al azar la columna (0 a 12)
50 } while(mazo[fila][columna] != 0); // fin de do...while
51
52 // coloca el número de la carta en la posición elegida del mazo
53 mazo[fila][columna] = carta;
54 } // fin de for
55 } // fin de la función barajar
56
57 // reparte las cartas en el mazo
58 void MazoDeCartas::repartir()
59 {
60 // inicializa el arreglo palo
61 static const char *palos[4] =
62 { "Corazones", "Diamantes", "Bastos", "Espadas" };
63
64 // inicializa el arreglo cara
65 static const char *cara[13] =
66 { "As", "Dos", "Tres", "Cuatro", "Cinco", "Seis", "Siete",
67 "Ocho", "Nueve", "Diez", "Sota", "Reina", "Rey" };
68
69 // para cada una de las 52 cartas
70 for (int carta = 1; carta <= 52; carta++)
71 {
72 // itera a través de las filas del mazo
73 for (int fila = 0; fila <= 3; fila++)
74 {
75 // itera a través de las columnas del mazo para la fila actual
76 for (int columna = 0; columna <= 12; columna++)
77 {
78 // si la posición contiene la carta actual, la muestra
79 if (mazo[fila][columna] == carta)

```

Figura 8.26 | Definiciones de las funciones miembro para barajar y repartir. (Parte 2 de 3).

```

80 {
81 cout << setw(5) << right << cara[columna]
82 << " de " << setw(8) << left << palo[fila]
83 << (carta % 2 == 0 ? '\n' : '\t');
84 } // fin de if
85 } // fin de for más interior
86 } // fin de for interior
87 } // fin de for exterior
88 } // fin de la función repartir

```

**Figura 8.26** | Definiciones de las funciones miembro para barajar y repartir. (Parte 3 de 3).

La instrucción de salida imprime la cara justificada a la derecha en un campo de cinco caracteres, e imprime el palo justificado a la izquierda en un campo de ocho caracteres (figura 8.27). La salida se imprime en formato de dos columnas; si la carta que se va a imprimir está en la primera columna, se imprime un tabulador después de la carta para desplazarse a la segunda columna (línea 83); en caso contrario, se imprime una nueva línea.

También hay una debilidad en el algoritmo para repartir. Una vez que se encuentra una concordancia, aun si se encuentra en el primer intento, las dos instrucciones `for` interiores siguen buscando una concordancia en el resto de los elementos de `mazo`. En los ejercicios, corregimos esta deficiencia.

```

1 // Fig. 8.27: fig08_27.cpp
2 // Programa para barajar y repartir cartas.
3 #include "MazoDeCartas.h" // definición de la clase MazoDeCartas
4
5 int main()
6 {
7 MazoDeCartas mazoDeCartas; // crea un objeto MazoDeCartas
8
9 mazoDeCartas.barajar(); // baraja las cartas en el mazo
10 mazoDeCartas.repartir(); // reparte las cartas en el mazo
11 return 0; // indica que terminó correctamente
12 } // fin de main

```

|                    |                     |
|--------------------|---------------------|
| Rey de Corazones   | Nueve de Bastos     |
| Seis de Espadas    | Seis de Corazones   |
| Cuatro de Bastos   | Tres de Diamantes   |
| Dos de Bastos      | Nueve de Espadas    |
| Cinco de Espadas   | Sota de Bastos      |
| Seis de Diamantes  | Ocho de Corazones   |
| Rey de Diamantes   | Cuatro de Corazones |
| Siete de Bastos    | Nueve de Diamantes  |
| As de Corazones    | Cuatro de Diamantes |
| As de Espadas      | Cuatro de Espadas   |
| Diez de Corazones  | Tres de Espadas     |
| As de Diamantes    | Reina de Espadas    |
| Seis de Bastos     | Ocho de Espadas     |
| Reina de Corazones | Rey de Espadas      |
| Reina de Diamantes | Diez de Diamantes   |
| As de Bastos       | Siete de Corazones  |
| Ocho de Diamantes  | Dos de Espadas      |
| Sota de Espadas    | Sota de Diamantes   |
| Rey de Bastos      | Dos de Diamantes    |
| Diez de Espadas    | Nueve de Corazones  |
| Ocho de Bastos     | Dos de Corazones    |
| Tres de Bastos     | Cinco de Bastos     |
| Reina de Bastos    | Siete de Diamantes  |
| Tres de Corazones  | Diez de Bastos      |
| Sota de Corazones  | Cinco de Diamantes  |
| Cinco de Corazones | Siete de Espadas    |

**Fig. 8.27** | Programa para barajar y repartir cartas.

## 8.12 Apuntadores a funciones

Un apuntador a una función contiene la dirección a la función en la memoria. En el capítulo 7 vimos que el nombre de un arreglo es en realidad la dirección en memoria del primer elemento del mismo. De manera similar, el nombre de una función es en realidad la dirección inicial en memoria del código que realiza la tarea de la función. Los apuntadores a funciones se pueden pasar a las funciones, devolver de las funciones, almacenar en arreglos, asignar a otros apuntadores a funciones y usarse para llamar a la función subyacente.

### *Ordenamiento por selección multipropósito mediante el uso de apuntadores a funciones*

Para ilustrar el uso de los apuntadores a funciones, en la figura 8.28 se modifica el programa de ordenamiento por selección de la figura 8.15. La figura 8.28 consiste de `main` (líneas 17 a 55) y las funciones `ordenamientoSelección` (líneas 59 a 76), `intercambiar` (líneas 80 a 85), `ascendente` (líneas 89 a 92) y `descendente` (líneas 96 a 99). La función `ordenamientoSelección` recibe un apuntador a una función (ya sea `ascendente` o `descendente`) como argumento, además del arreglo de enteros a ordenar y del tamaño del arreglo. Las funciones `ascendente` y `descendente` determinan el tipo de ordenamiento. El programa pide al usuario que elija si el arreglo se debe ordenar en forma ascendente o descendente (líneas 24 a 26). Si el usuario introduce 1, se pasa a la función `ordenamientoSelección` un apuntador a la función `ascendente` (línea 37), lo cual hace que el arreglo se ordene en forma ascendente. Si el usuario introduce 2, se pasa a la función `ordenamientoSelección` un apuntador a la función `descendente` (línea 45), lo cual hace que el arreglo se ordene en forma descendente.

El siguiente parámetro aparece en la línea 60 del encabezado de la función `ordenamientoSelección`:

```
bool (*compara)(int, int)
```

Este parámetro especifica un apuntador a una función. La palabra clave `bool` indica que la función a la que apunta devuelve un valor `bool`. El texto `(*compara)` indica el nombre del apuntador a la función (el `*` indica que el parámetro `compara` es un apuntador). El texto `(int, int)` indica que la función a la que apunta `compara` recibe dos argumentos enteros. Los paréntesis se necesitan alrededor de `*compara` para indicar que `compara` es un apuntador a una función. Si no hubiéramos incluido los paréntesis, la declaración sería

```
bool *compara(int, int)
```

la cual declara una función que recibe dos enteros como parámetros, y devuelve un apuntador a un valor `bool`.

```

1 // Fig. 8.28: fig08_28.cpp
2 // Programa de ordenamiento multipropósito que usa apuntadores a funciones.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 // prototipos
12 void ordenamientoSelección(int [], const int, bool (*) (int, int));
13 void intercambiar(int * const, int * const);
14 bool ascendente(int, int); // implementa el orden ascendente
15 bool descendente(int, int); // implementa el orden descendente
16
17 int main()
18 {
19 const int tamanoArreglo = 10;
20 int orden; // 1 = ascendente, 2 = descendente
21 int contador; // subíndice del arreglo
22 int a[tamanoArreglo] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
23
24 cout << "Escriba 1 para orden ascendente,\n"
25 << "Escriba 2 para orden descendente: ";
26 cin >> orden;
27 cout << "\nElementos de datos en el orden original\n";

```

Fig. 8.28 | Programa de ordenamiento multipropósito mediante el uso de apuntadores a funciones. (Parte I de 3).

```
28 // imprime el arreglo original
29 for (contador = 0; contador < tamanoArreglo; contador++)
30 cout << setw(4) << a[contador];
31
32
33 // ordena el arreglo en forma ascendente; pasa la función ascendente
34 // como un argumento para especificar el orden ascendente
35 if (orden == 1)
36 {
37 ordenamientoSeleccion(a, tamanoArreglo, ascendente);
38 cout << "\nElementos de datos en orden ascendente\n";
39 } // fin de if
40
41 // ordena el arreglo en forma descendente; pasa la función descendente
42 // como argumento para especificar el orden en forma descendente
43 else
44 {
45 ordenamientoSeleccion(a, tamanoArreglo, descendente);
46 cout << "\nElementos de datos en orden descendente\n";
47 } // fin de la parte del if...else correspondiente al else
48
49 // imprime el arreglo ordenado
50 for (contador = 0; contador < tamanoArreglo; contador++)
51 cout << setw(4) << a[contador];
52
53 cout << endl;
54 return 0; // indica que terminó correctamente
55 } // fin de main
56
57 // ordenamiento por selección multipropósito; el parámetro compara es
58 // un apuntador a la función de comparación que determina el tipo de orden
59 void ordenamientoSeleccion(int trabajo[], const int tamano,
60 bool (*compara)(int, int))
61 {
62 int menorOMayor; // índice del elemento más pequeño (o grande)
63
64 // itera a través de tamano - 1 elementos
65 for (int i = 0; i < tamano - 1; i++)
66 {
67 menorOMayor = i; // primer índice para el vector remanente
68
69 // itera para encontrar el índice del elemento más pequeño (o más grande)
70 for (int index = i + 1; index < tamano; index++)
71 if (!(*compara)(trabajo[menorOMayor], trabajo[index]))
72 menorOMayor = index;
73
74 intercambiar(&trabajo[menorOMayor], &trabajo[i]);
75 } // fin de if
76 } // fin de la función ordenamientoSeleccion
77
78 // intercambia los valores en las ubicaciones de memoria
79 // a las que apuntan elemento1Ptr y elemento2Ptr
80 void intercambiar(int * const elemento1Ptr, int * const elemento2Ptr)
81 {
82 int contenido = *elemento1Ptr;
83 *elemento1Ptr = *elemento2Ptr;
84 *elemento2Ptr = contenido;
85 } // fin de la función intercambiar
86
87 // determina si el elemento a es menor que
88 // el elemento b para un orden ascendente
89 bool ascendente(int a, int b)
```

Fig. 8.28 | Programa de ordenamiento multipropósito mediante el uso de apunadores a funciones. (Parte 2 de 3).

```

90 {
91 return a < b; // devuelve true si a es menor que b
92 } // fin de la función ascendente
93
94 // determina si el elemento a es mayor que
95 // el elemento b para un orden descendente
96 bool descendente(int a, int b)
97 {
98 return a > b; // devuelve true si a es mayor que b
99 } // fin de la función descendente

```

Escriba 1 para orden ascendente,  
Escriba 2 para orden descendente: 1

Elementos de datos en el orden original  
2 6 4 8 10 12 89 68 45 37

Elementos de datos en orden ascendente  
2 4 6 8 10 12 37 45 68 89

Escriba 1 para orden ascendente,  
Escriba 2 para orden descendente: 2

Elementos de datos en el orden original  
2 6 4 8 10 12 89 68 45 37

Elementos de datos en orden descendente  
89 68 45 37 12 10 8 6 4 2

**Figura 8.28** | Programa de ordenamiento multipropósito mediante el uso de apuntadores a funciones. (Parte 3 de 3).

El parámetro correspondiente en el prototipo de función de `ordenamientoSeleccion` es

```
bool (*) (int, int)
```

Observe que sólo se han incluido los tipos. Como siempre, para fines de documentación, podemos incluir nombres que el compilador ignorará.

La función que se pasa a `ordenamientoSeleccion` se llama en la línea 71, de la siguiente manera:

```
(*compara)(trabajo[menorOMayor], trabajo[indice])
```

Así como se desreferencia un apuntador a una variable para acceder al valor de la variable, también se desreferencia un apuntador a una función para ejecutar la función. Los paréntesis alrededor de `*compara` son necesarios; si se omitieran, el operador `*` trataría de desreferenciar el valor devuelto de la llamada a la función. La llamada a la función se podría haber realizado sin desreferenciar el apuntador, como en

```
compara(trabajo[menorOMayor], trabajo[indice])
```

que utiliza el apuntador directamente como nombre de la función. Nosotros preferimos el primer método de llamar a una función a través de un apuntador, ya que ilustra de manera explícita que `compara` es un apuntador a una función que se desreferencia para llamar a la función. El segundo método de llamar a una función a través de un apuntador hace que parezca como si `compara` fuera el nombre de una función actual en el programa. Esto puede ser confuso para un usuario del programa que quisiera ver la definición de la función `compara` y descubrir que no está definida en el archivo.

El capítulo 22, Biblioteca de plantillas estándar (STL), presenta muchos usos comunes de los apuntadores a funciones.

### Arreglos de apuntadores a funciones

Uno de los usos de los apuntadores a funciones está en los sistemas controlados por menús. Por ejemplo, un programa podría pedir a un usuario que seleccionara una opción de un menú, introduciendo un valor entero. La elección del usuario se puede usar como subíndice en un arreglo de apuntadores a funciones, y el apuntador en el arreglo se puede utilizar para llamar a la función.

La figura 8.29 proporciona un ejemplo mecánico que demuestra cómo declarar y usar un arreglo de apuntadores a funciones. El programa define tres funciones: `funcion0`, `funcion1` y `funcion2`; cada una de ellas recibe un argumento entero y no devuelve un valor. En la línea 17 se almacenan apuntadores a estas funciones en el arreglo `f`. En este caso, todas las funciones a las que apunta el arreglo deben tener el mismo tipo de valor de retorno y los mismos tipos de parámetros.

```
1 // Fig. 8.29: fig08_29.cpp
2 // Demostración de un arreglo de apunadores a funciones.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 // prototipos de funciones -- cada función realiza acciones similares
9 void funcion0(int);
10 void funcion1(int);
11 void funcion2(int);
12
13 int main()
14 {
15 // inicializa un arreglo de 3 apunadores a funciones, cada una
16 // de las cuales recibe un argumento int y devuelve void
17 void (*f[3])(int) = { funcion0, funcion1, funcion2 };
18
19 int opcion;
20
21 cout << "Escriba un numero entre 0 y 2, 3 para terminar: ";
22 cin >> opcion;
23
24 // procesa la opción del usuario
25 while ((opcion >= 0) && (opcion < 3))
26 {
27 // invoca la función en la selección de ubicación en
28 // el arreglo f y pasa la opción como un argumento
29 (*f[opcion])(opcion);
30
31 cout << "Escriba un numero entre 0 y 2, 3 para terminar: ";
32 cin >> opcion;
33 } // fin de while
34
35 cout << "Se completo la ejecucion del programa." << endl;
36 return 0; // indica que terminó correctamente
37 } // fin de main
38
39 void funcion0(int a)
40 {
41 cout << "Usted escribio " << a << " por lo que se llamo a la funcion0\n\n";
42 } // fin de la función funcion0
43
44 void funcion1(int b)
45 {
46 cout << "Usted escribio " << b << " por lo que se llamo a la funcion1\n\n";
47 } // fin de la función funcion1
48
49 void funcion2(int c)
50 {
51 cout << "Usted escribio " << c << " por lo que se llamo a la funcion2\n\n";
52 } // fin de la función funcion2
```

Escriba un numero entre 0 y 2, 3 para terminar: 0  
Usted escribio 0 por lo que se llamo a la funcion0

Escriba un numero entre 0 y 2, 3 para terminar: 1  
Usted escribio 1 por lo que se llamo a la funcion1

Escriba un numero entre 0 y 2, 3 para terminar: 2  
Usted escribio 2 por lo que se llamo a la funcion2

Escriba un numero entre 0 y 2, 3 para terminar: 3  
Se completo la ejecucion del programa.

Figura 8.29 | Arreglo de apunadores a funciones.

La declaración en la línea 17 se lee empezando en el conjunto de paréntesis de más a la izquierda de la siguiente forma: “`f` es un arreglo de tres apuntadores a funciones, cada una de las cuales recibe un valor `int` como argumento y devuelve `void`”. El arreglo se inicializa con los nombres de las tres funciones (que, de nuevo, son apuntadores). El programa pide al usuario que introduzca un número entre 0 y 2, o 3 para finalizar. Cuando el usuario introduce un valor entre 0 y 2 éste se utiliza como subíndice en el arreglo de apuntadores a funciones. En la línea 29 se invoca una de las funciones en el arreglo `f`. En la llamada, `f[opcion]` selecciona el apuntador en la ubicación `opcion` en el arreglo. El apuntador se desreferencia para llamar a la función, y se pasa `opcion` como argumento para la función. Cada función imprime el valor de su argumento y el nombre de su función para indicar que la función se llama en forma correcta. En los ejercicios el lector desarrollará un sistema controlado por menús. En el capítulo 13, Programación orientada a objetos: polimorfismo, veremos que los desarrolladores de compiladores utilizan arreglos de apuntadores a funciones para implementar los mecanismos que dan soporte a las funciones `virtual`; la tecnología clave detrás del polimorfismo.

## 8.13 Introducción al procesamiento de cadenas basadas en apuntador

En esta sección presentaremos algunas funciones comunes de la Biblioteca estándar de C++ que facilitan el procesamiento de cadenas. Las técnicas aquí descritas son apropiadas para desarrollar editores de texto, procesadores de palabras, software para diseño de páginas, sistemas de tipografía computarizados y demás tipos de software de procesamiento de texto. Ya hemos usado la clase `string` de la Biblioteca estándar de C++ en varios ejemplos para representar cadenas como objetos completos. Por ejemplo, el ejemplo práctico `LibroCalificaciones` de los capítulos 3 a 7 representa el nombre de un curso mediante el uso de un objeto `string`. En el capítulo 18 presentaremos la clase `string` con detalle. Aunque el uso de objetos `string` es bastante simple, en esta sección vamos a usar cadenas basadas en apuntador con terminación nula. Muchas funciones de la Biblioteca estándar de C++ operan sólo con cadenas basadas en apuntador con terminación nula, que son más complicadas de usar que los objetos `string`. Además, si trabaja con programas de C++ heredados, tal vez tenga que manipular estas cadenas basadas en apuntador.

### 8.13.1 Fundamentos de caracteres y cadenas basadas en apuntador

Los caracteres son los bloques de construcción fundamentales de los programas de código fuente de C++. Todo programa está compuesto de una secuencia de caracteres que (al agruparse de una manera significativa) el compilador interpreta como una serie de instrucciones que se utilizan para realizar una tarea. Un programa puede contener **constantes tipo carácter**. Una constante carácter es un valor entero que se representa como un carácter entre comillas sencillas. El valor de una constante carácter es el valor entero del carácter en el conjunto de caracteres del equipo. Por ejemplo, '`z`' representa el valor entero de `z` (122 en el conjunto de caracteres ASCII; vea el apéndice B), y '`\n`' representa el valor entero de nueva línea (10 en el conjunto de caracteres ASCII).

Una cadena es una serie de caracteres se trata como una sola unidad. Una cadena puede incluir letras, dígitos y diversos **caracteres especiales** tales como `+`, `-`, `*`, `/` y `$`. Las **literales de cadena**, o **constantes de cadena** en C++ se escriben entre dobles comillas, como se muestra a continuación:

|                                       |                          |
|---------------------------------------|--------------------------|
| <code>"John Q. Doe"</code>            | (un nombre)              |
| <code>"9999 Main Street"</code>       | (una calle)              |
| <code>"Maynard, Massachusetts"</code> | (una ciudad y un estado) |
| <code>"(201) 555-1212"</code>         | (un número telefónico)   |

Una cadena basada en apuntador en C++ es un arreglo de caracteres que termina con el carácter nulo (`'\0'`), el cual indica dónde termina la cadena en memoria. Una cadena se utiliza a través de un apuntador a su primer carácter. El valor de una cadena es la dirección del primer carácter, pero el valor `sizeof` de una literal de cadena es la longitud de la cadena, incluyendo el carácter nulo de terminación. En este sentido, las cadenas son como los arreglos, debido a que el nombre de un arreglo es también un apuntador a su primer elemento.

Una literal de cadena se puede utilizar como inicializador en la declaración de un arreglo de caracteres, o de una variable de tipo `char *`. Cada una de las declaraciones

```
char color[] = "azul";
const char *colorPtr = "azul";
```

inicializa una variable con la cadena `"azul"`. La primera declaración crea un arreglo de cinco elementos llamado `color`, el cual contiene los caracteres '`a`', '`z`', '`u`', '`l`' y '`\0`'. La segunda declaración crea la variable apuntador `colorPtr` que apunta a la letra `b` en la cadena `"azul"` (que termina en '`\0`') en alguna parte de la memoria. Las literales de cadena tienen la clase de almacenamiento `static` (existen durante la ejecución del programa) y pueden o no compartirse, si se hace referencia a la misma literal de cadena de varias ubicaciones en un programa. De acuerdo con el estándar de C++

(sección 2.13.4), el efecto de tratar de modificar una literal de cadena es indefinido; por ende, siempre se debe declarar un apuntador a una literal de cadena como `const char *`.

La declaración `char color[] = "azul";` también se podría escribir como

```
char color[] = { 'a', 'z', 'u', 'l', '\0' };
```

Al declarar un arreglo de caracteres para que contenga una cadena, el arreglo debe ser lo bastante grande como para almacenar la cadena con su carácter nulo de terminación. La anterior declaración determina el tamaño del arreglo, con base en el número de inicializadores proporcionados en la lista inicializadora.



### Error común de programación 8.15

*Si no se asigna suficiente espacio en un arreglo de caracteres para almacenar el carácter nulo que termina una cadena, se produce un error.*



### Error común de programación 8.16

*Crear o usar una cadena estilo C que no contiene un carácter nulo de terminación puede producir errores lógicos.*



### Tip para prevenir errores 8.4

*Al almacenar una cadena de caracteres en un arreglo de caracteres, asegúrese que el arreglo sea lo bastante grande como para contener la cadena más larga que se vaya a almacenar. C++ permite almacenar cadenas de cualquier longitud. Si una cadena es mayor que el arreglo de caracteres en la que se va a almacenar, los caracteres que estén más allá del final del arreglo sobrescribirán datos en las ubicaciones de memoria que sigan después del arreglo, lo cual produce errores lógicos.*

Una cadena se puede leer y colocar en un arreglo de caracteres mediante la extracción de flujos con `cin`. Por ejemplo, la siguiente instrucción se puede utilizar para colocar una cadena en el arreglo de caracteres `palabra[ 20 ]`:

```
cin >> palabra;
```

La cadena introducida por el usuario se almacena en `palabra`. La anterior instrucción lee caracteres hasta encontrar un carácter de espacio en blanco o un indicador de fin de archivo. Observe que la cadena no debe ser mayor de 19 caracteres, para dejar espacio al carácter nulo de terminación. Se puede utilizar el manipulador de flujo `setw` para asegurar que la cadena que se coloque en `palabra` no excede al tamaño del arreglo. Por ejemplo, la instrucción

```
cin >> setw(20) >> palabra;
```

especifica que `cin` debe leer un máximo de 19 caracteres en el arreglo `palabra` y guardar la ubicación número 20 en el arreglo para almacenar el carácter nulo de terminación para la cadena. El manipulador de flujo `setw` sólo se aplica al siguiente valor que se va a recibir como entrada. Si se introducen más de 19 caracteres, el resto de los mismos no se almacena en `palabra`, sino que se leerá y se puede guardar en otra variable.

En algunos casos es conveniente recibir como entrada una línea completa de texto y colocarla en un arreglo. Para este fin, C++ proporciona la función `cin.getline` en el archivo de encabezado `<iostream>`. En el capítulo 3 le presentamos una función similar, llamada `getline` del archivo de encabezado `<string>`, la cual lee la entrada hasta que se introduce un carácter de nueva línea, y almacena la entrada (sin el carácter de nueva línea) en un objeto `string` que se especifica como argumento. La función `cin.getline` recibe tres argumentos: un arreglo de caracteres en el que se va a almacenar la línea de texto, una longitud y un carácter delimitador. Por ejemplo, el siguiente segmento de programa

```
char enunciado[80];
cin.getline(enunciado, 80, '\n');
```

declara el arreglo `enunciado` de 80 caracteres y lee una línea de texto del teclado, para colocarla en el arreglo. La función deja de leer caracteres al momento de encontrar el carácter delimitador '`\n`', cuando se introduce el indicador de fin de archivo o cuando el número de caracteres leídos hasta ese momento sea uno menos que la longitud especificada en el segundo argumento. (El último carácter en el arreglo se reserva para el carácter nulo de terminación). Al encontrar el carácter delimitador, éste se lee y se descarta. El tercer argumento para `cin.getline` tiene '`\n`' como valor predeterminado, por lo que la llamada a la función anterior podría escribirse de la siguiente manera:

```
cin.getline(enunciado, 80);
```

El capítulo 15, Entrada y salida de flujos, proporciona una discusión detallada sobre `cin.getline` y otras funciones de entrada/salida.



### Error común de programación 8.17

Procesar un carácter individual como una cadena `char *` puede producir un error fatal en tiempo de ejecución. Una cadena `char *` es un apuntador; probablemente un entero bastante grande. Sin embargo, un carácter es un pequeño entero (los valores ASCII varían de 0 a 255). En muchos sistemas, al desreferenciar un valor `char` se produce un error, ya que las direcciones inferiores de memoria están reservadas para fines especiales, como los manejadores de interrupciones del sistema operativo; por lo tanto, ocurren “violaciones de acceso a memoria”.



### Error común de programación 8.18

Pasar una cadena como argumento para una función cuando se espera un carácter es un error de compilación.

## 8.13.2 Funciones para manipular cadenas de la biblioteca para manejo de cadenas

La biblioteca para manejar cadenas proporciona muchas funciones útiles para manipular datos de cadena, comparar cadenas, buscar caracteres y otras cadenas en cadenas, descomponer cadenas en tokens (separar cadenas en piezas lógicas, como las palabras separadas en un enunciado) y determinar la longitud de cadenas. En esta sección presentaremos algunas funciones comunes de manipulación de cadenas de la biblioteca para manejo de cadenas (de la biblioteca estándar de C++). Las funciones se sintetizan en la figura 8.30; después se utiliza cada una de ellas en un ejemplo de código activo. Los prototipos para estas funciones se encuentran en el archivo de encabezado `<cstring>`.

Observe que varias funciones de la figura 8.30 contienen parámetros con el tipo de datos `size_t`. Este tipo está definido en el archivo de encabezado `<cstring>` como un tipo integral sin signo, tal como `unsigned int` o `unsigned long`.



### Error común de programación 8.19

Olvidar incluir el archivo de encabezado `<cstring>` al utilizar funciones de la biblioteca de manejo de cadenas produce errores de compilación.

| Prototipo de función                                                  | Descripción de función                                                                                                                                                                                                                                                                                                                                       |
|-----------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>char *strcpy( char *s1, const char *s2 );</code>                | Copia la cadena <code>s2</code> en el arreglo de caracteres <code>s1</code> . Se devuelve el valor de <code>s1</code> .                                                                                                                                                                                                                                      |
| <code>char *strncpy( char *s1, const char *s2, size_t n );</code>     | Copia como máximo <code>n</code> caracteres de la cadena <code>s2</code> y los coloca en el arreglo de caracteres <code>s1</code> . Se devuelve el valor de <code>s1</code> .                                                                                                                                                                                |
| <code>char *strcat( char *s1, const char *s2 );</code>                | Adjunta la cadena <code>s2</code> a <code>s1</code> . El primer carácter de <code>s2</code> sobrescribe el carácter nulo de terminación de <code>s1</code> . Se devuelve el valor de <code>s1</code> .                                                                                                                                                       |
| <code>char *strncat( char *s1, const char *s2, size_t n );</code>     | Adjunta como máximo <code>n</code> caracteres de la cadena <code>s2</code> a la cadena <code>s1</code> . El primer carácter de <code>s2</code> sobrescribe el carácter nulo de terminación de <code>s1</code> . Se devuelve el valor de <code>s1</code> .                                                                                                    |
| <code>int strcmp( const char *s1, const char *s2 );</code>            | Compara la cadena <code>s1</code> con la cadena <code>s2</code> . La función devuelve un valor de cero, menor que cero o mayor que cero si <code>s1</code> es igual a, menor que, o mayor que <code>s2</code> , respectivamente.                                                                                                                             |
| <code>int strncmp( const char *s1, const char *s2, size_t n );</code> | Compara hasta <code>n</code> caracteres de la cadena <code>s1</code> con la cadena <code>s2</code> . La función devuelve cero, menor que cero o mayor que cero, si la porción del carácter <code>n</code> de <code>s1</code> es igual a, menor que, o mayor que la correspondiente porción del carácter <code>n</code> de <code>s2</code> , respectivamente. |

Figura 8.30 | Funciones de manipulación de cadenas de la biblioteca para manejo de cadenas. (Parte I de 2).

| Prototipo de función                               | Descripción de función                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|----------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| char * <b>strtok</b> ( char *s1, const char *s2 ); | Una secuencia de llamadas a <b>strtok</b> divide la cadena s1 en "tokens" (piezas lógicas, como las palabras en una línea de texto). La cadena se divide con base en los caracteres contenidos en la cadena s2. Por ejemplo, si descomponemos la cadena "esta:es:una:cadena" en tokens con base en el carácter ':', los tokens resultantes serían "esta", "es", "una" y "cadena". La función <b>strtok</b> sólo devuelve un token a la vez; la primera llamada contiene s1 como primer argumento, y las llamadas subsiguientes para seguir dividiendo en tokens la misma cadena contienen NULL como primer argumento. En cada llamada se devuelve un apuntador al token actual. Si no hay más tokens al momento de llamar a la función, se devuelve NULL. |
| size_t <b>strlen</b> ( const char *s );            | Determina la longitud de la cadena s. Se devuelve el número de caracteres antes del carácter nulo de terminación.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |

Figura 8.30 | Funciones de manipulación de cadenas de la biblioteca para manejo de cadenas. (Parte 2 de 2).

### Copia de cadenas mediante **strcpy** y **strncpy**

La función **strcpy** copia su segundo argumento (una cadena) en su primer argumento: un arreglo de caracteres que debe ser lo bastante grande como para almacenar la cadena y su carácter nulo de terminación (que también se copia). La función **strncpy** es muy parecida a **strcpy**, sólo que **strncpy** especifica el número de caracteres a copiar de la cadena al arreglo. Observe que la función **strncpy** no necesariamente copia el carácter nulo de terminación de su segundo argumento; sólo se escribe un carácter nulo de terminación si el número de caracteres a copiar es por lo menos uno más que la longitud de la cadena. Por ejemplo, si "prueba" es el segundo argumento, sólo se escribe un carácter nulo de terminación si el tercer argumento para **strncpy** es por lo menos 5 (cuatro caracteres en "prueba" más un carácter nulo de terminación). Si el tercer argumento es mayor que 5, se adjuntan caracteres nulos al arreglo hasta que se escriba el número total de caracteres especificados por el tercer argumento.

### Error común de programación 8.20



Al usar **strncpy**, el carácter nulo de terminación del segundo argumento (una cadena **char** \*) no se copiará si el número de caracteres especificado por el tercer argumento de **strncpy** no es mayor que la longitud del segundo argumento. En ese caso, puede ocurrir un error fatal si no se termina en forma manual la cadena **char** \* resultante con un carácter nulo.

La figura 8.31 usa **strcpy** (línea 17) para copiar la cadena completa del arreglo x al arreglo y, y usa una instrucción **strncpy** (línea 23) para copiar los primeros 14 caracteres del arreglo x al arreglo z. En la línea 24 se adjunta un carácter nulo ('\0') al arreglo z, ya que la llamada a **strncpy** en el programa no escribe un carácter nulo de terminación. (El tercer argumento es menor que la longitud de cadena del segundo argumento más uno).

```

1 // Fig. 8.31: fig08_31.cpp
2 // Uso de strcpy y strncpy.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // prototipos para strcpy y strncpy
8 using std::strcpy;
9 using std::strncpy;
10
11 int main()
12 {
13 char x[] = "Feliz cumpleaños a ti"; // longitud de cadena: 22
14 char y[25];
15 char z[18];

```

Figura 8.31 | **strcpy** y **strncpy**. (Parte 1 de 2).

```

16 strcpy(y, x); // copia el contenido de x en y
17
18 cout << "La cadena en el arreglo x es: " << x
19 << "\nLa cadena en el arreglo y es: " << y << '\n';
20
21
22 // copia los primeros 17 caracteres de x a z
23 strncpy(z, x, 17); // no copia el carácter nulo
24 z[17] = '\0'; // adjunta '\0' al contenido de z
25
26 cout << "La cadena en el arreglo z es: " << z << endl;
27 return 0; // indica que terminó correctamente
28 } // fin de main

```

```

La cadena en el arreglo x es: Feliz cumpleanios a ti
La cadena en el arreglo y es: Feliz cumpleanios a ti
La cadena en el arreglo z es: Feliz cumpleanios

```

Figura 8.31 | strcpy y strncpy. (Parte 2 de 2).

### Concatenación de cadenas con strcat y strncat

La función **strcat** adjunta su segundo argumento (una cadena) a su primer argumento (un arreglo de caracteres que contiene una cadena). El primer carácter del segundo argumento reemplaza al carácter nulo ('\0') que termina la cadena en el primer argumento. Hay que asegurarse que el arreglo utilizado para almacenar la primera cadena sea lo bastante grande como para almacenar la combinación de la primera cadena, la segunda cadena y el carácter nulo de terminación (que se copia de la segunda cadena). La función **strncat** adjunta un número específico de caracteres de la segunda cadena a la primera cadena, y adjunta un carácter nulo de terminación al resultado. El programa de la figura 8.32 demuestra la función **strcat** (líneas 19 y 29) y la función **strncat** (línea 24).

```

1 // Fig. 8.32: fig08_32.cpp
2 // Uso de strcat y strncat.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // prototipos para strcat y strncat
8 using std::strcat;
9 using std::strncat;
10
11 int main()
12 {
13 char s1[20] = "Feliz "; // longitud: 6
14 char s2[] = "Anio Nuevo "; // longitud: 11
15 char s3[40] = "";
16
17 cout << "s1 = " << s1 << "\ns2 = " << s2;
18
19 strcat(s1, s2); // concatena s2 con s1 (longitud: 17)
20
21 cout << "\n\nDespues de strcat(s1, s2):\ns1 = " << s1 << "\ns2 = " << s2;
22
23 // concatena los primeros 6 caracteres de s1 con s3
24 strncat(s3, s1, 6); // coloca '\0' despues del ultimo caracter
25
26 cout << "\n\nDespues de strncat(s3, s1, 6):\ns1 = " << s1
27 << "\ns3 = " << s3;
28
29 strcat(s3, s1); // concatena s1 con s3

```

Figura 8.32 | strcat y strncat. (Parte 1 de 2).

```

30 cout << "\n\nDespues de strcat(s3, s1):\ns1 = " << s1
31 << "\ns3 = " << s3 << endl;
32 return 0; // indica que terminó correctamente
33 } // fin de main

```

```

s1 = Feliz
s2 = Anio Nuevo

Despues de strcat(s1, s2):
s1 = Feliz Anio Nuevo
s2 = Anio Nuevo

Despues de strcat(s3, s1, 6):
s1 = Feliz Anio Nuevo
s3 = Feliz

Despues de strcat(s3, s1):
s1 = Feliz Anio Nuevo
s3 = Feliz Feliz Anio Nuevo

```

Figura 8.32 | `strcat` y `strncat`. (Parte 2 de 2).

### Comparación de cadenas con `strcmp` y `strncmp`

La figura 8.33 compara tres cadenas mediante el uso de `strcmp` (líneas 21, 22 y 23) y `strncmp` (líneas 26, 27 y 28). La función `strcmp` compara su primer argumento con su segundo argumento de cadena, carácter por carácter. La función devuelve cero si las cadenas son iguales, un valor negativo si la primera cadena es menor que la segunda, y un valor positivo si la primera cadena es mayor que la segunda. La función `strncmp` es equivalente a `strcmp`, excepto que `strncmp` compara hasta cierto número especificado de caracteres. La función `strncmp` deja de comparar caracteres si llega al carácter nulo en uno de sus argumentos de cadena. El programa imprime el valor entero devuelto por cada llamada a la función.

### Error común de programación 8.21



*Suponer que `strcmp` y `strncmp` devuelven uno (un valor true) cuando sus argumentos son iguales es un error lógico. Ambas funciones devuelven cero (el valor false de C++) para la igualdad. Por lo tanto, al evaluar la igualdad de dos cadenas, el resultado de la función `strcmp` o `strncmp` debe compararse con cero para determinar si las cadenas son iguales.*

Para comprender cuál es el significado de que una cadena sea “mayor” o “menor” que otra, considere el proceso de alfabetizar una serie de apellidos. Sin duda, colocaríamos a “Jones” antes que “Smith”, ya que la primera letra de “Jones” está antes que la primera letra de “Smith” en el alfabeto. Pero el alfabeto es algo más que sólo una lista de letras; es una lista *ordenada* de caracteres. Cada letra ocurre en una posición específica dentro de la lista. “Z” es más que sólo una letra del alfabeto; “Z” es específicamente la última letra del alfabeto.

```

1 // Fig. 8.33: fig08_33.cpp
2 // Uso de strcmp y strncmp.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 #include <cstring> // prototipos para strcmp y strncmp
11 using std::strcmp;
12 using std::strncmp;
13
14 int main()
15 {
16 char *s1 = "Felices Fiestas";
17 char *s2 = "Felices Fiestas";
18 char *s3 = "Felices Dias de fiesta";

```

Figura 8.33 | `strcmp` y `strncmp`. (Parte 1 de 2).

```

19
20 cout << "s1 = " << s1 << "\ns2 = " << s2 << "\ns3 = " << s3
21 << "\n\nstrcmp(s1, s2) = " << setw(2) << strcmp(s1, s2)
22 << "\nstrcmp(s1, s3) = " << setw(2) << strcmp(s1, s3)
23 << "\nstrcmp(s3, s1) = " << setw(2) << strcmp(s3, s1);
24
25 cout << "\n\nstrncmp(s1, s3, 8) = " << setw(2)
26 << strncmp(s1, s3, 8) << "\nstrncmp(s1, s3, 9) = " << setw(2)
27 << strncmp(s1, s3, 9) << "\nstrncmp(s3, s1, 9) = " << setw(2)
28 << strncmp(s3, s1, 9) << endl;
29 return 0; // indica que terminó correctamente
30 } // fin de main

```

```

s1 = Felices Fiestas
s2 = Felices Fiestas
s3 = Felices Dias de fiesta

strcmp(s1, s2) = 0
strcmp(s1, s3) = 1
strcmp(s3, s1) = -1

strncmp(s1, s3, 8) = 0
strncmp(s1, s3, 9) = 1
strncmp(s3, s1, 9) = -1

```

Figura 8.33 | strcmp y strncmp. (Parte 2 de 2).

¿Cómo sabe la computadora qué letra se encuentra antes de otra? Dentro de la computadora, todos los caracteres se representan en forma de código numérico; cuando la computadora compara dos cadenas, en realidad compara los códigos numéricos de los caracteres en las cadenas.

En un esfuerzo por estandarizar las representaciones de los caracteres, la mayoría de los fabricantes de computadora han diseñado sus equipos para utilizar uno de dos esquemas populares de codificación: ASCII o EBCDIC. Recuerde que ASCII significa “Código estándar estadounidense para el intercambio de información”. EBCDIC significa “Código extendido de intercambio decimal codificado en binario”. También existen otros esquemas de codificación.

ASCII y EBCDIC se conocen como **códigos de caracteres**, o conjuntos de caracteres. La mayoría de los lectores de este libro utilizan computadoras de escritorio o portátiles que utilizan el conjunto de caracteres ASCII. Las computadoras mainframe de IBM utilizan el conjunto de caracteres EBCDIC. A medida que el uso de Internet y World Wide Web se ha vuelto dominante, el conjunto de caracteres Unicode, que es más reciente, ha crecido en popularidad ([www.unicode.org](http://www.unicode.org)). En realidad, las manipulaciones de cadenas y caracteres implican la manipulación de los códigos numéricos apropiados, y no los caracteres en sí. Esto explica que puedan intercambiarse caracteres y enteros pequeños en C++. Como es importante decir que un código numérico es mayor que, menor o igual que otro código numérico, es posible relacionar varios caracteres o cadenas entre sí al hacer referencia a los códigos de caracteres. El apéndice B contiene los códigos de caracteres ASCII.

#### Tip de portabilidad 8.4

*Los códigos numéricos internos utilizados para representar caracteres pueden ser distintos en las distintas computadoras que utilizan distintos conjuntos de caracteres.*

#### Tip de portabilidad 8.5

*No debemos evaluar explícitamente los códigos ASCII, como en if ( calificacion == 65 ); en vez de ello, hay que utilizar la correspondiente constante carácter, como en if ( calificacion == 'A' ).*

[Nota: con algunos compiladores, las funciones strcmp y strncmp siempre devuelven -1, 0 o 1, como en los resultados de ejemplo de la figura 8.33. Con otros compiladores, estas funciones devuelven 0 o la diferencia entre los códigos numéricos de los primeros caracteres que difieren en las cadenas que se están comparando. Por ejemplo, cuando se comparan s1 y s3, los primeros caracteres que difieren entre ellos son el primer carácter de la segunda palabra en cada cadena: F (código numérico 70) en s1 y D (código numérico 68) en s3, respectivamente. En este caso, el valor de retorno será 6 (o -6 si s3 se compara con s1)].

### Dividir una cadena en tokens mediante strtok

La función `strtok` divide una cadena en una serie de **tokens**. Un token es una secuencia de caracteres separada por **caracteres delimitadores** (por lo general, espacios o signos de puntuación). Por ejemplo, en una línea de texto cada palabra se puede considerar como un token, y los espacios que separan a las palabras se pueden considerar como delimitadores.

Se requieren varias llamadas a `strtok` para dividir una cadena en tokens (asumiendo que la cadena contenga más de un token). La primera llamada a `strtok` contiene dos argumentos, una cadena a dividir en tokens y una cadena que contiene los caracteres que separan a los tokens (es decir, delimitadores). En la línea 19 de la figura 8.34 se asigna a `tokenPtr` un apuntador al primer token en `enunciado`. El segundo argumento, " ", indica que los tokens en `enunciado` se separan mediante espacios. La función `strtok` busca el primer carácter en `enunciado` que no sea delimitador (espacio). Esto empieza el primer token. Después, la función busca el siguiente carácter delimitador en la cadena y lo reemplaza con un carácter nulo ('\0'). Esto termina el token actual. La función `strtok` guarda (en una variable `static`) un apuntador al siguiente carácter después del token en `enunciado` y devuelve un apuntador al token actual.

```
1 // Fig. 8.34: fig08_34.cpp
2 // Uso de strtok para dividir una cadena en tokens.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // prototipo para strtok
8 using std::strtok;
9
10 int main()
11 {
12 char enunciado[] = "Este es un enunciado con 7 tokens";
13 char *tokenPtr;
14
15 cout << "La cadena a dividir en tokens es:\n" << enunciado
16 << "\n\nLos tokens son:\n\n";
17
18 // empieza la division de enunciado en tokens
19 tokenPtr = strtok(enunciado, " ");
20
21 // continua dividiendo enunciado en tokens hasta que tokenPtr se vuelve NULL
22 while (tokenPtr != NULL)
23 {
24 cout << tokenPtr << '\n';
25 tokenPtr = strtok(NULL, " "); // obtiene el siguiente token
26 } // fin de while
27
28 cout << "\nDespues de strtok, enunciado = " << enunciado << endl;
29 return 0; // indica que terminó correctamente
30 } // fin de main
```

La cadena a dividir en tokens es:  
Este es un enunciado con 7 tokens

Los tokens son:

Este  
es  
un  
enunciado  
con  
7  
tokens

Despues de strtok, enunciado = Este

Figura 8.34 | Uso de `strtok` para dividir una cadena en tokens.

Las llamadas subsiguientes a `strtok` para seguir dividiendo `enunciado` en tokens contienen NULL como su primer argumento (línea 25). El argumento NULL indica que la llamada a `strtok` debe seguir dividiendo el enunciado en tokens, desde la ubicación en `enunciado` guardada por la última llamada a `strtok`. Observe que `strtok` mantiene esta información almacenada de una forma que no es visible para el programador. Si no quedan tokens cuando se llama a `strtok`, la función devuelve NULL. El programa de la figura 8.34 utiliza a `strtok` para dividir en tokens la cadena "Este es un enunciado con 7 tokens". El programa imprime cada token en una línea separada. En la línea 28 se imprime `enunciado` después de dividirlo en tokens. Observe que `strtok` modifica la cadena de entrada; por lo tanto, debe realizarse una copia de la cadena si el programa requiere la original después de las llamadas a `strtok`. Cuando `enunciado` se imprime después de la división en tokens, observe que sólo se imprime la palabra "Este", ya que `strtok` reemplazó cada espacio en blanco en `enunciado` con un carácter nulo ('\0') durante el proceso de división en tokens.

### Error común de programación 8.22



*Si no se toma en cuenta que `strtok` modifica la cadena que se va a dividir en tokens, al tratar de utilizar esa cadena como si fuera la original sin modificación se produce un error lógico.*

### Determinar longitudes de cadenas

La función `strlen` recibe una cadena como argumento y devuelve el número de caracteres en la misma; el carácter nulo de terminación no se incluye en la longitud. La longitud es también el subíndice del carácter nulo. El programa de la figura 8.35 demuestra el uso de la función `strlen`.

```

1 // Fig. 8.35: fig08_35.cpp
2 // Uso de strlen.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // prototipo para strlen
8 using std::strlen;
9
10 int main()
11 {
12 char *cadena1 = "abcdefghijklmnopqrstuvwxyz";
13 char *cadena2 = "cuatro";
14 char *cadena3 = "Boston";
15
16 cout << "La longitud de \"\" << cadena1 << "\"" es " << strlen(cadena1)
17 << "\nLa longitud de \"\" << cadena2 << "\"" es " << strlen(cadena2)
18 << "\nLa longitud de \"\" << cadena3 << "\"" es " << strlen(cadena3)
19 << endl;
20
21 return 0; // indica que terminó correctamente
22 } // fin de main

```

```

La longitud de "abcdefghijklmnopqrstuvwxyz" es 26
La longitud de "cuatro" es 6
La longitud de "Boston" es 6

```

Figura 8.35 | `strlen` devuelve la longitud de una cadena `char *`.

## 8.14 Repaso

En este capítulo proporcionamos una introducción detallada a los apuntadores, o variables que contienen direcciones como sus valores. Empezamos por demostrar cómo declarar e inicializar apuntadores. Vimos cómo utilizar el operador dirección (&) para asignar la dirección de una variable a un apuntador, y el operador indirección (\*) para acceder a los datos almacenados en la variable a la que un apuntador hace referencia indirecta. Hablamos sobre cómo pasar argumentos por referencia, usando argumentos tipo apuntador y tipo referencia.

Aprendió a usar `const` con apuntadores para hacer valer el principio del menor privilegio. Demostramos cómo usar apuntadores no constantes a datos no constantes, apuntadores no constantes a datos constantes, apuntadores constantes a

datos no constantes y apuntadores constantes a datos constantes. Después utilizamos el ordenamiento por selección para demostrar cómo pasar arreglos y elementos individuales de arreglos por referencia. Hablamos sobre el operador `sizeof`, que se puede usar para determinar los tamaños de los tipos de datos y las variables en bytes, durante la compilación de un programa.

Demostramos cómo usar apuntadores en aritmética y expresiones de comparación. Vimos cómo se puede utilizar la aritmética de apuntadores para saltar de un elemento de un arreglo a otro. Aprendió a usar arreglos de apuntadores, y en forma más específica los arreglos de cadenas. Hablamos sobre los apuntadores a funciones, que permiten a los programadores pasar funciones como parámetros. Presentamos varias funciones de C++ que manipulan cadenas basadas en apuntador. Aprendió acerca de las herramientas de procesamiento de cadenas, como las que se usan para copiar cadenas, dividir cadenas en tokens y determinar la longitud de las cadenas.

En el siguiente capítulo comenzaremos con nuestro tratamiento más detallado sobre las clases. Aprenderá acerca del alcance de los miembros de una clase, y cómo mantener los objetos en un estado consistente. También aprenderá acerca del uso de funciones miembro especiales, llamadas constructores y destructores, las cuales se ejecutan cuando un objeto se crea y se destruye, respectivamente; después hablaremos sobre cuándo se hacen las llamadas a los constructores y destructores. Además, demostraremos el uso de argumentos predeterminados con constructores, y el uso de la asignación a nivel de miembros para asignar un objeto de una clase con otro objeto de la misma clase. También hablaremos sobre el peligro de devolver una referencia a un miembro de datos `private` de una clase.

## Resumen

### Sección 8.2 Declaraciones e inicialización de variables apuntadores

- Los apuntadores son variables que contienen direcciones de memoria de otras variables como sus valores.
- La declaración

```
int *ptr;
```

declara a `ptr` como un apuntador a una variable de tipo `int*` y se lee así: “`ptr` es un apuntador a un valor `int`”. El uso que se da aquí al carácter `*` en una declaración indica que la variable es un apuntador.

- Hay tres valores que se pueden usar para inicializar un apuntador: 0, `NULL` o una dirección de un objeto del mismo tipo. Inicializar un apuntador con 0 e inicializar ese mismo apuntador con `NULL` son procesos idénticos; 0 es la convención en C++.
- El único entero que se puede asignar a un apuntador sin conversión de tipos es cero.

### Sección 8.3 Operadores de apuntadores

- El operador `&` (dirección) obtiene la dirección de memoria de su operando.
- El operando del operador dirección debe ser el nombre de una variable (o de otro valor *lvalue*); el operador dirección no se puede aplicar a constantes o expresiones que no devuelvan una referencia.
- El operador `*`, que se conoce como el operador de indirección (o desreferencia), devuelve un sinónimo, alias o sobrenombre para el objeto al que apunta su operando en la memoria. A esto se le conoce como desreferenciar el apuntador.

### Sección 8.4 Paso de argumentos a funciones por referencia mediante apuntadores

- Al llamar a una función con un argumento que la función que hace la llamada desea que la función llamada modifique, se puede pasar la dirección del argumento. Así, la función llamada utiliza el operador indirección (`*`) para desreferenciar el apuntador y modificar el valor del argumento en la función que hace la llamada.
- Una función que recibe una dirección como argumento debe tener un apuntador como su correspondiente parámetro.

### Sección 8.5 Uso de `const` con apuntadores

- El calificador `const` nos permite informar al compilador que el valor de una variable específica no se debe modificar a través del identificador especificado. Si hay un intento de modificar un valor `const`, el compilador genera una advertencia o un error, dependiendo del compilador específico.
- Hay cuatro formas de pasar un apuntador a una función: un apuntador no constante a datos no constantes, un apuntador no constante a datos constantes, un apuntador constante a datos no constantes y un apuntador constante a datos constantes.
- El valor del nombre de un arreglo es la dirección del primer elemento del arreglo.
- Para pasar un solo elemento de un arreglo por referencia mediante el uso de apuntadores, se debe pasar la dirección del elemento del arreglo.

### Sección 8.6 Ordenamiento por selección mediante el uso del paso por referencia

- El algoritmo de ordenamiento por selección es un algoritmo de ordenamiento fácil de programar, pero por desgracia ineficiente. En la primera iteración del algoritmo se selecciona el elemento más pequeño en el arreglo y se intercambia con el primer elemento. En la segunda iteración se selecciona el segundo elemento más pequeño (que es el elemento más pequeño del elemento restante) y se intercambia con el segundo elemento. El algoritmo continúa hasta que en la última iteración se selecciona el segundo elemento más grande y se intercambia con el penúltimo subíndice, dejando el elemento más grande en el último subíndice. Después de la  $i$ -ésima iteración, los  $i$  elementos más pequeños del arreglo se ordenarán en forma ascendente en los primeros  $i$  elementos del arreglo.

### Sección 8.7 Operador `sizeof`

- C++ proporciona el operador unario `sizeof` para determinar el tamaño de un arreglo (o de cualquier otro tipo de datos, variable o constante) en bytes, en tiempo de compilación.
- Cuando se aplica al nombre de un arreglo, el operador `sizeof` devuelve el número total de bytes en el arreglo como un entero.

### Sección 8.8 Expresiones y aritmética de apuntadores

- Las operaciones aritméticas que se pueden realizar con los apuntadores son: incrementar (++) o decrementar (--) un apuntador, sumar (+ o +=) un entero a un apuntador, restar (- o -=) un entero de un apuntador, y restar un apuntador de otro apuntador del mismo tipo.
- Cuando se suma o resta un entero a un apuntador, éste se incrementa o decremente con base en ese entero multiplicado por el tamaño del objeto al que hace referencia el apuntador.
- Un apuntador se puede asignar a otro, si ambos son del mismo tipo. En caso contrario, debe usarse una conversión de tipos. La excepción a esto es el apuntador `void *`, el cual es un tipo de apuntador genérico que puede contener valores de apuntadores de cualquier tipo. A los apuntadores a `void` se les puede asignar apuntadores de otros tipos de datos. Un apuntador `void *` puede asignarse a un apuntador de otro tipo sólo mediante una conversión de tipo explícita.
- Las únicas operaciones válidas en un apuntador `void *` son: comparar apuntadores `void *` con otros apuntadores, asignar direcciones a apuntadores `void *` y convertir apuntadores `void *` a tipos de apuntadores válidos.
- Los apuntadores se pueden comparar mediante el uso de los operadores de igualdad y relacionales. Las comparaciones mediante operadores relacionales sólo tienen significado si los apuntadores apuntan a miembros del mismo arreglo.

### Sección 8.9 Relación entre apuntadores y arreglos

- Los apuntadores que apuntan a arreglos aceptan el uso de subíndices de la misma forma que los nombres de arreglos.
- En la notación apuntador/desplazamiento, si el apuntador apunta al primer elemento del arreglo, el desplazamiento es igual que un subíndice de arreglo.
- Todas las expresiones de arreglos con subíndices se pueden escribir con un apuntador y un desplazamiento, usando el nombre del arreglo como apuntador, o usando un apuntador separado que apunte al arreglo.

### Sección 8.10 Arreglos de apuntadores

- Los arreglos pueden contener apuntadores.
- Dicha estructura de datos se puede utilizar para formar un arreglo de cadenas basadas en apuntadores, a lo cual se le conoce como arreglo de cadenas. Cada entrada en el arreglo es una cadena, pero en C++ una cadena es en esencia un apuntador a su primer carácter, por lo que cada entrada en un arreglo de cadenas es simplemente un apuntador al primer carácter de una cadena.
- Los arreglos de cadenas se utilizan comúnmente con argumentos de línea de comandos que se pasan a `main` cuando un programa empieza a ejecutarse. Dichos argumentos van después del nombre del programa, cuando éste se ejecuta desde la línea de comandos.

### Sección 8.11 Ejemplo práctico: simulación para barajar y repartir cartas

- El aplazamiento indefinido (también conocido como inanición) ocurre cuando un algoritmo se puede ejecutar durante un período de tiempo indefinidamente largo.

### Sección 8.12 Apuntadores a funciones

- Un apuntador a una función es la dirección donde reside el código para la función.
- Los apuntadores a funciones se pueden pasar a las funciones, devolver de las funciones, almacenar en arreglos y asignar a otros apuntadores.
- Un uso común de los apuntadores a funciones es en los sistemas controlados por menús. Los apuntadores a funciones se utilizan para seleccionar la función que se va a llamar para un elemento de menú específico.

### Sección 8.13 Introducción al procesamiento de cadenas basadas en apuntador

- La función `strcpy` copia su segundo argumento (una cadena) a su primer argumento (un arreglo de caracteres). El programador debe asegurar que el arreglo de destino sea lo bastante grande como para almacenar la cadena y su carácter nulo de terminación.

- La función `strncpy` es equivalente a `strcpy`, excepto que una llamada a `strncpy` especifica el número de caracteres a copiar de la cadena hacia el arreglo. El carácter nulo de terminación se copia sólo si el número de caracteres a copiar es por lo menos uno más que la longitud de la cadena.
- La función `strcat` adjunta su segundo argumento de cadena (incluyendo el carácter nulo de terminación) a su primer argumento de cadena. El primer carácter de la segunda cadena reemplaza al carácter nulo ('\0') de la primera cadena. El programador debe asegurar que el arreglo de destino usado para almacenar la primera cadena sea lo bastante grande como para almacenar la primera y la segunda cadenas.
- La función `strncat` es equivalente a `strcat`, excepto que una llamada a `strncat` adjunta un número especificado de caracteres de la segunda cadena a la primera cadena. Al resultado se adjunta un carácter nulo de terminación.
- La función `strcmp` compara su primer argumento de cadena con su segundo argumento de cadena, carácter por carácter. La función devuelve cero si las cadenas son iguales, un valor negativo si la primera cadena es menor que la segunda, y un valor positivo si la primera cadena es mayor que la segunda.
- La función `strncmp` es equivalente a `strcmp`, excepto que `strncmp` compara un número especificado de caracteres. Si el número de caracteres en una de las cadenas es menor que el número de caracteres especificado, `strncmp` compara caracteres hasta encontrar el carácter nulo en la cadena más corta.
- Una secuencia de llamadas a `strtok` divide una cadena en tokens, que están separados por los caracteres contenidos en un segundo argumento de cadena. La primera llamada especifica la cadena que se va a dividir en tokens como el primer argumento, y las llamadas subsiguientes para seguir dividiendo en tokens la misma cadena especifican NULL como primer argumento. La función devuelve un apuntador al token actual de cada llamada. Si no hay más tokens al llamar a `strtok`, se devuelve NULL.
- La función `strlen` recibe una cadena como argumento y devuelve el número de caracteres en la cadena; el carácter nulo de terminación no se incluye en la longitud de la cadena.

## Terminología

|                                                                           |                                                                                      |
|---------------------------------------------------------------------------|--------------------------------------------------------------------------------------|
| & (operador dirección)                                                    | decrementar un apuntador                                                             |
| * (desreferencia de apuntador, u operador indirección)                    | desplazamiento para un apuntador                                                     |
| '\0' (carácter nulo)                                                      | desreferenciar un apuntador                                                          |
| algoritmo de ordenamiento por selección                                   | desreferenciar un apuntador a 0                                                      |
| aplazamiento indefinido                                                   | dividir cadenas en tokens                                                            |
| apuntador a función                                                       | EBCDIC (Código extendido de intercambio decimal codificado en binario)               |
| apuntador a una función                                                   | <code>getline</code> , función de <code>cin</code>                                   |
| apuntador constante                                                       | hacer referencia directa a un valor                                                  |
| apuntador constante a datos constantes                                    | hacer referencia indirecta a un valor                                                |
| apuntador constante a datos no constantes                                 | inanición                                                                            |
| apuntador no constante a datos constantes                                 | incrementar un apuntador                                                             |
| apuntador no constante a datos no constantes                              | indirección                                                                          |
| apuntador nulo                                                            | intercambiar arreglos y apuntadores                                                  |
| argumentos de línea de comandos                                           | <code>islower</code> , función ( <code>&lt;cctype&gt;</code> )                       |
| aritmética de apuntadores                                                 | llamar funciones por referencia                                                      |
| arreglo de apuntadores a funciones                                        | modificar la dirección almacenada en una variable apuntador                          |
| arreglo de cadenas                                                        | modificar un apuntador constante                                                     |
| ASCII (Código estándar estadounidense para el intercambio de información) | operador desreferencia (*)                                                           |
| cadena con terminación nula                                               | operador desreferencia de apuntador (*)                                              |
| cadena que se va a dividir en tokens                                      | operador dirección (&)                                                               |
| cadenas basadas en apuntador                                              | operador indirección (*)                                                             |
| carácter delimitador                                                      | paso por referencia con argumentos tipo apuntador                                    |
| carácter nulo ('\0')                                                      | paso por referencia con argumentos tipo referencia                                   |
| carácter nulo de terminación                                              | referencia a datos constantes                                                        |
| caracteres especiales                                                     | referencia a los elementos de un arreglo                                             |
| código de caracteres                                                      | resta de apuntadores                                                                 |
| comparación de cadenas                                                    | <code>size_t</code> , tipo                                                           |
| concatenación de cadenas                                                  | <code>sizeof</code> , operador                                                       |
| <code>const</code> con parámetros de función                              | <code>strcat</code> , función del archivo de encabezado <code>&lt;cstring&gt;</code> |
| constante carácter                                                        | <code>strcmp</code> , función del archivo de encabezado <code>&lt;cstring&gt;</code> |
| constante de cadena                                                       | <code>strcpy</code> , función del archivo de encabezado <code>&lt;cstring&gt;</code> |
| copia de cadenas                                                          | <code>strlen</code> , función del archivo de encabezado <code>&lt;cstring&gt;</code> |

`strncat`, función del archivo de encabezado `<cstring>`  
`strcmp`, función del archivo de encabezado `<cstring>`  
`strncpy`, función del archivo de encabezado `<cstring>`  
`strtok`, función del archivo de encabezado `<cstring>`

token  
`toupper`, función (`<cctype>`)  
 Unicode, conjunto de caracteres

## Ejercicios de autoevaluación

**8.1** Complete los siguientes enunciados:

- a) Un apuntador es una variable que contiene como valor la \_\_\_\_\_ de otra variable.
- b) Los tres valores que se pueden utilizar para inicializar un apuntador son \_\_\_\_\_, \_\_\_\_\_ y \_\_\_\_\_.
- c) El único entero que se puede asignar directamente a un apuntador es \_\_\_\_\_.

**8.2** Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

- a) El operador dirección & se puede aplicar sólo a constantes y expresiones.
- b) Un apuntador que se declara de tipo `void *` se puede desreferenciar.
- c) Los apuntadores de distintos tipos no se pueden asignar entre sí sin una operación de conversión de tipos.

**8.3** Para cada uno de los siguientes enunciados, escriba instrucciones en C++ que realicen la tarea especificada. Suponga que los números de punto flotante con precisión doble se almacenan en ocho bytes, y que la dirección inicial del arreglo está en la ubicación 1002500 en la memoria. Cada parte del ejercicio debe usar los resultados de incisos anteriores, donde sea apropiado.

- a) Declarar un arreglo de tipo `double` llamado `numeros` con 10 elementos, e inicializar los elementos con los valores 0.0, 1.1, 2.2, ..., 9.9. Suponga que se ha definido la constante simbólica `TAMANIO` como 10.
- b) Declarar un apuntador `nPtr` que apunte a una variable de tipo `double`.
- c) Usar una instrucción `for` para imprimir los elementos del arreglo `numeros` mediante el uso de notación de subíndice. Imprima cada número con una posición de precisión a la derecha del punto decimal.
- d) Escribir dos instrucciones separadas, cada una de las cuales debe asignar la dirección inicial del arreglo `numeros` a la variable apuntador `nPtr`.
- e) Usar una instrucción `for` para imprimir los elementos del arreglo `numeros`, usando la notación apuntador/desplazamiento con el apuntador `nPtr`.
- f) Usar una instrucción `for` para imprimir los elementos del arreglo `numeros`, usando la notación apuntador/desplazamiento con el nombre del arreglo como el apuntador.
- g) Usar una instrucción `for` para imprimir los elementos del arreglo `numeros`, usando la notación apuntador/subíndice con el apuntador `nPtr`.
- h) Hacer referencia al cuarto elemento del arreglo `numeros`, usando la notación de subíndice de arreglo, la notación apuntador/desplazamiento con el nombre del arreglo como apuntador, la notación de subíndice de apuntador con `nPtr` y la notación apuntador/desplazamiento con `nPtr`.
- i) Suponiendo que `nPtr` apunte al inicio del arreglo `numeros`, ¿qué dirección se desreferencia mediante `nPtr + 8`? ¿Qué valor se almacena en esa ubicación?
- j) Suponiendo que `nPtr` apunta a `numeros[ 5 ]`, ¿qué dirección se referencia mediante `nPtr` después de ejecutar `nPtr -= 4`? ¿Cuál es el valor almacenado en esa ubicación?

**8.4** Para cada uno de los siguientes enunciados, escriba una sola instrucción que realice la tarea especificada. Suponga que se han declarado las variables de punto flotante `numero1` y `numero2`, y que `numero1` se ha inicializado con 7.3. Suponga que la variable `ptr` es de tipo `char *`. Suponga que los arreglos `s1` y `s2` son arreglos `char` de 100 elementos cada uno, y que se inicializan con literales de cadena.

- a) Declarar la variable `fPtr` para que sea un apuntador a un objeto de tipo `double`.
- b) Asignar la dirección de la variable `numero1` a la variable apuntador `fPtr`.
- c) Imprimir el valor del objeto al que apunta `fPtr`.
- d) Asignar a la variable `numero2` el valor del objeto al que apunta `fPtr`.
- e) Imprimir el valor de `numero2`.
- f) Imprimir la dirección de `numero1`.
- g) Imprimir la dirección almacenada en `fPtr`. ¿El valor que se imprime es el mismo que la dirección de `numero1`?
- h) Copiar la cadena almacenada en el arreglo `s2` al arreglo `s1`.
- i) Comparar la cadena en `s1` con la cadena en `s2`, e imprimir el resultado.
- j) Adjuntar los primeros 10 caracteres de la cadena en `s1` a la cadena en `s2`.
- k) Determinar la longitud de la cadena en `s1` e imprimir el resultado.
- l) Asignar a `ptr` la ubicación del primer token en `s2`. Los delimitadores de los tokens son comas ( , ).

- 8.5** Realice la tarea especificada por cada uno de los siguientes enunciados:
- Escribir el encabezado para una función llamada `intercambiar`, que reciba dos apuntadores a los números de punto flotante con precisión doble `x` y `y` como parámetros, y no devuelva un valor.
  - Escribir el prototipo para la función en la parte (a).
  - Escribir el encabezado para una función llamada `evaluar`, que devuelva un entero y reciba como parámetros el entero `x` y un apuntador a la función `poly`. Esta función debe recibir un parámetro entero y devolver un entero.
  - Escribir el prototipo para la función en la parte (c).
  - Escribir dos instrucciones, cada una de las cuales debe inicializar el arreglo de caracteres `vocal` con la cadena de vocales "AEIOU".
- 8.6** Busque el error en cada uno de los siguientes segmentos de programa. Suponga las siguientes declaraciones e instrucciones:
- ```
int *zPtr;           // zPtr hará referencia al arreglo z
int *aPtr = 0;
void *sPtr = 0;
int numero;
int z[ 5 ] = { 1, 2, 3, 4, 5 };
a) ++zPtr;
b) // usa el apuntador para obtener el primer valor del arreglo
   numero = zptr;
c) // asigna el elemento 2 del arreglo (el valor 3) a numero
   numero = *zPtr[ 2 ];
d) // imprime el arreglo z completo
   for ( int i = 0; i <= 5; i++ )
       cout << zPtr[ i ] << endl;
e) // asigna a numero el valor al que apunta sPtr
   numero = *sPtr;
f) ++z;
g) char s[ 10 ];
   cout << strncpy( s, "holo", 4 ) << endl;
h) char s[ 12 ];
   strcpy( s, "Bienvenido a casa";
i) if ( strcmp( cadena1, cadena2 ) )
    cout << "Las cadenas son iguales" << endl;
```
- 8.7** ¿Qué se imprime (si acaso) cuando se ejecuta cada una de las siguientes instrucciones? Si la instrucción contiene un error, descríbalo e indique cómo corregirlo. Suponga las siguientes declaraciones de variables:

```
char s1[ 50 ] = "jack";
char s2[ 50 ] = "jill";
char s3[ 50 ];

a) cout << strcpy( s3, s2 ) << endl;
b) cout << strcat( strcat( strcpy( s3, s1 ), " y " ), s2 )
   << endl;
c) cout << strlen( s1 ) + strlen( s2 ) << endl;
d) cout << strlen( s3 ) << endl;
```

Respuestas a los ejercicios de autoevaluación

- 8.1** a) dirección. b) 0, NULL, una dirección. c) 0.
- 8.2**
- Falso. El operando del operador dirección debe ser un *lvalue*; el operador dirección no se puede aplicar a constantes o expresiones que no den referencias como resultado.
 - Falso. Un apuntador a `void` no se puede desreferenciar. Dicho apuntador no tiene un tipo que permita al compilador determinar el número de bytes de memoria a desreferenciar, y el tipo de datos a los que apunta el apuntador.
 - Falso. Se pueden asignar apuntadores de cualquier tipo a apuntadores `void`. Los apuntadores de tipo `void` se pueden asignar a apuntadores de otros tipos sólo con una conversión de tipos explícita.
- 8.3**
- `double numeros[TAMANIO] = { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9 };`
 - `double *nPtr;`

390 Capítulo 8 Apuntadores y cadenas basadas en apuntadores

- c)

```
cout << fixed << showpoint << setprecision( 1 );
for ( int i = 0; i < TAMANIO; i++ )
    cout << numeros[ i ] << ' ';
```
- d)

```
nPtr = numeros;
nPtr = &numeros[ 0 ];
```
- e)

```
cout << fixed << showpoint << setprecision( 1 );
for ( int j = 0; j < TAMANIO; j++ )
    cout << *( nPtr + j ) << ' ';
```
- f)

```
cout << fixed << showpoint << setprecision( 1 );
for ( int k = 0; k < TAMANIO; k++ )
    cout << *( numeros + k ) << ' ';
```
- g)

```
cout << fixed << showpoint << setprecision( 1 );
for ( int m = 0; m < TAMANIO; m++ )
    cout << nPtr[ m ] << ' ';
```
- h)

```
numeros[ 3 ]
*( numeros + 3 )
nPtr[ 3 ]
*( nPtr + 3 )
```
- i) La dirección es $1002500 + 8 * 8 = 1002564$. El valor es 8.8.
- j) La dirección de numeros[5] es $1002500 + 5 * 8 = 1002540$.
La dirección de nPtr -= 4 es $1002540 - 4 * 8 = 1002508$.
El valor en esa ubicación es 1.1.

8.4

- a)

```
double *fPtr;
```
- b)

```
fPtr = &numero1;
```
- c)

```
cout << "El valor de *fPtr es " << *fPtr << endl;
```
- d)

```
numero2 = *fPtr;
```
- e)

```
cout << "El valor de numero2 es " << numero2 << endl;
```
- f)

```
cout << "La direccion de numero1 es " << &numero1 << endl;
```
- g)

```
cout << "La direccion almacenada en fPtr es " << fPtr << endl;
```


Sí, el valor es el mismo.
- h)

```
strcpy( s1, s2 );
```
- i)

```
cout << "strcmp(s1, s2) = " << strcmp( s1, s2 ) << endl;
```
- j)

```
strncat( s1, s2, 10 );
```
- k)

```
cout << "strlen(s1) = " << strlen( s1 ) << endl;
```
- l)

```
ptr = strtok( s2, "," );
```

8.5

- a)

```
void intercambiar( double *x, double *y )
```
- b)

```
void intercambiar( double *, double * );
```
- c)

```
int evaluar( int x, int (*poly)( int ) )
```
- d)

```
int evaluar( int, int (*)( int ) );
```
- e)

```
char vocal[] = "AEIOU";
char vocal[] = { 'A', 'E', 'I', 'O', 'U', '\0' };
```

8.6

- a) *Error:* no se ha inicializado zPtr.
Corrección: inicializar zPtr con $zPtr = z$;
- b) *Error:* el apuntador no se desreferencia.

Corrección: cambiar la instrucción a $numero = *zPtr$;

- c) *Error:* zPtr[2] no es un apuntador y no se debe desreferenciar.
Corrección: cambiar $*zPtr[2]$ a $zPtr[2]$.

- d) *Error:* se hace referencia a un elemento del arreglo fuera de los límites de éste, con subíndice de apuntador.
Corrección: para evitar esto, cambie el operador relacional en la instrucción for a <.

- e) *Error:* se desreferencia un apuntador a void.

Corrección: para desreferenciar el apuntador void, primero se debe convertir en un apuntador entero. Cambie la instrucción a $numero = *static_cast< int *>(sPtr)$;

- f) *Error:* tratar de modificar el nombre de un arreglo con la aritmética de apuntadores.

Corrección: usar una variable apuntador en vez del nombre del arreglo para realizar la aritmética de apuntadores, o usar un subíndice con el nombre del arreglo para hacer referencia a un elemento específico.

- g) *Error:* la función `strncpy` no escribe un carácter nulo de terminación en el arreglo `s`, ya que su tercer argumento es igual a la longitud de la cadena "holá".
Corrección: hacer que 5 sea el tercer argumento de `strncpy`, o asignar '\0' a `s[4]` para asegurar que se agregue el carácter nulo de terminación a la cadena.
- h) *Error:* el arreglo de caracteres `s` no es lo bastante grande como para almacenar el carácter nulo de terminación.
Corrección: declarar el arreglo con más elementos.
- i) *Error:* la función `strcmp` devolverá 0 si las cadenas son iguales; por lo tanto, la condición en la instrucción `if` será falsa, y la instrucción de salida no se ejecutará.
Corrección: comparar explícitamente el resultado de `strcmp` con 0 en la condición de la instrucción `if`.

- 8.7**
- a) `jill`
 - b) `jack y jill`
 - c) 8
 - d) 13

Ejercicios

- 8.8** Indique si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.
- a) Dos apuntadores que apuntan a distintos arreglos no se pueden comparar de una forma que tenga sentido.
 - b) Como el nombre de un arreglo es un apuntador al primer elemento del arreglo, los nombres de los arreglos se pueden manipular precisamente de la misma forma que los apuntadores.
- 8.9** Para cada uno de los siguientes enunciados, escriba instrucciones de C++ que realicen la tarea especificada. Suponga que los enteros sin signo se almacenan en dos bytes y que la dirección inicial del arreglo está en la ubicación 1002500 en memoria.
- a) Declarar un arreglo de tipo `unsigned int` llamado `valores` con cinco elementos, e inicializar los elementos con los enteros pares del 2 al 10. Suponga que la constante simbólica `TAMANIO` se ha definido como 5.
 - b) Declarar un apuntador `vPtr` que apunte a un objeto del tipo `unsigned int`.
 - c) Usar una instrucción `for` para imprimir los elementos del arreglo `valores` mediante el uso de la notación de subíndices.
 - d) Escribir dos instrucciones separadas que asigan la dirección inicial del arreglo `valores` a la variable apuntador `vPtr`.
 - e) Usar una instrucción `for` para imprimir los elementos del arreglo `valores` usando la notación apuntador/desplazamiento.
 - f) Usar una instrucción `for` para imprimir los elementos del arreglo `valores` usando la notación apuntador/desplazamiento, con el nombre del arreglo como apuntador.
 - g) Usar una instrucción `for` para imprimir los elementos del arreglo `valores`, mediante el uso de subíndices con el apuntador al arreglo.
 - h) Hacer referencia al quinto elemento de `valores` mediante el uso de la notación de subíndices de arreglo, la notación apuntador/desplazamiento con el nombre del arreglo como apuntador, la notación de subíndice de apuntador y la notación apuntador/desplazamiento.
 - i) ¿Qué dirección se referencia mediante `vPtr + 3`? ¿Qué valor se almacena en esa ubicación?
 - j) Suponiendo que `vPtr` apunta a `valores[4]`, ¿qué dirección se referencia mediante `vPtr - 4`? ¿Qué valor se almacena en esa ubicación?
- 8.10** Para cada uno de los siguientes enunciados, escriba una sola instrucción que realice la tarea especificada. Suponga que las variables `long` llamadas `valor1` y `valor2` se hayan declarado, y que `valor1` se haya inicializado con 200000.
- a) Declarar la variable `longPtr` para que sea un apuntador a un objeto de tipo `long`.
 - b) Asignar la dirección de la variable `valor1` a la variable apuntador `longPtr`.
 - c) Imprimir el valor del objeto al que apunta `longPtr`.
 - d) Asignar a la variable `valor2` el valor del objeto al que apunta `longPtr`.
 - e) Imprimir el valor de `valor2`.
 - f) Imprimir la dirección de `valor1`.
 - g) Imprimir la dirección almacenada en `longPtr`. ¿El valor que se imprimió es igual que la dirección de `valor1`?
- 8.11** Realice la tarea especificada en cada uno de los siguientes enunciados:
- a) Escribir el encabezado para la función `cero`, que reciba un parámetro tipo arreglo entero largo llamado `enterosGrandes`, y que no devuelva un valor.
 - b) Escriba el prototipo para la función en la parte(a).

- c) Escriba el encabezado para la función `sumar1YSumar`, que reciba un parámetro tipo arreglo entero `unoDemasiado-Chico` y devuelva un entero.
- d) Escribir el prototipo para la función descrita en la parte (c).

Nota: los ejercicios 8.12 al 8.15 son razonablemente complicados. Una vez que haya resuelto estos problemas, podrá implementar muchos juegos de cartas populares.

8.12 Modifique el programa de la figura 8.27, de manera que la función para repartir cartas reparta una mano de póquer de cinco cartas. Después escriba funciones para realizar cada una de las siguientes acciones:

- a) Determinar si la mano contiene un par.
- b) Determinar si la mano contiene dos pares.
- c) Determinar si la mano contiene tres cartas de un tipo (por ejemplo, tres sotas).
- d) Determinar si la mano contiene cuatro cartas de un tipo (por ejemplo, cuatro ases).
- e) Determinar si la mano contiene un “flush” (es decir, cinco cartas del mismo palo).
- f) Determinar si la mano contiene cinco cartas con valores de cara consecutivos.

8.13 Use las funciones desarrolladas en el ejercicio 8.12 para escribir un programa que reparta dos manos de póquer de cinco cartas, que evalúe cada mano y determine cuál es mejor.

8.14 Modifique el programa desarrollado en el ejercicio 8.13, de manera que pueda simular el repartidor. La mano de cinco cartas del repartidor se reparte “boca abajo”, de manera que el jugador no pueda verla. El programa debe entonces evaluar la mano del repartidor, y con base en la calidad de la misma, el repartidor debe pedir una, dos o tres cartas más para reemplazar el número correspondiente de cartas innecesarias en la mano original. El programa debe entonces volver a evaluar la mano del repartidor. [Precaución: ¡éste es un problema difícil!]

8.15 Modifique el programa desarrollado en el ejercicio 8.14 de manera que maneje la mano del repartidor, pero que permita al repartidor cuáles cartas de la mano desea reemplazar. El programa deberá entonces evaluar ambas manos y determinar quién ganó. Ahora use este nuevo programa para jugar 20 veces contra la computadora. ¿Quién gana más juegos, usted o la computadora? Haga que uno de sus amigos juegue 20 veces contra la computadora. ¿Quién gana más juegos? Con base en los resultados de estos juegos, realice las modificaciones apropiadas para refinar su programa para jugar póquer. [Nota: esto también es un problema difícil]. Juegue 20 veces más. ¿Su programa modificado juega mejor?

8.16 En el programa para barajar y repartir cartas de las figuras 8.25 a 8.27, utilizamos de manera intencional un algoritmo inefficiente para barajar, el cual introdujo la posibilidad de aplazamiento indefinido. En este problema, creará un algoritmo de alto rendimiento para barajar, que evita el aplazamiento indefinido.

Modifique las figuras 8.25 a 8.27 como se muestra a continuación. Inicialice el arreglo `mazo` como se muestra en la figura 8.36. Modifique la función `barajar` para iterar fila por fila y columna por columna a través del arreglo, pasando por cada elemento una vez. Cada elemento deberá intercambiarse con un elemento del arreglo seleccionado al azar. Imprima el arreglo resultante para determinar si el mazo se baraja en forma satisfactoria (como en la figura 8.37, por ejemplo). Tal vez quiera que su programa llame a la función `barajar` varias veces para asegurar un proceso de barajado satisfactorio.

Arreglo mazo sin barajar												
0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11	12
1	14	15	16	17	18	19	20	21	22	23	24	25
2	27	28	29	30	31	32	33	34	35	36	37	38
3	40	41	42	43	44	45	46	47	48	49	50	51
												52

Figura 8.36 | El arreglo `mazo` sin barajar.

Ejemplo del arreglo mazo barajado												
0	1	2	3	4	5	6	7	8	9	10	11	12
0	19	40	27	25	36	46	10	34	35	41	18	2
1	13	28	14	16	21	30	8	11	31	17	24	7
2	12	33	15	42	43	23	45	3	29	32	4	47
3	50	38	52	39	48	51	9	5	37	49	22	6
												20

Figura 8.37 | Ejemplo del arreglo `mazo` barajado.

Observe que, aunque el método en este problema mejora el algoritmo para barajar, el algoritmo para repartir aún requiere buscar en el arreglo `mazo` la carta 1, después la carta 2, después la carta 3, y así en lo sucesivo. Peor aún, incluso después de que el algoritmo para repartir localiza y reparte la carta, el algoritmo continúa buscando a través del resto del mazo. Modifique el programa de las figuras 8.25 a 8.27 de manera que una vez que se reparta una carta, no haya más intentos por relacionar ese número de carta, y el programa proceda de inmediato a repartir la siguiente carta.

8.17 (Simulación: La tortuga y la liebre) En este ejercicio usted recreará la clásica carrera de la tortuga y la liebre. Utilizará la generación de números aleatorios para desarrollar una simulación de este memorable suceso.

Nuestros competidores empezarán la carrera en la “posición 1” de 70 posiciones. Cada posición representa a una posible posición a lo largo del curso de la carrera. La línea de meta se encuentra en la posición 70. El primer competidor en llegar a la posición 70 recibirá una cubeta llena con zanahorias y lechuga frescas. El recorrido se abre paso hasta la cima de una resbalosa montaña, por lo que ocasionalmente los competidores pierden terreno.

Un reloj hace tic tac una vez por segundo. Con cada tic del reloj, su programa debe ajustar la posición de los animales de acuerdo con las reglas de la figura 8.38.

Use variables para llevar el registro de las posiciones de los animales (los números de las posiciones son del 1 al 70). Empiece con cada animal en la posición 1 (la “puerta de inicio”). Si un animal se resbala hacia la izquierda antes de la posición 1, regrésselo a la posición 1.

Genere los porcentajes que se muestran en la figura 8.38 produciendo un entero aleatorio i en el rango $1 \leq i \leq 10$. Para la tortuga, realice un “paso pesado rápido” cuando $1 \leq i \leq 5$, un “resbalón” cuando $6 \leq i \leq 7$ o un “paso pesado lento” cuando $8 \leq i \leq 10$. Utilice una técnica similar para mover a la liebre.

Empiece la carrera imprimiendo el mensaje

PUM!!!

Y ARRANCAN!!!

Luego, para cada tic del reloj (es decir, cada repetición de un ciclo) imprima una línea de 70 posiciones, mostrando la letra T en la posición de la tortuga y la letra H en la posición de la liebre. En ocasiones los competidores se encontrarán en la misma posición. En este caso, la tortuga muerde a la liebre y su programa debe imprimir OUCH!!! empezando en esa posición. Todas las posiciones de impresión distintas de la T, la H o el mensaje OUCH!!! (en caso de un empate) deben estar en blanco.

Después de imprimir cada línea, compruebe si uno de los animales ha llegado o se ha pasado de la posición 70. De ser así, imprima quién fue el ganador y termine la simulación. Si la tortuga gana, imprima LA TORTUGA GANA!!! YAY!!! Si la liebre gana, imprima La liebre gana. Que mal. Si ambos animales ganan en el mismo tic del reloj, tal vez usted quiera favorecer a la tortuga (la más débil) o tal vez quiera imprimir Es un empate. Si ninguno de los dos animales gana, ejecute el ciclo de nuevo para simular el siguiente tic del reloj.

Animal	Tipo de movimiento	Porcentaje del tiempo	Movimiento actual
Tortuga	Paso pesado rápido	50%	3 posiciones a la derecha
	Resbalón	20%	6 posiciones a la izquierda
	Paso pesado lento	30%	1 posición a la derecha
Liebre	Dormir	20%	Ningún movimiento
	Gran salto	20%	9 posiciones a la derecha
	Gran resbalón	10%	12 posiciones a la izquierda
	Pequeño salto	30%	1 posición a la derecha
	Pequeño resbalón	20%	2 posiciones a la izquierda

Figura 8.38 | Reglas para ajustar las posiciones de la tortuga y la liebre.

Sección especial: construya su propia computadora

En los siguientes problemas nos desviaremos temporalmente del mundo de la programación en lenguajes de alto nivel. Vamos a “abrir de par en par” una computadora y ver su estructura interna. Presentaremos la programación en lenguaje máquina y escribiremos varios programas en este lenguaje. Para que ésta sea una experiencia valiosa, crearemos también una computadora (mediante la técnica de la *simulación* basada en software) en la que pueda ejecutar sus programas en lenguaje máquina.

8.18 (Programación en lenguaje máquina) Vamos a crear una computadora a la que llamaremos Simpletron. Como su nombre lo implica, es una máquina simple pero, como veremos pronto, también es poderosa. Simpletron sólo ejecuta programas escritos en el único lenguaje que entiende directamente: el lenguaje máquina de Simpletron, o LMS.

Simpletron contiene un *acumulador*, un “registro especial” en el cual se coloca la información antes de que Simpletron la utilice en los cálculos, o que la analice de distintas maneras. Toda la información dentro de Simpletron se manipula en términos de *palabras*. Una palabra es un número decimal con signo de cuatro dígitos, tal como +3364, -1293, +0007 y -0001. Simpletron está equipada con una memoria de 100 palabras, y se hace referencia a estas palabras mediante sus números de ubicación 00, 01, ..., 99.

Antes de ejecutar un programa LMS debemos *cargar*, o colocar, el programa en memoria. La primera instrucción de cada programa LMS se coloca siempre en la ubicación 00. El simulador empezará a ejecutarse en esta ubicación.

Cada instrucción escrita en LMS ocupa una palabra de la memoria de Simpletron; por lo tanto, las instrucciones son números decimales de cuatro dígitos con signo. Vamos a suponer que el signo de una instrucción LMS siempre será positivo, pero el signo de una palabra de información puede ser positivo o negativo. Cada una de las ubicaciones en la memoria de Simpletron puede contener una instrucción, un valor de datos utilizado por un programa o un área no utilizada (y por lo tanto indefinida) de memoria. Los primeros dos dígitos de cada instrucción LMS son el *código de operación* que especifica la operación a realizar. Los códigos de operación de LMS se sintetizan en la figura 8.39.

Los últimos dos dígitos de una instrucción LMS son el *operando* (la dirección de la ubicación en memoria que contiene la palabra a la cual se aplica la operación).

Ahora consideremos varios programas simples en LMS. El primero (figura 8.40) lee dos números del teclado, calcula e imprime su suma. La instrucción +1007 lee el primer número del teclado y lo coloca en la ubicación 07 (que se ha inicializado con 0). Después, la instrucción +1008 lee el siguiente número y lo coloca en la ubicación 08. La instrucción *carga*, +2007,

Código de operación	Significado
<i>Operaciones de entrada/salida</i>	
const int LEE = 10;	Lee una palabra desde el teclado y la introduce en una ubicación específica de memoria.
const int ESCRIBE = 11;	Escribe una palabra de una ubicación específica de memoria y la imprime en la pantalla.
<i>Operaciones de carga/almacenamiento:</i>	
const int CARGA = 20;	Carga una palabra de una ubicación específica de memoria y la coloca en el acumulador.
final int ALMACENA = 21;	Almacena una palabra del acumulador dentro de una ubicación específica de memoria.
<i>Operaciones aritméticas:</i>	
const int SUMA = 30;	Suma una palabra de una ubicación específica de memoria a la palabra en el acumulador (deja el resultado en el acumulador).
const int RESTA = 31;	Resta una palabra de una ubicación específica de memoria a la palabra en el acumulador (deja el resultado en el acumulador).
final int DIVIDE = 32;	Divide una palabra de una ubicación específica de memoria entre la palabra en el acumulador (deja el resultado en el acumulador).
const int MULTIPLICA = 33;	Multiplica una palabra de una ubicación específica de memoria por la palabra en el acumulador (deja el resultado en el acumulador).
<i>Operaciones de transferencia de control:</i>	
final int BIFURCA = 40;	Bifurca hacia una ubicación específica de memoria.
final int BIFURCANEG = 41;	Bifurca hacia una ubicación específica de memoria si el acumulador es negativo.
final int BIFURCACERO = 42;	Bifurca hacia una ubicación específica de memoria si el acumulador es cero.
const int ALTO = 43;	Alto. El programa completó su tarea.

Figura 8.39 | Códigos de operación del Lenguaje máquina Simpletron (LMS).

coloca (copia) el primer número en el acumulador y la instrucción *suma*, +3008, suma el segundo número al número en el acumulador. *Todas las instrucciones LMS aritméticas dejan sus resultados en el acumulador.* La instrucción *almacena*, +2109, coloca (copia) el resultado de vuelta en la ubicación de memoria 09. Después la instrucción *escribe*, +1109, toma el número y lo imprime (como un número decimal de cuatro dígitos con signo). La instrucción *alto*, +4300, termina la ejecución.

Ubicación	Número	Instrucción
00	+1007	(Lee A)
01	+1008	(Lee B)
02	+2007	(Carga A)
03	+3008	(Suma B)
04	+2109	(Almacena C)
05	+1109	(Escribe C)
06	+4300	(Alto)
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Resultado C)

Figura 8.40 | Programa en LMS que lee dos enteros y calcula la suma.

El programa en LMS de la figura 8.41 lee dos números desde el teclado, determina e imprime el valor más grande. Observe el uso de la instrucción +4107 como una transferencia de control condicional, en forma muy similar a la instrucción *if* de C++.

Ubicación	Número	Instrucción
00	+1009	(Lee A)
01	+1010	(Lee B)
02	+2009	(Carga A)
03	+3110	(Resta B)
04	+4107	(Bifurcación negativa a 07)
05	+1109	(Escribe A)
06	+4300	(Alto)
07	+1110	(Escribe B)
08	+4300	(Alto)
09	+0000	(Variable A)
10	+0000	(Variable B)

Figura 8.41 | Ejemplo 2 de LMS.

Ahora escriba programas en LMS para realizar cada una de las siguientes tareas:

- Usar un ciclo controlado por centinela para leer números positivos, calcular e imprimir la suma. Terminar la entrada cuando se introduzca un número negativo.
- Usar un ciclo controlado por contador para leer siete números, algunos positivos y otros negativos, y calcular e imprimir su promedio.
- Leer una serie de números, determinar e imprimir el número más grande. El primer número leído indica cuántos números deben procesarse.

8.19 (*Un simulador de computadora*) Tal vez a primera instancia parezca extravagante, pero en este problema usted va a crear su propia computadora. No, no va a soldar componentes, sino que utilizará la poderosa técnica de la *simulación basada en software* para crear un *modelo de software* de Simpletron. No quedará defraudado. Su simulador Simpletron convertirá la computadora que usted utiliza en Simpletron, y será capaz de ejecutar, probar y depurar los programas LMS que escribió en el ejercicio 8.18.

Cuando ejecute su simulador Simpletron, debe empezar mostrando lo siguiente:

```
*** Bienvenido a Simpletron! ***
*** Por favor, introduzca en su programa una instruccion ***
*** (o palabra de datos) a la vez . Yo le mostrare el ***
*** numero de ubicacion y un signo de interrogacion (?). ***
*** Entonces usted escribirá la palabra para esa ubicacion. ***
*** Escriba el valor centinela -99999 para dejar de ***
*** introducir su programa. ***
```

Su programa debe simular la memoria del Simpletron con un arreglo de un solo subíndice llamado `memoria`, que cuente con 100 elementos. Ahora suponga que el simulador se está ejecutando y examinaremos el diálogo a medida que introduzcamos el programa del ejemplo 2 del ejercicio 8.18:

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999
*** Se completo la carga del programa ***
*** Empieza la ejecucion del programa ***
```

Observe que los números a la derecha de cada ? en el diálogo anterior representan las instrucciones del programa de LMS introducidas por el usuario.

Ahora el programa en LMS se ha colocado (o cargado) en el arreglo `memoria`. Simpletron debe a continuación ejecutar el programa en LMS. La ejecución comienza con la instrucción en la ubicación 00 y, como en C++, continúa secuencialmente a menos que se lleve a otra parte del programa mediante una transferencia de control.

Use la variable `acumulador` para representar el registro acumulador. Use la variable `contador` para llevar el registro de la ubicación en memoria que contiene la instrucción que se está ejecutando. Use la variable `codigoDeOperacion` para indicar la operación que se está realizando actualmente (es decir, los dos dígitos a la izquierda en la palabra de instrucción). Use la variable `operando` para indicar la ubicación de memoria en la que va a operar la instrucción actual. Por lo tanto, `operando` está compuesta por los dos dígitos más a la derecha de la instrucción que se esté ejecutando en esos momentos. No ejecute las instrucciones directamente desde la memoria. En vez de eso, transfiera la siguiente instrucción a ejecutar desde la memoria hasta una variable llamada `registroDeInstruccion`. Luego “recoja” los dos dígitos a la izquierda y colóquelos en `codigoDeOperacion`, después “recoja” los dos dígitos a la derecha y colóquelos en `operando`. Cuando Simpletron comience con la ejecución, todos los registros especiales se deben inicializar con cero.

Ahora vamos a “dar un paseo” por la ejecución de la primera instrucción LMS, +1009 en la ubicación de memoria 00. A este procedimiento se le conoce como *ciclo de ejecución de una instrucción*.

El `contador` nos indica la ubicación de la siguiente instrucción a ejecutar. Nosotros *obtenemos* el contenido de esa ubicación de `memoria`, utilizando la siguiente instrucción de C++:

```
registroDeInstruccion = memoria[ contador ];
```

El código de operación y el operando se extraen del registro de instrucción, mediante las instrucciones

```
codigoDeOperacion = registroDeInstruccion / 100;
operando = registroDeInstruccion % 100;
```

Ahora, Simpletron debe determinar que el código de operación es en realidad un *lee* (en comparación con un *escribe*, *carga*, etcétera). Una instrucción `switch` establece la diferencia entre las 12 operaciones de LMS.

En la instrucción `switch` se simula el comportamiento de varias instrucciones LMS, como se muestra en la figura 8.42 (dejaremos las otras a usted).

<i>lee:</i>	<code>cin >> memoria[operando];</code>
<i>carga:</i>	<code>acumulador = memoria[operando];</code>
<i>suma:</i>	<code>acumulador += memoria[operando];</code>
<i>bifurca:</i>	En breve hablaremos sobre las instrucciones de bifurcación.
<i>alto:</i>	Esta instrucción imprime el mensaje <code>*** Termino la ejecucion de Simpletron ***</code>

Figura 8.42 | Comportamiento de las instrucciones de LMS.

La instrucción *alto* también hace que Simpletron imprima el nombre y contenido de cada registro, así como el contenido completo de la memoria. A este tipo de impresión se le denomina *vaciado de memoria y registro*. Para ayudarlo a programar su método de vaciado, en la figura 8.43 se muestra un formato de vaciado de muestra. Observe que un vaciado, después de la ejecución de un programa de Simpletron, muestra los valores actuales de las instrucciones y los valores de los datos al momento en que se terminó la ejecución. Para dar formato a los números con su signo como se muestra en el vaciado, use el manipulador de flujo `showpos`. Para deshabilitar la visualización del signo, use el manipulador de flujo `noshowpos`. Para los números que tienen menos de cuatro dígitos, se puede dar formato a éstos con ceros a la izquierda entre el signo y el valor, usando la siguiente instrucción antes de imprimir los valores:

```
cout << setfill( '0' ) << internal;
```

El manipulador de flujo parametrizado `setfill` (del encabezado `<iomanip>`) especifica el carácter de relleno que debe aparecer entre el signo y el valor, cuando un número se muestra con una anchura de campo de cinco caracteres, pero no tiene cuatro dígitos. (Se reserva una posición en la anchura de campo para el signo). El manipulador de flujo `internal` indica que los caracteres de relleno deben aparecer entre el signo y el valor numérico.

Procedamos ahora con la ejecución de la primera instrucción de nuestro programa, +1009 en la ubicación 00. Como lo hemos indicado, la instrucción `switch` simula esta tarea ejecutando la siguiente instrucción de C++:

```
cin >> memoria[ operando ];
```

Se debe mostrar un signo de interrogación (?) en la pantalla, antes de que se ejecute la instrucción `cin` para pedir la entrada al usuario. Simpletron espera a que el usuario introduzca un valor y oprima la clave *Intro*. Después, el valor se lee en la ubicación 09.

En este punto se ha completado la simulación de la primera instrucción. Todo lo que resta es preparar a Simpletron para que ejecute la siguiente instrucción. Como la instrucción que acaba de ejecutarse no es una transferencia de control, sólo necesitamos incrementar el registro contador de instrucciones de la siguiente manera:

```
++contador;
```

Esta acción completa la ejecución simulada de la primera instrucción. Todo el proceso (es decir, el ciclo de ejecución de una instrucción) empieza de nuevo, con la búsqueda de la siguiente instrucción a ejecutar.

REGISTROS									
acumulador	+0000								
contador	00								
registroDeInstruccion	+0000								
codigoDeOperacion	00								
operando	00								
MEMORIA:									
0	1	2	3	4	5	6	7	8	9
0	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000
10	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000
20	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000
30	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000
40	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000
50	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000
60	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000
70	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000
80	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000
90	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000	+0000

Figura 8.43 | Ejemplo de un vaciado de registro y memoria.

Ahora veremos cómo se simulan las instrucciones de bifurcación (las transferencias de control). Todo lo que necesitamos hacer es ajustar el valor en el contador de instrucciones de manera apropiada. Por lo tanto, la instrucción de bifurcación condicional (40) se simula dentro de la instrucción `switch` como

```
contador = operando;
```

La instrucción condicional “bifurcar si el acumulador es cero” se simula como

```
if ( acumulador == 0 )
    contador = operando;
```

En este punto, usted debe implementar su simulador Simpletron y ejecutar cada uno de los programas que escribió en el ejercicio 8.18. Si lo desea, puede embellecer al LMS con características adicionales y ofrecerlas en su simulador.

Su simulador debe comprobar diversos tipos de errores. Por ejemplo, durante la fase de carga del programa, cada número que el usuario escribe en la memoria de Simpletron debe encontrarse dentro del rango de -9999 a +9999. Su simulador debe usar un ciclo `while` para probar que cada número introducido se encuentre dentro de este rango y, en caso contrario, seguir pidiendo al usuario que vuelva a introducir el número hasta que introduzca un número correcto.

Durante la fase de ejecución, su simulador debe comprobar varios errores graves, como los intentos de dividir entre cero, intentos de ejecutar códigos de operación inválidos, desbordamientos del acumulador (es decir, las operaciones aritméticas que den como resultado valores mayores que +9999 o menores que -9999) y demás. Dichos errores graves se conocen como **errores fatales**. Al detectar un error fatal, su simulador deberá imprimir un mensaje de error tal como

```
*** Intento de dividir entre cero ***
*** La ejecucion de Simpletron se termino en forma anormal ***
```

y deberá imprimir un vaciado de registro y memoria completo en el formato que vimos anteriormente. Este análisis ayudará al usuario a localizar el error en el programa.

Más ejercicios de apuntadores

8.20 Modifique el programa para barajar y repartir cartas de las figuras 8.25 a 8.27, de manera que las operaciones de barajar y repartir se realicen mediante la misma función (`barajarYRepartir`). La función debe contener una instrucción de ciclo anidada que sea similar a la función `barajar` en la figura 8.26.

8.21 ¿Qué hace el siguiente programa?

```

1 // Ex. 8.21: ex08_21.cpp
2 // ¿Qué hace este programa?
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 void misterio1( char *, const char * ); // prototipo
9
10 int main()
11 {
12     char cadena1[ 80 ];
13     char cadena2[ 80 ];
14
15     cout << "Escriba dos cadenas: ";
16     cin >> cadena1 >> cadena2;
17     misterio1( cadena1, cadena2 );
18     cout << cadena1 << endl;
19     return 0; // indica que terminó correctamente
20 } // fin de main
21
22 // ¿Qué hace esta función?
23 void misterio1( char *s1, const char *s2 )
24 {
25     while ( *s1 != '\0' )
26         s1++;
27
28     for ( ; *s1 = *s2; s1++, s2++ )
29         ; // instrucción vacía
30 } // fin de la función misterio1

```

8.22 ¿Qué hace este programa?

```

1 // Ej. 8.22: ej08_22.cpp
2 // ¿Qué hace este programa?
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 int misterio2( const char * ); // prototipo
9
10 int main()
11 {
12     char cadena1[ 80 ];
13
14     cout << "Escriba una cadena: ";
15     cin >> cadena1;
16     cout << misterio2( cadena1 ) << endl;
17     return 0; // indica que terminó correctamente
18 } // fin de main
19
20 // ¿Qué hace esta función?
21 int misterio2( const char *s )
22 {
23     int x;
24
25     for ( x = 0; *s != '\0'; s++ )
26         x++;
27
28     return x;
29 } // fin de la función misterio2

```

8.23 Busque el error en cada uno de los siguientes segmentos. Si se puede corregir, explique cómo.

- a)

```
int *numero;
    cout << numero << endl;
```
- b)

```
double *realPtr;
    long *enteroPtr;
    enteroPtr = realPtr;
```
- c)

```
int *x, y;
    x = y;
```
- d)

```
char s[] = "este es un arreglo de caracteres";
    for ( ; *s != '\0'; s++ )
        cout << *s << ' ';
```
- e)

```
short *numPtr, resultado;
    void *genericoPtr = numPtr;
    resultado = *genericoPtr + 7;
```
- f)

```
double z = 19.34;
    double xPtr = &x;
    cout << xPtr << endl;
```
- g)

```
char *s;
    cout << s << endl;
```

8.24 (*Quicksort*) Ya hemos visto antes las técnicas de ordenamiento de cubeta y de ordenamiento por selección. Ahora presentaremos la técnica de ordenamiento recursiva llamada Quicksort. El algoritmo básico para un arreglo de valores con un solo subíndice es el siguiente:

- a) *Paso de particionamiento*: tomar el primer elemento del arreglo desordenado y determinar su ubicación final en el arreglo ordenado (es decir, todos los valores a la izquierda del elemento en el arreglo son menores que el elemento, y todos los valores a la derecha del elemento en el arreglo son mayores). Ahora tenemos un elemento en su ubicación apropiada y dos subarreglos desordenados.
- b) *Paso recursivo*: llevar a cabo el *paso 1* en cada subarreglo desordenado.

Cada vez que se realiza el *paso 1* en un subarreglo, se coloca otro elemento en su ubicación final en el arreglo ordenado, y se crean dos subarreglos desordenados. Cuando un subarreglo consiste en un elemento, ese subarreglo ya debe estar ordenado; por lo tanto, el elemento está en su ubicación final.

El algoritmo básico parece lo bastante simple, pero ¿cómo determinamos la posición final del primer elemento de cada subarreglo? Como ejemplo, considere el siguiente conjunto de valores (el elemento en negritas es el elemento de particionamiento; se colocará en su ubicación final en el arreglo ordenado):

37 2 6 4 89 8 10 12 68 45

- Empezando desde el elemento de más a la derecha del arreglo, se compara cada elemento con **37** hasta que se encuentra un elemento menor que **37**; después se intercambian el **37** y ese elemento. El primer elemento menor que **37** es 12, por lo que se intercambian el **37** y el 12. Los valores que residen ahora en el arreglo son:

12 2 6 4 89 8 10 **37** 68 45

El elemento 12 está en cursivas, para indicar que se acaba de intercambiar con el **37**.

- Empezando desde la parte izquierda del arreglo, pero con el elemento que está después de 12, se compara cada elemento con **37** hasta encontrar un elemento mayor que **37**. Después se intercambian el **37** y ese elemento. El primer elemento mayor que **37** es 89, por lo que se intercambian el **37** y el 89. Ahora los valores residen en el arreglo de la siguiente manera:

12 2 6 4 **37** 8 10 89 68 45

- Empezando desde la derecha, pero con el elemento antes del 89, se compara cada elemento con **37** hasta encontrar un elemento menor que **37**. Después se intercambian el **37** y ese elemento. El primer elemento menor que **37** es 10, por lo que se intercambian **37** y 10. Ahora los valores residen en el arreglo de la siguiente manera:

12 2 6 4 10 8 **37** 89 68 45

- Empezando desde la izquierda, pero con el elemento que está después de 10, se compara cada elemento con **37** hasta encontrar un elemento mayor que **37**; después se intercambian el **37** y ese elemento. No hay más elementos mayores que **37**, por lo que al comparar el **37** consigo mismo, sabemos que se ha colocado en su ubicación final en el arreglo ordenado.

Una vez que se ha aplicado la partición en el arreglo anterior, hay dos subarreglos desordenados. El subarreglo con valores menores que 37 contiene 12, 2, 6, 4, 10 y 8. El subarreglo con valores mayores que 37 contiene 89, 68 y 45. El ordenamiento continúa con el particionamiento de ambos subarreglos de la misma forma que el arreglo original.

Con base en la anterior discusión, escriba la función recursiva `quickSort` para ordenar un arreglo entero con un solo subíndice. La función debe recibir como argumentos un arreglo de enteros, un subíndice inicial y un subíndice final. La función `quickSort` sólo debe llamar a la función `partición` para realizar el paso de particionamiento.

8.25 (Recorrido de un laberinto) La cuadrícula que contiene caracteres # y puntos (.) en la figura 8.44 es una representación de un laberinto mediante un arreglo bidimensional. En este arreglo bidimensional, los caracteres # representan las paredes del laberinto, y los puntos representan las ubicaciones en las posibles rutas a través del laberinto. Sólo pueden realizarse movimientos hacia una ubicación en el arreglo que contenga un punto.

Hay un algoritmo simple para recorrer un laberinto, que garantiza encontrar la salida (suponiendo que la haya). Si no hay salida, el algoritmo lo llevará a la ubicación inicial de nuevo. Coloque su mano derecha en la pared a su derecha y empiece a caminar hacia adelante. Nunca quite su mano de la pared. Si el laberinto gira a la derecha, siga la pared a la derecha. Mientras que no quite su mano de la pared, en un momento dado llegará a la salida del laberinto. Puede haber una ruta más corta que la que usted haya tomado, pero se garantiza que saldrá del laberinto si sigue el algoritmo.

```
# # # # # # # # # #
# . . . # . . . . .
. . # . # . # # . #
# # # . # . . . # .
# . . . . # # # . #
# # # . # . # . # .
# . . # . # . # . #
# # . # . # . # . #
# . . . . . . . . .
# # # # # . # # # .
# . . . . . . . . .
# # # # # # # # #
```

Figura 8.44 | Representación de un laberinto mediante un arreglo bidimensional.

Escriba una función recursiva llamada `recorrerLaberinto` para avanzar a través del laberinto. La función debe recibir como argumentos un arreglo de caracteres de 12 por 12 que representa el laberinto, y la posición inicial en el mismo. A medida que `recorrerLaberinto` trate de localizar la salida, debe colocar el carácter `x` en cada posición en la ruta. La función debe mostrar el laberinto después de cada movimiento, de manera que el usuario pueda observar a medida que se va resolviendo.

8.26 (*Generación de laberintos al azar*) Escriba una función llamada `generarLaberintos`, que reciba como argumento un arreglo bidimensional de 12 por 12 caracteres, y que produzca un laberinto al azar. Esta función también deberá proporcionar las posiciones inicial y final del laberinto. Pruebe su función `recorrerLaberinto` del ejercicio 8.25, usando varios laberintos generados al azar.

8.27 (*Laberintos de cualquier tamaño*) Generalice las funciones `recorrerLaberinto` y `generarLaberintos` de los ejercicios 8.25 y 8.26 para procesar laberintos de cualquier anchura y altura.

8.28 (*Modificaciones al simulador Simpletron*) En el ejercicio 8.19 usted escribió una simulación de software de una computadora que ejecuta programas escritos en el Lenguaje máquina Simpletron (LMS). En este ejercicio proponemos varias modificaciones y mejoras al simulador Simpletron. En los ejercicios 20.26 y 20.27 propondremos la creación de un compilador que convierta los programas escritos en un lenguaje de programación de alto nivel (una variación de BASIC) a LMS. Algunas de las siguientes modificaciones y mejoras pueden requerirse para ejecutar los programas producidos por el compilador. [Nota: algunas modificaciones pueden estar en conflicto con otras, y por lo tanto deberán realizarse por separado].

- a) Extienda la memoria del simulador Simpletron, de manera que contenga 1000 ubicaciones de memoria para permitir a Simpletron manejar programas más grandes.
- b) Permita al simulador realizar cálculos de módulo. Esta modificación requiere de una instrucción adicional en lenguaje máquina Simpletron.
- c) Permita al simulador realizar cálculos de exponentiación. Esta modificación requiere una instrucción adicional en lenguaje máquina Simpletron.
- d) Modifique el simulador para que pueda utilizar valores hexadecimales, en vez de valores enteros para representar instrucciones en lenguaje máquina Simpletron.
- e) Modifique el simulador para permitir la impresión de una nueva línea. Esta modificación requiere una instrucción adicional en lenguaje máquina Simpletron.
- f) Modifique el simulador para procesar valores de punto flotante además de valores enteros.
- g) Modifique el simulador para manejar la introducción de cadenas. [Sugerencia: cada palabra de Simpletron puede dividirse en dos grupos, cada una de las cuales guarda un entero de dos dígitos. Cada entero de dos dígitos representa el equivalente decimal de código ASCII de un carácter. Agregue una instrucción de lenguaje máquina que reciba como entrada una cadena y la almacene, empezando en una ubicación de memoria específica de Simpletron. La primera mitad de la palabra en esa ubicación será una cuenta del número de caracteres en la cadena (es decir, la longitud de la cadena). Cada media palabra subsiguiente contiene un carácter ASCII, expresado como dos dígitos decimales. La instrucción en lenguaje máquina convierte cada carácter en su equivalente ASCII y lo asigna a una media palabra].
- h) Modifique el simulador para manejar la impresión de cadenas almacenadas en el formato de la parte (g). [Sugerencia: agregue una instrucción en lenguaje máquina que imprima una cadena, empezando en cierta ubicación de memoria de Simpletron. La primera mitad de la palabra en esa ubicación es una cuenta del número de caracteres en la cadena (es decir, la longitud de la misma). Cada media palabra subsiguiente contiene un carácter ASCII expresado como dos dígitos decimales. La instrucción en lenguaje máquina comprueba la longitud e imprime la cadena, traduciendo cada número de dos dígitos en su carácter equivalente].
- i) Modifique el simulador para incluir la instrucción `LMS_DEPURA` que imprima un vaciado de memoria después de que se ejecute cada instrucción. Proporcione a `LMS_DEPURA` un código de operación de 44. La palabra `+4401` debe activar el modo de depuración, y `+4400` debe desactivarlo.

8.29 ¿Qué hace este programa?

```

1 // Ej. 8.29: ej08_29.cpp
2 // ¿Qué hace este programa?
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 bool misterio3( const char *, const char * ); // prototipo
9

```

```

10 int main()
11 {
12     char cadena1[ 80 ], cadena2[ 80 ];
13
14     cout << "Escriba dos cadenas: ";
15     cin >> cadena1 >> cadena2;
16     cout << "El resultado es " << misterio3( cadena1, cadena2 ) << endl;
17     return 0; // indica que terminó correctamente
18 } // fin de main
19
20 // ¿Qué hace esta función?
21 bool misterio3( const char *s1, const char *s2 )
22 {
23     for ( ; *s1 != '\0' && *s2 != '\0'; s1++, s2++ )
24
25         if ( *s1 != *s2 )
26             return false;
27
28     return true;
29 } // fin de la función misterio3

```

Ejercicios de manipulación de cadenas

[Nota: los siguientes ejercicios se deben implementar mediante el uso de cadenas basadas en apuntador estilo C].

8.30 Escriba un programa que utilice la función `strcmp` para comparar dos cadenas que introduzca el usuario. El programa deberá indicar si la primera cadena es menor, igual o mayor que la segunda.

8.31 Escriba un programa que utilice la función `strncmp` para comparar dos cadenas introducidas por el usuario. El programa deberá recibir como entrada el número de caracteres a comparar. También deberá indicar si la primera cadena es menor, igual o mayor que la segunda.

8.32 Escriba un programa que utilice la generación de números aleatorios para crear enunciados. El programa debe usar cuatro arreglos de apuntadores a char llamados `articulo`, `sustantivo`, `verbo` y `preposicion`. El programa debe crear un enunciado, seleccionando una palabra al azar de cada arreglo en el siguiente orden: `articulo`, `sustantivo`, `verbo`, `preposición`, `articulo` y `sustantivo`. A medida que se seleccione cada palabra, debe concatenarse con las palabras anteriores en un arreglo de caracteres que sea lo bastante grande como para contener el enunciado completo. Las palabras deben separarse por espacios. Cuando se imprima el enunciado final, deberá empezar con letra mayúscula y terminar con un punto. El programa debe generar 20 enunciados de ese tipo.

Los arreglos deberán llenarse de la siguiente manera: el arreglo `articulo` debe contener los artículos "el", "un", "algun" y "ningun"; el arreglo `sustantivo` deberá contener los sustantivos "nino", "senior", "perro", "ciudad" y "auto"; el arreglo `verbo` deberá contener los verbos "manejo", "salto", "corrio", "camino" y "paso"; el arreglo `preposicion` deberá contener las preposiciones "a", "desde", "encima de", "debajo de" y "sobre".

Una vez que escriba el programa anterior, modifíquelo para producir una historia breve que consista de varias de estas oraciones (¿qué le parecería un escritor de tesis al azar?).

8.33 (*Quintillas*) Una quintilla es un verso con humor de cinco líneas en el cual la primera y segunda línea riman con la quinta, y la tercera línea rima con la cuarta. Utilizando técnicas similares a las desarrolladas en el ejercicio 8.32, escriba un programa en C++ que produzca quintillas al azar. Pulir el programa para producir buenas quintillas es un problema desafiante, ¡pero el resultado valdrá la pena!

8.34 (*Latín cerdo*) Escriba un programa que codifique frases en español a frases en latín cerdo. El latín cerdo es una forma de lenguaje codificado que se utiliza con frecuencia por diversión. Existen muchas variaciones en los métodos utilizados para formar frases en latín cerdo. Por cuestiones de simpleza, utilice el siguiente algoritmo: para formar una frase en latín cerdo a partir de una frase en español, divida la frase en palabras con la función `strtok`. Para traducir cada palabra en español a una palabra en latín cerdo, coloque la primera letra de la palabra en español al final de la palabra, y agregue las letras "ae". De esta forma, la palabra "salta" se convierte a "altasae", la palabra "el" se convierte en "leae" y la palabra "computadora" se convierte en "omputadoracae". Los espacios en blanco entre las palabras permanecen como espacios en blanco. Suponga que la frase en español consiste en palabras separadas por espacios en blanco, que no hay signos de puntuación y que todas las palabras tienen dos o más letras. El método `imprimirPalabraEnLatin` deberá mostrar cada palabra. [Sugerencia: cada vez que se encuentre un token en la llamada a `strtok`, se pasará el apuntador al token a la función `imprimirPalabraEnLatin` para imprimir la palabra en latín cerdo].

8.35 Escriba un programa que reciba como entrada un número telefónico como una cadena de la forma (555) 555-5555. El programa deberá utilizar la función `strtok` para extraer el código de área como un token, los primeros tres dígitos del número telefónico como otro token y los últimos cuatro dígitos del número telefónico como otro token. Los siete dígitos del número telefónico deberán concatenarse en una cadena. Deberán imprimirse tanto el código de área como el número telefónico.

8.36 Escriba un programa que reciba como entrada una línea de texto, que divida la línea en tokens mediante la función `strtok` e imprima los tokens en orden inverso.

8.37 Use las funciones de comparación de cadenas que se describieron en la sección 8.13.2, junto con las técnicas para ordenar arreglos que se desarrollaron en el capítulo 7, para escribir un programa que ordene alfabéticamente una lista de cadenas. Use los nombres de 10 ciudades en su área como datos para su programa.

8.38 Escriba dos versiones de cada una de las funciones de copia y concatenación de cadenas de la figura 8.30. La primera versión debe usar subíndices de arreglos, y la segunda debe usar apuntadores y aritmética de apuntadores.

8.39 Escriba dos versiones de cada una de las funciones de comparación de cadenas de la figura 8.30. La primera versión debe usar subíndices de arreglos, y la segunda debe usar apuntadores y aritmética de apuntadores.

8.40 Escriba dos versiones de la función `strlen` de la figura 8.30. La primera versión debe usar subíndices de arreglos, y la segunda debe usar apuntadores y aritmética de apuntadores.

Sección especial: ejercicios avanzados de manipulación de cadenas

Los siguientes ejercicios son claves para el libro y están diseñados para evaluar la comprensión del lector sobre los conceptos fundamentales de la manipulación de cadenas. Esta sección incluye una colección de ejercicios intermedios y avanzados de manipulación de cadena. El lector encontrará estos ejercicios desafiantes, pero divertidos. Los problemas varían considerablemente en dificultad. Algunos requieren una hora o dos para escribir e implementar el programa. Otros son útiles como tareas de laboratorio que pudieran requerir dos o tres semanas de estudio e implementación. Algunos son proyectos de fin de curso desafiantes.

8.41 (*Análisis de textos*) La disponibilidad de computadoras con capacidades de manipulación de cadenas ha dado como resultado algunos métodos interesantes para analizar los escritos de grandes autores. Se ha puesto mucha atención en saber si realmente vivió William Shakespeare. Algunos estudiosos creen que existe evidencia importante que indica que, en realidad fueron Francis Bacon, Christopher Marlowe u otros autores quienes escribieron las obras maestras que se atribuyen a Shakespeare. Los investigadores han utilizado computadoras para buscar similitudes en los escritos de estos dos autores. En este ejercicio se examinan tres métodos para analizar textos mediante una computadora. Observe que hay miles de textos, incluyendo los de Shakespeare, disponibles en línea en www.gutenberg.org.

- a) Escriba un programa que lea varias líneas de texto desde el teclado e imprima una tabla que indique el número de ocurrencias de cada letra del alfabeto en el texto. Por ejemplo, la frase:

Ser o no ser: ése es el dilema:

contiene una “a”, ninguna “b”, ninguna “c”, etcétera.

- b) Escriba un programa que lea varias líneas de texto e imprima una tabla que indique el número de palabras de una letra, de dos letras, de tres letras, etcétera, que aparezcan en el texto. Por ejemplo, la frase:

¿Qué es más noble para el espíritu?

contiene las siguientes longitudes de palabra y ocurrencias:

Longitud de palabras	Ocurrencias
1	0
2	2
3	1
4	2
5	1
6	0
7	0
8	0
9	1

- c) Escriba un programa que lea varias líneas de texto e imprima una tabla que indique el número de ocurrencias de cada palabra distinta en el texto. La primera versión de su programa debe incluir las palabras en la tabla, en el mismo orden en el cual aparecen en el texto. Por ejemplo, las líneas:

Ser o no ser: ése es el dilema:
¿Qué es más noble para el espíritu?

contiene la palabra “ser” dos veces, La palabra “o” una vez, la palabra “ese” una vez, etcétera. Una muestra más interesante (y útil) podría ser intentar con las palabras ordenadas alfabéticamente.

8.42 (*Procesamiento de palabras*) Una importante función en los sistemas de procesamiento de palabras es la *justificación de tipos*: la alineación de palabras a los márgenes izquierdo y derecho de una página. Esto genera un documento con apariencia profesional, que aparenta ser una composición tipográfica, en vez de haber sido preparado en una máquina de escribir. La justificación de tipos se puede llevar a cabo en los sistemas computacionales mediante la inserción de caracteres en blanco entre las palabras en una línea, de manera que la palabra de más a la derecha se alinee con el margen derecho.

Escriba un programa que lea varias líneas de texto e imprima este texto en formato de justificación de tipos. Suponga que el texto se va a imprimir en papel de 8 ½ pulgadas de anchura, y que se van a permitir márgenes de una pulgada en los lados izquierdo y derecho. Suponga que la computadora imprime 10 caracteres en una pulgada horizontal. Por lo tanto, su programa debe imprimir 6 ½ pulgadas de texto, o 65 caracteres por línea.

8.43 (*Impresión de fechas en varios formatos*) Las fechas se imprimen en varios formatos comunes. Dos de los formatos más utilizados son:

21/07/1955
21 Julio, 1955

Escriba un programa que lea una fecha en el primer formato e imprima dicha fecha en el segundo formato.

8.44 (*Protección de cheques*) Las computadoras se utilizan frecuentemente en sistemas de escritura de cheques tales como aplicaciones para nóminas y para cuentas por pagar. Existen muchas historias extrañas acerca de cheques de pago que se imprimen (por error) con montos que se exceden de \$1 millón. Los sistemas de escritura de cheques computarizados imprimen cantidades incorrectas debido al error humano o a una falla de la máquina. Los diseñadores de sistemas construyen controles en sus sistemas para evitar la emisión de dichos cheques erróneos.

Otro problema grave es la alteración intencional del monto de un cheque por alguien que planee cobrar un cheque de manera fraudulenta. Para evitar la alteración de un monto, la mayoría de los sistemas computarizados emplean una técnica llamada *protección de cheques*.

Los cheques diseñados para impresión por computadora contienen un número fijo de espacios en los cuales la computadora puede imprimir un monto. Suponga que un cheque contiene ocho espacios en blanco en los cuales la computadora puede imprimir el monto de un cheque de nómina semanal. Si el monto es grande, entonces se llenarán los ocho espacios. Por ejemplo:

1,230.60 (monto del cheque)

12345678 (números de posición)

Por otra parte, si el monto es menor de \$1,000, entonces varios espacios quedarían vacíos. Por ejemplo:

99.87

12345678

contiene tres espacios en blanco. Si se imprime un cheque con espacios en blanco, es más fácil para alguien alterar el monto del cheque. Para evitar que se altere el cheque, muchos sistemas de escritura de cheques insertan *asteriscos al principio* para proteger la cantidad, como se muestra a continuación:

99.87

12345678

Escriba un programa que reciba como entrada un monto a imprimir sobre un cheque y que lo escriba mediante el formato de protección de cheques, con asteriscos al principio si es necesario. Suponga que existen nueve espacios disponibles para imprimir el monto.

8.45 (*Escritura en letras del código de un cheque*) Para continuar con la discusión del ejercicio anterior, reiteramos la importancia de diseñar sistemas de escritura de cheques para evitar la alteración de los montos de los cheques. Un método común

de seguridad requiere que el monto del cheque se escriba tanto en números como en letras. Incluso si alguien puede alterar el monto numérico del cheque, es extremadamente difícil modificar el monto en letras.

Escriba una aplicación que reciba como entrada un monto numérico de un cheque y que escriba el equivalente del monto en palabras. Su programa debe ser capaz de manejar montos de cheques tan grandes como \$99.99. Por ejemplo, el monto 112.43 debe escribirse como

CIENTO DOCE CON 43/100

8.46 (*Código Morse*) Quizá el más famoso de todos los esquemas de codificación es el código Morse, desarrollado por Samuel Morse en 1832 para usarlo con el sistema telegráfico. El código Morse asigna una serie de puntos y guiones a cada letra del alfabeto, cada dígito y algunos caracteres especiales (tales como el punto, la coma, los dos puntos y el punto y coma). En los sistemas orientados a sonidos, el punto representa un sonido corto y el guión representa un sonido largo. Otras representaciones de puntos y guiones se utilizan en sistemas orientados a luces y sistemas de señalización con banderas.

La separación entre palabras se indica mediante un espacio o, simplemente, con la ausencia de un punto o un guión. En un sistema orientado a sonidos, un espacio se indica por un tiempo breve durante el cual no se transmite sonido alguno. La versión internacional del código Morse aparece en la figura 8.45.

Escriba un programa que lea una frase en español y que codifique la frase en clave Morse. Además, escriba un programa que lea una frase en código Morse y que la convierta en su equivalente en español. Use un espacio en blanco entre cada letra en clave Morse y tres espacios en blanco entre cada palabra en clave Morse.

Carácter	Código	Carácter	Código
A	.-	N	-.
B	-...	O	---
C	-.-.	P	.---.
D	-..	Q	---.
E	.	R	.-.
F	...-.	S	...
G	--.	T	-
H	U-
I	..	V-
J	.---	W	.--
K	-.-	X	-...-
L	.-..	Y	-.--
M	--	Z	---..
Dígitos			
1	.----	6	-.....
2-	7	-....-
3-	8	-----.
4-	9	-----.
5	0	-----

Figura 8.45 | Alfabeto en código Morse.

8.47 (*Programa de conversión al sistema métrico*) Escriba un programa que ayude al usuario a realizar conversiones métricas. Su programa debe permitir al usuario especificar los nombres de las unidades como cadenas (es decir, centímetros, litros, gramos, etcétera, para el sistema métrico, y pulgadas, cuartos, libras, etcétera, para el sistema inglés) y debe responder a preguntas simples tales como:

"¿Cuántas pulgadas hay en 2 metros?"
 "¿Cuántos litros hay en 10 cuartos?"

Su programa debe reconocer conversiones inválidas. Por ejemplo, la pregunta:

"¿Cuántos pies hay en 5 kilogramos?"

no es correcta, debido a que los "pies" son unidades de longitud, mientras que los "kilogramos" son unidades de masa.

Un proyecto desafiante sobre manipulación de cadenas

8.48 (*Un generador de crucigramas*) La mayoría de las personas han resuelto crucigramas, pero pocos han intentado generar uno. La generación de un crucigrama es un problema difícil. Aquí lo sugerimos como un proyecto de manipulación de cadenas que requiere una cantidad considerable de sofisticación y esfuerzo. Hay muchas cuestiones que el programador tiene que resolver para hacer que funcione incluso hasta el programa generador de crucigramas más simple. Por ejemplo, ¿cómo representaría la cuadrícula de un crucigrama dentro de la computadora? ¿Debería utilizar una serie de cadenas o arreglos bidimensionales? El programador necesita una fuente de palabras (es decir, un diccionario computarizado) a las que el programa pueda hacer referencia de manera directa. ¿De qué manera deben almacenarse estas palabras para facilitar las complejas manipulaciones requeridas por el programa? El lector verdaderamente ambicioso querrá generar la porción de “claves” del crucigrama, en la que se imprimen las breves pistas para cada palabra “horizontal” y cada palabra “vertical”. La sola impresión del crucigrama en blanco no es un problema sencillo.



*Mi objeto, en todo sublime,
lograré con el tiempo.*

—W. S. Gilbert

*¿Es éste un mundo en el
cual se deben ocultar las
virtudes?*

—William Shakespeare

*No hay que ser “duros”,
sino simplemente sinceros.*

—Oliver Wendell Holmes, Jr.

*Por encima de todo:
hay que ser sinceros con
nosotros mismos.*

—William Shakespeare

Clases: un análisis más detallado, parte I

OBJETIVOS

En este capítulo aprenderá a:

- Usar una envoltura del preprocesador para evitar los errores de múltiples definiciones, ocasionados por incluir más de una copia de un archivo de encabezado en un archivo de código fuente.
- Comprender el alcance de las clases y el acceso a los miembros de una clase a través del nombre de un objeto, una referencia a un objeto o un apuntador a un objeto.
- Definir los constructores con argumentos predeterminados.
- Conocer la forma en que se utilizan los destructores para realizar “tareas de mantenimiento” en un objeto antes de destruirlo.
- Saber cuándo se hacen llamadas a los constructores y destructores, y el orden en el que se llaman.
- Conocer los errores lógicos que pueden ocurrir cuando una función miembro `public` de una clase devuelve una referencia a datos `private`.
- Asignar los miembros de datos de un objeto a los de otro objeto, mediante la asignación predeterminada a nivel de miembros.

- 9.1 Introducción
- 9.2 Ejemplo práctico con la clase `Tiempo`
- 9.3 Alcance de las clases y acceso a los miembros de una clase
- 9.4 Separar la interfaz de la implementación
- 9.5 Funciones de acceso y funciones utilitarias
- 9.6 Ejemplo práctico de la clase `Tiempo`: constructores con argumentos predeterminados
- 9.7 Destructores
- 9.8 Cuándo se hacen llamadas a los constructores y destructores
- 9.9 Ejemplo práctico con la clase `Tiempo`: una trampa sutil (devolver una referencia a un miembro de datos `private`)
- 9.10 Asignación predeterminada a nivel de miembros
- 9.11 (Opcional) Ejemplo práctico de Ingeniería de Software: inicio de la programación de las clases del sistema ATM
- 9.12 Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)

9.1 Introducción

En los capítulos anteriores, presentamos muchos términos y conceptos básicos de la programación orientada a objetos en C++. También hablamos sobre nuestra metodología para desarrollar programas: seleccionamos los atributos y comportamientos apropiados para cada clase y especificamos la forma en que los objetos de nuestras clases colaboraron con objetos de las clases de la Biblioteca estándar de C++ para llevar a cabo los objetivos generales de cada programa.

En este capítulo analizaremos las clases de una forma más detallada. Usaremos un ejemplo práctico integrado con la clase `Tiempo`, tanto en este capítulo (tres ejemplos) como en el capítulo 10, Clases: un análisis más detallado, parte 2 (dos ejemplos) para demostrar varias herramientas de construcción de clases. Empezaremos con una clase llamada `Tiempo`, que repasa varias de las características presentadas en los capítulos anteriores. El ejemplo también demuestra un importante concepto de ingeniería de software de C++: el uso de una “envoltura del preprocesador” en los archivos de encabezado para evitar que el código en el encabezado se incluya en el mismo archivo de código fuente más de una vez. Como una clase sólo se puede definir una vez, al usar dichas directivas del preprocesador evitamos los errores por múltiples definiciones.

A continuación, vamos a hablar sobre el alcance de las clases y las relaciones entre los miembros de una clase. También vamos a demostrar cómo el código cliente puede acceder a los miembros `public` de una clase, a través de tres tipos de “manejadores”: el nombre de un objeto, una referencia a un objeto o un apuntador a un objeto. Como podemos ver, los nombres de objetos y las referencias se pueden usar con el operador punto (.) de selección de miembros para acceder a un miembro `public`, y los apuntadores se pueden usar con el operador flecha (->) de selección de miembros.

Vamos a hablar sobre las funciones de acceso que pueden leer o mostrar los datos en un objeto. Un uso común de las funciones de acceso es evaluar la veracidad o falsedad de las condiciones; dichas funciones se conocen como funciones predicado. También demostraremos la noción de una función utilitaria (también conocida como función ayudante): una función miembro `private` que soporta la operación de las funciones miembro `public` de la clase, pero no está diseñada para que los clientes de la clase la utilicen.

En el segundo ejemplo del ejemplo práctico con la clase `Tiempo`, demostraremos cómo pasar argumentos a los constructores y cómo se pueden usar los argumentos predeterminados en un constructor, para permitir que el código cliente inicialice objetos de una clase mediante el uso de una variedad de argumentos. Después, hablaremos sobre una función miembro especial llamada destructor, la cual forma parte de toda clase y se utiliza para realizar “tareas de mantenimiento de terminación” en un objeto antes de destruirlo. Luego demostraremos el orden en el que se hacen las llamadas a los constructores y destructores, debido a que el hecho de que un programa sea correcto depende del uso apropiado de los objetos inicializados que todavía no se han destruido.

Nuestro último ejemplo práctico con la clase `Tiempo` en este capítulo muestra una peligrosa práctica de programación, en la cual una función miembro devuelve una referencia a datos `private`. Hablaremos acerca de cómo esto quebranta la encapsulación de una clase, y permite que el código cliente acceda directamente a los datos de un objeto. Este último ejemplo muestra que los objetos de la misma clase se pueden asignar entre sí, mediante el uso de la asignación predeterminada a nivel de miembros, en la cual los miembros de datos que están en el objeto del lado derecho de la asignación se copian en los correspondientes miembros de datos que están en el objeto del lado izquierdo de la asignación. Este capítulo concluye con una discusión sobre la reutilización de software.

9.2 Ejemplo práctico con la clase Tiempo

Nuestro primer ejemplo (figuras 9.1 a 9.3) crea la clase `Tiempo` y un programa controlador que prueba la clase. Ya hemos creado muchas clases en este libro. En esta sección repasaremos muchos de los conceptos cubiertos en el capítulo 3, y demostraremos un importante concepto de ingeniería de software de C++: el uso de una “envoltura del preprocesador” en los archivos de encabezado, para evitar que el código del encabezado se incluya en el mismo archivo de código fuente más de una vez. Como una clase sólo se puede definir una vez, al usar estas directivas del preprocesador evitamos los errores por múltiples definiciones.

Definición de la clase `Tiempo`

La definición de la clase (figura 9.1) contiene prototipos (líneas 13 a 16) para las funciones miembro `Tiempo`, establecerTiempo, imprimirUniversal e imprimirEstandar. Esta clase incluye los miembros enteros `private hora, minuto` y `segundo` (líneas 18 a 20). Sólo esas cuatro funciones miembro pueden acceder a los miembros de datos `private` de la clase `Tiempo`. El capítulo 12 introduce un tercer especificador de acceso llamado `protected`, a medida que estudiemos la herencia y el papel que desempeña en la programación orientada a objetos.



Buena práctica de programación 9.1

Por cuestión de claridad y legibilidad, use cada especificador de acceso sólo una vez en una definición de clase. Coloque los miembros `public` primero, donde sean fáciles de localizar.



Observación de Ingeniería de Software 9.1

Cada elemento de una clase debe tener visibilidad `private`, a menos que pueda demostrarse que el elemento necesita visibilidad `public`. Éste es otro ejemplo del principio del menor privilegio.

En la figura 9.1, observe que la definición de la clase va encerrada en la siguiente **envoltura del preprocesador** (líneas 6, 7 y 23):

```
// evita múltiples inclusiones del archivo de encabezado
#ifndef TIEMPO_H
#define TIEMPO_H
...
#endif
```

```

1 // Fig. 9.1: Tiempo.h
2 // Declaración de la clase Tiempo.
3 // Las funciones miembro están definidas en Tiempo.cpp
4
5 // evita múltiples inclusiones del archivo de encabezado
6 #ifndef TIEMPO_H
7 #define TIEMPO_H
8
9 // definición de la clase Tiempo
10 class Tiempo
11 {
12 public:
13     Tiempo(); // constructor
14     void establecerTiempo( int, int, int ); // establece hora, minuto y segundo
15     void imprimirUniversal(); // imprime la hora en formato universal
16     void imprimirEstandar(); // imprime la hora en formato estándar
17 private:
18     int hora; // 0 - 23 (formato de reloj de 24 horas)
19     int minuto; // 0 - 59
20     int segundo; // 0 - 59
21 }; // fin de la clase Tiempo
22
23 #endif
```

Figura 9.1 | Definición de la clase `Tiempo`.

Cuando construyamos programas más grandes, las demás definiciones y declaraciones también se colocarán en archivos de encabezado. La envoltura del preprocesador anterior evita que el código entre `#ifndef` (que significa “si no está definido”) y `#endif` se incluya, si ya se ha definido el nombre `TIEMPO_H`. Si el encabezado no se ha incluido antes en un archivo, el nombre `TIEMPO_H` se define mediante la directiva `#define` y se incluyen las instrucciones del archivo de encabezado. Si el encabezado ya se incluyó antes, `TIEMPO_H` se encuentra definido de antemano y el archivo de encabezado no se vuelve a incluir. Por lo general, los intentos de incluir un archivo de encabezado varias veces (inadvertidamente) ocurren en programas extensos con muchos archivos de encabezado, que a su vez pueden incluir otros archivos de encabezado. [Nota: la convención de uso común para el nombre de la constante simbólica en las directivas del preprocesador es simplemente el nombre del archivo de encabezado en mayúsculas con el carácter de guión bajo como sustituto para el punto].



Tip para prevenir errores 9.1

Use las directivas del preprocesador `#ifndef`, `#define` y `#endif` para formar una envoltura del preprocesador que evite incluir los archivos de encabezado más de una vez en un programa.



Buena práctica de programación 9.2

Use el nombre del archivo de encabezado en mayúsculas, sustituyendo el punto por un guion bajo en las directivas del preprocesador `#ifndef` y `#define` de un archivo de encabezado.

Funciones miembro de la clase `Tiempo`

En la figura 9.2, el constructor de `Tiempo` (líneas 14 a 17) inicializa los miembros de datos con 0 (es decir, el equivalente en tiempo universal de 12 AM). Esto asegura que el objeto empiece en un estado consistente. No se pueden almacenar valores inválidos en los miembros de datos de un objeto `Tiempo`, ya que se hace una llamada al constructor cuando se crea el objeto `Tiempo`, y todos los intentos subsiguientes de un cliente por modificar los miembros de datos son escudriñados por la función `establecerTiempo` (que veremos en breve). Por último, es importante observar que podemos definir varios constructores sobrecargados para una clase.

```

1 // Fig. 9.2: Tiempo.cpp
2 // Definiciones de las funciones miembro para la clase Tiempo.
3 #include <iostream>
4 using std::cout;
5
6 #include <iomanip>
7 using std::setfill;
8 using std::setw;
9
10 #include "Tiempo.h" // incluye la definición de la clase Tiempo de Tiempo.h
11
12 // el constructor de Tiempo inicializa cada miembro de datos con cero.
13 // Asegura que todos los objetos Tiempo empiecen en un estado consistente.
14 Tiempo::Tiempo()
15 {
16     hora = minuto = segundo = 0;
17 } // fin del constructor de Tiempo
18
19 // establece el nuevo valor de Tiempo usando la hora universal; asegura que
20 // los datos sean consistentes al establecer los valores inválidos en cero
21 void Tiempo::establecerTiempo( int h, int m, int s )
22 {
23     hora = ( h >= 0 && h < 24 ) ? h : 0; // valida la hora
24     minuto = ( m >= 0 && m < 60 ) ? m : 0; // valida el minuto
25     segundo = ( s >= 0 && s < 60 ) ? s : 0; // valida el segundo
26 } // fin de la función establecerTiempo
27
28 // imprime el Tiempo en formato de hora universal (HH:MM:SS)
29 void Tiempo::imprimirUniversal()
```

Figura 9.2 | Definiciones de las funciones miembro de la clase `Tiempo`. (Parte 1 de 2).

```

30  {
31      cout << setfill( '0' ) << setw( 2 ) << hora << ":" 
32      << setw( 2 ) << minuto << ":" << setw( 2 ) << segundo;
33 } // fin de la función imprimirUniversal
34
35 // imprime el Tiempo en formato de hora estándar (HH:MM:SS AM or PM)
36 void Tiempo::imprimirEstandar()
37 {
38     cout << ( ( hora == 0 || hora == 12 ) ? 12 : hora % 12 ) << ":" 
39     << setfill( '0' ) << setw( 2 ) << minuto << ":" << setw( 2 )
40     << segundo << ( hora < 12 ? " AM" : " PM" );
41 } // fin de la función imprimirEstandar

```

Figura 9.2 | Definiciones de las funciones miembro de la clase *Tiempo*. (Parte 2 de 2).

Los miembros de datos de una clase no se pueden inicializar donde se declaran en el cuerpo de la clase. Se recomienda ampliamente que estos miembros de datos se inicialicen mediante el constructor de la clase (ya que no hay inicialización predeterminada para los miembros de datos de tipo fundamental). A los miembros de datos también se les pueden asignar valores mediante las funciones *establecer* de *Tiempo*. [Nota: el capítulo 10 demuestra que sólo los miembros de datos *static const* de una clase de los tipos integral o *enum* se pueden inicializar en el cuerpo de la clase].

Error común de programación 9.1



Tratar de inicializar un miembro de datos no *static* de una clase, explícitamente en la definición de la misma, es un error de sintaxis.

La función *establecerHora* (líneas 21 a 26) es una función *public* que declara tres parámetros *int* y los utiliza para establecer la hora. Una expresión condicional evalúa cada argumento para determinar si el valor no se encuentra en un rango especificado. Por ejemplo, el valor *hora* (línea 23) debe ser mayor o igual que 0 y menor que 24, debido a que el formato de hora universal representa las horas como enteros del 0 al 23 (por ejemplo, 1 PM es la hora 13 y 11 PM es la hora 23; medianoche es la hora 0 y mediodía es la hora 12). De manera similar, los valores de *minuto* y *segundo* (líneas 24 y 25) deben ser mayores o iguales que 0, y menores que 60. Cualquier valor fuera de estos rangos se establece en cero para asegurar que un objeto *Tiempo* siempre contenga datos consistentes; es decir, los valores de los datos del objeto siempre se mantienen dentro del rango, aun si los valores que se proporcionan como argumentos para la función *establecerTiempo* son incorrectos. En este ejemplo, cero es un valor consistente para *hora*, *minuto* y *segundo*.

Un valor que se pasa a *establecerTiempo* es un valor correcto si se encuentra dentro del rango permitido para el miembro que va a inicializar. Por lo tanto, cualquier número en el rango de 0 a 23 sería un valor correcto para la *hora*. Un valor correcto siempre es un valor consistente. Sin embargo, un valor consistente no necesariamente es un valor correcto. Si *establecerTiempo* establece *hora* en 0 debido a que el argumento que se recibió está fuera de rango, entonces *hora* es correcta sólo si el tiempo actual es por coincidencia medianoche.

La función *imprimirUniversal* (líneas 29 a 33 de la figura 9.2) no recibe argumentos e imprime el tiempo en formato universal, el cual consiste de tres pares de dígitos separados por dos puntos; para hora, minuto y segundo, respectivamente. Por ejemplo, si el tiempo es 1:30:07 PM, la función *imprimirUniversal* devuelve 13:30:07. Observe que en la línea 31 se utiliza el manipulador de flujo parametrizado *setfill* para especificar el carácter de relleno que se muestra cuando se imprime un entero en un campo más ancho que el número de dígitos en el valor. De manera predeterminada, los caracteres de relleno aparecen a la izquierda de los dígitos en el número. En este ejemplo, si el valor de *minuto* es 2, se mostrará como 02 ya que el carácter de relleno se estableció en cero ('0'). Si el número que se va a imprimir rellena el campo especificado, no se muestra el carácter de relleno. Observe que, una vez que se especifica el carácter de relleno mediante *setfill*, se aplica para todos los valores subsiguientes que se muestran en campos más amplios que el valor que se va a mostrar (es decir, *setfill* es una opción “pegajosa”). Esto es en contraste con *setw*, que sólo se aplica al siguiente valor mostrado (*setw* es una opción “no pegajosa”).

Tip para prevenir errores 9.2



Cada opción pegajosa (como un carácter de relleno o una precisión de punto flotante) se debe restaurar a su opción anterior cuando ya no se necesite. Si no se hace esto, se puede producir una salida con formato incorrecto posteriormente en un programa. En el capítulo 15, Entrada y salida de flujos, veremos cómo restablecer el carácter de relleno y la precisión.

La función `imprimirEstandar` (líneas 36 a 41) no recibe argumentos e imprime la fecha en formato de tiempo estándar, el cual consiste en los valores de hora, `minuto` y `segundo` separados por dos puntos, y va seguido de un indicador AM o PM (por ejemplo, 1:27:06 PM). Al igual que la función `imprimirUniversal`, la función `imprimirEstandar` usa `setfill('0')` para dar formato a `minuto` y `segundo` como valores de dos dígitos con ceros a la izquierda, en caso de ser necesario. En la línea 38 se utiliza el operador condicional (`?:`) para determinar el valor de `hora` a mostrar; si `hora` es 0 o 12 (AM o PM), aparece como 12; en caso contrario, `hora` aparece como un valor de 1 a 11. El operador condicional en la línea 40 determina si se va a mostrar AM o PM.

Definición de funciones miembro fuera de la definición de la clase: alcance de las clases

Aun y cuando una función miembro declarada en una definición de clase se puede definir fuera de esa definición de clase (y “enlazarse” a la clase mediante el operador de resolución de ámbito binario), esa función miembro aún está dentro del **alcance de esa clase**; su nombre es conocido sólo para los otros miembros de la clase, a menos que se haga referencia a ésta a través de un objeto de la clase, una referencia a un objeto de la clase, un apuntador a un objeto de la clase o del operador de resolución de ámbito binario. En breve hablaremos más acerca del alcance de las clases.

Si una función miembro se define en el cuerpo de la definición de una clase, el compilador trata de poner en línea las llamadas a la función miembro. Las funciones miembro definidas fuera de una definición de clase se pueden poner en línea, usando de manera explícita la palabra clave `inline`. Recuerde que el compilador se reserva el derecho de no poner en línea cualquier función.

Tip de rendimiento 9.1



Al definir una función miembro dentro de la definición de clase, se pone en línea la función miembro (si el compilador opta por hacer esto). Esto puede mejorar el rendimiento.

Observación de Ingeniería de Software 9.2



Al definir una función miembro pequeña dentro de la definición de clase no se promueve la mejor ingeniería de software, ya que los clientes de la clase podrán ver la implementación de la función, y el código cliente se debe volver a compilar si cambia la definición de la función.

Observación de Ingeniería de Software 9.3



Sólo las funciones miembro más simples y estables (es decir, aquellas cuyas implementaciones tengan pocas probabilidades de cambiar) deben definirse en el encabezado de la clase.

Comparación entre funciones miembro y funciones globales

Es interesante ver que las funciones miembro `imprimirUniversal` e `imprimirEstandar` no reciben argumentos. Esto se debe a que esas funciones miembro saben de manera implícita que deben imprimir los miembros de datos del objeto `Tiempo` específico para el cual se invocaron. Esto puede hacer que las llamadas a las funciones miembro sean más concisas que las llamadas a las funciones convencionales en la programación por procedimientos.

Observación de Ingeniería de Software 9.4



Por lo general, el uso de una metodología de programación orientada a objetos puede simplificar las llamadas a las funciones, al reducir el número de parámetros que se deben pasar. Este beneficio de la programación orientada a objetos se deriva del hecho de que al encapsular los miembros de datos y las funciones miembro dentro de un objeto, se proporciona a las funciones miembro el derecho de acceder a los miembros de datos.

Observación de Ingeniería de Software 9.5



Por lo general, las funciones miembro son más cortas que las funciones en los programas no orientados a objetos, ya que los datos almacenados en los miembros de datos se han validado idealmente mediante un constructor, o mediante funciones miembro que almacenan nuevos datos. Debido a que los datos ya se encuentran en el objeto, comúnmente las llamadas a funciones miembro no tienen argumentos, o tienen menos argumentos que las llamadas a funciones comunes en los lenguajes no orientados a objetos. Por ende, las llamadas son más cortas, las definiciones de las funciones son más cortas y los prototipos de las funciones son más cortos. Esto mejora muchos aspectos del desarrollo de programas.



Tip para prevenir errores 9.3

El hecho de que las llamadas a funciones miembro generalmente no reciben argumentos, o reciben mucho menos argumentos que las llamadas a funciones convencionales en los lenguajes no orientados a objetos, se reduce la probabilidad de pasar los argumentos incorrectos, los tipos incorrectos de argumentos o el número incorrecto de los mismos.

Uso de la clase Tiempo

Una vez que se ha definido la clase `Tiempo`, se puede usar como un tipo en las declaraciones de un objeto, arreglo, apuntador y referencia, como se muestra a continuación:

```

Tiempo puestaDeSol; // objeto de tipo Tiempo
Tiempo arregloDeTiempos[ 5 ], // arreglo de 5 objetos Tiempo
Tiempo &horaDeComer = puestaDeSol; // referencia a un objeto Tiempo
Tiempo *tiempoPtr = &horaDeComer, // apuntador a un objeto Tiempo

```

La figura 9.3 usa la clase `Tiempo`. En la línea 12 se crea una instancia de un solo objeto de la clase `Tiempo` llamado `t`. Cuando se crea la instancia del objeto, se hace una llamada al constructor de `Tiempo` para inicializar cada miembro de datos `private` con 0. Después, en las líneas 16 y 18 se imprime el tiempo en los formatos universal y estándar, respectivamente, para confirmar que los miembros se hayan inicializado en forma apropiada. En la línea 20 se establece un nuevo tiempo, para lo cual se hace una llamada a la función miembro `establecerTiempo`, y en las líneas 24 y 26 se imprime el tiempo de nuevo en ambos formatos. En la línea 28 se intenta usar `establecerTiempo` para establecer los miembros de datos con valores inválidos; la función `establecerTiempo` reconoce esto y establece los valores inválidos en 0 para mantener el objeto en un estado consistente. Por último, en las líneas 33 y 35 se imprime el tiempo de nuevo, en ambos formatos.

Un avance acerca de la composición y la herencia

A menudo, las clases no se tienen que crear “desde cero”. En vez de ello, pueden incluir objetos de otras clases como miembros, o pueden derivarse de otras clases que proporcionen atributos y comportamientos que las nuevas clases puedan usar. Dicha reutilización de software puede mejorar de manera considerable la productividad del programador, y simplificar el mantenimiento del código. Al proceso de incluir objetos de clases como miembros de otras clases se le llama **composición** (o **agregación**), y se describe en el capítulo 10. Al proceso de derivar nuevas clases a partir de clases existentes se le llama **herencia** y se describe en el capítulo 12.

Tamaño de los objetos

A menudo, las personas con poca experiencia en la programación orientada a objetos suponen que los objetos deben ser bastante grandes, ya que contienen miembros de datos y funciones miembro. En sentido lógico, esto es verdad; podemos considerar que los objetos contienen datos y funciones (y nuestra discusión sin duda ha fomentado este punto de vista); sin embargo, físicamente esto no es verdad.



Tip de rendimiento 9.2

Los objetos sólo contienen datos, por lo que son mucho menores que si también contuvieran funciones miembro. Al aplicar el operador `sizeof` al nombre de una clase o a un objeto de esa clase se reportará sólo el tamaño de los miembros de datos de la clase. El compilador crea una copia (sólo) de las funciones miembro, separada de todos los objetos de la clase. Estos objetos comparten esta única copia. Desde luego que cada objeto de la clase necesita su propia copia de los datos de la clase, ya que éstos pueden variar entre un objeto y otro. El código de la función no se puede modificar (a esto también se le llama código reentrante o procedimiento puro) y, por ende, puede compartirse entre todos los objetos de una clase.

```

1 // Fig. 9.3: fig09_03.cpp
2 // Programa para probar la clase Tiempo.
3 // NOTA: Este archivo se debe compilar con Tiempo.cpp.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "Tiempo.h" // incluye la definición de la clase Tiempo de Tiempo.h
9
10 int main()

```

Figura 9.3 | Programa para probar la clase `Tiempo`. (Parte 1 de 2).

```

11  {
12      Tiempo t; // instancia un objeto t de la clase Tiempo
13
14      // imprime los valores iniciales del objeto Tiempo t
15      cout << "El tiempo universal inicial es ";
16      t.imprimirUniversal(); // 00:00:00
17      cout << "\nEl tiempo universal estandar es ";
18      t.imprimirEstandar(); // 12:00:00 AM
19
20      t.establecerTiempo( 13, 27, 6 ); // cambia el tiempo
21
22      // imprime los nuevos valores del objeto Tiempo t
23      cout << "\n\nEl tiempo universal despues de establecerTiempo es ";
24      t.imprimirUniversal(); // 13:27:06
25      cout << "\nEl tiempo estandar despues de establecerTiempo es ";
26      t.imprimirEstandar(); // 1:27:06 PM
27
28      t.establecerTiempo( 99, 99, 99 ); // intenta hacer ajustes inválidos
29
30      // imprime los valores de t después de especificar valores inválidos
31      cout << "\n\nDespues de intentar ajustes invalidos:"
32          << "\nTiempo universal: ";
33      t.imprimirUniversal(); // 00:00:00
34      cout << "\nTiempo estandar: ";
35      t.imprimirEstandar(); // 12:00:00 AM
36      cout << endl;
37
38  } // fin de main

```

```

El tiempo universal inicial es 00:00:00
El tiempo universal estandar es 12:00:00 AM

El tiempo universal despues de establecerTiempo es 13:27:06
El tiempo estandar despues de establecerTiempo es 1:27:06 PM

Despues de intentar ajustes invalidos:
Tiempo universal: 00:00:00
Tiempo estandar: 12:00:00 AM

```

Figura 9.3 | Programa para probar la clase Tiempo. (Parte 2 de 2).

9.3 Alcance de las clases y acceso a los miembros de una clase

Los miembros de datos de una clase (variables declaradas en la definición de la clase) y las funciones miembro (funciones declaradas en la definición de la clase) pertenecen al alcance de esa clase. Las funciones que no son miembro se definen en **alcance de archivo**.

Dentro del alcance de una clase, los miembros de ésta se pueden utilizar inmediatamente por todas las funciones miembro de esa clase, y se pueden referenciar por nombre. Fuera del alcance de una clase, los miembros **public** de la clase se referencian a través de uno de los **manejadores** en un objeto: el nombre de un objeto, una referencia a un objeto, o un apuntador a un objeto. El tipo del objeto, referencia o apuntador especifica la interfaz (es decir, las funciones miembro) accesible para el cliente. [En el capítulo 10 veremos que el compilador inserta un manejador implícito en cada referencia a un miembro de datos o función miembro, desde el interior de un objeto].

Las funciones miembro de una clase se pueden sobrecargar, pero sólo mediante otras funciones miembro de esa clase. Para sobrecargar una función miembro, sólo hay que proporcionar en la definición de la clase un prototipo para cada versión de la función sobrecargada, y hay que proporcionar una definición de función separada para cada versión de la misma.

Las variables que se declaran en una función miembro tienen alcance de bloque y sólo esa función las conoce. Si una función miembro define una variable con el mismo nombre que una variable con alcance de clase, la variable con alcance de clase se oculta debido a la variable con alcance de bloque, en el alcance de bloque. Para acceder a dicha variable oculta, hay que colocar antes de su nombre el nombre de la clase, seguido del operador de resolución de ámbito (: :). Se puede acceder a las variables globales ocultas con el operador de resolución de ámbito unario (vea el capítulo 6).

Antes del operador punto (.) de selección de miembro se coloca el nombre de un objeto o una referencia a un objeto para acceder a los miembros de ese objeto. Antes del operador flecha (->) de selección de miembros se coloca un apuntador a un objeto, para acceder a los miembros de ese objeto.

En la figura 9.4 se utiliza una clase simple llamada `Cuenta` (líneas 8 a 25) con un miembro de datos `private` llamado `x` de tipo `int` (línea 24), la función miembro `public` llamada `establecerX` (líneas 12 a 15) y la función miembro `public` llamada `imprimir` (líneas 18 a 21) para ilustrar el acceso a los miembros de una clase con los operadores de selección de miembros. Por cuestión de simpleza, hemos incluido esta pequeña clase en el mismo archivo que la función `main` que la utiliza. En las líneas 29 a 31 se crean tres variables relacionadas con el tipo `Cuenta`: `contador` (un objeto `Cuenta`), `contadorPtr` (un apuntador a un objeto `Cuenta`) y `refContador` (una referencia a un objeto `Cuenta`). La variable `refContador`

```

1 // Fig. 9.4: fig09_04.cpp
2 // Demostración de los operadores . y -> para acceder a los miembros de una clase
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // definición de la clase Cuenta
8 class Cuenta
9 {
10 public: // los datos public son peligrosos
11     // establece el valor del miembro de datos private x
12     void setX( int valor )
13     {
14         x = valor;
15     } // fin de la función setX
16
17     // imprime el valor del miembro de datos private x
18     void imprimir()
19     {
20         cout << x << endl;
21     } // fin de la función imprimir
22
23 private:
24     int x;
25 }; // fin de la clase Cuenta
26
27 int main()
28 {
29     Cuenta contador; // crea objeto contador
30     Cuenta *contadorPtr = &contador; // crea apuntador a contador
31     Cuenta &contadorRef = contador; // crea referencia a contador
32
33     cout << "Establecer x en 1 e imprimir usando el nombre del objeto: ";
34     contador.setX( 1 ); // establece el miembro de datos x en 1
35     contador.imprimir(); // llama a la función miembro imprimir
36
37     cout << "Establecer x en 2 e imprimir usando una referencia a un objeto: ";
38     contadorRef.setX( 2 ); // establece el miembro de datos x en 2
39     contadorRef.imprimir(); // llama a la función miembro imprimir
40
41     cout << "Establecer x en 3 e imprimir usando un apuntador a un objeto: ";
42     contadorPtr->setX( 3 ); // establece el miembro de datos x en 3
43     contadorPtr->imprimir(); // llama a la función miembro imprimir
44     return 0;
45 } // fin de main

```

```

Establecer x en 1 e imprimir usando el nombre del objeto: 1
Establecer x en 2 e imprimir usando una referencia a un objeto: 2
Establecer x en 3 e imprimir usando un apuntador a un objeto: 3

```

Figura 9.4 | Acceso a las funciones miembro de un objeto a través de cada tipo de manejador de objeto: el nombre de un objeto, una referencia al objeto y un apuntador al mismo.

hace referencia a `contador`, y la variable `contadorPtr` apunta a `contador`. En las líneas 34 a 35 y 38 a 39, observe que el programa puede invocar a las funciones miembro `establecerX` e `imprimir` mediante el uso del operador punto (.) de selección de miembros, anteponiendo a éste el nombre del objeto (`contador`) o una referencia al objeto (`refContador`, que es un alias para `contador`). De manera similar, en las líneas 42 y 43 se demuestra que el programa puede invocar a las funciones miembro `establecerX` e `imprimir` mediante el uso de un apuntador (`cuentaPtr`) y el operador flecha (→) de selección de miembros.

9.4 Separar la interfaz de la implementación

En el capítulo 3, empezamos por incluir la definición de una clase y las definiciones de las funciones miembro en un archivo. Después demostramos cómo separar este código en dos archivos: un archivo de encabezado para la definición de la clase (es decir, su interfaz) y un archivo de código fuente para las definiciones de las funciones miembro de la clase (es decir, la implementación de la clase). Recuerde que esto facilita la modificación de los programas; en cuanto a lo que a los clientes de una clase concierne, las modificaciones en la implementación de la clase no afectan al cliente, siempre y cuando la interfaz de la clase que se proporcionó originalmente al cliente permanezca sin modificaciones.



Observación de Ingeniería de Software 9.6

Los clientes de una clase no necesitan acceso al código fuente de la clase para poder utilizarla. Sin embargo, los clientes necesitan poder enlazarse con el código objeto de la clase (es decir, la versión compilada de la clase). Esto alienta a los distribuidores independientes de software (ISV, por sus siglas en inglés) a proporcionar bibliotecas de clases para venta o licenciamiento. Los ISVs proporcionan en sus productos sólo los archivos de encabezado y los módulos de objetos. No se revela información propietaria, como sería el caso si se proporcionara el código fuente. La comunidad de usuarios de C++ se beneficia al tener a su disposición más bibliotecas de clases producidas por ISVs.

En realidad, las cosas no son tan prometedoras. Los archivos de encabezado contienen ciertas porciones de la implementación y sugerencias sobre otras. Por ejemplo, las funciones miembro en línea necesitan estar en un archivo de encabezado, de manera que cuando el compilador compile un cliente, éste pueda incluir la definición de la función `inline` en el lugar adecuado. Los miembros `private` de una clase se listan en la definición de la clase en el archivo de encabezado, por lo cual estos miembros están visibles para los clientes, aun y cuando éstos tal vez no tengan acceso a los miembros `private`. En el capítulo 10 mostraremos cómo usar una “clase proxy” para ocultar hasta los datos `private` de una clase a los clientes de la misma.



Observación de Ingeniería de Software 9.7

La información importante para la interfaz de una clase se debe incluir en el archivo de encabezado. La información que se utilizará sólo de manera interna en la clase y que no necesitarán los clientes de la misma deberá incluirse en el archivo de código fuente sin publicar. Esto es otro ejemplo más del principio del menor privilegio.

9.5 Funciones de acceso y funciones utilitarias

Las **funciones de acceso** pueden leer o mostrar datos. Otro uso común para las funciones de acceso es para evaluar la veracidad o falsedad de las condiciones; a menudo, a dichas funciones se les conoce como **funciones predicado**. Un ejemplo de una función predicado sería una función `estaVacio` para cualquier clase contenedora (una clase capaz de contener muchos objetos), tal como una lista enlazada, una pila o una cola. Un programa podría evaluar a `estaVacio` antes de tratar de leer otro elemento del objeto contenedor. Una función predicado `estaLleno` podría evaluar un objeto de una clase contenedora para determinar si no tiene espacio adicional. Dos funciones predicado útiles para nuestra clase `Tiempo` podrían ser `esAM` y `esPM`.

El programa de las figuras 9.5 a 9.7 demuestra la noción de una función utilitaria (también conocida como **función ayudante**). Una función utilitaria no forma parte de la interfaz `public` de una clase; en vez de ello, es una función miembro `private` que soporta la operación de las funciones miembro `public` de la clase. Las funciones utilitarias no están diseñadas para ser utilizadas por los clientes de una clase (pero pueden ser utilizadas por las funciones amigas de una clase, como veremos en el capítulo 10).

La clase `Vendedor` (figura 9.5) declara un arreglo de cifras de ventas de 12 meses (línea 16) y los prototipos para el constructor de la clase y las funciones miembro que manipulan el arreglo.

```

1 // Fig. 9.5: Vendedor.h
2 // Definición de la clase Vendedor.
3 // Funciones miembro definidas en Vendedor.cpp.
4 #ifndef VENDEDOR_H
5 #define VENDEDOR_H
6
7 class Vendedor
8 {
9 public:
10    Vendedor(); // constructor
11    void obtenerVentasDelUsuario(); // recibe las ventas del teclado
12    void establecerVentas( int, double ); // establece las ventas para un mes específico
13    void imprimirVentasAnuales(); // resume e imprime las ventas
14 private:
15    double totalVentasAnuales(); // prototipo para la función utilitaria
16    double ventas[ 12 ]; // cifras de ventas de 12 meses
17 }; // fin de la clase Vendedor
18
19 #endif

```

Figura 9.5 | Definición de la clase Vendedor.

En la figura 9.6, el constructor de `Vendedor` (líneas 15 a 19) inicializa el arreglo `ventas` con cero. La función miembro `public establecerVentas` (líneas 36 a 43) establece la cifra de ventas para un mes en el arreglo `ventas`. La función miembro `public imprimirVentasAnuales` (líneas 46 a 51) imprime las ventas totales para los últimos 12 meses. La función utilitaria `private totalVentasAnuales` (líneas 54 a 62) totaliza las cifras de ventas de 12 meses para beneficio de `imprimirVentasAnuales`. La función miembro `imprimirVentasAnuales` edita las cifras de ventas en formato monetario.

```

1 // Fig. 9.6: Vendedor.cpp
2 // Definiciones de las funciones miembro de la clase Vendedor.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7 using std::fixed;
8
9 #include <iomanip>
10 using std::setprecision;
11
12 #include "Vendedor.h" // incluye la definición de la clase Vendedor
13
14 // inicializa los elementos del arreglo ventas con 0.0
15 Vendedor::Vendedor()
16 {
17     for ( int i = 0; i < 12; i++ )
18         ventas[ i ] = 0.0;
19 } // fin del constructor de Vendedor
20
21 // obtiene 12 cifras de ventas del usuario mediante el teclado
22 void Vendedor::obtenerVentasDelUsuario()
23 {
24     double cifraVentas;
25
26     for ( int i = 1; i <= 12; i++ )
27     {
28         cout << "Escriba el monto de ventas para el mes " << i << ": ";
29         cin >> cifraVentas;
30         establecerVentas( i, cifraVentas );
31     } // fin de for

```

Figura 9.6 | Definiciones de las funciones miembro de la clase Vendedor. (Parte I de 2).

```

32 } // fin de la función obtenerVentasDelUsuario
33
34 // establece una de las 12 cifras de ventas mensuales; la función resta
35 // uno al valor del mes para el subíndice apropiado en el arreglo ventas
36 void Vendedor::establecerVentas( int mes, double monto )
37 {
38     // prueba que los valores de mes y monto sean válidos
39     if ( mes >= 1 && mes <= 12 && monto > 0 )
40         ventas[ mes - 1 ] = monto; // ajusta los subíndices de 0 a 11
41     else // valor de mes o monto inválido
42         cout << "Mes o cifra de ventas invalidos" << endl;
43 } // fin de la función establecerVentas
44
45 // imprime el total de ventas anuales (con la ayuda de la función utilitaria)
46 void Vendedor::imprimirVentasAnuales()
47 {
48     cout << setprecision( 2 ) << fixed
49         << "\nLas ventas anuales totales son: $"
50         << totalVentasAnuales() << endl; // llama a la función utilitaria
51 } // fin de la función imprimirVentasAnuales
52
53 // función utilitaria privada para totalizar las ventas anuales
54 double Vendedor::totalVentasAnuales()
55 {
56     double total = 0.0; // inicializa el total
57
58     for ( int i = 0; i < 12; i++ ) // sintetiza los resultados de las ventas
59         total += ventas[ i ]; // suma las ventas del mes i al total
60
61     return total;
62 } // fin de la función totalVentasAnuales

```

Figura 9.6 | Definiciones de las funciones miembro de la clase `Vendedor`. (Parte 2 de 2).

En la figura 9.7, observe que la función `main` de la aplicación sólo incluye una secuencia simple de llamadas a funciones miembro; no hay instrucciones de control. La lógica de manipular el arreglo `ventas` se encapsula por completo en las funciones miembro de la clase `Vendedor`.



Observación de Ingeniería de Software 9.8

Un fenómeno de la programación orientada a objetos es que, una vez definida una clase, los procesos de crear y manipular objetos de esa clase implican a menudo generar sólo una secuencia simple de llamadas a las funciones miembro; se requieren pocas (si acaso) instrucciones de control. En contraste, es común tener instrucciones de control en la implementación de las funciones miembro de una clase.

```

1 // Fig. 9.7: fig09_07.cpp
2 // Demostración de una función utilitaria.
3 // Compile este programa con Vendedor.cpp
4
5 // incluye la definición de la clase Vendedor de Vendedor.h
6 #include "Vendedor.h"
7
8 int main()
9 {
10     Vendedor s; // crea el objeto Vendedor s
11
12     s.obtenerVentasDelUsuario(); // observe el código secuencial simple; no
13     s.imprimirVentasAnuales(); // hay instrucciones de control en main

```

Figura 9.7 | Demostración de una función utilitaria. (Parte 1 de 2).

```

14     return 0;
15 } // fin de main

```

Escriba el monto de ventas para el mes 1: 5314.76
Escriba el monto de ventas para el mes 2: 4292.38
Escriba el monto de ventas para el mes 3: 4589.83
Escriba el monto de ventas para el mes 4: 5534.03
Escriba el monto de ventas para el mes 5: 4376.34
Escriba el monto de ventas para el mes 6: 5698.45
Escriba el monto de ventas para el mes 7: 4439.22
Escriba el monto de ventas para el mes 8: 5893.57
Escriba el monto de ventas para el mes 9: 4909.67
Escriba el monto de ventas para el mes 10: 5123.45
Escriba el monto de ventas para el mes 11: 4024.97
Escriba el monto de ventas para el mes 12: 5923.92

Las ventas anuales totales son: \$60120.59

Figura 9.7 | Demostración de una función utilitaria. (Parte 2 de 2).

9.6 Ejemplo práctico de la clase Tiempo: constructores con argumentos predeterminados

El programa de las figuras 9.8 a 9.10 mejora la clase **Tiempo** para demostrar cómo se pasan los argumentos implícitamente a un constructor. El constructor definido en la figura 9.2 inicializa **hora**, **minuto** y **segundo** con 0 (es decir, media noche en el formato de tiempo universal). Al igual que otras funciones, los constructores pueden especificar argumentos predeterminados. En la línea 13 de la figura 9.8 se declara el constructor de **Tiempo** para incluir argumentos predeterminados, especificando un valor predeterminado de cero para cada argumento que se pasa al constructor. En la figura 9.9, en las líneas 14 a 17 se define la nueva versión del constructor de **Tiempo** que recibe valores para los parámetros **hr**, **min** y **seg** que se utilizarán para inicializar los miembros de datos **private hora**, **minuto** y **segundo**, respectivamente. Observe que la clase **Tiempo** proporciona funciones **set** y **get** para cada miembro de datos. El constructor de **Tiempo** ahora llama a **setTiempo**, que a su vez llama a las funciones **setHora**, **setMinuto** y **setSegundo** para validar y asignar valores a los miembros de datos. Los argumentos predeterminados para el constructor aseguran que, aun si no se proporcionan valores en la llamada a un constructor, éste de todas formas inicializa los miembros de datos para mantener el objeto **Tiempo** en un estado consistente. Un constructor que utiliza valores predeterminados para todos sus argumentos también es un constructor predeterminado; es decir, un constructor que se puede invocar sin argumentos. Puede haber por lo menos un constructor predeterminado por clase.

```

1 // Fig. 9.8: Tiempo.h
2 // Clase Tiempo que contiene un constructor con argumentos predeterminados.
3 // Las funciones miembro se definen en Tiempo.cpp.
4
5 // evita múltiples inclusiones del archivo de encabezado
6 #ifndef TIEMPO_H
7 #define TIEMPO_H
8
9 // Definición del tipo de datos abstracto Tiempo
10 class Tiempo
11 {
12 public:
13     Tiempo( int = 0, int = 0, int = 0 ); // constructor predeterminado
14
15     // funciones "establecer"
16     void establecerTiempo( int, int, int ); // establece hora, minuto, segundo
17     void establecerHora( int ); // establece la hora (después de la validación)
18     void establecerMinuto( int ); // establece el minuto (después de la validación)
19     void establecerSegundo( int ); // establece el segundo (después de la validación)

```

Figura 9.8 | Clase **Tiempo** que contiene un constructor con argumentos predeterminados. (Parte 1 de 2).

```

20 // funciones "obtener"
21 int obtenerHora(); // devuelve la hora
22 int obtenerMinuto(); // devuelve el minuto
23 int obtenerSegundo(); // devuelve el segundo
24
25
26 void imprimirUniversal(); // imprime Tiempo en formato universal
27 void imprimirEstandar(); // imprime Tiempo en formato estándar
28 private:
29     int hora; // 0 - 23 (formato de reloj de 24 horas)
30     int minuto; // 0 - 59
31     int segundo; // 0 - 59
32 }; // fin de la clase Tiempo
33
34 #endif

```

Figura 9.8 | Clase `Tiempo` que contiene un constructor con argumentos predeterminados. (Parte 2 de 2).

En la figura 9.9, en la línea 16 del constructor se hace una llamada a la función miembro `setTiempo` con los valores que se pasan al constructor (o a los valores predeterminados). La función `setTiempo` llama a `setHora` para asegurar que el valor suministrado para `hora` esté en el rango de 0 a 23, y después llama a `setMinuto` y `setSegundo` para asegurar que los valores para `minuto` y `segundo` se encuentren en el rango de 0 a 59. Si un valor está fuera de rango, se establece a cero (para asegurar que cada miembro de datos permanezca en un estado consistente). En el capítulo 16, Manejo de excepciones, lanzaremos excepciones cuando un valor se encuentre fuera de rango, en vez de simplemente asignar un valor consistente predeterminado.

```

1 // Fig. 9.9: Tiempo.cpp
2 // Definiciones de las funciones miembro para la clase Tiempo.
3 #include <iostream>
4 using std::cout;
5
6 #include <iomanip>
7 using std::setfill;
8 using std::setw;
9
10 #include "Tiempo.h" // incluye la definición de la clase Tiempo de Tiempo.h
11
12 // el constructor de Tiempo inicializa cada miembro de datos con cero;
13 // asegura que los objetos Tiempo empiecen en un estado consistente
14 Tiempo::Tiempo( int hr, int min, int seg )
15 {
16     establecerTiempo( hr, min, seg ); // valida y establece Tiempo
17 } // fin del constructor de Tiempo
18
19 // establece nuevo valor de Tiempo usando el formato universal; asegura que
20 // los datos permanezcan consistentes al establecer los valores inválidos en cero
21 void Tiempo::establecerTiempo( int h, int m, int s )
22 {
23     establecerHora( h ); // establece el campo private hora
24     establecerMinuto( m ); // establece el campo private minuto
25     establecerSegundo( s ); // establece el campo private segundo
26 } // fin de la función establecerTiempo
27
28 // establece el valor de hora
29 void Tiempo::establecerHora( int h )
30 {
31     hora = ( h >= 0 && h < 24 ) ? h : 0; // valida la hora

```

Figura 9.9 | Definiciones de las funciones miembro de la clase `Tiempo`, que incluyen un constructor que recibe argumentos. (Parte 1 de 2).

```

32 } // fin de la función establecerHora
33
34 // establece el valor de minuto
35 void Tiempo::establecerMinuto( int m )
36 {
37     minuto = ( m >= 0 && m < 60 ) ? m : 0; // valida el minuto
38 } // fin de la función establecerMinuto
39
40 // establece el valor de segundo
41 void Tiempo::establecerSegundo( int s )
42 {
43     segundo = ( s >= 0 && s < 60 ) ? s : 0; // valida el segundo
44 } // fin de la función establecerSegundo
45
46 // devuelve el valor de la hora
47 int Tiempo::obtenerHora()
48 {
49     return hora;
50 } // fin de la función obtenerHora
51
52 // devuelve el valor del minuto
53 int Tiempo::obtenerMinuto()
54 {
55     return minuto;
56 } // fin de la función obtenerMinuto
57
58 // devuelve el valor del segundo
59 int Tiempo::obtenerSegundo()
60 {
61     return segundo;
62 } // fin de la función obtenerSegundo
63
64 // imprime el Tiempo en formato universal (HH:MM:SS)
65 void Tiempo::imprimirUniversal()
66 {
67     cout << setfill( '0' ) << setw( 2 ) << obtenerHora() << ":"
68         << setw( 2 ) << obtenerMinuto() << ":" << setw( 2 ) << obtenerSegundo();
69 } // fin de la función imprimirUniversal
70
71 // imprime el Tiempo en formato estándar (HH:MM:SS AM or PM)
72 void Tiempo::imprimirEstandar()
73 {
74     cout << ( ( obtenerHora() == 0 || obtenerHora() == 12 ) ? 12 : obtenerHora() % 12 )
75         << ":" << setfill( '0' ) << setw( 2 ) << obtenerMinuto()
76         << ":" << setw( 2 ) << obtenerSegundo() << ( hora < 12 ? " AM" : " PM" );
77 } // fin de la función imprimirEstandar

```

Figura 9.9 | Definiciones de las funciones miembro de la clase `Tiempo`, que incluyen un constructor que recibe argumentos. (Parte 2 de 2).

Observe que el constructor de `Tiempo` podría escribirse de manera que incluya las mismas instrucciones que la función miembro `setTiempo`, o incluso las instrucciones individuales en las funciones `setHora`, `setMinuto` y `setSegundo`. Llamar a `setHora`, `setMinuto` y `setSegundo` desde el constructor puede ser ligeramente más eficiente, ya que se elimina la llamada adicional a `setTiempo`. De manera similar, al copiar el código de las líneas 31, 37 y 43 en el constructor se eliminaría la sobrecarga de llamar a `setTiempo`, `setHora`, `setMinuto` y `setSegundo`. Al codificar el constructor de `Tiempo` o la función miembro `setTiempo` como una copia del código en las líneas 31, 37 y 43 se dificultaría más el mantenimiento de esta clase. Si se fuera a modificar las implementaciones de `setHora`, `setMinuto` y `setSegundo`, la implementación de cualquier función miembro que duplicara las líneas 31, 37 y 43 tendría que modificarse de manera acorde. Al hacer que el constructor de `Tiempo` llame a `setTiempo`, y que `setTiempo` llame a `setHora`, `setMinuto` y `setSegundo`, podemos limitar las modificaciones en el código que validan la hora, minuto o segundo con la correspon-

diente función `set`. Esto reduce la probabilidad de errores a la hora de alterar la implementación de la clase. Además, el rendimiento del constructor de `Tiempo` y de `setTiempo` se puede mejorar al declararlos explícitamente como `inline`, o al definirlos en la definición de la clase (que pone implícitamente en línea la definición de la función).



Observación de Ingeniería de Software 9.9

Si una función miembro de una clase proporciona de antemano toda o parte de la funcionalidad requerida por un constructor (u otra función miembro) de la clase, hay que llamar a esa función miembro desde el constructor (o de otra función miembro). Esto simplifica el mantenimiento del código y reduce la probabilidad de un error si la implementación del código se modifica. Como regla general: evite repetir código.



Observación de Ingeniería de Software 9.10

Cualquier modificación a los valores predeterminados de los argumentos de una función requiere que el código cliente se vuelva a compilar (para asegurar que el programa siga funcionando correctamente).

La función `main` en la figura 9.10 inicializa cinco objetos `Tiempo`: uno con los tres argumentos predeterminados en la llamada implícita al constructor (línea 11), uno con un argumento especificado (línea 12), uno con dos argumentos especificados (línea 13), uno con tres argumentos especificados (línea 14) y uno con tres argumentos inválidos especificados (línea 15). Así, el programa muestra cada objeto en formatos de tiempo universal y tiempo estándar.

```

1 // Fig. 9.10: fig09_10.cpp
2 // Demostración de un constructor predeterminado para la clase Tiempo.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Tiempo.h" // incluye la definición de la clase Tiempo de Tiempo.h
8
9 int main()
10 {
11     Tiempo t1; // valor predeterminado en todos los argumentos
12     Tiempo t2( 2 ); // se especifica hora; valores predeterminados para minuto y segundo
13     Tiempo t3( 21, 34 ); // se especifican hora y minuto; valor predeterminado para segundo
14     Tiempo t4( 12, 25, 42 ); // se especifican hora, minuto y segundo
15     Tiempo t5( 27, 74, 99 ); // se especifican valores incorrectos
16
17     cout << "Se construyo con:\n\tt1: todos los argumentos predeterminados\n\t";
18     t1.imprimirUniversal(); // 00:00:00
19     cout << "\n\t";
20     t1.imprimirEstandar(); // 12:00:00 AM
21
22     cout << "\n\tt2: se especifico hora; minuto y segundo predeterminados\n\t";
23     t2.imprimirUniversal(); // 02:00:00
24     cout << "\n\t";
25     t2.imprimirEstandar(); // 2:00:00 AM
26
27     cout << "\n\tt3: se especificaron hora y minuto; segundo predeterminado\n\t";
28     t3.imprimirUniversal(); // 21:34:00
29     cout << "\n\t";
30     t3.imprimirEstandar(); // 9:34:00 PM
31
32     cout << "\n\tt4: se especificaron hora, minuto y segundo\n\t";
33     t4.imprimirUniversal(); // 12:25:42
34     cout << "\n\t";
35     t4.imprimirEstandar(); // 12:25:42 PM
36
37     cout << "\n\tt5: se especificaron valores invalidos\n\t";
38     t5.imprimirUniversal(); // 00:00:00

```

Figura 9.10 | Constructor con argumentos predeterminados. (Parte I de 2).

```

39     cout << "\n ";
40     t5.imprimirEstandar(); // 12:00:00 AM
41     cout << endl;
42     return 0;
43 } // fin de main

```

Se construyo con:

```

t1: todos los argumentos predeterminados
    00:00:00
    12:00:00 AM

t2: se especifico hora; minuto y segundo predeterminados
    02:00:00
    2:00:00 AM

t3: se especificaron hora y minuto; segundo predeterminado
    21:34:00
    9:34:00 PM

t4: se especificaron hora, minuto y segundo
    12:25:42
    12:25:42 PM

t5: se especificaron valores invalidos
    00:00:00
    12:00:00 AM

```

Figura 9.10 | Constructor con argumentos predeterminados. (Parte 2 de 2).

Observaciones en relación con el constructor y las funciones Establecer y Obtener de la clase Tiempo

Las funciones *set* y *get* de *Tiempo* se llaman a través del cuerpo de la clase. En especial, la función *setTiempo* (líneas 21 a 26 de la figura 9.9) llama a las funciones *setHora*, *setMinuto* y *setSegundo*, y las funciones *imprimirUniversal* e *imprimirEstandar* llaman a las funciones *getHora*, *getMinuto* y *getSegundo* en las líneas 67 a 68 y 74 a 76, respectivamente. En cada caso, estas funciones podrían haber accedido a los datos *private* de la clase directamente. Sin embargo, considere el modificar la representación del tiempo, de tres valores *int* (que requieren 12 bytes de memoria) a un solo valor *int* que represente el número total de segundos transcurridos desde medianoche (que sólo requiere cuatro bytes de memoria). Si realizáramos dicha modificación, sólo tendrían que cambiar los cuerpos de las funciones que acceden directamente a los datos *private*; en especial, las funciones individuales *set* y *get* para *hora*, *minuto* y *segundo*. No habría necesidad de modificar los cuerpos de las funciones *setTiempo*, *imprimirUniversal* o *imprimirEstandar*, ya que no acceden directamente a los datos. Al diseñar la clase de esta forma se reduce la probabilidad de que se produzcan errores de programación al alterar la implementación de la clase.

De manera similar, el constructor de *Tiempo* podría escribirse de manera que incluya una copia de las instrucciones apropiadas de la función *setTiempo*. Esto puede ser ligeramente más eficiente, debido a que se eliminan la llamada adicional al constructor y la llamada a *setTiempo*. Sin embargo, al duplicar instrucciones en varias funciones o constructores se dificulta más el proceso de modificar la representación interna de datos de la clase. Para que el constructor de *Tiempo* llame directamente a la función *establecerTiempo* se requiere que las modificaciones a la implementación de *establecerTiempo* se realicen sólo una vez.



Error común de programación 9.2

Un constructor puede llamar a otras funciones miembro de la clase, como las funciones establecer u obtener, pero debido a que el constructor está inicializando el objeto, los miembros de datos tal vez no se encuentren todavía en un estado consistente. Si se utilizan los miembros de datos antes de que se hayan inicializado en forma apropiada, se pueden producir errores lógicos.

9.7 Destructores

Un destructor es otro tipo de función miembro especial. El nombre del destructor para una clase es el carácter tilde (~) seguido del nombre de la clase. Esta convención de nomenclatura tiene un atractivo intuitivo, ya que como veremos en

un capítulo posterior, el operador tilde es el operador de complemento a nivel de bits, y en cierto sentido, el destructor es el complemento del constructor. Observe que un destructor se representa comúnmente con la abreviación “dtor” en la literatura. Nosotros preferimos no usar esta abreviación.

El destructor de una clase se llama de manera implícita cuando se destruye un objeto. Por ejemplo, esto ocurre a medida que un objeto automático se destruye cuando la ejecución del programa sale del alcance en el que se instanció el objeto. *El destructor en sí no libera la memoria del objeto*; realiza tareas de mantenimiento de terminación antes de que se reclame la memoria del objeto, de forma que ésta se pueda reutilizar para contener nuevos objetos.

Un destructor no recibe parámetros y no devuelve un valor. Un destructor no puede especificar un tipo de valor de retorno; ni siquiera `void`. Una clase sólo puede tener un destructor; no se permite la sobrecarga de destructores. Un destructor debe ser `public`.

Error común de programación 9.3



Es un error de sintaxis tratar de pasar argumentos a un destructor, especificar un tipo de valor de retorno para un destructor (ni siquiera se puede especificar void), devolver valores de un destructor o sobrecargarlo.

Aun y cuando no se han proporcionado destructores para las clases presentadas hasta ahora, toda clase tiene un destructor. Si el programador no especifica un destructor de manera explícita, el compilador crea un “destructor” vacío. [Nota: más adelante veremos que un destructor creado de manera implícita desempeña, de hecho, operaciones importantes sobre los objetos que se crean a través de la composición (capítulo 10) y la herencia (capítulo 12)]. En el capítulo 11 crearemos destructores apropiados para las clases cuyos objetos contienen memoria asignada en forma dinámica (por ejemplo, para arreglos y cadenas) o que utilizan otros recursos del sistema (por ejemplo, archivos en disco, que veremos en el capítulo 17). En el capítulo 10 veremos cómo asignar y desasignar la memoria en forma dinámica.



Observación de Ingeniería de Software 9.11

Como veremos en el resto del libro, los constructores y los destructores tienen mucha más prominencia en C++ y la programación orientada a objetos de la que es posible transmitir sólo con base en nuestra breve introducción en este capítulo.

9.8 Cuándo se hacen llamadas a los constructores y destructores

El compilador llama de manera implícita a los constructores y destructores. El orden en el que ocurren estas llamadas a funciones depende del orden en el que la ejecución entra y sale de los alcances en los que se instancian los objetos. Por lo general, las llamadas a los destructores se realizan en orden inverso a las llamadas correspondientes a los constructores, pero como veremos en las figuras 9.11 a 9.13, las clases de almacenamiento de los objetos pueden alterar el orden en el que se llama a los destructores.

Los constructores se llaman para los objetos que se definen en alcance global antes de que cualquier otra función (incluyendo a `main`) en ese archivo empiece a ejecutarse (aunque no se garantiza el orden de ejecución de los constructores de objetos globales entre los archivos). Los destructores correspondientes se llaman cuando termina `main`. La función `exit` obliga a un programa a terminar de inmediato y no ejecuta a los destructores de objetos automáticos. A menudo, la función se utiliza para terminar un programa cuando se detecta un error en la entrada, o si no se puede abrir un archivo que va a ser procesado por el programa. La función `abort` se ejecuta de manera similar a la función `exit`, pero obliga al programa a terminar de inmediato, sin permitir que se llamen los destructores de ningún objeto. La función `abort` se utiliza comúnmente para indicar una terminación anormal del programa. (Vea el apéndice E para obtener más información acerca de las funciones `exit` y `abort`).

El constructor para un objeto local automático se llama cuando la ejecución llega al punto en el que se define ese objeto; el correspondiente destructor se llama cuando la ejecución sale del alcance del objeto (es decir, el bloque en el que se define el objeto ha terminado de ejecutarse). Los constructores y destructores para los objetos automáticos se llaman cada vez que la ejecución entra y sale del alcance del objeto. Los destructores no se llaman para los objetos automáticos si el programa termina con una llamada a la función `exit` o a la función `abort`.

El constructor para un objeto local `static` se llama sólo una vez, cuando la ejecución llega por primera vez al punto en el que se define el objeto; el destructor correspondiente se llama cuando termina `main`, o cuando el programa llama a la función `exit`. Los objetos globales y `static` se destruyen en el orden inverso de su creación. Los destructores no se llaman para los objetos `static` si el programa termina con una llamada a la función `abort`.

El programa de las figuras 9.11 a 9.13 demuestra el orden en el que se llaman los constructores y destructores para los objetos de la clase `CrearYDestruir` (figuras 9.11 y 9.12) de varias clases de almacenamiento en varios alcances. Cada objeto de la clase `CrearYDestruir` contiene un entero (`idObjeto`) y una cadena (`mensaje`), los cuales se utilizan en la salida del

```

1 // Fig. 9.11: CrearYDestruir.h
2 // Definición de la clase CrearYDestruir.
3 // Las funciones miembro se definen en CrearYDestruir.cpp.
4 #include <string>
5 using std::string;
6
7 #ifndef CREAR_H
8 #define CREAR_H
9
10 class CrearYDestruir
11 {
12 public:
13     CrearYDestruir( int, string ); // constructor
14     ~CrearYDestruir(); // destructor
15 private:
16     int idObjeto; // número de ID para el objeto
17     string mensaje; // mensaje que describe al objeto
18 }; // fin de la clase CrearYDestruir
19
20 #endif

```

Figura 9.11 | Definición de la clase CrearYDestruir.

```

1 // Fig. 9.12: CrearYDestruir.cpp
2 // Definiciones de las funciones miembro de CrearYDestruir.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "CrearYDestruir.h" // incluye la definición de la clase CrearYDestruir
8
9 // constructor
10 CrearYDestruir::CrearYDestruir( int ID, string cadenaMensaje )
11 {
12     idObjeto = ID; // establece el número de ID del objeto
13     mensaje = cadenaMensaje; // establece el mensaje descriptivo del objeto
14
15     cout << "El constructor del objeto " << idObjeto << " se ejecuta "
16         << mensaje << endl;
17 } // fin del constructor de CrearYDestruir
18
19 // destructor
20 CrearYDestruir::~CrearYDestruir()
21 {
22     // imprime nueva línea para ciertos objetos; mejora la legibilidad
23     cout << ( idObjeto == 1 || idObjeto == 6 ? "\n" : "" );
24
25     cout << "El destructor del objeto " << idObjeto << " se ejecuta "
26         << mensaje << endl;
27 } // fin del destructor ~CrearYDestruir

```

Figura 9.12 | Definiciones de las funciones miembro de la clase CrearYDestruir.

programa para identificar al objeto (figura 9.11, líneas 16 y 17). Este ejemplo mecánico es sólo para fines pedagógicos. Por esta razón, la línea 23 del destructor en la figura 9.12 determina si el objeto que se va a destruir tiene un valor de `idObjeto` de 1 o 6 y, de ser así, imprime un carácter de nueva línea. Esta línea facilita más el seguimiento de la salida del programa.

En la figura 9.13 se define el objeto `primero` (línea 12) en alcance global. Su constructor se llama antes de que se ejecute cualquier instrucción en `main`, y su destructor se llama al momento en que termina el programa, después de los destructores para todos los demás objetos que se hayan ejecutado.

La función `main` (líneas 14 a 26) declara tres objetos. Los objetos `segundo` (línea 17) y `cuarto` (línea 23) son objetos locales automáticos, y el objeto `tercero` (línea 18) es un objeto local `static`. El constructor para cada uno de estos objetos

se llama cuando la ejecución llega al punto en el que se declara el objeto. Los destructores para los objetos **cuarto** y después **segundo** se llaman (es decir, el orden inverso en que se llamaron sus constructores) cuando la ejecución llega al final de **main**. Como el objeto **tercero** es **static**, existe hasta que el programa termina su ejecución. El destructor para el objeto **tercero** se llama antes del destructor para el objeto global **primero**, pero después de que se destruyen todos los demás objetos.

La función **crear** (líneas 29 a 36) declara tres objetos: **quinto** (línea 32) y **séptimo** (línea 34) como objetos automáticos locales, y **sexto** (línea 33) como objeto local **static**. Los destructores para los objetos **séptimo** y después **quinto** se llaman (es decir, el orden inverso en que se llamaron sus constructores) cuando **crear** termina. Como **sexto** es **static**, existe hasta que el programa termina su ejecución. El destructor de **sexto** se llama antes de los destructores para **tercero** y **primero**, pero después de que se destruyen todos los demás objetos.

```

1 // Fig. 9.13: fig09_13.cpp
2 // Demostración del orden en el que se llama a los
3 // constructores y destructores.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "CrearYDestruir.h" // incluye la definición de la clase CrearYDestruir
9
10 void crear( void ); // prototipo
11
12 CrearYDestruir primero( 1, "(global antes de main)" ); // objeto global
13
14 int main()
15 {
16     cout << "\nFUNCION MAIN: EMPIEZA LA EJECUCION" << endl;
17     CrearYDestruir segundo( 2, "(local automatico en main)" );
18     static CrearYDestruir tercero( 3, "(local static en main)" );
19
20     crear(); // llama a la función para crear objetos
21
22     cout << "\nFUNCION MAIN: CONTINUA LA EJECUCION" << endl;
23     CrearYDestruir cuarto( 4, "(local automatico en main)" );
24     cout << "\nFUNCION MAIN: TERMINA LA EJECUCION" << endl;
25     return 0;
26 } // fin de main
27
28 // función para crear objetos
29 void crear( void )
30 {
31     cout << "\nFUNCION crear: EMPIEZA LA EJECUCION" << endl;
32     CrearYDestruir quinto( 5, "(local automatico en crear)" );
33     static CrearYDestruir sexto( 6, "(local static en crear)" );
34     CrearYDestruir septimo( 7, "(local automatico en crear)" );
35     cout << "\nFUNCION crear: TERMINA LA EJECUCION" << endl;
36 } // fin de la función crear

```

```

El constructor del objeto 1  se ejecuta  (global antes de main)
FUNCION MAIN: EMPIEZA LA EJECUCION
El constructor del objeto 2  se ejecuta  (local automatico en main)
El constructor del objeto 3  se ejecuta  (local static en main)

FUNCION crear: EMPIEZA LA EJECUCION
El constructor del objeto 5  se ejecuta  (local automatico en crear)
El constructor del objeto 6  se ejecuta  (local static en crear)
El constructor del objeto 7  se ejecuta  (local automatico en crear)

FUNCION crear: TERMINA LA EJECUCION
El destructor del objeto 7  se ejecuta  (local automatico en crear)
El destructor del objeto 5  se ejecuta  (local automatico en crear)

```

Figura 9.13 | Orden en el que se llaman los constructores y los destructores. (Parte I de 2).

```

FUNCION MAIN: CONTINUA LA EJECUCION
El constructor del objeto 4 se ejecuta (local automatico en main)

FUNCION MAIN: TERMINA LA EJECUCION
El destructor del objeto 4 se ejecuta (local automatico en main)
El destructor del objeto 2 se ejecuta (local automatico en main)
El destructor del objeto 6 se ejecuta (local static en crear)
El destructor del objeto 3 se ejecuta (local static en main)
El destructor del objeto 1 se ejecuta (global antes de main)

```

Figura 9.13 | Orden en el que se llaman los constructores y los destructores. (Parte 2 de 2).

9.9 Ejemplo práctico con la clase Tiempo: una trampa sutil (devolver una referencia a un miembro de datos private)

Una referencia a un objeto es un alias para el nombre del objeto y, por ende, puede usarse del lado izquierdo de una instrucción de asignación. En este contexto, la referencia se convierte en un *lvalue* perfectamente aceptable que puede recibir un valor. Una manera de usar esta herramienta (¡por desgracia!) es hacer que una función miembro `public` de una clase devuelva una referencia a un miembro de datos `private` de esa clase. Observe que si una función devuelve una referencia `const`, esa referencia no se puede utilizar como *lvalue* modificable.

El programa de las figuras 9.14 a 9.16 utiliza una clase `Tiempo` simplificada (figuras 9.14 y 9.15) para demostrar cómo devolver una referencia a un miembro de datos `private` con la función miembro `setHoraIncorrecta` (declarada en la línea 15 de la figura 9.14 y definida en las líneas 29 a 33 de la figura 9.15). Dicha devolución de referencia en realidad hace una llamada a la función miembro `setHoraIncorrecta`, ¡un alias para el miembro de datos `private hora`! La llamada a la función se puede utilizar en cualquier forma que se pueda usar el miembro de datos `private`, incluso como un *lvalue* en una instrucción de asignación, ¡con lo cual se permite a los clientes de la clase hacer lo que quieran con los datos `private` de ésta! Observe que ocurriría el mismo problema si la función devolviera un apuntador a los datos `private`.

En la figura 9.16 se declaran el objeto `Tiempo` llamado `t` (línea 12) y la referencia `horaRef` (línea 15), que se inicializa con la referencia devuelta por la llamada `t.setHoraIncorrecta(20)`. En la línea 17 se muestra el valor del alias `horaRef`. Esto muestra cómo `horaRef` quebranta el encapsulamiento de la clase; las instrucciones en `main` no deben tener acceso a los datos `private` de la clase. A continuación, en la línea 18 se usa el alias para establecer el valor de `hora` en 30 (un valor inválido) y en la línea 19 se muestra el valor devuelto por la función `getHora`, para mostrar que al asignar un valor a `horaRef` en realidad se modifican los datos `private` en el objeto `Tiempo t`. Por último, en la línea 23 se usa la misma llamada a la función `setHoraIncorrecta` como un *lvalue* y se asigna 74 (otro valor inválido) a la referencia devuelta por la función.

```

1 // Fig. 9.14: Tiempo.h
2 // Declaración de la clase Tiempo.
3 // Las funciones miembro se definen en Tiempo.cpp
4
5 // evita múltiples inclusiones del archivo de encabezado
6 #ifndef TIEMPO_H
7 #define TIEMPO_H
8
9 class Tiempo
10 {
11 public:
12     Tiempo( int = 0, int = 0, int = 0 );
13     void establecerTiempo( int, int, int );
14     int obtenerHora();
15     int &establecerHoraIncorrecta( int ); // devolución de referencia PELIGROSA
16 private:
17     int hora;

```

Figura 9.14 | Declaración de la clase `Tiempo`. (Parte 1 de 2).

```

18     int minuto;
19     int segundo;
20 } // fin de la clase Tiempo
21
22 #endif

```

Figura 9.14 | Declaración de la clase Tiempo. (Parte 2 de 2).

```

1 // Fig. 9.15: Tiempo.cpp
2 // Definiciones de las funciones miembro de la clase Tiempo.
3 #include "Tiempo.h" // incluye la definición de la clase Tiempo
4
5 // función constructor para inicializar los datos privados;
6 // llama a la función miembro establecerTiempo para establecer las variables;
7 // los valores predeterminados son 0 (vea la definición de la clase)
8 Tiempo::Tiempo( int hr, int min, int seg )
9 {
10     establecerTiempo( hr, min, seg );
11 } // fin del constructor de Tiempo
12
13 // establece los valores de hora, minuto y segundo
14 void Tiempo::establecerTiempo( int h, int m, int s )
15 {
16     hora = ( h >= 0 && h < 24 ) ? h : 0; // valida la hora
17     minuto = ( m >= 0 && m < 60 ) ? m : 0; // valida el minuto
18     segundo = ( s >= 0 && s < 60 ) ? s : 0; // valida el segundo
19 } // fin de la función establecerTiempo
20
21 // devuelve el valor de hora
22 int Tiempo::obtenerHora()
23 {
24     return hora;
25 } // fin de la función obtenerHora
26
27 // MALA PRÁCTICA DE PROGRAMACIÓN:
28 // Devolver una referencia a un miembro de datos privado.
29 int &Tiempo::establecerHoraIncorrecta( int hh )
30 {
31     hora = ( hh >= 0 && hh < 24 ) ? hh : 0;
32     return hora; // devolución de referencia PELIGROSA
33 } // fin de la función establecerHoraIncorrecta

```

Figura 9.15 | Definiciones de las funciones miembro de la clase Tiempo

```

1 // Fig. 9.16: fig09_16.cpp
2 // Demostración de una función miembro pública que
3 // devuelve una referencia a un miembro de datos privado.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "Tiempo.h" // incluye la definición de la clase Tiempo
9
10 int main()
11 {
12     Tiempo t; // crea un objeto Tiempo
13
14     // inicializa horaRef con la referencia devuelta por establecerHoraIncorrecta
15     int &horaRef = t.establecerHoraIncorrecta( 20 ); // 20 es una hora válida

```

Figura 9.16 | Devolución de una referencia a un miembro de datos **private**. (Parte 1 de 2).

```

16    cout << "Hora valida antes de la modificacion: " << horaRef;
17    horaRef = 30; // usa horaRef para establecer un valor inválido en el objeto Tiempo t
18    cout << "\nHora invalida despues de la modificacion: " << t.obtenerHora();
19
20
21    // Peligroso: ¡La llamada a una función que devuelve
22    // una referencia se puede usar como un lvalue!
23    t.establecerHoraIncorrecta( 12 ) = 74; // asigna otro valor inválido a hora
24
25    cout << "\n\n*****\n"
26    << "MALA PRACTICA DE PROGRAMACION!!!!!!\n"
27    << "t.establecerHoraIncorrecta( 12 ) como un lvalue, hora invalida: "
28    << t.obtenerHora()
29    << "\n*****" << endl;
30
31    return 0;
31 } // fin de main

```

Hora valida antes de la modificacion: 20
 Hora invalida despues de la modificacion: 30

 MALA PRACTICA DE PROGRAMACION!!!!!!
 t.establecerHoraIncorrecta(12) como un lvalue, hora invalida: 74

Figura 9.16 | Devolución de una referencia a un miembro de datos **private**. (Parte 2 de 2).



Tip para prevenir errores 9.4

Al devolver una referencia a un apuntador a un miembro de datos **private** se quebra la encapsulación y se hace dependiente el código cliente de la representación de los datos de la clase; ésta es una práctica peligrosa que debe evitarse.

9.10 Asignación predeterminada a nivel de miembros

El operador de asignación (=) se puede utilizar para asignar un objeto a otro objeto del mismo tipo. De manera predeterminada, dicha asignación se realiza mediante la **asignación a nivel de miembros**; cada miembro de datos del objeto a la derecha del operador de asignación se asigna de manera individual al mismo miembro de datos en el objeto a la izquierda del operador de asignación. En las figuras 9.17 y 9.18 se define la clase Fecha para usarla en este ejemplo. En la línea 20 de la figura 9.19 usa la asignación predeterminada a nivel de miembros para asignar los miembros de datos del objeto Fecha llamado fecha1 a los correspondientes miembros de datos del objeto Fecha llamado fecha2. En este caso, el miembro mes de fecha1 se asigna al miembro mes de fecha2, el miembro dia de fecha1 se asigna al miembro dia de fecha2 y el miembro anio de fecha1 se asigna al miembro anio de fecha2. [Precaución: la asignación a nivel de miembros puede producir problemas graves al utilizarse con una clase cuyos miembros de datos contengan apunadores a la memoria asignada en forma dinámica; veremos estos problemas en el capítulo 11 y le mostraremos cómo lidiar con ellos]. Observe que el constructor de Fecha no contiene comprobación de errores; dejaremos esto para los ejercicios.

```

1 // Fig. 9.17: Fecha.h
2 // Declaración de la clase Fecha.
3 // Las funciones miembro se definen en Fecha.cpp
4
5 // evita múltiples inclusiones del archivo de encabezado
6 #ifndef FECHA_H
7 #define FECHA_H
8
9 // definición de la clase Fecha

```

Figura 9.17 | Declaración de la clase Fecha. (Parte 1 de 2).

```

10 class Fecha
11 {
12 public:
13     Fecha( int = 1, int = 1, int = 2000 ); // constructor predeterminado
14     void imprimir();
15 private:
16     int mes;
17     int dia;
18     int anio;
19 }; // fin de la clase Fecha
20
21 #endif

```

Figura 9.17 | Declaración de la clase Fecha. (Parte 2 de 2).

```

1 // Fig. 9.18: Fecha.cpp
2 // Definiciones de las funciones miembro de la clase Fecha.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Fecha.h" // incluye la definición de la clase Fecha de Fecha.h
8
9 // constructor de Fecha (deber realizar comprobación de rangos)
10 Fecha::Fecha( int m, int d, int y )
11 {
12     mes = m;
13     dia = d;
14     anio = y;
15 } // fin del constructor de Fecha
16
17 // imprime la Fecha en el formato mm/dd/aaaa
18 void Fecha::imprimir()
19 {
20     cout << mes << '/' << dia << '/' << anio;
21 } // fin de la función imprimir

```

Figura 9.18 | Definiciones de las funciones miembro de la clase Fecha.

```

1 // Fig. 9.19: fig09_19.cpp
2 // Demuestra que los objetos de una clase se pueden asignar
3 // unos a otros mediante la asignación predeterminada a nivel de bits.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "Fecha.h" // incluye la definición de la clase Fecha de Fecha.h
9
10 int main()
11 {
12     Fecha fecha1( 7, 4, 2004 );
13     Fecha fecha2; // el valor predeterminado de Fecha2 es 1/1/2000
14
15     cout << "fecha1 = ";
16     fecha1.imprimir();
17     cout << "\nfecha2 = ";
18     fecha2.imprimir();
19
20     fecha2 = fecha1; // asignación predeterminada a nivel de bits
21

```

Figura 9.19 | Asignación predeterminada a nivel de bits. (Parte I de 2).

```

22     cout << "\n\nDespues de la asignacion predeterminada a nivel de bits, fecha2 = ";
23     fecha2.imprimir();
24     cout << endl;
25     return 0;
26 } // fin de main

```

```

fecha1 = 7/4/2004
fecha2 = 1/1/2000

```

Despues de la asignacion predeterminada a nivel de bits, fecha2 = 7/4/2004

Figura 9.19 | Asignación predeterminada a nivel de bits. (Parte 2 de 2).

Los objetos se pueden pasar como argumentos para funciones y se pueden devolver de las funciones. Dichos procesos de pasar y devolver valores se llevan a cabo usando el paso por valor de manera predeterminada; se pasa o devuelve una copia del objeto. En tales casos, C++ crea un nuevo objeto y utiliza un **constructor de copia** para copiar los valores del objeto original en el nuevo objeto. Para cada clase, el compilador proporciona un constructor de copia predeterminado que copie cada miembro del objeto original en el miembro correspondiente del nuevo objeto. Al igual que la asignación a nivel de miembros, los constructores de copia pueden provocar problemas graves cuando se utilizan con una clase cuyos miembros de datos contienen apuntadores a memoria asignada en forma dinámica. En el capítulo 11 veremos cómo los programadores pueden definir constructores de copia personalizados, que copien de manera apropiada los objetos que contengan apuntadores a memoria asignada en forma dinámica.

Tip de rendimiento 9.3



Es bueno pasar un objeto por valor desde el punto de vista de la seguridad, ya que la función llamada no tiene acceso al objeto original en la función que la llamó, pero el paso por valor puede degradar el rendimiento cuando se hace una copia de un objeto extenso. Un objeto se puede pasar por referencia, ya sea pasando un apuntador o una referencia al objeto. El paso por referencia ofrece un buen rendimiento, pero es más débil desde el punto de vista de seguridad, ya que la función llamada recibe acceso al objeto original. El paso por referencia const es una alternativa segura y con buen rendimiento (esta se puede implementar con un parámetro de referencia const o con un parámetro de apuntador a datos const).

9.11 (Opcional) Ejemplo práctico de Ingeniería de Software: inicio de la programación de las clases del sistema ATM

En las secciones del Ejemplo práctico de Ingeniería de Software de los capítulos 1 al 7, introducimos los fundamentos de la orientación a objetos y desarrollamos un diseño orientado a objetos para nuestro sistema ATM. Anteriormente en este capítulo, vimos muchos de los detalles de programación con clases en C++. Ahora empezaremos a implementar nuestro diseño orientado a objetos en C++. Al final de esta sección, le mostraremos cómo convertir los diagramas de clases en archivos de encabezado de C++. En la sección final del Ejemplo práctico de Ingeniería de Software (sección 13.10), modificaremos los archivos de encabezado para incorporar el concepto orientado a objetos de herencia. En el apéndice G, Código del caso de estudio del ATM, presentamos la implementación completa del código en C++.

Visibilidad

Ahora vamos a aplicar especificadores de acceso a los miembros de nuestras clases. En el capítulo 3 presentamos los especificadores de acceso **public** y **private**. Los especificadores de acceso determinan la **visibilidad**, o accesibilidad, de los atributos y operaciones de un objeto para otros objetos. Antes de empezar a implementar nuestro diseño, debemos considerar cuáles atributos y operaciones de nuestras clases deben ser **public** y cuáles deben ser **private**.

En el capítulo 3 observamos que por lo general los miembros de datos deben ser **private**, y que las funciones miembro invocadas por los clientes de una clase dada deben ser **public**. Sin embargo, las funciones miembro que se llaman sólo por otras funciones miembro de la clase como “funciones utilitarias” deben ser generalmente **private**. UML emplea **marcadores de visibilidad** para modelar la visibilidad de los atributos y las operaciones. La visibilidad pública se indica mediante la colocación de un signo más (+) antes de una operación o atributo; un signo menos (-) indica una visibilidad privada. La figura 9.20 muestra nuestro diagrama de clases actualizado, en el cual se incluyen los marcadores de visibilidad. [Nota: no incluimos parámetros de operación en la figura 9.20. Esto es perfectamente normal. Agregar los marcadores de visibilidad no afecta a los parámetros que ya están modelados en los diagramas de clases de las figuras 6.36 a 6.39].

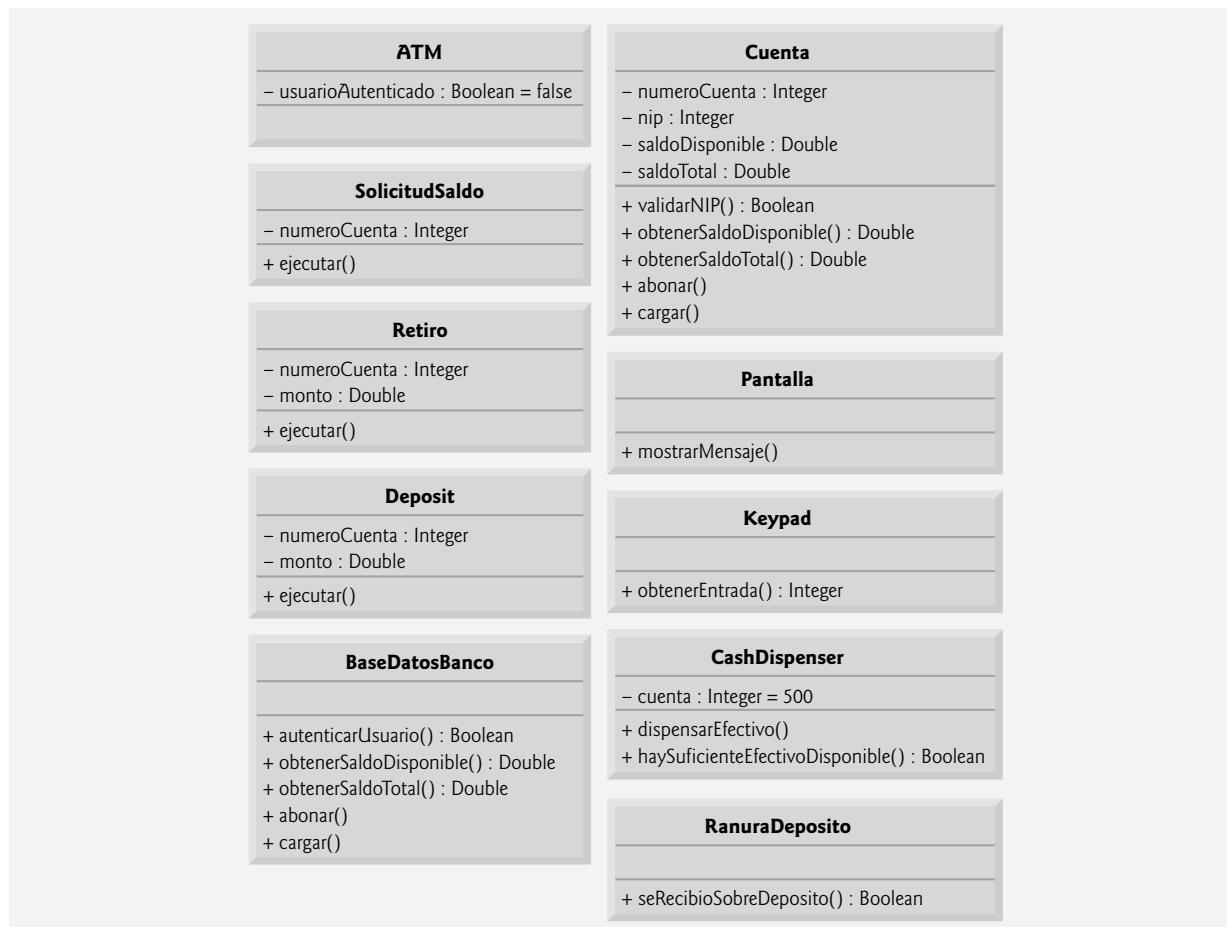


Figura 9.20 | Diagrama de clases con marcadores de visibilidad.

Navegabilidad

Antes de empezar a implementar nuestro diseño en C++, presentaremos una notación adicional de UML. El diagrama de clases de la figura 9.21 refina aún más las relaciones entre las clases del sistema ATM, al agregar flechas de navegabilidad a las líneas de asociación. Las **flechas de navegabilidad** (representadas como flechas con puntas delgadas en el diagrama de clases) indican en qué dirección puede recorrerse una asociación, y se basan en las colaboraciones modeladas en los diagramas de comunicación y de secuencia (vea la sección 7.12). Al implementar un sistema diseñado mediante el uso de UML, los programadores utilizan flechas de navegabilidad para ayudar a determinar cuáles objetos necesitan referencias o apunadores a otros objetos. Por ejemplo, la flecha de navegabilidad que apunta de la clase ATM a la clase `BaseDatosBanco` indica que podemos navegar de una a la otra, con lo cual se permite a la clase ATM invocar a las operaciones de `BaseDatosBanco`. No obstante, como la figura 9.21 no contiene una flecha de navegabilidad que apunte de la clase `BaseDatosBanco` a la clase ATM, la clase `BaseDatosBanco` no puede acceder a las operaciones de la clase ATM. Observe que las asociaciones en un diagrama de clases que tienen flechas de navegabilidad en ambos extremos, o que no tienen ninguna flecha de navegabilidad, indican una **navegabilidad bidireccional**: la navegación puede proceder en cualquier dirección a lo largo de la asociación.

Al igual que el diagrama de clases de la figura 3.23, el de la figura 9.21 omite las clases `SolicitudSaldo` y `Deposito` para simplificarlo. La navegabilidad de las asociaciones en las que participan estas dos clases se asemeja mucho a la navegabilidad de las asociaciones de la clase `Retiro`. En la sección 3.11 vimos que `SolicitudSaldo` tiene una asociación con la clase `Pantalla`. Podemos navegar de la clase `SolicitudSaldo` a la clase `Pantalla` a lo largo de esta asociación, pero no podemos navegar de la clase `Pantalla` a la clase `SolicitudSaldo`. Por ende, si modeláramos la clase `SolicitudSaldo` en la figura 9.21, colocaríamos una flecha de navegabilidad en el extremo de la clase `Pantalla` de esta asociación. Recuerde también que la clase `Deposito` se asocia con las clases `Pantalla`, `Keypad` y `RanuraDeposito`. Podemos navegar de la clase `Deposito` a cada una de estas clases, pero no al revés. Por lo tanto, podríamos colocar flechas de navegabilidad en

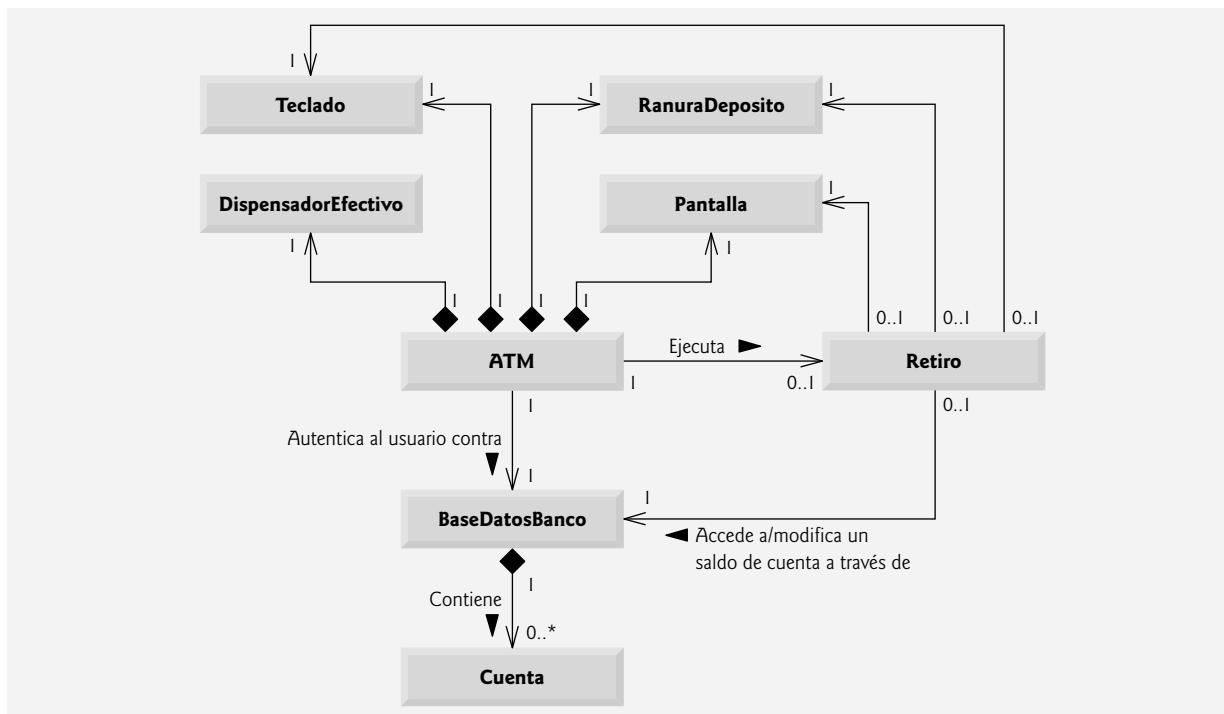


Figura 9.21 | Diagrama de clases con flechas de navegabilidad.

los extremos de las clases **Pantalla**, **Teclado** y **RanuraDeposito** de estas asociaciones. [Nota: modelaremos estas clases y asociaciones adicionales en nuestro diagrama de clases final en la sección 13.10, una vez que hayamos simplificado la estructura de nuestro sistema, al incorporar el concepto orientado a objetos de la herencia].

Implementación del sistema ATM a partir de su diseño de UML

Ahora estamos listos para empezar a implementar el sistema ATM. Primero convertiremos las clases de los diagramas de las figuras 9.20 y 9.21 en archivos de encabezado de C++. Este código representará el “esqueleto” del sistema. En el capítulo 13 modificaremos los archivos de encabezado para incorporar el concepto orientado a objetos de la herencia. En el apéndice G, presentaremos el código de C++ completo y funcional para nuestro modelo.

Como ejemplo, empezaremos a desarrollar el archivo de encabezado para la clase **Retiro**, a partir de nuestro diseño de la clase **Retiro** en la figura 9.20. Utilizaremos esta figura para determinar los atributos y operaciones de la clase. Usaremos el modelo de UML en la figura 9.21 para determinar las asociaciones entre las clases. Seguiremos estos cinco lineamientos para cada clase:

1. Use el nombre que se localiza en el primer compartimiento de una clase en un diagrama de clases para definir la clase en un archivo de encabezado (figura 9.22). Use las directivas del preprocesador `#ifndef`, `#define` y `#endif` para evitar que el archivo de encabezado se incluya más de una vez en un programa.
2. Use los atributos que se localizan en el segundo compartimiento de la clase para declarar los miembros de datos. Por ejemplo, los atributos `private numeroCuenta` y `moneda` de la clase **Retiro** producen el código de la figura 9.23.
3. Use las asociaciones descritas en el diagrama de clases para declarar las referencias (o apuntadores, según sea apropiado) a otros objetos. Por ejemplo, de acuerdo con la figura 9.21, **Retiro** puede acceder a un objeto de la clase **Pantalla**, a un objeto de la clase **Teclado**, a un objeto de la clase **DispensadorEfectivo** y a un objeto de la clase **BaseDatosBanco**. La clase **Retiro** debe mantener manejadores en estos objetos para enviarles mensajes, por lo que en las líneas 19 a 22 de la figura 9.24 se declaran cuatro referencias como miembros de datos `private`. En la implementación de la clase **Retiro** en el apéndice G, un constructor inicializa estos miembros de datos con referencias a objetos actuales. Observe que en las líneas 6 a 9 se incluyen (mediante `#include`) los archivos de encabezado que contienen las definiciones de las clases **Pantalla**, **Teclado**, **DispensadorEfectivo** y **BaseDatosBanco**, de manera que podamos declarar referencias a objetos de esas clases en las líneas 19 a 22.

```

1 // Fig. 9.22: Retiro.h
2 // Definición de la clase Retiro que representa una transacción de retiro
3 #ifndef RETIRO_H
4 #define RETIRO_H
5
6 class Retiro
7 {
8 }; // fin de la clase Retiro
9
10#endif // RETIRO_H

```

Figura 9.22 | Definición de la clase *Retiro* en envolturas del preprocesador.

```

1 // Fig. 9.23: Retiro.h
2 // Definición de la clase Retiro que representa una transacción de retiro
3 #ifndef RETIRO_H
4 #define RETIRO_H
5
6 class Retiro
7 {
8 private:
9     // atributos
10    int numeroCuenta; // cuenta de la que se van a retirar los fondos
11    double monto; // monto a retirar
12}; // fin de la clase Retiro
13
14#endif // RETIRO_H

```

Figura 9.23 | Agregar atributos al archivo de encabezado de la clase *Retiro*.

```

1 // Fig. 9.24: Retiro.h
2 // Definición de la clase Retiro que representa una transacción de retiro
3 #ifndef RETIRO_H
4 #define RETIRO_H
5
6 #include "Pantalla.h" // incluye la definición de la clase Pantalla
7 #include "Teclado.h" // incluye la definición de la clase Teclado
8 #include "DispensadorEfectivo.h" // incluye la definición de la clase DispensadorEfectivo
9 #include "BaseDatosBanco.h" // incluye la definición de la clase BaseDatosBanco
10
11 class Retiro
12 {
13 private:
14     // atributos
15     int numeroCuenta; // cuenta de la que se van a retirar los fondos
16     double monto; // monto a retirar
17
18     // referencias a los objetos asociados
19     Pantalla &pantalla; // referencia a la pantalla del ATM
20     Teclado &teclado; // referencia al teclado del ATM
21     DispensadorEfectivo &DispensadorEfectivo; // referencia al dispensador de efectivo del ATM
22     BaseDatosBanco &baseDatosBanco; // referencia a la información de la base de datos de cuentas
23}; // fin de la clase Retiro
24
25#endif // RETIRO_H

```

Figura 9.24 | Declaración de referencias a objetos asociados con la clase *Retiro*.

4. Resulta ser que al incluir los archivos de encabezado para las clases *Pantalla*, *Teclado*, *DispensadorEfectivo* y *BaseDatosBanco* en la figura 9.24 se hace más de lo necesario. La clase *Retiro* contiene *referencias* a objetos de estas clases; no contiene verdaderos objetos; y la cantidad de información requerida por el compilador para

crear una referencia difiere de la que se requiere para crear un objeto. Recuerde que para crear un objeto, el programador debe proporcionar al compilador una definición de la clase que introduzca el nombre de la clase como un nuevo tipo definido por el usuario, y que indique los miembros de datos que determinen cuánta memoria se requiere para almacenar el objeto. Sin embargo, al declarar una *referencia* (o apuntador) a un objeto sólo se requiere que el compilador sepa que la clase del objeto existe; no necesita conocer el tamaño del objeto. Cualquier referencia (o apuntador), sin importar la clase del objeto al que hace referencia, sólo contiene la dirección de memoria de dicho objeto. La cantidad de memoria requerida para almacenar una dirección es una característica física del hardware de la computadora. Por ende, el compilador conoce el tamaño de cualquier referencia (o apuntador). Como resultado, es innecesario incluir el archivo de encabezado completo de una clase cuando se declara sólo una referencia a un objeto de esa clase; necesitamos introducir el nombre de la clase, pero no necesitamos proporcionar la distribución de los datos del objeto, ya que el compilador conoce de antemano el tamaño de todas las referencias. C++ proporciona una instrucción conocida como **declaración anticipada**, la cual indica que un archivo de encabezado contiene referencias o apuntadores a una clase, pero ésta se encuentra fuera del archivo de encabezado. Podemos reemplazar las instrucciones `#include` en la definición de la clase **Retiro** de la figura 9.24 con declaraciones anticipadas de las clases **Pantalla**, **Teclado**, **DispensadorEfectivo** y **BaseDatosBanco** (líneas 6 a 9 en la figura 9.25). En vez de incluir (mediante `#include`) el archivo de encabezado completo para cada una de estas clases, sólo colocamos una declaración anticipada de cada clase en el archivo de encabezado para la clase **Retiro**. Observe que, si la clase **Retiro** contara con objetos actuales en vez de referencias (es decir, si se omitieran los signos & en las líneas 19 a 22), entonces tendríamos que incluir (mediante `#include`) los archivos de encabezado completos.

Observe que usar una declaración anticipada (en donde sea posible) en vez de incluir un archivo de encabezado completo nos ayuda a evitar un problema del preprocesador, conocido como **inclusión circular**. Este problema ocurre cuando el archivo de encabezado para la clase A incluye el archivo de encabezado para la clase B, y viceversa. Algunos preprocesadores no pueden resolver tales directivas `#include`, lo cual produce un error de compilación. Por ejemplo, si la clase A sólo utiliza una referencia a un objeto de la clase B, entonces la instrucción `#include` en el archivo de encabezado de la clase A se puede reemplazar por una declaración anticipada de la clase B, para evitar la inclusión circular.

5. Use las operaciones que se localizan en el tercer compartimiento de la figura 9.20 para escribir los prototipos de las funciones miembro de la clase. Si todavía no hemos especificado un tipo de valor de retorno para una

```
1 // Fig. 9.25: Retiro.h
2 // Definición de la clase Retiro que representa una transacción de Retiro
3 #ifndef RETIRO_H
4 #define RETIRO_H
5
6 class Pantalla; // declaración anticipada de la clase Pantalla
7 class Teclado; // declaración anticipada de la clase Teclado
8 class DispensadorEfectivo; // declaración anticipada de la clase DispensadorEfectivo
9 class BaseDatosBanco; // declaración anticipada de la clase BaseDatosBanco
10
11 class Retiro
12 {
13 private:
14     // atributos
15     int numeroCuenta; // cuenta de la que se van a retirar los fondos
16     double monto; // monto a retirar
17
18     // referencias a los objetos asociados
19     Pantalla &pantalla; // referencia a la pantalla del ATM
20     Teclado &teclado; // referencia al teclado del ATM
21     DispensadorEfectivo &dispensadorEfectivo; // referencia al dispensador de efectivo del ATM
22     BaseDatosBanco &baseDatosBanco; // referencia a la base de datos de información de cuentas
23 }; // fin de la clase Retiro
24
25 #endif // RETIRO_H
```

Figura 9.25 | Uso de declaraciones anticipadas en vez de directivas `#include`.

operación, declaramos la función miembro con el tipo de valor de retorno `void`. Consulte los diagramas de clases de las figuras 6.22 a 6.25 para declarar cualquier parámetro necesario. Por ejemplo, al agregar la operación `public ejecutar` en la clase `Retiro`, que tiene una lista de parámetros vacía, se produce el prototipo en la línea 15 de la figura 9.26. [Nota: codificaremos los cuerpos de las funciones miembro en archivos .cpp cuando implementemos el sistema ATM completo en el apéndice G].



Observación de Ingeniería de Software 9.12

Varias herramientas de modelado de UML pueden convertir los diseños basados en UML en código C++, con lo que se agiliza considerablemente el proceso de implementación. Para obtener más información sobre estos generadores de código “automáticos”, consulte los recursos de Internet y Web que se listan al final de la sección 2.8.

```

1 // Fig. 9.26: Retiro.h
2 // Definición de la clase Retiro que representa una transacción de Retiro
3 #ifndef RETIRO_H
4 #define RETIRO_H
5
6 class Pantalla; // declaración anticipada de la clase Pantalla
7 class Teclado; // declaración anticipada de la clase Teclado
8 class DispensadorEfectivo; // declaración anticipada de la clase DispensadorEfectivo
9 class BaseDatosBanco; // declaración anticipada de la clase BaseDatosBanco
10
11 class Retiro
12 {
13 public:
14     // operaciones
15     void ejecutar(); // realiza la transacción
16 private:
17     // atributos
18     int numeroCuenta; // cuenta de la que se van a retirar los fondos
19     double monto; // monto a retirar
20
21     // referencias a los objetos asociados
22     Pantalla &pantalla; // referencia a la pantalla del ATM
23     Teclado &teclado; // referencia al teclado del ATM
24     DispensadorEfectivo &dispensadorEfectivo; // referencia al dispensador de efectivo del ATM
25     BaseDatosBanco &baseDatosBanco; // referencia a la base de datos de información de cuentas
26 }; // fin de la clase Retiro
27
28 #endif // RETIRO_H

```

Figura 9.26 | Agregar operaciones al archivo de encabezado de la clase `Retiro`.

Esto concluye nuestra discusión sobre los fundamentos de la generación de archivos de encabezado de clases a partir de diagramas de UML. En la sección final del Ejemplo práctico de Ingeniería de Software (sección 13.10), demostraremos cómo modificar los archivos de encabezado para incorporar el concepto orientado a objetos de la herencia.

Ejercicios de autoevaluación del ejemplo práctico de Ingeniería de Software

- 9.1 Indique si el siguiente enunciado es *verdadero* o *falso*, y si es *falso*, explique por qué: si un atributo de una clase se marca con un signo menos (-) en un diagrama de clases, el atributo no es directamente accesible fuera de la clase.
- 9.2 En la figura 9.21, la asociación entre los objetos ATM y Pantalla indica:
 - a) que podemos navegar de la Pantalla al ATM.
 - b) que podemos navegar del ATM a la Pantalla.
 - c) (a) y (b); la asociación es bidireccional.
 - d) Ninguna de las anteriores.
- 9.3 Escriba código de C++ para empezar a implementar el diseño para la clase Cuenta.

Respuestas a los ejercicios de autoevaluación del Ejemplo práctico de Ingeniería de Software

9.1 Verdadero. El signo menos (–) indica visibilidad privada. Mencionamos la “amistad” como una excepción a la visibilidad privada. En el capítulo 10 hablaremos sobre la amistad.

9.2 b.

9.3 El diseño para la clase Cuenta produce el archivo de encabezado de la figura 9.27.

```

1 // Fig. 9.27: Cuenta.h
2 // Definición de la clase Cuenta. Representa una cuenta bancaria.
3 #ifndef CUENTA_H
4 #define CUENTA_H
5
6 class Cuenta
7 {
8 public:
9     bool validarNIP( int ); // ¿es correcto el NIP especificado por el usuario?
10    double obtenerSaldoDisponible(); // devuelve el saldo disponible
11    double obtenerSaldoTotal(); // devuelve el saldo total
12    void abonar( double ); // suma una cantidad a la Cuenta
13    void cargar( double ); // resta un monto a la Cuenta
14 private:
15     int numeroCuenta; // número de cuenta
16     int nip; // NIP para autenticar
17     double saldoDisponible; // fondos disponibles para retirar
18     double saldoTotal; // fondos disponibles + fondos esperando verificación
19 }; // fin de la clase Cuenta
20
21 #endif // CUENTA_H

```

Figura 9.27 | Archivo de encabezado de la clase Cuenta, con base en las figuras 9.20 y 9.21.

9.12 Repaso

En este capítulo profundizamos nuestro conocimiento acerca de las clases, usando un enriquecedor ejemplo práctico con la clase Tiempo para presentar varias nuevas características de las clases. Vimos que las funciones miembro son por lo general más cortas que las funciones globales, ya que las funciones miembro pueden acceder directamente a los miembros de datos de un objeto, de manera que las funciones miembro puedan recibir menos argumentos que las funciones en los lenguajes de programación por procedimientos. Usted aprendió a usar el operador flecha para acceder a los miembros de un objeto a través de un apuntador del tipo de la clase del objeto.

También aprendió que las funciones miembro tienen alcance de clase; es decir, el nombre de la función miembro sólo es conocido para los demás miembros de la clase, a menos que se haga referencia a la función miembro a través de un objeto de la clase, una referencia a un objeto de la clase, un apuntador a un objeto de la clase o el operador de resolución de ámbito binario. También hablamos sobre las funciones de acceso (que se utilizan comúnmente para obtener los valores de los miembros de datos, o para evaluar la veracidad o falsedad de las condiciones) y las funciones utilitarias (funciones miembro private que soportan la operación de las funciones miembro public de la clase).

Aprendió además que un constructor puede especificar argumentos predeterminados que le permitan ser llamado en una variedad de formas. También aprendió que cualquier constructor que se pueda llamar sin argumentos es un constructor predeterminado, y que puede haber por lo menos un constructor predeterminado por clase. Hablamos sobre los destructores y su propósito de realizar tareas de mantenimiento de terminación en un objeto de una clase, antes de destruir ese objeto. Demostramos también el orden en el que se llaman los constructores y destructores de un objeto.

Demostramos los problemas que pueden ocurrir cuando una función miembro hace referencia a un miembro de datos private, lo cual quebranta el encapsulamiento de la clase. Mostramos además que los objetos del mismo tipo se pueden asignar entre sí, usando la asignación predeterminada a nivel de miembros. Por último, hablamos sobre los beneficios de usar bibliotecas de clases para mejorar la velocidad con la que se puede crear el código, e incrementar la calidad del software.

En el capítulo 10 presentaremos características adicionales sobre las clases. Demostraremos cómo se puede utilizar const para indicar que una función miembro no modifica a un objeto de una clase. Aprenderá a crear clases mediante la composición: la herramienta que permite a una clase contener objetos de otras clases como miembros. Le mostraremos

cómo una clase puede permitir lo que se conoce como funciones “amigas” para acceder a los miembros no `public` de la clase. También mostraremos cómo las funciones miembro no `static` de una clase pueden utilizar un apuntador especial llamado `this` para acceder a los miembros de un objeto. Después aprenderá a usar los operadores `new` y `delete` de C++, que permiten a los programadores obtener y liberar memoria según sea necesario, durante la ejecución de un programa.

Resumen

Sección 9.2 Ejemplo práctico con la clase Tiempo

- Las directivas del preprocesador `#ifndef` (que significa “si no está definido”) y `#endif` se utilizan para evitar múltiples inclusiones de un archivo de encabezado. Si el código entre estas directivas no se ha incluido antes en una aplicación, `#define` define un nombre que se puede utilizar para evitar futuras inclusiones, y el código se incluye en el archivo de código fuente.
- Los miembros de datos de una clase no se pueden inicializar en donde se declaran en el cuerpo de la clase (excepto para los miembros de datos `static const` de tipos integrales o `enum` de una clase, como veremos en el capítulo 10). Se recomienda enérgicamente que estos miembros de datos se inicialicen mediante el constructor de la clase (ya que no hay inicialización predeterminada para los miembros de datos de tipos fundamentales).
- El manipulador de flujo `setfill` especifica el carácter de relleno a mostrar cuando se imprime un entero en un campo que sea más ancho que el número de dígitos en el valor.
- De manera predeterminada, los caracteres de relleno aparecen antes de los dígitos en el número.
- El manipulador de flujo `setfill` es una opción “pegajosa”, lo cual significa que una vez que se establece el carácter de relleno, se aplica para todos los campos subsiguientes que se impriman.
- Aun y cuando una función miembro declarada en una definición de clase pueda definirse fuera de la definición de esa clase (y “enlazarse” a la clase a través del operador de resolución de ámbito binario), esa función miembro sigue dentro del alcance de esa clase; es decir, su nombre es conocido sólo para los demás miembros de la clase, a menos que se haga referencia a la función a través de un objeto de la clase, de una referencia a un objeto de la clase o de un apuntador a un objeto de la clase.
- Si una función miembro se define en el cuerpo de una definición de clase, el compilador de C++ trata de poner en línea las llamadas a la función miembro.
- Las clases no tienen que crearse “desde cero”. En vez de ello, pueden incluir objetos de otras clases como miembros, o pueden derivarse de otras clases que proporcionen atributos y comportamientos que puedan usar las nuevas clases. Al proceso de incluir objetos de una clase como miembros de otras clases se le conoce como composición.

Sección 9.3 Alcance de las clases y acceso a los miembros de una clase

- Los miembros de datos de una clase y las funciones miembro pertenecen al alcance de esa clase.
- Las funciones que no son miembro se definen en alcance de archivo.
- Dentro del alcance de una clase, los miembros de ésta son inmediatamente accesibles para todas las funciones miembro de esa clase, y se puede hacer referencia a ellos por su nombre.
- Fuera del alcance de una clase, los miembros de ésta se refieren a través de uno de los manejadores en un objeto: el nombre de un objeto, una referencia a un objeto o un apuntador a un objeto.
- Las funciones miembro de una clase se pueden sobrecargar, pero sólo por otras funciones miembro de esa clase.
- Para sobrecargar una función miembro, se debe proporcionar en la definición de la clase un prototipo para cada versión de la función sobrecargada, además de una definición separada para cada versión de la función.
- Las variables que se declaran en una función miembro tienen alcance de bloque, y sólo esa función las conoce.
- Si una función miembro define una variable con el mismo nombre que una variable con alcance de clase, esta última se oculta debido a la variable con alcance de bloque en el alcance de bloque.
- Al operador punto (.) de selección de miembros se le antepone el nombre de un objeto o una referencia al objeto para acceder a los miembros `public` del objeto.
- Al operador flecha (->) de selección de miembros se le antepone un apuntador a un objeto para acceder a los miembros `public` de ese objeto.

Sección 9.4 Separar la interfaz de la implementación

- Los archivos de encabezado contienen ciertas porciones de la implementación y sugerencias sobre otras. Por ejemplo, las funciones miembro en línea necesitan estar en un archivo de encabezado, de manera que cuando el compilador compile un cliente, éste pueda incluir la definición de la función `inline` en el lugar adecuado.
- Los miembros `private` de una clase se listan en la definición de la clase en el archivo de encabezado, por lo cual estos miembros están visibles para los clientes, aun cuando éstos tal vez no tengan acceso a los miembros `private`.

Sección 9.5 Funciones de acceso y funciones utilitarias

- Una función utilitaria (también conocida como función ayudante) es una función miembro `private` que soporta la operación de las funciones miembro `public` de la clase. Las funciones utilitarias no están diseñadas para que las utilicen los clientes de una clase (pero las amigas de una clase las pueden utilizar).

Sección 9.6 Ejemplo práctico de la clase `Tiempo`: constructores con argumentos predeterminados

- Al igual que otras funciones, los constructores pueden especificar argumentos predeterminados.

Sección 9.7 Destructores

- El destructor de una clase se llama de manera implícita cuando se destruye un objeto de la clase.
- El nombre del destructor para una clase es el carácter tilde (~) seguido del nombre de la clase.
- Un destructor no libera el almacenamiento de un objeto; realiza tareas de mantenimiento de terminación antes de que el sistema reclame la memoria de un objeto, de manera que ésta se pueda reutilizar para contener nuevos objetos.
- Un destructor no recibe parámetros y no devuelve valores. Una clase sólo puede tener un destructor.
- Si no se proporciona un destructor de manera explícita, el compilador crea un destructor “vacío”, de manera que cada clase tiene sólo un destructor.

Sección 9.8 Cuándo se hacen llamadas a los constructores y destructores

- El orden en el que se llaman los constructores y destructores depende del orden en el que la ejecución entra y sale de los alcances en los que se instancian los objetos.
- Por lo general, las llamadas al destructor se realizan en orden inverso a las llamadas correspondientes al constructor, pero las clases de almacenamiento de los objetos pueden alterar el orden en el que se llama a los destructores.

Sección 9.9 Ejemplo práctico con la clase `Tiempo`: una trampa sutil (devolver una referencia a un miembro de datos `private`)

- Una referencia a un objeto es un alias para el nombre del objeto y, por ende, se puede usar del lado izquierdo de una instrucción de asignación. En este contexto, la referencia se convierte en un *lvalue* perfectamente aceptable que puede recibir un valor. Una manera de usar esta herramienta (¡por desgracia!) es hacer que una función miembro `public` de una clase devuelva una referencia a un miembro de datos `private` de esa clase. Si la función devuelve una referencia `const`, entonces la referencia no se puede utilizar como *lvalue* modificable.

Sección 9.10 Asignación predeterminada a nivel de miembros

- El operador de asignación (=) se puede utilizar para asignar un objeto a otro del mismo tipo. De manera predeterminada, dicha asignación se realiza mediante la asignación a nivel de miembros; cada miembro del objeto a la derecha del operador de asignación se asigna de manera individual al mismo miembro en el objeto a la izquierda del operador de asignación.
- Los objetos se pueden pasar como argumentos de función y se pueden devolver de las funciones. Los procesos de paso y devolución se llevan a cabo mediante el paso por valor de manera predeterminada; se pasa o devuelve una copia del objeto. En dichos casos, C++ crea un nuevo objeto y utiliza un constructor de copia para copiar los valores del objeto original al nuevo objeto.
- Para cada clase, el compilador proporciona un constructor de copia predeterminado que copia cada miembro del objeto original en el miembro correspondiente del nuevo objeto.

Terminología

<code>abort</code> , función	declaración anticipada
agregación	<code>#define</code> , directiva del preprocesador
alcance de archivo	derivar una clase de otra
alcance de clase	destructor
argumentos predeterminados con constructores	el objeto se sale del alcance
asignación a nivel de miembros	<code>#endif</code> , directiva del preprocesador
asignación de objetos de clases	envoltura del preprocesador
asignación predeterminada a nivel de miembros	<code>exit</code> , función
bibliotecas de clases	función ayudante
carácter de relleno	función de acceso
carácter tilde (~) en el nombre de un destructor	función miembro sobrecargada
código reentrant	función predicado
componentes reutilizables	herencia
composición	<code>#ifndef</code> , directiva del preprocesador
constructor de copia	inizializador
constructor sobrecargado	manejador de apuntador en un objeto

manejador de nombre en un objeto	orden en el que se llama a los constructores y destructores
manejador de objeto	pasar un objeto por valor
manejador de referencia en un objeto	procedimiento puro
manejador en un objeto	<code>setfill</code> , manipulador de flujo parametrizado
manejador implícito en un objeto	tareas de mantenimiento de terminación
operador flecha (\rightarrow) de selección de miembros	

Ejercicios de autoevaluación

9.1 Complete los siguientes enunciados:

- Los miembros de una clase se utilizan mediante el operador _____ en conjunción con el nombre de un objeto (o referencia a un objeto) de la clase, o a través del operador _____ en conjunción con un apuntador a un objeto de la clase.
- Los miembros de una clase que se especifican como _____ están accesibles sólo para las funciones miembro de la clase y sus funciones amigas.
- Los miembros de una clase que se especifican como _____ están accesibles en cualquier parte en la que un objeto de la clase se encuentre dentro del alcance.
- _____ se puede utilizar para asignar un objeto de una clase a otro objeto de la misma clase.

9.2 Busque el (los) error(es) en cada uno de los siguientes incisos, y explique cómo corregirlo(s).

- Suponga que se declara el siguiente prototipo en la clase `Tiempo`:

```
void ~Tiempo( int );
```

- A continuación se muestra una definición parcial de la clase `Tiempo`:

```
class Tiempo
{
public:
    // prototipos de funciones
private:
    int hora = 0;
    int minuto = 0;
    int segundo = 0;
}; // fin de la clase Tiempo
```

- Suponga que se declara el siguiente prototipo en la clase `Empleado`:

```
int Empleado( const char *, const char * );
```

Respuestas a los ejercicios de autoevaluación

9.1 a) punto (.), flecha (\rightarrow). b) `private`. c) `public`. d) La asignación predeterminada a nivel de miembros (realizada por el operador de asignación).

9.2 a) *Error*: los destructores no pueden devolver valores (o incluso especificar un tipo de valor de retorno) ni recibir argumentos.

Corrección: elimine el tipo de valor de retorno `void` y el parámetro `int` de la declaración.

b) *Error*: los miembros no se pueden inicializar de manera explícita en la definición de la clase.

Corrección: elimine la inicialización explícita de la definición de la clase e inicialice los miembros de datos en un constructor.

c) *Error*: los constructores no pueden devolver valores.

Corrección: elimine el tipo de valor de retorno `int` de la declaración.

Ejercicios

9.3 ¿Cuál es el propósito del operador de resolución de ámbito?

9.4 (*Mejora de la clase Tiempo*) Proporcione un constructor que sea capaz de usar el tiempo actual de la función `time()` (declarada en el encabezado `<ctime>` de la Biblioteca estándar de C++) para inicializar un objeto de la clase `Tiempo`.

9.5 (*Clase Complejo*) Cree una clase llamada `Complejo` para realizar operaciones aritméticas con números complejos. Escriba un programa para evaluar su clase.

Los números complejos tienen la forma

```
parteReal + parteImaginaria * i
```

donde i es

$$\sqrt{-1}$$

Use variables `double` para representar los datos `private` de la clase. Proporcione un constructor que permita a un objeto de la clase inicializarse al momento de ser declarado. El constructor debe contener valores predeterminados en caso de que no se proporcionen inicializadores. Proporcione funciones miembro `public` que realicen las siguientes tareas:

- Sumar dos números `Complejo`: las partes reales se suman entre sí y las partes imaginarias se suman entre sí.
- Restar dos números `Complejo`: la parte real del operando derecho se resta de la parte real del operando izquierdo, y la parte imaginaria del operando derecho se resta de la parte imaginaria del operando izquierdo.
- Imprimir números `Complejo` de la forma (a, b) , en donde a es la parte real y b es la parte imaginaria.

9.6 (Clase Racional) Cree una clase llamada `Racional` para realizar operaciones aritméticas con fracciones. Escriba un programa para evaluar su clase.

Use variables enteras para representar los datos `private` de la clase: el `numerador` y el `denominador`. Proporcione un constructor que permita a un objeto de esta clase inicializarse cuando se declare. El constructor debe contener valores predeterminados, en caso de que no se proporcionen inicializadores, y debe almacenar la fracción en forma reducida. Por ejemplo, la fracción

$$\frac{2}{4}$$

se almacenaría en el objeto como 1 en el `numerador` y 2 en el `denominador`. Proporcione funciones miembro `public` que realicen cada una de las siguientes tareas:

- Sumar dos números `Racional`. El resultado debe almacenarse en forma reducida.
- Restar dos números `Racional`. El resultado debe almacenarse en forma reducida.
- Multiplicar dos números `Racional`. El resultado debe almacenarse en forma reducida.
- Dividir dos números `Racional`. El resultado debe almacenarse en forma reducida.
- Imprimir números `Racional` en la forma a/b , en donde a es el numerador y b es el denominador.
- Imprimir números `Racional` en formato de punto flotante.

9.7 (Mejora a la clase Tiempo) Modifique la clase `Tiempo` de las figuras 9.8 y 9.9 para incluir una función `tictac`, que incremente el tiempo almacenado en un objeto `Tiempo` por un segundo. El objeto `Tiempo` debe permanecer siempre en un estado consistente. Escriba un programa para probar la función miembro `tictac` en un ciclo que imprima la hora en formato estándar durante cada iteración del ciclo, para ilustrar que la función miembro funcione correctamente. Asegúrese de evaluar los siguientes casos:

- Incrementar el minuto, de manera que cambie al siguiente minuto.
- Incrementar la hora, de manera que cambie a la siguiente hora.
- Incrementar el tiempo de manera que cambie al siguiente día (por ejemplo, de 11:59:59 PM a 12:00:00 AM).

9.8 (Mejora a la clase Fecha) Modifique la clase `Fecha` de las figuras 9.17 y 9.18 para realizar la comprobación de errores en los valores inicializadores para los miembros de datos `mes`, `dia` y `anio`. Además, proporcione una función llamada `siguienteDia` para incrementar el `dia` en uno. El objeto `Fecha` siempre deberá permanecer en un estado consistente. Escriba un programa que evalúe la función `siguienteDia` en un ciclo que imprima la fecha durante cada iteración del ciclo, para mostrar que `siguienteDia` funciona correctamente. Asegúrese de evaluar los siguientes casos:

- Incrementar la fecha de manera que cambie al siguiente mes.
- Incrementar la fecha de manera que cambie al siguiente año.

9.9 (Combinar la clase Tiempo y la clase Fecha) Combine la clase `Tiempo` modificada del ejercicio 9.7 y la clase `Fecha` modificada del ejercicio 9.8 en una sola clase llamada `FechaYHora`. (En el capítulo 12 hablaremos sobre la herencia, que nos permitirá realizar esta tarea rápidamente sin tener que modificar las definiciones de las clases existentes). Modifique la función `tictac` para que llame a la función `siguienteDia` si el tiempo se incrementa y cambia al siguiente día. Modifique las funciones `imprimirEstandar` e `imprimirUniversal` para imprimir la fecha y hora. Escriba un programa para evaluar la nueva clase `FechaYHora`. En específico, incremente el tiempo para que cambie al siguiente día.

9.10 (Devolver indicadores de error de las funciones set de la clase Tiempo) Modifique las funciones `set` en la clase `Tiempo` de las figuras 9.8 y 9.9 para que devuelvan valores de error apropiados si hay un intento de `set` un miembro de datos de un objeto de la clase `Tiempo` en un valor inválido. Escriba un programa que evalúe su nueva versión de la clase `Tiempo`. Muestre mensajes de error cuando las funciones `set` devuelvan los valores de error.

9.11 (Clase Rectángulo) Cree una clase `Rectangulo` con los atributos `longitud` y `anchura`, cada una de las cuales tiene un valor predeterminado de 1. Proporcione funciones miembro que calculen el `perímetro` y el `area` del rectángulo. Además, proporcione funciones `set` y `get` para los atributos `longitud` y `anchura`. Las funciones `set` deben verificar que `longitud` y `anchura` sean números de punto flotante mayores que 0.0 y menores que 20.0.

9.12 (Clase Rectángulo mejorada) Cree una clase `Rectangulo` más sofisticada que la del ejercicio 9.11. Esta clase sólo almacena las coordenadas Cartesianas de las cuatro esquinas del rectángulo. El constructor llama a una función `set` que acepta cuatro conjuntos de coordenadas y verifica que cada una de éstas se encuentre en el primer cuadrante, en donde ninguna coordenada `x` o `e individual` debe ser mayor que 20.0. La función `set` también verifica que las coordenadas suministradas especifiquen, de hecho, un rectángulo. Proporcione funciones miembro que calculen los valores de `longitud`, `anchura`, `perímetro` y `area`. La longitud es el valor mayor de las dos dimensiones. Incluya una función predicado llamada `cuadrado` que determine si el rectángulo es un cuadrado.

9.13 (Clase Rectángulo mejorada) Modifique la clase `Rectangulo` del ejercicio 9.12 para que incluya una función `dibujar` que muestre el rectángulo dentro de un cuadro de 25 por 25, que encierre la porción correspondiente al primer cuadrante en el que reside el rectángulo. Incluya una función `setCaracterRelleno` para especificar el carácter a partir del cual se dibujará el rectángulo. Incluya una función `setCaracterPerímetro` para especificar el carácter que se utilizará para dibujar el borde del rectángulo. Si se siente ambicioso, tal vez quiera incluir funciones para escalar el tamaño del rectángulo, girarlo y desplazarlo alrededor del interior de la porción designada del primer cuadrante.

9.14 (Clase EnteroEnorme) Cree una clase llamada `EnteroEnorme` que utilice un arreglo de 40 elementos de dígitos, para almacenar enteros de hasta 40 dígitos cada uno. Proporcione las funciones miembro `recibir`, `imprimir`, `sumar` y `restar`. Para comparar objetos `EnteroEnorme`, proporcione las funciones `esIgualA`, `noEsIgualA`, `esMayorQue`, `esMenorQue`, `esMayori0IgualQue` y `esMenor0IgualQue`; cada una de éstas es una función “predicado” que simplemente devuelve `true` si la relación se mantiene entre los dos objetos `EnteroEnorme` y devuelve `false` si la relación no se mantiene. Además, proporcione una función predicado `esCero`. Si se siente ambicioso, proporcione las funciones miembro `multiplicar`, `dividir` y `modulo`.

9.15 (Clase TresEnRaya) Cree una clase llamada `TresEnRaya` que le permita escribir un programa completo para jugar al “tres en raya” (o tres en línea). La clase debe contener como datos `private` un arreglo bidimensional de enteros, con un tamaño de 3 por 3. El constructor debe inicializar el tablero vacío con ceros. Permita dos jugadores humanos. Siempre que el primer jugador realice un movimiento, coloque un 1 en el cuadro especificado. Coloque un 2 siempre que el segundo jugador realice un movimiento. Cada movimiento debe hacerse en un cuadro vacío. Después de cada movimiento, determine si el juego se ha ganado o si hay un empate. Si desea hacer algo más, modifique su programa de manera que la computadora realice los movimientos para uno de los jugadores. Además, permita que el jugador especifique si desea el primer o segundo turno. Si se siente todavía más motivado, desarrolle un programa que reproduzca un juego de tres en raya tridimensional, en un tablero de 4 por 4 por 4. [Nota: ¡Éste es un proyecto extremadamente retador, que podría requerir de muchas semanas de esfuerzo!].



¿Pero qué es lo que, para servir a nuestros fines privados, prohíbe que nuestros amigos hagan trampa?

—Charles Churchill

En vez de esta absurda división de sexos, deberían clasificar a las personas como estáticas y dinámicas.

—Evelyn Waugh

No tengas amigos diferentes a ti mismo.

—Confucio

Clases: un análisis más detallado, parte 2

OBJETIVOS

En este capítulo aprenderá a:

- Especificar objetos `const` (constantes) y funciones miembro `const`.
- Crear objetos compuestos de otros objetos.
- Usar funciones `friend` y clases `friend`.
- Usar el apuntador `this`.
- Crear y destruir objetos en forma dinámica mediante los operadores `new` y `delete`, respectivamente.
- Usar datos miembros y funciones miembro `static`.
- El concepto de una clase contenedora.
- La noción de las clases iteradoras que recorren los elementos de las clases contenedoras.
- Usar clases proxy para ocultar los detalles de implementación de los clientes de una clase.

- 10.1** Introducción
- 10.2** Objetos `const` (constantes) y funciones miembro `const`
- 10.3** Composición: objetos como miembros de clases
- 10.4** Funciones `friend` y clases `friend`
- 10.5** Uso del apuntador `this`
- 10.6** Administración dinámica de memoria con los operadores `new` y `delete`
- 10.7** Miembros de clase `static`
- 10.8** Abstracción de datos y ocultamiento de información
 - 10.8.1** Ejemplo: tipo de datos abstracto arreglo
 - 10.8.2** Ejemplo: tipo de datos abstracto cadena
 - 10.8.3** Ejemplo: tipo de datos abstracto cola
- 10.9** Clases contenedoras e iteradores
- 10.10** Clases proxy
- 10.11** Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)

10.1 Introducción

En este capítulo, continuaremos nuestro estudio sobre las clases y la abstracción de datos con varios temas más avanzados. Usaremos objetos `const` y funciones miembro `const` para evitar modificaciones de objetos y hacer valer el principio del menor privilegio. Hablaremos sobre la composición: una forma de reutilización en la que una clase puede tener objetos de otras clases como miembros. A continuación, presentaremos la amistad, que permite a un diseñador de clases especificar funciones no miembro que puedan acceder a los miembros no `public` de una clase; una técnica que se utiliza comúnmente en la sobrecarga de operadores (capítulo 11) por cuestiones de rendimiento. Hablaremos sobre un apuntador especial (llamado `this`), que es un argumento implícito para cada una de las funciones miembro no `static` de una clase. Permite a esas funciones miembro acceder correctamente a los datos miembro y otras funciones miembro no `static` del objeto. Después hablaremos sobre la administración dinámica de memoria y mostraremos cómo crear y destruir objetos de manera dinámica, mediante los operadores `new` y `delete`. A continuación, motivaremos la necesidad de miembros de clase `static` y mostraremos cómo usar datos miembro y funciones miembro `static` en sus propias clases. Por último, mostraremos cómo crear una clase proxy para ocultar los detalles de implementación de una clase (incluyendo sus datos miembro `private`) de los clientes de la clase.

Recuerde que en el capítulo 3 se introdujo la clase `string` de la Biblioteca Estándar de C++ para representar cadenas como objetos de clase completos. Sin embargo, en este capítulo usaremos las cadenas basadas en apuntador que presentamos en el capítulo 8 para ayudar al lector a dominar el tema de los apuntadores y prepararse para el mundo profesional, en el que verá una gran cantidad de código heredado de C que se ha implementado durante las últimas décadas. Por ende, se familiarizará con los dos métodos más prevalecientes para crear y manipular cadenas en C++.

10.2 Objetos `const` (constantes) y funciones miembro `const`

Hemos enfatizado el principio del menor privilegio como uno de los principios más fundamentales de buena ingeniería de software. Veamos cómo se aplica este principio a los objetos.

Algunos objetos necesitan ser modificables y otros no. Podemos usar la palabra clave `const` para especificar que un objeto no es modificable, y que cualquier intento por modificar el objeto debe producir un error de compilación. La instrucción

```
const Tiempo mediodía( 12, 0, 0 );
```

declara un objeto `const` `mediodia` de la clase `Tiempo` y lo inicializa con las 12 del mediodía.



Observación de Ingeniería de Software 10.1

Declarar un objeto como `const` ayuda a hacer valer el principio del menor privilegio. Los intentos de modificar el objeto se atrapan en tiempo de compilación, en vez de producir errores en tiempo de ejecución. El uso de `const` de manera apropiada es crucial para el diseño apropiado de las clases, de los programas y para la codificación.



Tip de rendimiento 10.1

Declarar variables y objetos const puede mejorar el rendimiento; los compiladores optimizadores sofisticados de la actualidad pueden realizar ciertas optimizaciones sobre constantes que no se pueden llevar a cabo sobre variables.

C++ no permite llamadas a funciones miembro para objetos `const`, a menos que las mismas funciones miembro también se declaren como `const`. Esto se aplica incluso para las funciones miembro `get` que no modifican el objeto. Además, el compilador no permite que las funciones miembro declaradas como `const` modifiquen el objeto.

Una función se especifica como `const` tanto en su prototipo (figura 10.1; líneas 19 a 24) como en su definición (figura 10.2; líneas 47, 53, 59 y 65), para lo cual se inserta la palabra clave `const` después de la lista de parámetros de la función `y`, en el caso de la definición de la función, antes de la llave izquierda que empieza el cuerpo de la función.



Error común de programación 10.1

Definir como const una función miembro que modifica a datos miembro de un objeto es un error de compilación.



Error común de programación 10.2

Definir como const una función miembro que llama a una función miembro no const de la clase en la misma instancia de ésta, es un error de compilación.



Error común de programación 10.3

Invocar a una función miembro no const en un objeto const es un error de compilación.



Observación de Ingeniería de Software 10.2

Una función miembro const se puede sobrecargar con una versión no const. El compilador selecciona cuál función miembro sobrecargada puede utilizar, con base en el objeto en el que se invoca la función. Si el objeto es const, el compilador usa la versión const. Si el objeto no es const, el compilador usa la versión no const.

Hay un interesante problema que surge para los constructores y destructores, cada uno de los cuales por lo general modifica objetos. La declaración `const` no se permite para los constructores y destructores. Un constructor debe tener la capacidad de modificar un objeto, para que éste se pueda inicializar de manera apropiada. Un destructor debe tener la capacidad de realizar sus tareas de mantenimiento de terminación antes de que el sistema reclame la memoria ocupada por el objeto.



Error común de programación 10.4

Tratar de declarar un constructor o un destructor como const es un error de compilación.

Definición y uso de funciones miembro const

El programa de las figuras 10.1 a 10.3 modifica la clase `Tiempo` de las figuras 9.8 a 9.9, para lo cual hace que sus funciones `get` y su función `imprimirUniversal` sean `const`. En el archivo de encabezado `Tiempo.h` (figura 10.1), en las líneas 19 a 21 y 24 se incluye ahora la palabra clave `const` después de la lista de parámetros de cada función. La definición correspondiente de cada función en la figura 10.2 (líneas 47, 53, 59 y 65, respectivamente) también especifica la palabra clave `const` después de la lista de parámetros de cada función.

```

1 // Fig. 10.1: Tiempo.h
2 // Definición de la clase Tiempo con funciones miembro const.
3 // Las funciones miembro se definen en Tiempo.cpp.
4 #ifndef TIEMPO_H
5 #define TIEMPO_H
6
7 class Tiempo
8 {
9 public:
10    Tiempo( int = 0, int = 0, int = 0 ); // constructor predeterminado
11

```

Figura 10.1 | Definición de la clase `Tiempo` con funciones miembro `const`. (Parte I de 2).

```

12 // funciones "set"
13 void setTiempo( int, int, int ); // establece el tiempo
14 void setHora( int ); // establece la hora
15 void setMinuto( int ); // establece el minuto
16 void setSegundo( int ); // establece el segundo
17
18 // funciones "get" (por lo general se declaran const)
19 int getHora() const; // devuelve la hora
20 int getMinuto() const; // devuelve el minuto
21 int getSegundo() const; // devuelve el segundo
22
23 // funciones para imprimir (por lo general se declaran const)
24 void imprimirUniversal() const; // imprime el tiempo universal
25 void imprimirEstandar(); // imprime el tiempo estándar(debe ser const)
26 private:
27     int hora; // 0 - 23 (formato de reloj de 24 horas)
28     int minuto; // 0 - 59
29     int segundo; // 0 - 59
30 }; // fin de la clase Tiempo
31
32 #endif

```

Figura 10.1 | Definición de la clase **Tiempo** con funciones miembro **const**. (Parte 2 de 2).

```

1 // Fig. 10.2: Tiempo.cpp
2 // Definiciones de las funciones miembro de la clase Tiempo.
3 #include <iostream>
4 using std::cout;
5
6 #include <iomanip>
7 using std::setfill;
8 using std::setw;
9
10 #include "Tiempo.h" // incluye la definición de la clase Tiempo
11
12 // función del constructor para inicializar los datos privados;
13 // llama a la función miembro setTiempo para establecer las variables;
14 // los valores predeterminados son 0 (vea la definición de la clase)
15 Tiempo::Tiempo( int hora, int minuto, int segundo )
16 {
17     setTiempo( hora, minuto, segundo );
18 } // fin del constructor de Tiempo
19
20 // establece los valores de hora, minuto y segundo
21 void Tiempo::setTiempo( int hora, int minuto, int segundo )
22 {
23     setHora( hora );
24     setMinuto( minuto );
25     setSegundo( segundo );
26 } // fin de la función setTiempo
27
28 // establece el valor de hora
29 void Tiempo::setHora( int h )
30 {
31     hora = ( h >= 0 && h < 24 ) ? h : 0; // valida la hora
32 } // fin de la función setHora
33
34 // establece el valor de minuto
35 void Tiempo::setMinuto( int m )
36 {

```

Figura 10.2 | Definiciones de las funciones miembro de la clase **Tiempo**, incluyendo las funciones miembro **const**. (Parte 1 de 2).

```

37     minuto = ( m >= 0 && m < 60 ) ? m : 0; // valida el minuto
38 } // fin de la función setMinuto
39
40 // establece el valor de segundo
41 void Tiempo::setSegundo( int s )
42 {
43     segundo = ( s >= 0 && s < 60 ) ? s : 0; // valida el segundo
44 } // fin de la función setSegundo
45
46 // devuelve el valor de hora
47 int Tiempo::getHora() const // las funciones obtener deben ser const
48 {
49     return hora;
50 } // fin de la función getHora
51
52 // devuelve el valor de minuto
53 int Tiempo::getMinuto() const
54 {
55     return minuto;
56 } // fin de la función getMinuto
57
58 // devuelve el valor de segundo
59 int Tiempo::getSegundo() const
60 {
61     return segundo;
62 } // fin de la función getSegundo
63
64 // imprime el Tiempo en formato universal (HH:MM:SS)
65 void Tiempo::imprimirUniversal() const
66 {
67     cout << setfill( '0' ) << setw( 2 ) << hora << ":"
68         << setw( 2 ) << minuto << ":" << setw( 2 ) << segundo;
69 } // fin de la función imprimirUniversal
70
71 // imprime el Tiempo en formato estándar (HH:MM:SS AM or PM)
72 void Tiempo::imprimirEstandar() // observe que no hay declaración const
73 {
74     cout << ( ( hora == 0 || hora == 12 ) ? 12 : hora % 12 )
75         << ":" << setfill( '0' ) << setw( 2 ) << minuto
76         << ":" << setw( 2 ) << segundo << ( hora < 12 ? " AM" : " PM" );
77 } // fin de la función imprimirEstandar

```

Figura 10.2 | Definiciones de las funciones miembro de la clase `Tiempo`, incluyendo las funciones miembro `const`. (Parte 2 de 2).

En la figura 10.3 se instancian dos objetos `Tiempo`: el objeto no `const` `despertar` (línea 7) y el objeto `const` `mediodia` (línea 8). El programa trata de invocar a las funciones miembro no `const` `setHora` (línea 13) e `imprimirEstandar` (línea 20) en el objeto `const` `mediodia`. En cada caso el compilador genera un mensaje de error. El programa también ilustra las otras tres combinaciones de llamadas a funciones miembro en los objetos: una función miembro no `const` en un objeto no `const` (línea 11), una función miembro `const` en un objeto no `const` (línea 15) y una función miembro `const` en un objeto `const` (líneas 17 a 18). Los mensajes de error generados para las funciones miembro no `const` que se llaman desde un objeto `const` se muestran en la ventana de resultados. Observe que, aunque algunos compiladores actuales generan sólo mensajes de advertencia para las líneas 13 y 20 (con lo cual el programa se puede ejecutar), consideramos estas advertencias como errores; el estándar ISO/IEC de C++ no permite invocar una función miembro no `const` en un objeto `const`.

Observe que, aun y cuando un constructor debe ser una función miembro no `const` (figura 10.2, líneas 15 a 18), de todas formas se puede utilizar para inicializar un objeto `const` (figura 10.3, línea 8). La definición del constructor de `Tiempo` (figura 10.2, líneas 15 a 18) muestra que llama a otra función miembro no `const` (`setTiempo`, líneas 21 a 26) para realizar la inicialización de un objeto `Tiempo`. Se permite la invocación de una función miembro no `const` desde la llamada al constructor como parte de la inicialización de un objeto `const`. Lo “constante” de un objeto `const` se hace valer desde el momento en que el constructor completa la inicialización del objeto, hasta que se llama al destructor de ese objeto.

```

1 // Fig. 10.3: fig10_03.cpp
2 // Intento de acceder a un objeto const con funciones miembro no const.
3 #include "Tiempo.h" // incluye la definición de la clase Tiempo
4
5 int main()
6 {
7     Tiempo despertar( 6, 45, 0 ); // objeto no constante
8     const Tiempo mediodia( 12, 0, 0 ); // objeto constante
9
10            // OBJETO      FUNCIÓN MIEMBRO
11     despertar.setHora( 18 );      // no const    no const
12
13     mediodia.setHora( 12 );      // const      no const
14
15     despertar.getHora();        // no const    const
16
17     mediodia.getMinuto();       // const      const
18     mediodia.imprimirUniversal(); // const      const
19
20     mediodia.imprimirEstandar(); // const      no const
21
22 } // fin de main

```

Mensajes de error del compilador de línea de comandos Borland C++:

```

Warning W8037 fig10_03.cpp 13: Non-const function Tiempo::setHora(int)
called for const object in function main()
Warning W8037 fig10_03.cpp 20: Non-const function Tiempo::imprimirEstandar()
Called for const object in function main()

```

Mensajes de error del compilador Microsoft Visual C++ 2005:

```

C:\cpphttp6_ejemplos\cap10\Fig10_01_03\fig10_03.cpp(13) : error C2662:
'Tiempo::setHora' : cannot convert 'this' pointer from 'const Tiempo' to
'Tiempo &'

Conversion loses qualifiers
C:\cpphttp6_ejemplos\cap10\Fig10_01_03\fig10_03.cpp(20) : error C2662:
'Tiempo::imprimirEstandar' : cannot convert 'this' pointer from 'const Tiempo' to
'Tiempo &'

Conversion loses qualifiers

```

Mensajes de error del compilador GNU C++:

```

fig10_03.cpp:13: error: passing 'const Tiempo' as 'this' argument of
  'void Tiempo::setHora(int)' discards qualifiers
fig10_03.cpp:20: error: passing 'const Tiempo' as 'this' argument of
  'void Tiempo::imprimirEstandar()' discards qualifiers

```

Figura 10.3 | Objetos **const** y funciones miembro **const**.

Observe además que la línea 20 en la figura 10.3 genera un error de compilación, aun y cuando la función miembro **imprimirEstandar** de la clase **Tiempo** no modifica el objeto en el que se le invoca. El hecho de que una función miembro no modifica un objeto no basta para indicar que la función es constante; se debe declarar explícitamente como **const**.

Inicialización de datos miembro **const con una función miembro inicializadora**

El programa de las figuras 10.4 a 10.6 introduce el uso de la sintaxis de **inicializador de miembros**. Todos los datos miembro se *pueden* inicializar mediante la sintaxis de inicializador de miembros, pero los datos miembro **const** y los datos miembro que son referencias *deben* inicializarse mediante inicializadores de miembros. Más adelante en este capítulo veremos que los objetos miembro se deben inicializar de esta forma también. En el capítulo 12, Programación Orientada a objetos: herencia, veremos que las porciones de la clase base en las clases derivadas también se deben inicializar de esta forma.

```

1 // Fig. 10.4: Incremento.h
2 // Definición de la clase Incremento.
3 #ifndef INCREMENTO_H
4 #define INCREMENTO_H
5
6 class Incremento
7 {
8 public:
9     Incremento( int c = 0, int i = 1 ); // constructor predeterminado
10
11    // definición de la función agregarIncremento
12    void agregarIncremento()
13    {
14        cuenta += incremento;
15    } // fin de la función agregarIncremento
16
17    void imprimir() const; // imprime cuenta e incremento
18 private:
19     int cuenta;
20     const int incremento; // miembro de datos const
21 }; // fin de la clase Incremento
22
23 #endif

```

Figura 10.4 | Definición de la clase **Incremento** que contiene los datos miembro no **const** **cuenta** y los datos miembro **const** **incremento**.

```

1 // Fig. 10.5: Incremento.cpp
2 // Las definiciones de las funciones miembro para la clase Incremento demuestran el uso
3 // de un inicializador de miembros para inicializar una constante de un tipo de datos
4 // integrado.
5 #include <iostream>
6 using std::cout;
7 using std::endl;
8
9 #include "Incremento.h" // incluye la definición de la clase Incremento
10
11 // constructor
12 Incremento::Incremento( int c, int i )
13     : cuenta( c ), // inicializador para un miembro no const
14       incremento( i ) // inicializador requerido para un miembro const
15 {
16     // cuerpo vacío
17 } // fin del constructor de Incremento
18
19 // imprime los valores de cuenta e incremento
20 void Incremento::imprimir() const
21 {
22     cout << "cuenta = " << cuenta << ", incremento = " << incremento << endl;
23 } // fin de la función imprimir

```

Figura 10.5 | Inicializador de miembros utilizado para inicializar una constante de un tipo de datos integrado.

```

1 // Fig. 10.6: fig10_06.cpp
2 // Programa para probar la clase Incremento.
3 #include <iostream>
4 using std::cout;
5
6 #include "Incremento.h" // incluye la definición de la clase Incremento

```

Figura 10.6 | Invocación de las funciones miembro **imprimir** y **agregarIncremento** de un objeto **Incremento**. (Parte I de 2).

```

7 int main()
8 {
9     Incremento valor( 10, 5 );
10    cout << "Antes de incrementar: ";
11    valor.imprimir();
12
13    for ( int j = 1; j <= 3; j++ )
14    {
15        valor.agregarIncremento();
16        cout << "Despues de incrementar " << j << ": ";
17        valor.imprimir();
18    } // fin de for
19
20    return 0;
21
22 } // fin de main

```

```

Antes de incrementar: cuenta = 10, incremento = 5
Despues de incrementar 1: cuenta = 15, incremento = 5
Despues de incrementar 2: cuenta = 20, incremento = 5
Despues de incrementar 3: cuenta = 25, incremento = 5

```

Figura 10.6 | Invocación de las funciones miembro `imprimir` y `agregarIncremento` de un objeto `Incremento`. (Parte 2 de 2).

La definición del constructor (figura 10.5, líneas 11 a 16) usa una **lista de inicializadores de miembros** para inicializar los datos miembro de la clase `Incremento`: el entero no `const` `cuenta` y el entero `const` `incremento` (declarados en las líneas 19 y 20 de la figura 10.4). Los inicializadores de miembros aparecen entre la lista de parámetros de un constructor y la llave izquierda que empieza el cuerpo del constructor. La lista de inicializadores de miembros (figura 10.5, líneas 12 y 13) se separa de la lista de parámetros con un signo de dos puntos (:). Cada inicializador de miembros consiste en el nombre del dato miembro, seguido de paréntesis que contienen el valor inicial del dato miembro. En este ejemplo, `cuenta` se inicializa con el valor del parámetro `c` del constructor, y `incremento` se inicializa con el valor del parámetro `i` del constructor. Observe que varios inicializadores de miembros van separados por comas. Observe además que la lista de inicializadores de miembros se ejecuta antes que se ejecute el cuerpo del constructor.



Observación de Ingeniería de Software 10.3

Un objeto `const` no se puede modificar mediante la asignación, por lo que debe inicializarse. Cuando un miembro de datos de una clase se declara `const`, hay que utilizar un inicializador de miembros para proporcionar al constructor el valor inicial de los datos miembro para un objeto de la clase. Lo mismo se aplica para las referencias.

Intento erróneo de inicializar datos miembro `const` con una asignación

El programa de las figuras 10.7 a 10.9 ilustra los errores de compilación que se producen al tratar de inicializar los datos miembro `const` `incremento` con una instrucción de asignación (figura 10.8, línea 14) en el cuerpo del constructor de `Incremento`, en vez de hacerlo con un inicializador de miembros. Observe que en la línea 13 de la figura 10.8 no se genera un error de compilación, ya que `cuenta` no se declara como `const`.



Error común de programación 10.5

Si no se proporciona un inicializador de miembros para datos miembro `const`, se produce un error de compilación.



Observación de Ingeniería de Software 10.4

Los datos miembro constantes (objetos y variables `const`) y los datos miembro que se declaran como referencias se deben inicializar con la sintaxis de inicializador de miembros; las asignaciones para estos tipos de datos en el cuerpo del constructor no están permitidas.

Observe que la función `imprimir` (figura 10.8, líneas 18 a 21) se declara `const`. Podría parecer extraño etiquetar esta función `const`, ya que es probable que un programa nunca tenga un objeto `Incremento const`. Sin embargo, es

```

1 // Fig. 10.7: Incremento.h
2 // Definición de la clase Incremento.
3 #ifndef INCREMENTO_H
4 #define INCREMENTO_H
5
6 class Incremento
7 {
8 public:
9     Incremento( int c = 0, int i = 1 ); // constructor predeterminado
10
11    // definición de la función agregarIncremento
12    void agregarIncremento()
13    {
14        cuenta += incremento;
15    } // fin de la función agregarIncremento
16
17    void imprimir() const; // imprime cuenta e incremento
18 private:
19     int cuenta;
20     const int incremento; // miembro de datos const
21 }; // fin de la clase Incremento
22
23 #endif

```

Figura 10.7 | Definición de la clase `Incremento` que contiene datos miembro no `const` `cuenta` y datos miembro `const` `incremento`.

posible que un programa tenga una referencia `const` a un objeto `Incremento`, o un apuntador a `const` que apunte a un objeto `Incremento`. Por lo general, esto ocurre cuando se pasan objetos de la clase `Incremento` a las funciones, o cuando se devuelven de las funciones. En estos casos, sólo las funciones miembro `const` de la clase `Incremento` se pueden llamar a través de la referencia o apuntador. Por ende, es razonable declarar la función `imprimir` como `const`; esto evita errores en las situaciones en las que un objeto `Incremento` se trata como un objeto `const`.



Tip para prevenir errores 10.1

Declare como `const` todas las funciones miembro de una clase que no modifiquen el objeto en el que operan. Algunas veces esto puede parecer inapropiado, porque no se tendrá la intención de crear objetos `const` de esa clase, o de acceder a objetos de esa clase a través de referencias `const` o apuntadores a `const`. Sin embargo, declarar dichas funciones miembro como `const` ofrece un beneficio. Si la función miembro se escribe de manera inadvertida para modificar el objeto, el compilador generará un mensaje de error.

```

1 // Fig. 10.8: Incremento.cpp
2 // Intento erróneo de inicializar una constante de un tipo de datos
3 // integrado mediante la asignación.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "Incremento.h" // incluye la definición de la clase Incremento
9
10 // constructor; el miembro constante 'incremento' no se inicializa
11 Incremento::Incremento( int c, int i )
12 {
13     cuenta = c; // se permite, ya que cuenta no es constante
14     incremento = i; // ERROR: no se puede modificar un objeto const
15 } // fin del constructor de Incremento
16

```

Figura 10.8 | Intento erróneo de inicializar una constante de un tipo de datos integrado mediante la asignación. (Parte I de 2).

```

17 // imprime los valores de cuenta e incremento
18 void Incremento::imprimir() const
19 {
20     cout << "cuenta = " << cuenta << ", incremento = " << incremento << endl;
21 } // fin de la función imprimir

```

Figura 10.8 | Intento erróneo de inicializar una constante de un tipo de datos integrado mediante la asignación. (Parte 2 de 2).

```

1 // Fig. 10.9: fig10_09.cpp
2 // Programa para evaluar la clase Incremento.
3 #include <iostream>
4 using std::cout;
5
6 #include "Incremento.h" // incluye la definición de la clase Incremento
7
8 int main()
9 {
10     Incremento valor( 10, 5 );
11
12     cout << "Antes de incrementar: ";
13     valor.imprimir();
14
15     for ( int j = 10; j <= 3; j++ )
16     {
17         valor.agregarIncremento();
18         cout << "Despues de incrementar " << j << ": ";
19         valor.imprimir();
20     } // fin de for
21
22     return 0;
23 } // fin de main

```

Mensajes de error en la línea de comandos del compilador Borland C++:

```
Error E2024 Incremento.cpp 14: Cannot modify a const object in function
Incremento::Incremento(int, int)
```

Mensajes de error del compilador Microsoft Visual C++ 2005:

```
C:\cpphttp6_ejemplos\cap10\Fig10_07_09\Incremento.cpp(12) : error C2758:
'Incremento::incremento' : must be initialized in constructor
base/member initializer list
C:\cpphttp6_ejemplos\cap10\Fig10_07_09\Incremento.h(20) :
    see declaration of 'Incremento::incremento'
C:\cpphttp6_ejemplos\cap10\Fig10_07_09\Incremento.cpp(14) : error C2166:
    l-value specifies const object
```

Mensajes de error del compilador GNU C++:

```
Incremento.cpp:12: error: uninitialized member 'Incremento::incremento' with
    'const' type 'const int'
Incremento.cpp:14: error: assignment of read-only data-member
    'Incremento::incremento'
```

Figura 10.9 | Programa para probar la clase Incremento que genera errores de compilación.

10.3 Composición: objetos como miembros de clases

Un objeto RelojDespertador necesita saber cuándo se supone que debe sonar su alarma, así que ¿por qué no incluir un objeto Tiempo como miembro de la clase RelojDespertador? Dicha capacidad se conoce como **composición** y algunas veces como **relación “tiene un”**; una clase puede tener objetos de otras clases como miembros.



Observación de Ingeniería de Software 10.5

Una forma común de reutilización de software es la composición, en la cual una clase tiene objetos de otras clases como miembros.

Cuando se crea un objeto, su constructor se llama de manera automática. Anteriormente vimos cómo pasar argumentos al constructor de un objeto que creamos en `main`. En esta sección veremos cómo el constructor de un objeto puede pasar argumentos a los constructores de objetos miembro, lo cual se realiza mediante inicializadores de miembros.



Observación de Ingeniería de Software 10.6

Los objetos miembro se construyen en el orden en el que se declaran en la definición de la clase (no en el orden en el que se listan en la lista de inicializadores de miembros del constructor) y antes de que se construyan los objetos de la clase (algunas veces conocidos como objetos anfitriones).

El programa de las figuras 10.10 a 10.14 utiliza la clase `Fecha` (figuras 10.10 y 10.11) y la clase `Empleado` (figuras 10.12 y 10.13) para demostrar los objetos como miembros de otros objetos. La definición de la clase `Empleado` (figura 10.12) contiene los datos miembro `private primerNombre, apellidoPaterno, fechaNacimiento` y `fechaContratacion`. Los miembros `fechaNacimiento` y `fechaContratacion` son objetos `const` de la clase `Fecha`, que contiene los datos miembro `private mes, dia y anio`. El encabezado del constructor de `Empleado` (figura 10.13, líneas 18 a 21) especifica que el constructor tiene cuatro parámetros (`primero, ultimo, fechaDeNacimiento` y `fechaDeContratacion`). Los primeros dos parámetros se utilizan en el cuerpo del constructor para inicializar los arreglos de caracteres `primerNombre` y `apellidoPaterno`. Los últimos dos parámetros se pasan mediante inicializadores miembro al constructor para la clase `Fecha`. El signo de dos puntos (:) en el encabezado separa los inicializadores de miembros de la lista de parámetros. Los inicializadores de miembros especifican los parámetros del constructor de `Empleado` que se van a pasar a los constructores de los objetos `Fecha` miembros. El parámetro `fechaDeNacimiento` se pasa al constructor del objeto `fechaNacimiento` (figura 10.13, línea 20), y el parámetro `fechaDeContratacion` se pasa al constructor del objeto `fechaContratacion` (figura 10.13, línea 21). De nuevo, los inicializadores de miembros van separados por comas. Al estudiar la clase `Fecha` (figura 10.10), el lector observará que la clase no proporciona un constructor que reciba un parámetro de tipo `Fecha`. Entonces, ¿cómo puede la lista de inicializadores de miembros en el constructor de la clase `Empleado` inicializar los objetos `fechaNacimiento` y `fechaContratacion` al pasar el objeto `Fecha` a sus constructores de `Fecha`? Como mencionamos en el capítulo 9, el compilador proporciona a cada clase una copia pre-determinada del constructor que copia cada miembro de datos del objeto argumento del constructor en el miembro correspondiente del objeto que se va a inicializar. En el capítulo 11 veremos cómo se pueden definir constructores de copia personalizados.

```

1 // Fig. 10.10: Fecha.h
2 // Definición de la clase Fecha; las funciones miembro se definen en Fecha.cpp
3 #ifndef FECHA_H
4 #define FECHA_H
5
6 class Fecha
7 {
8 public:
9     Fecha( int = 1, int = 1, int = 1900 ); // constructor predeterminado
10    void imprimir() const; // imprime la fecha en formato mes/dia/año
11    ~Fecha(); // se proporciona para confirmar el orden de destrucción
12 private:
13    int mes; // 1-12 (Enero-Diciembre)
14    int dia; // 1-31 con base en el mes
15    int anio; // cualquier año
16
17    // función utilitaria para comprobar si el dia es apropiado para mes y anio
18    int comprobarDia( int ) const;
19 }; // fin de la clase Fecha
20
21 #endif

```

Figura 10.10 | Definición de la clase `Fecha`.

En la figura 10.14 se crean dos objetos `Fecha` (líneas 11 y 12) y se pasan como argumentos al constructor del objeto `Empleado` creado en la línea 13. En la línea 16 se imprimen los datos del objeto `Empleado`. Cuando se crea cada objeto `Fecha` en las líneas 11 y 12, el constructor de `Fecha` definido en las líneas 11 a 28 de la figura 10.11 despliega una línea de salida para mostrar que se llamó al constructor (vea las primeras dos líneas de los resultados de ejemplo). [Nota: en la línea 13 de la figura 10.14 se hacen dos llamadas adicionales al constructor de `Fecha` que no aparecen en la salida del programa. Cuando se inicializa cada uno de los objetos miembro `Fecha` de `Empleado` en la lista de inicializadores de miembros del constructor de `Empleado` (figura 10.13, líneas 20 y 21), se hace una llamada al constructor de copia predeterminado para la clase `Fecha`. El compilador define este constructor de manera implícita, y no contiene instrucciones de salida para demostrar cuándo se hace la llamada. En el capítulo 11 hablaremos con detalle sobre los constructores de copia y los constructores de copia predeterminados].

```

1 // Fig. 10.11: Fecha.cpp
2 // Definiciones de las funciones miembro de la clase Fecha.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Fecha.h" // incluye la definición de la clase Fecha
8
9 // el constructor confirma el valor apropiado para el mes; llama
10 // a la función utilitaria comprobarDia para confirmar un valor apropiado para dia
11 Fecha::Fecha( int mm, int dd, int aa )
12 {
13     if ( mm > 0 && mm <= 12 ) // valida el mes
14         mes = mm;
15     else
16     {
17         mes = 1; // mes inválido se establece en 1
18         cout << "Mes invalido (" << mm << ") se establecio en 1.\n";
19     } // fin else
20
21     anio = aa; // se pudo validar aa
22     dia = comprobarDia( dd ); // valida el dia
23
24     // imprime objeto Fecha para mostrar cuándo se llama a su constructor
25     cout << "Constructor del objeto Fecha para fecha ";
26     imprimir();
27     cout << endl;
28 } // fin del constructor de Fecha
29
30 // imprime objeto Fecha en el formato mes/dia/anio
31 void Fecha::imprimir() const
32 {
33     cout << mes << '/' << dia << '/' << anio;
34 } // fin de la función imprimir
35
36 // imprime objeto Fecha para mostrar cuándo se llama a su destructor
37 Fecha::~Fecha()
38 {
39     cout << "Destructor del objeto Fecha para fecha ";
40     imprimir();
41     cout << endl;
42 } // fin del destructor ~Fecha
43
44 // función utilitaria para confirmar el valor de dia apropiado con base
45 // en mes y anio; maneja años bisiestos también
46 int Fecha::comprobarDia( int diaPrueba ) const
47 {
48     static const int diasPorMes[ 13 ] =

```

Figura 10.11 | Definiciones de las funciones miembro de la clase `Fecha`. (Parte 1 de 2).

```

49     { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
50
51     // determina si diaPrueba es válido para el mes especificado
52     if ( diaPrueba > 0 && diaPrueba <= diasPorMes[ mes ] )
53         return diaPrueba;
54
55     // comprueba 29 de febrero para año bisiesto
56     if ( mes == 2 && diaPrueba == 29 && ( anio % 400 == 0 ||
57         ( anio % 4 == 0 && anio % 100 != 0 ) ) )
58         return diaPrueba;
59
60     cout << "Dia invalido (" << diaPrueba << ") se establecio en 1.\n";
61     return 1; // deja el objeto en estado consistente si hay un valor incorrecto
62 } // fin de la función comprobarDia

```

Figura 10.11 | Definiciones de las funciones miembro de la clase Fecha. (Parte 2 de 2).

```

1 // Fig. 10.12: Empleado.h
2 // Definición de la clase Empleado que muestra la composición.
3 // Las funciones miembro se definen en Empleado.cpp.
4 #ifndef EMPLEADO_H
5 #define EMPLEADO_H
6
7 #include "Fecha.h" // incluye la definición de la clase Fecha
8
9 class Empleado
10 {
11 public:
12     Empleado( const char * const, const char * const,
13             const Fecha &, const Fecha & );
14     void imprimir() const;
15     ~Empleado(); // se proporciona para confirmar el orden de destrucción
16 private:
17     char primerNombre[ 25 ];
18     char apellidoPaterno[ 25 ];
19     const Fecha fechaNacimiento; // composición: objeto miembro
20     const Fecha fechaContratacion; // composición: objeto miembro
21 }; // fin de la clase Empleado
22
23 #endif

```

Figura 10.12 | Definición de la clase Empleado que muestra la composición.

La clase Fecha y la clase Empleado incluyen un destructor (líneas 37 a 42 de la figura 10.11, y líneas 51 a 55 de la figura 10.13, respectivamente) que imprime un mensaje cuando se destruye un objeto de su clase. Esto nos permite confirmar en la salida del programa que los objetos se construyen de adentro hacia afuera y se destruyen en orden inverso, desde afuera hacia adentro (es decir, los objetos miembro de Fecha se destruyen después del objeto Empleado que los contiene). Observe las últimas cuatro líneas en la salida de la figura 10.14. Las últimas dos líneas son la salida del destructor de Fecha que se ejecuta en los objetos Fecha_contratacion (línea 12) y nacimiento (línea 11), respectivamente. Estos resultados confirman que los tres objetos creados en main se destruyen en el orden inverso al orden en el que se construyeron. (La salida del destructor de Empleado aparece cinco líneas antes de la última). Las líneas tercera y cuarta de la parte inferior de la ventana de resultados muestran la ejecución de los destructores para los objetos miembro fechaContratacion (figura 10.12, línea 20) y fechaNacimiento (figura 10.12, línea 19) de Empleado. Estos resultados confirman que el objeto Empleado se destruye desde el exterior hacia el interior; es decir, el destructor de Empleado se ejecuta primero (el resultado se muestra cinco líneas antes de la última de la ventana de resultados), y después los objetos miembro se destruyen en el orden inverso al que fueron construidos. De nuevo, los resultados en la figura 10.14 no muestran la ejecución de los constructores para estos objetos miembro, ya que fueron los constructores predeterminados que proporciona el compilador de C++.

```

1 // Fig. 10.13: Empleado.cpp
2 // Definiciones de las funciones miembro de la clase Empleado.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // prototipos de strlen y strncpy
8 using std::strlen;
9 using std::strncpy;
10
11 #include "Empleado.h" // definición de la clase Empleado
12 #include "Fecha.h" // definición de la clase Fecha
13
14 // el constructor usa la lista de inicializadores de miembros para pasar los valores
15 // de los inicializadores a los constructores de los objetos miembro fechaNacimiento y
16 // fechaContratacion
17 // [Nota: esto invoca al llamado "constructor de copia predeterminado" que el
18 // compilador de C++ proporciona de manera implícita.]
19 Empleado::Empleado( const char * const nombre, const char * const apellido,
20                     const Fecha &fechaDeNacimiento, const Fecha &fechaDeContratacion )
21   : fechaNacimiento( fechaDeNacimiento ), // inicializa fechaNacimiento
22     fechaContratacion( fechaDeContratacion ) // inicializa fechaContratacion
23 {
24   // copia nombre en primerNombre y se asegura de que quepa
25   int longitud = strlen( nombre );
26   longitud = ( longitud < 25 ? longitud : 24 );
27   strncpy( primerNombre, nombre, longitud );
28   primerNombre[ longitud ] = '\0';
29
30   // copy apellido into apellidoPaterno and be sure that it fits
31   longitud = strlen( apellido );
32   longitud = ( longitud < 25 ? longitud : 24 );
33   strncpy( apellidoPaterno, apellido, longitud );
34   apellidoPaterno[ longitud ] = '\0';
35
36   // imprime objeto Empleado para mostrar cuándo se llama al constructor
37   cout << "Constructor del objeto Empleado: "
38   << primerNombre << ' ' << apellidoPaterno << endl;
39 } // fin del constructor de Empleado
40
41 // imprime objeto Empleado
42 void Empleado::imprimir() const
43 {
44   cout << apellidoPaterno << ", " << primerNombre << " Contratacion: ";
45   fechaContratacion.imprimir();
46   cout << " Nacimiento: ";
47   fechaNacimiento.imprimir();
48   cout << endl;
49 } // fin de la función imprimir
50
51 // imprime objeto Empleado para mostrar cuándo se llama a su destructor
52 Empleado::~Empleado()
53 {
54   cout << "Destructor del objeto Empleado: "
55   << apellidoPaterno << ", " << primerNombre << endl;
56 } // fin del constructor ~Empleado

```

Figura 10.13 | Definiciones de las funciones miembro de la clase Empleado, incluyendo el constructor con una lista de inicializadores de miembros.

Un objeto miembro no necesita inicializarse de manera explícita a través de un inicializador de miembros. Si no se proporciona uno, se hará una llamada implícita al constructor predeterminado del objeto miembro. Los valores (si los hay)

```

1 // Fig. 10.14: fig10_14.cpp
2 // Demostración de la composición--un objeto con objetos miembro.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Empleado.h" // definición de la clase Empleado
8
9 int main()
10 {
11     Fecha nacimiento( 7, 24, 1949 );
12     Fecha contratacion( 3, 12, 1988 );
13     Empleado gerente( "Bob", "Blue", nacimiento, contratacion );
14
15     cout << endl;
16     gerente.imprimir();
17
18     cout << "\nPrueba del constructor de Fecha con valores inválidos:\n";
19     Fecha ultimoDiaDescanso( 14, 35, 1994 ); // mes y dia inválidos
20     cout << endl;
21     return 0;
22 } // fin de main

```

```

Constructor del objeto Fecha para fecha 7/24/1949
Constructor del objeto Fecha para fecha 3/12/1988
Constructor del objeto Empleado: Bob Blue _____
Blue, Bob Contratacion: 3/12/1988 Nacimiento: 7/24/1949
Prueba del constructor de Fecha con valores invalidos:
Mes invalido (14) se establecio en 1.
Dia invalido (35) se establecio en 1.
Constructor del objeto Fecha para fecha 1/1/1994

Destructor del objeto Fecha para fecha 1/1/1994
Destructor del objeto Empleado: Blue, Bob
Destructor del objeto Fecha para fecha 3/12/1988
Destructor del objeto Fecha para fecha 7/24/1949
Destructor del objeto Fecha para fecha 3/12/1988
Destructor del objeto Fecha para fecha 7/24/1949

```

Note que cuando se construye un **Empleado** se llaman tres constructores: dos llaman al constructor de copia por omisión de la clase **Fecha** (llamados en las líneas 20-21 de la figura 10.13) y el otro llama al constructor de la clase **Empleado**

Figura 10.14 | Demostración de la composición: un objeto con objetos miembros.

establecidos por el constructor predeterminado se pueden redefinir mediante funciones *set*. Sin embargo, para la inicialización compleja, esta metodología puede requerir una cantidad considerable de trabajo y tiempo adicionales.

Error común de programación 10.6



Si un objeto miembro no se inicializa con un inicializador de miembros y la clase del objeto miembro no proporciona un constructor predeterminado (es decir, que la clase del objeto miembro defina uno o más constructores, pero ninguno sea un constructor predeterminado), se produce un error de compilación.

Tip de rendimiento 10.2



*Inicialice los objetos miembro explícitamente a través de los inicializadores de miembros. Esto elimina la sobrecarga de “inicializar dos veces” los objetos miembro: una vez cuando se hace la llamada al constructor predeterminado del objeto miembro, y otra vez cuando se hacen las llamadas a las funciones *set* en el cuerpo del constructor (o después) para inicializar el objeto miembro.*

Observación de Ingeniería de Software 10.7



*Si el miembro de una clase es un objeto de otra clase, al hacer ese objeto miembro *public* no se viola el encapsulamiento ni el ocultamiento de los miembros *private* de ese objeto miembro. No obstante, sí se viola el encapsulamiento y el ocultamiento de la implementación de la clase que lo contiene, por lo que los objetos miembro de los tipos de clases deben seguir siendo *private*, al igual que todos los demás miembros de datos.*

En la línea 26 de la figura 10.11, observe la llamada a la función miembro `imprimir` de `Fecha`. Muchas funciones miembro de las clases en C++ no requieren argumentos. Esto se debe a que cada función miembro contiene un manejador implícito (en forma de un apuntador) al objeto en el que opera. Hablaremos sobre el apuntador implícito, el cual se representa mediante la palabra clave `this`, en la sección 10.5.

La clase `Empleado` usa dos arreglos de 25 caracteres (figura 10.12, líneas 17 y 18) para representar el primer nombre y el apellido paterno del `Empleado`. Estos arreglos pueden desperdiciar espacio para los nombres menores de 24 caracteres. (Recuerde, un carácter en cada arreglo es para el carácter nulo de terminación, '\0', de la cadena). Además, los nombres más largos de 24 caracteres deben truncarse para adaptarlos en estos arreglos de caracteres de tamaño fijo. En la sección 10.7 presentaremos otra versión de la clase `Empleado` que crea de manera dinámica la cantidad exacta de espacio requerido para contener el primer nombre y el apellido paterno.

Observe que la manera más simple de representar el primer nombre y el apellido paterno de un `Empleado` con la cantidad exacta de espacio requerido es utilizar dos objetos `string` (en el capítulo 3 presentamos la clase `string` de la Biblioteca estándar de C++). Si hicieramos esto, el constructor de `Empleado` sería como se muestra a continuación:

```
Empleado::Empleado( const string &nombre, const string &apellido,
                      const Fecha &fechaDeNacimiento, const Fecha &fechaDeContratacion )
    : primerNombre( nombre ), // inicializa primerNombre
      apellidoPaterno( apellido ), // inicializa apellidoPaterno
      fechaNacimiento( fechaDeNacimiento ), // inicializa fechaNacimiento
      fechaContratacion( fechaDeContratacion ) // inicializa fechaContratacion
{
    // imprime objeto Empleado para mostrar cuándo se llama al constructor
    cout << "Constructor del objeto Empleado: "
        << primerNombre << ' ' << apellidoPaterno << endl;
} // constructor del objeto Empleado
```

Observe que los datos miembro `primerNombre` y `apellidoPaterno` (que ahora son objetos `string`) se inicializan a través de inicializadores de miembros. Las clases `Empleado` que se presentan en los capítulos 12 y 13 utilizan objetos `string` de esta forma. En este capítulo vamos a usar cadenas basadas en apuntador, para que el lector obtenga una exposición adicional a la manipulación de apuntadores.

10.4 Funciones friend y clases friend

Una función `friend` de una clase se define fuera del alcance de ésta, pero de todas formas tiene el derecho de acceder a los miembros no `public` (y `public`) de la clase. Se pueden declarar funciones independientes o clases completas como amigas de otra clase.

El uso de funciones `friend` puede mejorar el rendimiento. En esta sección presentaremos un ejemplo mecánico acerca de cómo funciona una función `friend`. Más adelante en el libro, utilizaremos funciones `friend` para sobrecargar operadores y usarlos con objetos de clases (capítulo 11), y para crear clases iteradoras (capítulo 20, Estructuras de datos). Los objetos de una clase iteradora pueden seleccionar elementos de manera sucesiva, o realizar una operación con elementos en un objeto de clase contenedora (vea la sección 10.9). Los objetos de clases contenedoras pueden almacenar elementos. El uso de funciones amigas es comúnmente apropiado cuando no se puede usar una función miembro para ciertas operaciones, como veremos en el capítulo 11.

Para declarar una función como amiga de una clase, hay que anteponer la palabra clave `friend` al prototipo de la función en la definición de la clase. Para declarar todas las funciones miembro de la clase `ClaseDos` como amigas de la clase `ClaseUno`, coloque una declaración de la forma

```
friend class ClaseDos;
```

en la definición de la clase `ClaseUno`.



Observación de Ingeniería de Software 10.8

Aun cuando los prototipos para las funciones friend aparecen en la definición de la clase, las funciones amigas no son funciones miembro.



Observación de Ingeniería de Software 10.9

Las nociones private, protected y public de acceso a miembros no son relevantes para las declaraciones friend, por lo que estas declaraciones se pueden colocar en cualquier parte de la definición de una clase.



Buena práctica de programación 10.I

Coloque todas las declaraciones de amistad primero dentro del cuerpo de la definición de una clase, y no coloque un especificador de acceso antes de éstas.

La amistad se otorga, no se toma; por ejemplo, para que la clase B sea amiga (`friend`) de la clase A, ésta debe declarar explícitamente que la clase B es su amiga. Además, la relación de amistad no es simétrica ni transitiva; es decir, si la clase A es amiga de la clase B, y ésta es amiga de la clase C, no podemos inferir que la clase B es amiga de la clase A (de nuevo, la amistad no es simétrica), que la clase C es amiga de la clase B (también debido a que la amistad no es simétrica), o que la clase A es amiga de la clase C (la amistad no es transitiva).



Observación de Ingeniería de Software 10.II

Algunas personas en la comunidad de la POO sienten que la “amistad” corrompe el ocultamiento de información y debilita el valor de la metodología del diseño orientado a objetos. En este texto identificamos varios ejemplos del uso responsable de la amistad.

Modificación de los datos `private` de una clase con una función `friend`

La figura 10.15 es un ejemplo mecánico en el que definimos la función `friend` llamada `setX` para establecer el miembro de datos `private` `x` de la clase `Cuenta`. Observe que la declaración `friend` (línea 10) aparece primero (por convención) en la definición de la clase, incluso antes de declarar las funciones miembro `public`. De nuevo, esta declaración `friend` puede aparecer en cualquier parte de la clase.

```

1 // Fig. 10.15: fig10_15.cpp
2 // Las funciones amigas pueden acceder a los miembros privados de una clase.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // definición de la clase Cuenta
8 class Cuenta
9 {
10     friend void setX( Cuenta &, int ); // declaración friend
11 public:
12     // constructor
13     Cuenta()
14         : x( 0 ) // inicializa x en 0
15     {
16         // cuerpo vacío
17     } // fin del constructor Cuenta
18
19     // imprime x
20     void imprimir() const
21     {
22         cout << x << endl;
23     } // fin de la función imprimir
24 private:
25     int x; // miembro de datos
26 }; // fin de la clase Cuenta
27
28 // la función setX puede modificar los datos privados de Cuenta
29 // debido a que setX se declara como amiga de Cuenta (línea 10)
30 void setX( Cuenta &c, int val )
31 {
32     c.x = val; // se permite debido a que setX es amiga de Cuenta
33 } // fin de la función setX
34
35 int main()
```

Figura 10.15 | Las funciones `friend` pueden acceder a los miembros `private` de una clase. (Parte I de 2).

```

36  {
37      Cuenta contador; // crea objeto Cuenta
38
39      cout << "contador.x despues de crear la instancia: ";
40      contador.imprimir();
41
42      setX( contador, 8 ); // establece x usando una función friend
43      cout << "contador.x despues de la llamada a la función friend setX: ";
44      contador.imprimir();
45
46  } // fin de main

contador.x despues de crear la instancia: 0
contador.x despues de la llamada a la función friend setX: 8

```

Figura 10.15 | Las funciones **friend** pueden acceder a los miembros **private** de una clase. (Parte 2 de 2).

La función **setX** (líneas 30 a 33) es una función individual estilo C; no es una función miembro de la clase **Cuenta**. Por esta razón, cuando **setX** se invoca para el objeto **contador**, en la línea 42 se pasa **contador** como argumento a **setX**, en vez de usar un manejador (como el nombre del objeto) para llamar a la función, como en

```
contador.setX( 8 );
```

Como dijimos antes, la figura 10.15 es un ejemplo mecánico acerca del uso de la instrucción **friend**. Por lo general sería apropiado definir la función **setX** como función miembro de la clase **Cuenta**. También sería generalmente apropiado separar el programa de la figura 10.15 en tres archivos:

1. Un archivo de encabezado (por ejemplo, **Cuenta.h**) que contenga la definición de la clase **Cuenta**, que a su vez contenga el prototipo de la función **friend** llamada **setX**.
2. Un archivo de implementación (por ejemplo, **Cuenta.cpp**) que contenga las definiciones de las funciones miembro de la clase **Cuenta** y la definición de la función **friend** llamada **setX**.
3. Un programa de prueba (por ejemplo, **fig10_15.cpp**) con **main**.

*Intento erróneo de modificar un miembro **private** con una función no **friend***

La figura 10.16 demuestra los mensajes de error producidos por el compilador cuando se hace una llamada a la función **noPuedeSetX** (líneas 29 a 32) para modificar el miembro de datos **private** llamado **x**.

```

1 // Fig. 10.16: fig10_16.cpp
2 // Las funciones que no son friend ni miembro no pueden acceder a los datos privados de una
3 // clase.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 // definición de la clase Cuenta (observe que no hay declaración de amistad)
9 class Cuenta
10 {
11 public:
12     // constructor
13     Cuenta()
14     : x( 0 ) // inicializa x en 0
15     {
16         // cuerpo vacío
17     } // fin del constructor de Cuenta
18
19     // imprime x
20     void print() const

```

Figura 10.16 | Las funciones no **friend**/no miembro no pueden acceder a los miembros **private**. (Parte 1 de 2).

```

20      {
21          cout << x << endl;
22      } // fin de la función imprimir
23 private:
24     int x; // miembro de datos
25 }; // fin de la clase Cuenta
26
27 // La función noPuedeSetX trata de modificar los datos private de Cuenta,
28 // pero no puede debido a que la función no es amiga de Cuenta
29 void noPuedeSetX( Cuenta &c, int val )
30 {
31     c.x = val; // ERROR: no puede acceder al miembro de datos private en Cuenta
32 } // fin de la función noPuedeSetX
33
34 int main()
35 {
36     Cuenta contador; // crea un objeto Cuenta
37
38     noPuedeSetX( contador, 3 ); // noPuedeSetX no es amiga
39     return 0;
40 } // fin de main

```

Mensaje de error en la línea de comandos del compilador Borland C++:

```
Error E2247 Fig10_16/fig10_16.cpp 31: 'Cuenta::x' is not accesible in
function noPuedeSetX(Cuenta &,int)
```

Mensajes de error del compilador Microsoft Visual C++ 2005:

```
C:\cpphtp6_ejemplos\cap10\Fig10_16\fig10_16.cpp(31) : error C2248: 'Cuenta::x'
: cannot Access private member declared in class 'Cuenta'
    C:\cpphtp6_ejemplos\cap10\Fig10_16\fig10_16.cpp(24) : see declaration of 'Cuenta::x'
    C:\cpphtp6_ejemplos\cap10\Fig10_16\fig10_16.cpp(9) : see declaration of 'Cuenta'
```

Mensajes de error del compilador GNU C++:

```
fig10_16.cpp:24: error: 'int Cuenta::x' is private
fig10_16.cpp:31: error: within this context
```

Figura 10.16 | Las funciones no friend/no miembro no pueden acceder a los miembros **private**. (Parte 2 de 2).

Es posible especificar funciones sobrecargadas como **amigas** de una clase. Cada función sobrecargada destinada para ser **friend** debe declararse explícitamente en la definición de la clase como una **amiga** de ésta.

10.5 Uso del apuntador **this**

Hemos visto que las funciones miembro de un objeto pueden manipular los datos de éste. ¿Cómo saben las funciones miembro **cuáles** datos miembro del objeto deben manipular? Cada objeto tiene acceso a su propia dirección a través de un apuntador llamado **this** (una palabra clave de C++). El apuntador **this** de un objeto *no* es parte del objeto en sí; es decir, el tamaño de la memoria ocupada por el apuntador **this** no se refleja en el resultado de una operación **sizeof** en el objeto. En vez de ello, el apuntador **this** se pasa (por el compilador) como un argumento implícito para cada una de las funciones miembro no **static** del objeto. En la sección 10.7 se introducen los miembros de clase **static** y se explica por qué el apuntador **this** *no* se pasa implícitamente a las funciones miembro **static**.

Los objetos utilizan el apuntador **this** de manera implícita (como hemos hecho hasta este punto) o explícitamente para hacer referencia a sus datos miembro y funciones miembro. El tipo del apuntador **this** depende del tipo del objeto y si la función miembro en la que se utiliza **this** se declara **const**. Por ejemplo, en una función miembro no constante de la clase **Empleado**, el apuntador **this** tiene el tipo **Empleado * const** (un apuntador constante a un objeto **Empleado** no constante). En una función miembro constante de la clase **Empleado**, el apuntador **this** tiene el tipo de datos **const Empleado *const** (un apuntador constante a un objeto **Empleado** constante).

El siguiente ejemplo muestra el uso implícito y explícito del apuntador **this**; más adelante en este capítulo y en el capítulo 11, mostraremos algunos ejemplos sustanciales y sutiles acerca del uso de **this**.

Uso implícito y explícito del apuntador this para acceder a los datos miembro de un objeto

La figura 10.17 demuestra el uso implícito y explícito del apuntador **this** para permitir a una función miembro de la clase Prueba imprimir los datos **private x** de un objeto Prueba.

```

1 // Fig. 10.17: fig10_17.cpp
2 // Uso del apuntador this para hacer referencia a los miembros de un objeto.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 class Prueba
8 {
9 public:
10     Prueba( int = 0 ); // constructor predeterminado
11     void imprimir() const;
12 private:
13     int x;
14 } // fin de la clase Prueba
15
16 // constructor
17 Prueba::Prueba( int valor )
18     : x( valor ) // inicializa x con valor
19 {
20     // cuerpo vacío
21 } // fin del constructor de Prueba
22
23 // imprime x usando los apuntadores implícito y explícito;
24 // los paréntesis alrededor de *this son obligatorios
25 void Prueba::imprimir() const
26 {
27     // usa de manera implícita el apuntador this para acceder al miembro x
28     cout << "      x = " << x;
29
30     // usa de manera explícita el apuntador this y el operador flecha
31     // para acceder a la x del miembro
32     cout << "\n  this->x = " << this->x;
33
34     // usa de manera explícita el apuntador this desreferenciado y
35     // el operador punto para acceder a la x del miembro
36     cout << "\n(*this).x = " << ( *this ).x << endl;
37 } // fin de la función imprimir
38
39 int main()
40 {
41     Prueba objetoPrueba( 12 ); // instancia e inicializa objetoPrueba
42
43     objetoPrueba.imprimir();
44     return 0;
45 } // fin de main

```

```

x = 12
this->x = 12
(*this).x = 12

```

Figura 10.17 | Apuntador this que accede de manera implícita y explícita a los miembros de un objeto.

Para fines ilustrativos, la función miembro **imprimir** (líneas 25 a 37) primero imprime **x** mediante el uso del apuntador **this** de manera implícita (línea 28); sólo se especifica el nombre del miembro de datos. Después, **imprimir** usa dos notaciones distintas para acceder a **x** mediante el apuntador **this**: el operador flecha (**->**) del apuntador **this** (línea 32) y el operador punto (**.**) del apuntador **this** desreferenciado (línea 36).

Observe los paréntesis alrededor de `*this` (línea 36) cuando se utilizan con el operador punto (`.`) de selección de miembros. Los paréntesis son obligatorios debido a que el operador punto tiene mayor precedencia que el operador `*`. Sin los paréntesis, la expresión `*this.x` se evaluaría como `*(this.x)`, lo cual es un error de compilación, ya que el operador punto no se puede utilizar con un apuntador.



Error común de programación 10.7

Tratar de usar el operador `(.)` de selección de miembros con un apuntador a un objeto es un error de compilación; el operador punto de selección de miembros sólo se puede usar con un lvalue tal como el nombre de un objeto, una referencia a un objeto o un apuntador desreferenciado a un objeto.

Un uso interesante del apuntador `this` es para evitar que un objeto se asigne a sí mismo. Como veremos en el capítulo 11, la auto-asignación puede producir errores graves cuando el objeto contiene apuntadores a almacenamiento asignado en forma dinámica.

Uso del apuntador `this` para permitir llamadas en cascada a funciones

Otro uso del apuntador `this` es para permitir las **llamadas en cascada a funciones miembro**; es decir, invocar varias funciones en la misma instrucción (como en la línea 14 de la figura 10.20). El programa de las figuras 10.18 a 10.20 modifica las funciones `set`, `setTiempo`, `setHora`, `setMinuto` y `setSegundo` de la clase `Tiempo`, de manera que cada una devuelva una referencia a un objeto `Tiempo` para permitir las llamadas en cascada a funciones miembro. Observe en la figura 10.19 que la última instrucción en el cuerpo de cada una de estas funciones miembro devuelve `*this` (líneas 26, 33, 40 y 47) en un tipo de valor de retorno de `Tiempo &`.

```

1 // Fig. 10.18: Tiempo.h
2 // Llamadas en cascada a funciones miembro.
3
4 // Definición de la clase Tiempo.
5 // Las funciones miembro se definen en Tiempo.cpp.
6 #ifndef TIEMPO_H
7 #define TIEMPO_H
8
9 class Tiempo
10 {
11 public:
12     Tiempo( int = 0, int = 0, int = 0 ); // constructor predeterminado
13
14     // funciones "set" (los tipos de valores de retorno Tiempo & permiten las llamadas en
15     // cascada)
15     Tiempo &setTiempo( int, int, int ); // set hora, minuto, segundo
16     Tiempo &setHora( int ); // establece la hora
17     Tiempo &setMinuto( int ); // establece el minuto
18     Tiempo &setSegundo( int ); // establece el segundo
19
20     // funciones "get" (por lo general se declaran const)
21     int getHora() const; // devuelve la hora
22     int getMinuto() const; // devuelve el minuto
23     int getSegundo() const; // devuelve el segundo
24
25     // funciones para imprimir (por lo general se declaran const)
26     void imprimirUniversal() const; // imprime el tiempo universal
27     void imprimirEstandar() const; // imprime el tiempo estándar
28 private:
29     int hora; // 0 - 23 (formato de reloj de 24 horas)
30     int minuto; // 0 - 59
31     int segundo; // 0 - 59
32 }; // fin de clase Tiempo
33
34 #endif

```

Figura 10.18 | Definición de la clase `Tiempo` modificada para permitir las llamadas en cascada a funciones miembro.

```

1 // Fig. 10.19: Tiempo.cpp
2 // Definiciones de las funciones miembro de la clase Tiempo.
3 #include <iostream>
4 using std::cout;
5
6 #include <iomanip>
7 using std::setfill;
8 using std::setw;
9
10 #include "Tiempo.h" // Definición de la clase Tiempo
11
12 // función constructor para inicializar los datos privados;
13 // llama a la función miembro setTiempo para establecer las variables;
14 // los valores predeterminados son 0 (vea la definición de la clase)
15 Tiempo::Tiempo( int hr, int min, int seg )
16 {
17     setTiempo( hr, min, seg );
18 } // fin del constructor de Tiempo
19
20 // establece los valores de hora, minuto y segundo
21 Tiempo &Tiempo::setTiempo( int h, int m, int s ) // observe Tiempo & return
22 {
23     setHora( h );
24     setMinuto( m );
25     setSegundo( s );
26     return *this; // permite las llamadas en cascada
27 } // fin de la función setTiempo
28
29 // establece el valor de hora
30 Tiempo &Tiempo::setHora( int h ) // observe Tiempo & return
31 {
32     hora = ( h >= 0 && h < 24 ) ? h : 0; // valida la hora
33     return *this; // permite las llamadas en cascada
34 } // fin de la función setHora
35
36 // establece el valor de minuto
37 Tiempo &Tiempo::setMinuto( int m ) // observe Tiempo & return
38 {
39     minuto = ( m >= 0 && m < 60 ) ? m : 0; // valida el minuto
40     return *this; // permite las llamadas en cascada
41 } // fin de la función setMinuto
42
43 // establece el valor de segundo
44 Tiempo &Tiempo::setSegundo( int s ) // observe Tiempo & return
45 {
46     segundo = ( s >= 0 && s < 60 ) ? s : 0; // valida el segundo
47     return *this; // permite las llamadas en cascada
48 } // fin de la función setSegundo
49
50 // obtiene el valor de hora
51 int Tiempo::getHora() const
52 {
53     return hora;
54 } // fin de la función getHora
55
56 // obtiene el valor de minuto
57 int Tiempo::getMinuto() const
58 {
59     return minuto;
60 } // fin de la función getMinuto

```

Figura 10.19 | Definiciones de las funciones miembro de la clase `Tiempo`, modificadas para permitir las llamadas en cascada a funciones miembro. (Parte 1 de 2).

```

61 // obtiene el valor de segundo
62 int Tiempo::getSegundo() const
63 {
64     return segundo;
65 } // fin de la función getSegundo
66
67 // imprime el Tiempo en formato universal (HH:MM:SS)
68 void Tiempo::imprimirUniversal() const
69 {
70     cout << setfill( '0' ) << setw( 2 ) << hora << ":" 
71         << setw( 2 ) << minuto << ":" << setw( 2 ) << segundo;
72 } // fin de la función imprimirUniversal
73
74 // imprime el Tiempo en formato estándar (HH:MM:SS AM o PM)
75 void Tiempo::imprimirEstandar() const
76 {
77     cout << ( ( hora == 0 || hora == 12 ) ? 12 : hora % 12 )
78         << ":" << setfill( '0' ) << setw( 2 ) << minuto
79         << ":" << setw( 2 ) << segundo << ( hora < 12 ? " AM" : " PM" );
80 } // fin de la función imprimirEstandar

```

Figura 10.19 | Definiciones de las funciones miembro de la clase `Tiempo`, modificadas para permitir las llamadas en cascada a funciones miembro. (Parte 2 de 2).

```

1 // Fig. 10.20: fig10_20.cpp
2 // Llamadas en cascada a funciones miembro con el apuntador this.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Tiempo.h" // definición de la clase Tiempo
8
9 int main()
10 {
11     Tiempo t; // crea un objeto Tiempo
12
13     // Llamadas en cascada a funciones
14     t.setHora( 18 ).setMinuto( 30 ).setSegundo( 22 );
15
16     // imprime el tiempo en los formatos universal y estándar
17     cout << "Tiempo universal: ";
18     t.imprimirUniversal();
19
20     cout << "\nTiempo estandar: ";
21     t.imprimirEstandar();
22
23     cout << "\n\nNuevo tiempo estandar: ";
24
25     // Llamadas en cascada a funciones
26     t.setTiempo( 20, 20, 20 ).imprimirEstandar();
27     cout << endl;
28     return 0;
29 } // fin de main

```

```

Tiempo universal: 18:30:22
Tiempo estandar: 6:30:22 PM

Nuevo tiempo estandar: 8:20:20 PM

```

Figura 10.20 | Llamadas en cascada a funciones miembro con el apuntador `this`.

El programa de la figura 10.20 crea el objeto `Tiempo` llamado `t` (línea 11), y después lo utiliza en las llamadas en cascada a funciones miembro (líneas 14 y 26). ¿Por qué funciona la técnica de devolver `*this` como referencia? El operador punto (`.`) asocia de izquierda a derecha, por lo que la línea 14 evalúa primero a `t.setHora(18)`, y después devuelve una referencia al objeto `t` como el valor de la llamada a esta función. Luego, el resto de la expresión se interpreta de la siguiente manera:

```
t.setMinuto( 30 ).setSegundo( 22 );
```

La llamada a `t.setMinuto(30)` se ejecuta y devuelve una referencia al objeto `t`. El resto de la expresión se interpreta así:

```
t.setSegundo( 22 );
```

En la línea 26 también se usan las llamadas en cascada. Estas llamadas deben aparecer en el orden que se muestra en la línea 26, ya que la función `imprimirEstandar`, según su definición en la clase, no devuelve una referencia a `t`. Al colocar la llamada a `imprimirEstandar` antes de la llamada a `setTiempo` en la línea 26 se produce un error de compilación. En el capítulo 11 presentaremos varios ejemplos prácticos acerca del uso de llamadas en cascada a funciones. Uno de esos ejemplos utiliza varios operadores `<<` con `cout` para imprimir múltiples valores en una sola instrucción.

10.6 Administración dinámica de memoria con los operadores `new` y `delete`

C++ permite a los programadores controlar la asignación y desasignación de memoria en un programa, para cualquier tipo integrado o definido por el usuario. Esto se conoce como **administración dinámica de memoria** y se lleva a cabo mediante los operadores `new` y `delete`. Recuerde que la clase `Empleado` (figuras 10.12 y 10.13) usa dos arreglos de 25 caracteres para representar el primer nombre y el apellido paterno de un `Empleado`. La definición de la clase `Empleado` (figura 10.12) debe especificar el número de elementos en cada uno de estos arreglos cuando los declara como miembros de datos, ya que el tamaño de estos datos miembro indica la cantidad de memoria requerida para almacenar un objeto `Empleado`. Como vimos antes, estos arreglos pueden desperdiciar espacio para nombres con menos de 24 caracteres. Además, los nombres mayores de 24 caracteres deben truncarse para caber en estos arreglos de tamaño fijo.

¿No sería estupendo si pudiéramos usar arreglos que contuvieran exactamente el número de elementos necesarios para almacenar el primer nombre y apellido paterno de un `Empleado`? La asignación dinámica de memoria nos permite hacer eso exactamente. Como veremos en el ejemplo de la sección 10.7, si reemplazamos los datos miembro tipo arreglo llamados `primerNombre` y `apellidoPaterno` con apuntadores a `char`, podemos usar el operador `new` para asignar (reservar) en forma dinámica la cantidad exacta de memoria requerida para contener cada nombre en tiempo de ejecución. La asignación dinámica de memoria de esta forma hace que se cree un arreglo (o cualquier otro tipo integrado o definido por el usuario) en el **almacenamiento libre** (algunas veces conocido como el “heap” o **montón**): una región de memoria asignada a cada programa para almacenar los objetos que se asignan en forma dinámica. Una vez que se asigna la memoria para un arreglo en el almacenamiento libre, podemos obtener acceso a éste si apuntamos un apuntador al primer elemento del arreglo. Cuando ya no necesitemos el arreglo, podemos devolver la memoria al almacenamiento libre mediante el uso del operador `delete` para desasignar (liberar) la memoria, que las operaciones posteriores con `new` pueden reutilizar.

De nuevo, en el ejemplo de la sección 10.7 presentamos la clase `Empleado` modificada como se describe aquí. Primero, presentamos los detalles acerca del uso de los operadores `new` y `delete` para asignar la memoria en forma dinámica y almacenar objetos, tipos fundamentales y arreglos.

Considere la siguiente declaración e instrucción:

```
Tiempo *tiempoPtr;
tiempoPtr = new Tiempo;
```

El operador `new` asigna el almacenamiento del tamaño apropiado para un objeto de tipo `Tiempo`, llama al constructor predeterminado para inicializar el objeto y devuelve un apuntador al tipo especificado a la derecha del operador `new` (es decir, un `Tiempo *`). Observe que se puede utilizar `new` para asignar en forma dinámica cualquier tipo fundamental (como `int` o `double`) o el tipo de clase. Si `new` no puede encontrar suficiente espacio en memoria para el objeto, indica que ocurrió un error “generando una excepción”. En el capítulo 16, Manejo de excepciones, veremos cómo lidiar con las fallas de `new` en el contexto del estándar ISO/IEC de C++. En especial, mostraremos cómo “atravar” la excepción generada por `new` y lidiar con ella. Cuando un programa no “atrava” una excepción, termina de inmediato.

Tip de portabilidad 10.1



Al fallar, el operador `new` devuelve un apuntador a 0 en versiones de C++ anteriores al estándar ISO/IEC. Nosotros utilizaremos la versión estándar del operador `new` a lo largo de este libro.

Para destruir un objeto asignado en forma dinámica y liberar el espacio que éste ocupa, usamos el operador `delete` de la siguiente manera:

```
delete tiempoPtr;
```

Esta instrucción llama primero al destructor para el objeto al que apunta `tiempoPtr`, y después desasigna la memoria asociada con el objeto. Después de la instrucción anterior, el sistema puede reutilizar la memoria para asignar otros objetos.

Error común de programación 10.8



Si no se libera la memoria asignada en forma dinámica cuando ya no es necesaria, el sistema se puede quedar sin memoria antes de tiempo. A esto se le conoce algunas veces como “fuga de memoria”.

El programador puede proporcionar un **inicializador** para una variable de tipo fundamental recién creada, como en

```
double *ptr = new double( 3.14159 );
```

la cual inicializa un valor `double` recién creado con 3.14159 y asigna el apuntador resultante a `ptr`. La misma sintaxis se puede utilizar para especificar una lista de argumentos separados por comas para el constructor de un objeto. Por ejemplo,

```
Tiempo *tiempoPtr = new Tiempo( 12, 45, 0 );
```

inicializa un objeto `Tiempo` recién creado con 12:45 PM y asigna el apuntador resultante a `tiempoPtr`.

Como vimos antes, el operador `new` se puede utilizar para asignar arreglos en forma dinámica. Por ejemplo, un arreglo entero de 10 elementos se puede asignar a `arregloCalificaciones` de la siguiente manera:

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

esta instrucción declara el apuntador `int` llamado `arregloCalificaciones` y le asigna un apuntador al primer elemento de un arreglo de 10 elementos `int` asignado en forma dinámica. Recuerde que el tamaño de un arreglo creado en tiempo de compilación se debe especificar mediante una expresión integral constante. Sin embargo, el tamaño de un arreglo asignado en forma dinámica se puede especificar mediante *cualquier* expresión integral no negativa que se pueda evaluar en tiempo de ejecución. Observe además que, al asignar un arreglo de objetos en forma dinámica, no se pueden pasar argumentos al constructor de cada objeto. En vez de ello, cada objeto del arreglo se inicializa mediante su constructor predeterminado. Para eliminar el arreglo asignado en forma dinámica al que apunta `arregloCalificaciones`, use la siguiente instrucción:

```
delete [] arregloCalificaciones;
```

La instrucción anterior desasigna el arreglo al que apunta `arregloCalificaciones`. Si el apuntador en la instrucción anterior apunta a un arreglo de objetos, la instrucción primero llama al destructor para cada objeto en el arreglo, y después desasigna la memoria. Si la instrucción anterior no incluyera los corchetes (`[]`) y `arregloCalificaciones` apuntara a un arreglo de objetos, el resultado sería indefinido. Algunos compiladores llaman al destructor sólo para el primer objeto en el arreglo. El uso de `delete` en un apuntador nulo (es decir, un apuntador con el valor 0) no tiene efecto.

Error común de programación 10.9



El uso de `delete` en vez de `delete []` para los arreglos de objetos puede provocar errores lógicos en tiempo de ejecución. Para asegurarse que cada objeto en el arreglo reciba una llamada al destructor, siempre debe eliminar la memoria asignada como un arreglo con el operador `delete []`. De manera similar, siempre debe eliminar la memoria asignada como un elemento individual con el operador `delete`; en caso contrario, el resultado de la operación es indefinido.

10.7 Miembros de clase static

Hay una importante excepción a la regla que establece que cada objeto de una clase tiene su propia copia de todos los datos miembro de la misma. En ciertos casos, todos los objetos de una clase sólo deben compartir una copia de una variable. Por ésta y otras razones, se utiliza un **miembro de datos static**. Dicha variable representa información “a nivel de clase” (es decir, una propiedad de la clase compartida por todas las instancias, no una propiedad de un objeto específico de la clase). La declaración de un miembro `static` empieza con la palabra clave `static`. Recuerde que las versiones de la clase `LibroCalificaciones` en el capítulo 7 utilizan datos miembro `static` para almacenar constantes que representan el número de calificaciones que pueden contener todos los objetos `LibroCalificaciones`.

Vamos a esclarecer más la necesidad de datos `static` a nivel de clase con un ejemplo. Suponga que tenemos un videojuego con objetos `Marciano` y otras criaturas espaciales. Cada `Marciano` tiende a ser valiente y deseoso de atacar a otras criaturas espaciales cuando está consciente de que hay por lo menos cinco objetos `Marciano` presentes. Si hay menos de cinco, cada `Marciano` se vuelve cobarde. Por lo tanto, cada `Marciano` necesita conocer la `cuentaDeMarcianos`. Podríamos investir a cada instancia de la clase `Marciano` con `cuentaDeMarcianos` como datos miembro. Si lo hacemos, cada `Marciano` tendrá una copia separada de los datos miembro. Cada vez que creemos un nuevo `Marciano`, tendremos que actualizar los datos miembro `cuentaDeMarcianos` en todos los objetos `Marciano`. Para ello cada objeto `Marciano` tendría que tener acceso a los manejadores de todos los demás objetos `Marciano` en la memoria. Esto desperdicia espacio con las copias redundantes, y tiempo para actualizar las copias separadas. En vez de ello, declaramos a `cuentaDeMarcianos` como `static`. Esto convierte a `cuentaDeMarcianos` en datos a nivel de clase. Cada `Marciano` puede acceder a `cuentaDeMarcianos` como si fuera un miembro de datos del `Marciano`, pero C++ sólo mantiene una copia de la variable `static` `cuentaDeMarcianos`. Esto ahorra espacio. Ahorramos tiempo al hacer que el constructor de `Marciano` incremente la variable `static` `cuentaDeMarcianos`, y al hacer que el destructor de `Marciano` decremente `cuentaDeMarcianos`. Como sólo hay una copia, no tenemos que incrementar o decrementar copias separadas de `cuentaDeMarcianos` para cada objeto `Marciano`.



Tip de rendimiento 10.3

Use datos miembro static para ahorrar almacenamiento cuando sea suficiente con una sola copia de los datos para todos los objetos de una clase.

Aunque pueden parecer variables globales, los datos miembro `static` de una clase tienen alcance de clase. Además, los miembros `static` se pueden declarar `public`, `private` o `protected`. Un miembro de datos `static` de tipo fundamental se inicializa de manera predeterminada con 0. Si desea un valor inicial distinto, un miembro de datos `static` se puede inicializar *una vez* (y sólo una). Un miembro de datos `const static` de tipo `int` o `enum` se puede inicializar en su declaración en la definición de la clase. Sin embargo, todos los demás datos miembro `static` se deben definir en alcance de archivo (es decir, fuera del cuerpo de la definición de la clase) y sólo se pueden inicializar en esas definiciones. Observe que los datos miembro `static` de los tipos de clases (es decir, objetos miembro `static`) que tienen constructores predeterminados no necesitan inicializarse, ya que se llamará a sus constructores predeterminados.

Por lo general, se accede a los miembros `static private` y `protected` de una clase a través de las funciones miembro `public` de la misma, o a través de funciones `friend` de la clase. (En el capítulo 12, veremos que también se puede acceder a los miembros `static private` y `protected` de una clase a través de las funciones miembro `protected` de la clase). Los miembros `static` de una clase existen aun y cuando no existan objetos de la clase. Para acceder al miembro `public static` de una clase cuando no existen objetos de esa clase, simplemente se antepone el nombre de la clase y el operador de resolución de ámbito binario (`::`) al nombre del miembro de datos. Por ejemplo, si nuestra variable anterior `cuentaDeMarcianos` es `public`, se puede utilizar con la expresión `Marciano::cuentaDeMarcianos` cuando no haya objetos `Marciano`. (Desde luego que no se recomienda el uso de datos `public`).

También se puede acceder a los miembros `public static` de una clase a través de cualquier objeto de esa clase, usando el nombre del objeto, el operador punto y el nombre del miembro (por ejemplo, `miMarciano.cuentaDeMarcianos`). Para acceder a un miembro `private` o `protected` de la clase cuando no existen objetos de la misma, el programador debe proporcionar una función miembro `static public` y debe llamar a la función, anteponiendo a su nombre el nombre de la clase y el operador de resolución de ámbito binario. (Como veremos en el capítulo 12, una función miembro `protected static` puede servir a este fin también). Una función miembro `static` es un servicio de la clase, no un objeto específico de la misma.



Observación de Ingeniería de Software 10.11

Los datos miembro static y las funciones miembro static de una clase existen, y se pueden usar aun si no se han instanciado objetos de esa clase.

El programa de las figuras 10.21 a 10.23 demuestra el uso de un miembro de datos `private static` llamado `cuenta` (figura 10.21, línea 21) y una función miembro `public static` llamada `getCuenta` (figura 10.21, línea 15). En la figura 10.22, la línea 14 define e inicializa el miembro de datos `cuenta` con cero en *alcance de archivo*, y en las líneas 18 a 21 se define la función miembro `static` llamada `getCuenta`. Observe que ni la línea 14 ni la 18 incluyen la palabra clave `static`, pero de todas formas ambas líneas se refieren a los miembros `static` de la clase. Cuando se aplica `static` a un elemento con alcance de archivo, ese elemento sólo se vuelve conocido en ese archivo. Los miembros `static` de la clase necesitan estar disponibles desde cualquier código cliente que acceda al archivo, por lo cual no podemos declararlos

`static` en el archivo .cpp; los declaramos `static` sólo en el archivo .h. El miembro de datos `cuenta` mantiene un conteo del número de objetos de la clase `Empleado` que se han instanciado. Cuando existen objetos de la clase `Empleado`, el miembro `cuenta` se puede referenciar a través de cualquier función miembro de un objeto `Empleado`; en la figura 10.22, se hace referencia a `cuenta` tanto en la línea 33 en el constructor y en la línea 48 en el destructor. Además, observe que como `cuenta` es un valor `int`, se podría inicializar en el archivo de encabezado en la línea 21 de la figura 10.21.



Error común de programación 10.10

Incluir la palabra clave `static` en la definición de los datos miembro `static` en alcance de archivo es un error de compilación.

```

1 // Fig. 10.21: Empleado.h
2 // Definición de la clase Empleado.
3 #ifndef EMPLEADO_H
4 #define EMPLEADO_H
5
6 class Empleado
7 {
8 public:
9     Empleado( const char * const, const char * const ); // constructor
10    ~Empleado(); // destructor
11    const char *getPrimerNombre() const; // devuelve el primer nombre
12    const char *getApellidoPaterno() const; // devuelve el apellido paterno
13
14    // función miembro static
15    static int getCuenta(); // devuelve el número de objetos instanciados
16 private:
17    char *primerNombre;
18    char *apellidoPaterno;
19
20    // datos static
21    static int cuenta; // número de objetos instanciados
22 }; // fin de la clase Empleado
23
24 #endif

```

Figura 10.21 | Definición de la clase `Empleado` con datos miembro `static` para rastrear el número de objetos `Empleado` en la memoria.

En la figura 10.22, observe el uso del operador `new` (líneas 27 y 30) en el constructor de `Empleado` para asignar en forma dinámica la cantidad correcta de memoria para los miembros `primerNombre` y `apellidoPaterno`. Si el operador `new` no puede completar la petición de memoria para uno o ambos arreglos de caracteres, el programa terminará de inmediato. En el capítulo 16, proporcionaremos un mejor mecanismo para tratar con casos en los que `new` no puede asignar memoria.

Observe además en la figura 10.22 que las implementaciones de las funciones `getPrimerNombre` (líneas 52 a 58) y `getApellidoPaterno` (líneas 61 a 67) devuelven apuntadores a datos `const` tipo carácter. En esta implementación, si el cliente desea retener una copia del primer nombre o del apellido paterno, es responsable de copiar la memoria asignada en forma dinámica al objeto `Empleado`, después de obtener el apuntador a datos `const` tipo carácter del objeto. También es posible implementar `getPrimerNombre` y `getApellidoPaterno`, de manera que el cliente tenga que pasar un arreglo de caracteres y el tamaño del mismo a cada función. Después las funciones podrían copiar el primer nombre o el apellido paterno en el arreglo de caracteres proporcionado por el cliente. Una vez más, observe que podríamos haber usado la clase `string` aquí para devolver una copia de un objeto `string` a la función que hace la llamada, en vez de devolver un apuntador a los datos `private`.

```

1 // Fig. 10.22: Empleado.cpp
2 // Definiciones de las funciones miembro de la clase Empleado.
3 #include <iostream>
4 using std::cout;
5 using std::endl;

```

Figura 10.22 | Definiciones de las funciones miembro de la clase `Empleado`. (Parte I de 2).

```

6
7 #include <cstring> // prototipos de strlen y strcpy
8 using std::strlen;
9 using std::strcpy;
10
11 #include "Empleado.h" // definición de la clase Empleado
12
13 // define e inicializa el miembro de datos static en alcance de archivo
14 int Empleado::cuenta = 0; // no puede incluir la palabra clave static
15
16 // define la función miembro static que devuelve el número de
17 // objetos Empleado instanciados (se declara static en Empleado.h)
18 int Empleado::getCuenta()
19 {
20     return cuenta;
21 } // fin de la función static getCuenta
22
23 // el constructor asigna espacio en forma dinámica para nombre y apellido, y
24 // usa strcpy para copiar nombre y apellido en el objeto
25 Empleado::Empleado( const char * const nombre, const char * const apellido )
26 {
27     primerNombre = new char[ strlen( nombre ) + 1 ]; // crea espacio
28     strcpy( primerNombre, nombre ); // copia nombre en el objeto
29
30     apellidoPaterno = new char[ strlen( apellido ) + 1 ]; // crea espacio
31     strcpy( apellidoPaterno, apellido ); // copia apellido en el objeto
32
33     cuenta++; // incrementa la cuenta static de empleados
34
35     cout << "Se llamo al constructor de Empleado para " << primerNombre
36         << ' ' << apellidoPaterno << "." << endl;
37 } // fin del constructor de Empleado
38
39 // el destructor desasigna la memoria asignada en forma dinámica
40 Empleado::~Empleado()
41 {
42     cout << "Se llamo a ~Empleado() para " << primerNombre
43         << ' ' << apellidoPaterno << endl;
44
45     delete [] primerNombre; // libera la memoria
46     delete [] apellidoPaterno; // libera la memoria
47
48     cuenta--; // decrementa cuenta static de empleados
49 } // fin del destructor ~Empleado
50
51 // devuelve el primer nombre del empleado
52 const char *Empleado::getPrimerNombre() const
53 {
54     // const antes del tipo de retorno evita que el cliente modifique datos
55     // private; el cliente debe copiar la cadena devuelta antes de que el
56     // destructor elimine el almacenamiento para evitar un apuntador indefinido
57     return primerNombre;
58 } // fin de la función getPrimerNombre
59
60 // devuelve el apellido paterno del empleado
61 const char *Empleado::getApellidoPaterno() const
62 {
63     // const antes del tipo de retorno evita que el cliente modifique
64     // los datos privados; el cliente debe copiar la cadena devuelta antes
65     // que el destructor elimine el almacenamiento para evitar un apuntador indefinido
66     return apellidoPaterno;
67 } // fin de la función getApellidoPaterno

```

Figura 10.22 | Definiciones de las funciones miembro de la clase Empleado. (Parte 2 de 2).

La figura 10.23 utiliza la función miembro `static getConta` para determinar el número de objetos `Empleado` que se encuentran instanciados en un momento dado. Observe que, cuando no hay objetos instanciados en el programa, se genera la llamada a la función `Empleado::getConta()` (líneas 14 y 38). Sin embargo, cuando hay objetos instanciados, la función puede llamarse a través de cualquiera de los objetos, como se muestra en la instrucción de las líneas 22 y 23, en la que se utiliza el apuntador `e1Ptr` para invocar a la función `getConta`. Observe que el uso de `e2Ptr->getConta()` o de `Empleado::getConta()` en la línea 23 produciría el mismo resultado, ya que `getConta` siempre accede al mismo miembro `static cuenta`.

```

1 // Fig. 10.23: fig10_23.cpp
2 // Miembro de datos static que rastrea el número de objetos de una clase.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Empleado.h" // definición de la clase Empleado
8
9 int main()
10 {
11     // usa el nombre de la clase y el operador de resolución de ámbito binario
12     // para acceder a la función numérica static getConta
13     cout << "El numero de empleados antes de instanciar cualquier objeto es "
14         << Empleado::getConta() << endl; // usa el nombre de la clase
15
16     // usa new para crear en forma dinámica dos nuevos objetos Empleado
17     // el operador new también llama al constructor del objeto
18     Empleado *e1Ptr = new Empleado( "Susan", "Baker" );
19     Empleado *e2Ptr = new Empleado( "Robert", "Jones" );
20
21     // llama a getConta en el objeto Empleado nombre
22     cout << "El numero de empleados despues de instanciar los objetos es "
23         << e1Ptr->getConta();
24
25     cout << "\n\nEmpleado 1: "
26         << e1Ptr->getPrimerNombre() << " " << e1Ptr->getApellidoPaterno()
27         << "\nEmpleado 2: "
28         << e2Ptr->getPrimerNombre() << " " << e2Ptr->getApellidoPaterno() << "\n\n";
29
30     delete e1Ptr; // desasigna la memoria
31     e1Ptr = 0; // desconecta el apuntador del espacio de almacenamiento libre
32     delete e2Ptr; // desasigna la memoria
33     e2Ptr = 0; // desconecta el apuntador del espacio de almacenamiento libre
34
35     // no existen objetos, por lo que llama a la función miembro static getConta de nuevo
36     // usando el nombre de la clase y el operador de resolución de ámbito binario
37     cout << "El numero de empleados despues de eliminar los objetos es "
38         << Empleado::getConta() << endl;
39     return 0;
40 } // fin de main

```

El número de empleados antes de instanciar cualquier objeto es 0

Se llama al constructor de Empleado para Susan Baker.

Se llama al constructor de Empleado para Robert Jones.

El numero de empleados despues de instanciar los objetos es 2

Empleado 1: Susan Baker

Empleado 2: Robert Jones

Se llama a ~Empleado() para Susan Baker

Se llama a ~Empleado() para Robert Jones

El numero de empleados despues de eliminar los objetos es 0

Figura 10.23 | Datos miembro static que rastrea el número de objetos de una clase.



Observación de Ingeniería de Software 10.12

Algunas organizaciones especifican en sus estándares de ingeniería de software que todas las llamadas a las funciones miembro static se deben realizar usando el nombre de la clase, en vez de un manejador de objetos.

Una función miembro debe declararse como `static` si no accede a los datos miembro no `static` o a las funciones miembro no `static` de la clase. A diferencia de las funciones miembro no `static`, una función miembro `static` no tiene un apuntador `this`, ya que los datos miembro `static` y las funciones miembro `static` existen en forma independiente de cualquier objeto de una clase. El apuntador `this` debe hacer referencia a un objeto específico de la clase, y cuando se hace la llamada a una función miembro `static`, tal vez no haya ningún objeto de su clase en la memoria.



Error común de programación 10.11

Utilizar el apuntador this en una función miembro static es un error de compilación.



Error común de programación 10.12

Declarar const una función miembro static es un error de compilación. El calificador const indica que una función no puede modificar el contenido del objeto en el que opera, pero las funciones miembro static existen y operan en forma independiente de los objetos de la clase.

En las líneas 18 y 19 de la figura 10.23 se utiliza el operador `new` para asignar en forma dinámica dos objetos `Empleado`. Recuerde que el programa terminará de inmediato si no puede asignar uno o ambos objetos. Cuando se asigna cada objeto `Empleado`, se hace una llamada a su constructor. Cuando se utiliza `delete` en las líneas 30 y 32 para desasignar los objetos `Empleado`, se hace una llamada al destructor de cada objeto.



Tip para prevenir errores 10.2

Después de eliminar la memoria asignada en forma dinámica, establezca en 0 el apuntador que hacía referencia a esa memoria. Esto desconecta al apuntador del espacio previamente asignado en el almacenamiento libre. Este espacio en la memoria podría seguir teniendo información, a pesar de haberse eliminado. Al establecer el apuntador en 0, el programa pierde todo acceso a ese espacio de almacenamiento libre, que de hecho, podría haber sido reasignado ya para un propósito distinto. Si usted no estableció el apuntador en 0, su código podría acceder de manera inadvertida a esta nueva información, lo cual produciría errores lógicos extremadamente sencillos y no repetitivos.

10.8 Abstracción de datos y ocultamiento de información

Por lo general, una clase oculta sus detalles de implementación a sus clientes. A esto se le conoce como ocultamiento de información. Como ejemplo de esto, consideremos la estructura de datos tipo pila que presentamos en la sección 6.11.

Las pilas se pueden implementar con arreglos y con otras estructuras de datos, como listas enlazadas. (En el capítulo 14, Plantillas y en el capítulo 20, hablaremos acerca de las pilas y las listas enlazadas). Un cliente de una clase pila no necesita preocuparse por la implementación de la pila. El cliente sabe sólo que cuando se colocan elementos de datos en la pila, se recuperarán en el orden “último en entrar, primero en salir”. El cliente se debe preocupar por el *qué* de la funcionalidad que ofrece una pila, no el *cómo* se implementa esa funcionalidad. Este concepto se conoce como **abstracción de datos**. Aunque los programadores podrían conocer los detalles de la implementación de una clase, no deberían escribir código que dependa de esos detalles. Esto permite reemplazar una clase específica (como una que implemente a una pila y sus operaciones, *push* y *pop*) con otra versión sin afectar el resto del sistema. Siempre y cuando los servicios `public` de la clase no cambien (es decir, que cada función miembro `public` original siga teniendo el mismo prototipo en la definición de la nueva clase), el resto del sistema no se ve afectado.

Muchos lenguajes de programación enfatizan acciones. En estos lenguajes, los datos existen para soportar las acciones que los programas deben realizar. Los datos son “menos interesantes” que las acciones. Los datos son “crudos”. Sólo existen unos cuantos tipos de datos predefinidos, y es difícil para los programadores crear sus propios tipos. C++ y el estilo orientado a objetos de la programación elevan la importancia de los datos. Las principales actividades de la programación orientada a objetos en C++ son la creación de tipos (es decir, clases) y la expresión de las interacciones entre los objetos de esos tipos. Para crear lenguajes que hagan énfasis en los datos, la comunidad de lenguajes de programación necesitaba formalizar ciertas nociones acerca de los datos. La formalización que consideramos aquí es la noción de los **tipos de datos abstractos (ADTs)**, que mejoran el proceso de desarrollo de programas.

¿Qué es un tipo de datos abstractos? Considere el tipo integrado `int`, que la mayoría de las personas asociarían con un entero en matemáticas. En vez de ello, `int` es una representación abstracta de un entero. A diferencia de los enteros matemáticos, los `int` computacionales tienen un tamaño fijo. Por ejemplo, el tipo `int` en los equipos populares de 32 bits de la actualidad se limita generalmente al rango de -2,147,483,648 a +2,147,483,647. Si el resultado de un cálculo se encuentra fuera de este rango, se produce un error por “desbordamiento” y la computadora responde en cierta forma dependiente del equipo. Por ejemplo, podría producir “silenciosamente” un resultado incorrecto, como un valor demasiado grande como para caber en una variable `int` (lo que se conoce comúnmente como **desbordamiento aritmético**). Los enteros matemáticos no tienen este problema. Por lo tanto, la noción de un `int` de computadora es sólo una aproximación de la noción de un entero del mundo real. Lo mismo se aplica con `double`.

Incluso `char` es una aproximación; los valores `char` son por lo general patrones de ocho bits de unos y ceros; estos patrones no se parecen en nada a los caracteres que representan, como una Z mayúscula, una z minúscula, un signo de dólar (\$), un dígito (5), y así en lo sucesivo. Los valores de tipo `char` en la mayoría de las computadoras son bastante limitados, en comparación con el rango de caracteres del mundo real. El conjunto de caracteres ASCII de siete bits (apéndice B) proporciona 128 valores de caracteres distintos. Esto es inadecuado para representar lenguajes tales como japonés y chino, que requieren miles de caracteres. A medida que el uso de Internet y World Wide Web se va haciendo más dominante, el conjunto de caracteres Unicode más reciente está aumentando su popularidad rápidamente, debido a su habilidad para representar los caracteres de la mayoría de los lenguajes. Para obtener más información sobre Unicode, visite www.unicode.org.

El punto es que, hasta los tipos de datos integrados que se proporcionan con los lenguajes de programación como C++ son en realidad sólo aproximaciones o modelos imperfectos de conceptos y comportamientos reales. Hemos dado por sentado a `int` hasta este punto, pero ahora tenemos una nueva perspectiva a considerar. Los tipos como `int`, `double`, `char` y otros son ejemplos de tipos de datos abstractos. En esencia, son formas de representar nociones del mundo real hasta cierto nivel satisfactorio de precisión dentro de un sistema computacional.

En realidad, un tipo de datos abstracto capture dos nociones: una **representación de datos** y las **operaciones** que se pueden realizar con esos datos. Por ejemplo, en C++ un `int` contiene un valor entero (datos) y proporciona operaciones de suma, resta, multiplicación, división y módulo (entre otras); la división entre cero está indefinida. Estas operaciones permitidas se llevan a cabo de una manera sensible a los parámetros del equipo, como el tamaño de palabra fijo del sistema computacional subyacente. Otro ejemplo es la noción de los enteros negativos, cuyas operaciones y representación de datos están claros, pero la operación de obtener la raíz cuadrada de un entero negativo está indefinida. En C++, podemos usar clases para implementar tipos de datos abstractos y sus servicios. Por ejemplo, para implementar un ADT tipo pila, vamos a crear nuestras propias clases de pilas en los capítulos 14 y 20, y estudiaremos la clase `pila` de la biblioteca estándar en el capítulo 22, Biblioteca de plantillas estándar (STL).

10.8.1 Ejemplo: tipo de datos abstracto arreglo

En el capítulo 7 hablamos sobre los arreglos. Como se describe aquí, un arreglo no es más que un apuntador y cierto espacio en memoria. Esta herramienta primitiva es aceptable para realizar operaciones con arreglos, si el programador es cuidadoso y no demanda mucho. Existen muchas operaciones que sería excelente realizar con arreglos, pero que no están integradas en C++. Con las clases de C++, podemos desarrollar un ADT tipo arreglo preferible a los arreglos “crudos”. La clase arreglo puede proporcionar muchas nuevas herramientas útiles, como

- comprobación de rangos de subíndices.
- un rango arbitrario de subíndices, en vez de tener que empezar con 0.
- asignación de arreglos.
- comparación de arreglos.
- entrada/salida con arreglos.
- arreglos que conocen su tamaño.
- arreglos que se expanden en forma dinámica para alojar más elementos.
- arreglos que se pueden imprimir a sí mismos en formato tabular ordenado.

En el capítulo 11 vamos a crear nuestra propia clase de arreglo con muchas de estas herramientas. Recuerde que la plantilla de clase `vector` de la Biblioteca estándar de C++ (presentada en el capítulo 7) proporciona muchas de estas herramientas también. En el capítulo 22 explicaremos con detalle la plantilla de la clase `vector`. C++ tiene un conjunto pequeño de tipos predefinidos. Las clases extienden el lenguaje de programación con nuevos tipos.



Observación de Ingeniería de Software 10.3

Se pueden crear nuevos tipos a través de las clases. Los nuevos tipos se pueden designar tan fácilmente como los tipos predefinidos. Por lo tanto C++ es un lenguaje extensible. Aunque el lenguaje es fácilmente extendible con nuevos tipos, la base del lenguaje en sí misma no se puede modificar.

Nuevas clases creadas en los lenguajes de C++ pueden ser propiedad de un individuo, de pequeños grupos o de compañías. Las clases también pueden formar bibliotecas de clases para su distribución masiva. El C++ estándar incluye una biblioteca estándar de clases. Una vez que aprenda C++ y la programación orientada a objetos, estará listo para aprovechar los nuevos tipos de desarrollo rápido de software orientado a objetos que se hace posible mediante bibliotecas extensas, de las cuales cada vez abundan más.

10.8.2 Ejemplo: tipo de datos abstracto cadena

C++ es un lenguaje deliberadamente escaso, que proporciona a los programadores sólo las herramientas en crudo necesarias para construir un amplio rango de sistemas (considérelo como una herramienta para crear herramientas). El lenguaje está diseñado para minimizar las cargas de rendimiento. C++ es apropiado tanto para la programación de aplicaciones, como para la programación de sistemas; esta última demanda un extraordinario nivel de rendimiento de los programas. Sin duda, hubiera sido posible incluir un tipo de datos cadena entre los tipos de datos definidos de C++. En vez de ello, el lenguaje se diseñó para incluir mecanismos para crear e implementar tipos de datos abstractos de cadena a través de las clases. En el capítulo 3 presentamos la clase `string` de la Biblioteca estándar de C++, y en el capítulo 11 desarrollaremos nuestro propio ADT llamado `String`. En el capítulo 18 hablaremos con detalle sobre la clase `string`.

10.8.3 Ejemplo: tipo de datos abstracto cola

Cada uno de nosotros tiene que pararse en una fila de vez en cuando. A una línea de espera se le conoce también como **cola**. Esperamos en una fila en el supermercado para pagar, esperamos en fila para obtener gasolina, esperamos en fila para abordar un autobús, para pagar en una autopista de cuota, y los estudiantes saben muy bien acerca de esperar en una fila durante las inscripciones, para obtener los cursos que desean. Los sistemas computacionales utilizan líneas de espera de maneras interna, por lo que necesitamos escribir programas que simulen lo que son las colas y cuál es su función.

Una cola es otro ejemplo de un tipo de datos abstracto. Las colas ofrecen un comportamiento bien definido a sus clientes. Los clientes colocan cosas en la cola, una a la vez (mediante la invocación de la operación `enqueue` de la cola), y obtiene de vuelta esas cosas, una a la vez, bajo demanda (mediante la invocación de la operación `dequeue` de la cola). En concepto, una cola puede volverse infinitamente larga. Desde luego que una cola real es finita. Los elementos se devuelven de una cola en orden “**primero en entrar, primero en salir (PEPS)**”: el primer elemento que se inserta en la cola es el primer elemento que se saca de ella.

La cola oculta una representación de datos interna que lleva la cuenta de los elementos que esperan actualmente en línea, y ofrece un conjunto de operaciones a sus clientes, a saber, `enqueue` y `dequeue`. Los clientes no están preocupados acerca de la implementación de la cola. Los clientes simplemente quieren que la cola opere “según lo indicado”. Cuando un cliente pone un nuevo elemento en la cola, ésta debe aceptar ese elemento y colocarlo de manera interna en cierto tipo de estructura de datos en el orden “primero en entrar, primero en salir”. Cuando el cliente desea el siguiente elemento de la parte inicial de la cola, ésta debe eliminar el elemento de su representación interna y entregarlo al mundo exterior (es decir, al cliente de la cola) en orden PEPS (es decir, el elemento que haya estado más tiempo en la cola será el que se devuelva mediante la siguiente operación `dequeue`).

El ADT tipo cola garantiza la integridad de su estructura de datos interna. Los clientes no pueden manipular esta estructura de datos directamente. Sólo las funciones miembro de la cola tienen acceso a sus datos internos. Los clientes sólo pueden hacer que se realicen las operaciones permitidas en la representación de datos; las operaciones que no se proporcionen en la interfaz pública del ADT se rechazan de cierta forma apropiada. Esto podría indicar la generación de un mensaje de error, el lanzamiento de una excepción (vea el capítulo 16), terminar la ejecución o simplemente ignorar la petición de una operación.

Crearemos nuestra propia clase cola en el capítulo 20, y estudiaremos la clase `queue` de la Biblioteca estándar en el capítulo 22.

10.9 Clases contenedoras e iteradores

Entre los tipos más populares de clases están las **clases contenedoras** (también conocidas como **clases de colecciones**); es decir, clases diseñadas para contener colecciones de objetos. Por lo general, las clases contenedoras proporcionan servicios tales como inserción, eliminación, búsqueda, ordenamiento y prueba de un elemento para determinar si es un miembro

de la colección. Los arreglos, pilas, colas, árboles y listas enlazadas son ejemplos de clases contenedoras; en el capítulo 7 estudiamos los arreglos y en los capítulos 20 y 22 estudiaremos cada una de las otras estructuras de datos.

Es común asociar **objetos iteradores** (o simplemente **iteradores**) con las clases contenedoras. Un iterador es un objeto que recorre una colección, devolviendo el siguiente elemento (o realizando cierta acción con el siguiente elemento). Una vez que se ha escrito un iterador para una clase, el proceso de obtener el siguiente elemento de la clase se puede expresar de una manera simple. Así como un libro que comparten varias personas podría tener varios marcadores al mismo tiempo, también una clase contenedora puede tener varios iteradores operando en ella al mismo tiempo. Cada iterador mantiene su propia información de “posición”. Hablaremos sobre los contenedores y los iteradores con detalle en el capítulo 22.

10.10 Clases proxy

Recuerde que dos de los principios fundamentales de la buena ingeniería de software son: separar la interfaz de la implementación y ocultar los detalles de implementación. Nos esforzamos por lograr estos objetivos al definir una clase en un archivo de encabezado e implementar sus funciones miembro en un archivo de implementación. Sin embargo, como indicamos en el capítulo 9, los archivos de encabezado *contienen* una porción de la implementación de una clase y sugerencias acerca de otras. Por ejemplo, los miembros **private** de una clase se listan en la definición de la clase en un archivo de encabezado, por lo que estos miembros están visibles para los clientes, aun cuando éstos no puedan acceder a los miembros **private**. Al revelar los datos **private** de una clase de esta manera, hay una potencial exposición de información propietaria a los clientes de la clase. Ahora presentaremos la noción de una **clase proxy** que nos permite ocultar incluso hasta los datos **private** de una clase de los clientes de la clase. Al proporcionar a los clientes de su clase una clase proxy que sólo conozca la interfaz **public** para su clase, el programador permite a los clientes usar los servicios de su clase sin otorgar acceso a los clientes a los detalles de implementación de su clase.

Para implementar una clase proxy se requieren varios pasos, los cuales demostraremos en las figuras 10.24 a 10.27. Primero, vamos a crear la definición de la clase que contiene la implementación propietaria que nos gustaría ocultar. Nuestra clase de ejemplo, llamada **Implementacion**, se muestra en la figura 10.24. La clase proxy **Interfaz** se muestra en las figuras 10.25 a 10.26. El programa de prueba y los resultados de ejemplo se muestran en la figura 10.27.

La clase **Implementacion** (figura 10.24) proporciona un solo miembro de datos **private** llamado **valor** (los datos que nos gustaría ocultar del cliente), un constructor para inicializar **valor**, y las funciones **setValor** y **getValor**.

```

1 // Fig. 10.24: Implementacion.h
2 // Definición de la clase Implementacion.
3
4 class Implementacion
5 {
6 public:
7     // constructor
8     Implementacion( int v )
9         : valor( v ) // inicializa valor con v
10    {
11        // cuerpo vacío
12    } // fin del constructor de Implementacion
13
14    // establece valor en v
15    void setValor( int v )
16    {
17        valor = v; // debe validar v
18    } // fin de la función setValor
19
20    // devuelve valor
21    int getValor() const
22    {
23        return valor;
24    } // fin de la función getValor
25 private:
26     int valor; // datos que nos gustaría ocultar del cliente
27 } // fin de la clase Implementacion

```

Figura 10.24 | Definición de la clase **Implementacion**.

Vamos a definir una clase proxy llamada `Interfaz` (figura 10.25) con una interfaz `public` idéntica (excepto por los nombres del constructor y del destructor) a la de la clase `Implementacion`. El único miembro `private` de la clase proxy es un apuntador a un objeto de la clase `Implementacion`. Al usar un apuntador de esta manera, podemos ocultar al cliente los detalles de implementación de la clase `Implementacion`. Observe que las únicas menciones en la clase `Interfaz` acerca de la clase propietaria `Implementacion` se encuentran en la declaración del apuntador (línea 17) y en la línea 6, una **declaración de clase anticipada**. Cuando la definición de una clase (como la clase `Interfaz`) utiliza sólo un apuntador o referencia a un objeto de otra clase (por ejemplo, a un objeto de la clase `Implementacion`), no se tiene que incluir el archivo de encabezado de la clase para esa otra clase (que por lo general revelaría los datos `private` de esa clase) mediante `#include`. Esto se debe a que el compilador no necesita reservar espacio para un objeto de la clase. El compilador necesita reservar espacio para el apuntador o referencia. Los tamaños de los apuntadores y referencias son características de la plataforma de hardware en la que se ejecuta el compilador, por lo que éste ya conoce esos tamaños. Simplemente podemos declarar esa otra clase como un tipo de datos con una declaración de clase anticipada (línea 6) antes de usar el tipo en el archivo.

El archivo de implementación de la función miembro para la clase proxy `Interfaz` (figura 10.26) es el único archivo que incluye el archivo de encabezado `Implementacion.h` (línea 5), el cual contiene la clase `Implementacion`. El archivo `Interfaz.cpp` (figura 10.26) se proporciona al cliente como un código objeto precompilado, junto con el archivo de encabezado `Interfaz.h` que incluye los prototipos de función de los servicios proporcionados por la clase proxy. Como el archivo `Interfaz.cpp` está disponible para el cliente sólo como código objeto, el cliente no puede ver las interacciones entre la clase proxy y la clase propietaria (líneas 9, 17, 23 y 29). Observe que la clase proxy impone un “nivel” adicional de llamadas a funciones como el “precio a pagar” por ocultar los datos `private` de la clase `Implementacion`. Dada la velocidad de las computadoras de la actualidad y el hecho de que muchos compiladores pueden poner en línea las llamadas a funciones simples de manera automática, el efecto de esas llamadas a funciones adicionales sobre el rendimiento es a menudo insignificante.

```

1 // Fig. 10.25: Interfaz.h
2 // Definición de la clase proxy Interfaz.
3 // El cliente ve este código fuente, pero éste no revela
4 // la distribución de los datos de la clase Implementacion.
5
6 class Implementacion; // declaración anticipada de la clase, requerida por la línea 17
7
8 class Interfaz
9 {
10 public:
11     Interfaz( int ); // constructor
12     void setValor( int ); // misma interfaz public que
13     int getValor() const; // tiene la clase Implementacion
14     ~Interfaz(); // destructor
15 private:
16     // requiere la declaración anticipada anterior (línea 6)
17     Implementacion *ptr;
18 };// fin de la clase Interfaz

```

Figura 10.25 | Definición de la clase proxy `Interfaz`.

```

1 // Fig. 10.26: Interfaz.cpp
2 // Implementación de la clase Interfaz--el cliente recibe este archivo sólo
3 // como código objeto precompilado, y se mantiene oculta la implementación.
4 #include "Interfaz.h" // definición de la clase Interfaz
5 #include "Implementacion.h" // definición de la clase Implementacion
6
7 // constructor
8 Interfaz::Interfaz( int v )
9     : ptr ( new Implementacion( v ) ) // inicializa ptr para que apunte a
10 {                                     // un nuevo objeto Implementacion
11     // cuerpo vacío
12 } // fin del constructor de Interfaz
13

```

Figura 10.26 | Definiciones de las funciones miembro de la clase `Interfaz`. (Parte 1 de 2).

```

14 // llama a la función setValor de Implementacion
15 void Interfaz::setValor( int v )
16 {
17     ptr->setValor( v );
18 } // fin de la función setValor
19
20 // llama a la función getValor de Implementacion
21 int Interfaz::getValor() const
22 {
23     return ptr->getValor();
24 } // fin de la función getValor
25
26 // destructor
27 Interfaz::~Interfaz()
28 {
29     delete ptr;
30 } // fin del destructor ~Interfaz

```

Figura 10.26 | Definiciones de las funciones miembro de la clase `Interfaz`. (Parte 2 de 2).

La figura 10.27 prueba la clase `Interfaz`. Observe que sólo se incluye el archivo de encabezado para `Interfaz` en el código cliente (línea 7); no hay mención sobre la existencia de una clase separada llamada `Implementacion`. Por ende, el cliente nunca ve los datos `private` de la clase `Implementacion`, ni el código cliente se puede volver dependiente del código de `Implementacion`.



Observación de Ingeniería de Software 10.14

Una clase proxy aisla el código cliente de los cambios de la implementación.

```

1 // Fig. 10.27: fig10_27.cpp
2 // Ocultar los datos private de una clase con una clase proxy.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Interfaz.h" // definición de la clase Interfaz
8
9 int main()
10 {
11     Interfaz i( 5 ); // crea un objeto Interfaz
12
13     cout << "Interfaz contiene: " << i.getValor()
14     << " antes de setValor" << endl;
15
16     i.setValor( 10 );
17
18     cout << "Interfaz contiene: " << i.getValor()
19     << " despues de setValor" << endl;
20
21     return 0;
22 } // fin de main

```

```

Interfaz contiene: 5 antes de setValor
Interfaz contiene: 10 despues de setValor

```

Figura 10.27 | Implementación de una clase proxy.

10.11 Repaso

En este capítulo presentamos varios temas avanzados relacionados con las clases y la abstracción de datos. El lector aprendió a especificar objetos `const` y funciones miembro `const` para evitar modificaciones a los objetos, con lo cual se hace

valer el principio del menor privilegio. También aprendió que, mediante la composición, una clase puede tener objetos de otras clases como miembros. Presentamos el tema de la amistad y ejemplos que demuestran cómo usar las funciones `friend`.

Aprendió que el apuntador `this` se pasa como un argumento implícito a cada una de las funciones miembro no `static` de una clase, lo cual permite a las funciones acceder a los datos miembro correctos del objeto, junto con otras funciones miembro no `static`. También vimos el uso explícito del apuntador `this` para acceder a los miembros de una clase y permitir las llamadas a en cascada funciones miembro.

Presentamos el concepto de la administración dinámica de la memoria. Aprendió que puede crear y destruir objetos en forma dinámica con los operadores `new` y `delete`, respectivamente. Fomentamos la necesidad de datos miembro `static` y demostramos cómo declarar y usar datos miembro `static` y funciones miembro `static` en sus propias clases.

Aprendió acerca de la abstracción de datos y el ocultamiento de información: dos de los conceptos fundamentales de la programación orientada a objetos. Hablamos sobre los tipos de datos abstractos: formas de representar nociiones reales o conceptuales hasta cierto nivel satisfactorio de precisión dentro de un sistema computacional. Después aprendió acerca de tres tipos de datos abstractos de ejemplo: arreglos, cadenas y colas. Presentamos el concepto de una clase contenedora que contiene una colección de objetos, así como la noción de una clase iteradora que recorre los elementos de una clase contenedora. Por último, aprendió a crear una clase proxy para ocultar los detalles de implementación (incluyendo los datos miembro `private`) de una clase a los clientes de la misma.

En el capítulo 11, continuaremos nuestro estudio sobre las clases y los objetos, al mostrarle cómo permitir que los operadores de C++ funcionen con objetos; a este proceso se le conoce como sobrecarga. Por ejemplo, veremos cómo “sobrecargar” el operador `<<` de manera que se pueda utilizar para imprimir un arreglo completo, sin utilizar de manera explícita una instrucción de repetición.

Resumen

Sección 10.2 *Objetos const (constantes) y funciones miembro const*

- La palabra clave `const` se puede usar para especificar que un objeto no puede modificarse, y que cualquier intento por modificar el objeto debe producir un error de compilación.
- Los compiladores de C++ no permiten llamadas a funciones miembro no `const` en objetos `const`.
- El intento de una función miembro `const` por modificar un objeto de su clase es un error de compilación.
- Una función se especifica como `const`, tanto en su prototipo como en su definición.
- Un objeto `const` se debe inicializar, no asignarle un valor.
- Los constructores y destructores no se pueden declarar `const`.
- Los datos miembro `const` y los datos miembro tipo referencias *deben* inicializarse a través de inicializadores de miembros.

Sección 10.3 *Composición: objetos como miembros de clases*

- Una clase puede tener objetos de otras clases como miembros; a este concepto se le conoce como composición.
- Los objetos miembro se construyen en el orden en el que se declaran en la definición de la clase, y antes de construir los objetos de su clase circundante.
- Si no se proporciona un inicializador de miembros para un objeto miembro, se hará una llamada implícita al constructor predeterminado del objeto miembro.

Sección 10.4 *Funciones friend y clases friend*

- Una función `friend` de una clase se define fuera del alcance de esa clase, pero aun así tiene el derecho de acceder a los miembros no `public` (y `public`) de la clase. Pueden declararse funciones independientes o clases completas como amigas de otras clases.
- Una declaración `friend` puede aparecer en cualquier lugar dentro de una clase. Un `friend` es esencialmente una parte de la interfaz `public` de la clase.
- La relación de amistad no es simétrica ni transitiva.

Sección 10.5 *Uso del apuntador this*

- Todo objeto tiene acceso a su propia dirección, a través del apuntador `this`.
- El apuntador `this` de un objeto no forma parte del objeto en sí; es decir, el tamaño de la memoria ocupada por el apuntador `this` no se refleja en el resultado de una operación `sizeof` en el objeto.
- El apuntador `this` se pasa (por el compilador) como un argumento para cada una de las funciones miembro no `static` del objeto.

- Los objetos usan el apuntador `this` de manera implícita (como se ha hecho hasta este momento), o de manera explícita para hacer referencia a sus datos miembro y funciones miembro.
- El apuntador `this` permite llamadas en cascada a funciones miembro, donde se invocan varias funciones en la misma instrucción.

Sección 10.6 Administración dinámica de la memoria con los operadores `new` y `delete`

- La administración dinámica de memoria permite a los programadores controlar la asignación y desasignación de memoria en un programa, para cualquier tipo ya predefinido o definido por el usuario.
- El almacenamiento libre (algunas veces conocido como el “heap” o montón) es una región de memoria asignada a cada programa para almacenar objetos que se asignan en forma dinámica en tiempo de ejecución.
- El operador `new` asigna almacenamiento del tamaño apropiado para un objeto, ejecuta el constructor del objeto y devuelve un apuntador del tipo correcto. El operador `new` se puede utilizar para asignar en forma dinámica cualquier tipo fundamental (como `int` o `double`) o tipo de clase. Si `new` no puede encontrar espacio en memoria para el objeto, indica que ocurrió un error mediante el “lanzamiento” de una “excepción”. Esto por lo general hace que el programa termine de inmediato, a menos que se maneje la excepción.
- Para destruir un objeto asignado en forma dinámica y liberar el espacio para ese objeto, use el operador `delete`.
- Un arreglo de objetos se puede asignar en forma dinámica con `new`, como en la siguiente instrucción:

```
int *ptr = new int[ 100 ];
```

la cual asigna un arreglo de 100 enteros y asigna la ubicación inicial del arreglo a `ptr`. El arreglo anterior de enteros se elimina mediante la siguiente instrucción:

```
delete [] ptr;
```

Sección 10.7 Miembros de clase `static`

- Un miembro de datos `static` representa información a “nivel de clase” (es decir, una propiedad de la clase compartida por todas las instancias, no una propiedad de un objeto específico de la clase).
- Los datos miembro `static` tienen alcance de clase y se pueden declarar como `public`, `private` o `protected`.
- Los miembros `static` de una clase existen aun y cuando no existan objetos de esa clase.
- Para acceder a un miembro de clase `public static` cuando no existen objetos de la clase, simplemente hay que anteponer el nombre de clase y el operador de resolución de ámbito binario (`::`) al nombre del miembro de datos.
- Los miembros `public static` de una clase se pueden utilizar a través de cualquier objeto de esa clase.
- Una función miembro debe declararse como `static` si no accede a los datos miembro no `static` o a las funciones miembro no `static` de la clase. A diferencia de las funciones miembro no `static`, una función miembro `static` no tiene un apuntador `this`, ya que los datos miembro `static` y las funciones miembro `static` existen de manera independiente de cualquier objeto de una clase.

Sección 10.8 Abstracción de datos y ocultamiento de información

- Los tipos de datos abstractos son formas de representar nociones reales y conceptuales hasta cierto nivel satisfactorio de precisión dentro de un sistema computacional.
- Un tipo de datos abstracto captura dos nociones: una representación de datos y las operaciones que se pueden realizar sobre esos datos.
- C++ es un lenguaje deliberadamente escaso, que proporciona a los programadores sólo las herramientas crudas necesarias para construir un amplio rango de sistemas. C++ está diseñado para minimizar las cargas de rendimiento.
- Los elementos se devuelven de una cola en el orden “primero en entrar, primero en salir (PEPS)”; el primer elemento que se inserta en la cola es el primer elemento que se elimina de ésta.

Sección 10.9 Clases contenedoras e iteradores

- Las clases contenedoras (también conocidas como clases de colecciones) están diseñadas para contener colecciones de objetos. Por lo común, las clases contenedoras proporcionan servicios tales como inserción, eliminación, búsqueda, ordenamiento y prueba de un elemento, para determinar si es miembro de una colección.
- Es común asociar los iteradores con las clases contenedoras. Un iterador es un objeto que recorre una colección, devolviendo el siguiente elemento (o realizando cierta acción sobre el siguiente elemento).

Sección 10.10 Clases proxy

- Si el programador proporciona a los clientes de su clase una clase proxy que sólo conozca la interfaz `public` para su clase, permite a los clientes usar los servicios de su clase sin proporcionarles acceso a los detalles de implementación de la misma, como sus datos `private`.
- Cuando la definición de una clase sólo utiliza un apuntador o referencia a un objeto de otra clase, el archivo de encabezado para esa otra clase (que por lo general revelaría los datos `private` de la clase) no tiene que incluirse mediante `#include`.

Simplemente podemos declarar esa otra clase como un tipo de datos mediante una declaración anticipada de la clase antes de usar el tipo en el archivo.

- El archivo de implementación que contiene las funciones miembro para una clase proxy es el único archivo que incluye el archivo de encabezado para la clase cuyos datos `private` nos gustaría ocultar.
- El archivo de implementación que contiene las funciones miembro para la clase proxy se proporciona al cliente como un archivo de código objeto precompilado, junto con el archivo de encabezado que incluye los prototipos de función de los servicios proporcionados por la clase proxy.

Terminología

abstracción de datos	fuga de memoria
administración dinámica de memoria	inicializador de miembros
almacenamiento libre	iterador
asignar memoria	lista de inicializadores de miembros
clase contenedora	llamadas en cascada a funciones miembro
clase de colección	montón
clase proxy	<code>new</code> , operador
cola, tipo de datos abstracto	<code>new[]</code> , operador
composición	objeto anfitrión
<code>const</code> , función miembro	objeto miembro
<code>const</code> , objeto	objetos dinámicos
constructor de objeto miembro	ocultamiento de información
declaración anticipada de una clase	operaciones en un ADT
<code>delete</code> , operador	primero en entrar, primero en salir (PEPS)
<code>delete[]</code> , operador	representación de datos
<code>dequeue</code> (operación de una cola)	<code>static</code> , función miembro
desasignar memoria	<code>static</code> , miembro de datos
desbordamiento aritmético	<code>this</code> , apuntador
<code>enqueue</code> (operación de una cola)	<i>tiene un</i> , relación
<code>friend</code> , clase	tipo de datos abstracto (ADT)
<code>friend</code> , función	último en entrar, primero en salir (UEPS)

Ejercicios de autoevaluación

10.1 Complete los siguientes enunciados:

- _____ se debe usar para inicializar los miembros constantes de una clase.
- Una función no miembro se debe declarar como _____ de una clase para tener acceso a los datos miembro `private` de esa clase.
- El operador _____ asigna memoria en forma dinámica para un objeto de un tipo especificado, y devuelve un _____ a ese tipo.
- Un objeto constante debe _____; no se puede modificar después de crearlo.
- Un miembro de datos _____ representa la información a nivel de clase.
- Las funciones miembro no `static` de un objeto tienen acceso a un “auto-apuntador” al objeto, conocido como apuntador _____.
- La palabra clave _____ especifica que un objeto o variable no puede modificarse después de inicializarlo.
- Si no se proporciona un inicializador de miembros para un objeto miembro de una clase, se hace una llamada al _____ del objeto.
- Una función miembro debe declararse `static` si no accede a los datos miembro _____ de una clase.
- Los objetos miembro se construyen _____ del objeto de su clase circundante.
- El operador _____ reclama la memoria previamente asignada por `new`.

10.2 Busque los errores en la siguiente clase y explique cómo corregirlos:

```
class Ejemplo
{
public:
    Ejemplo( int y = 10 )
```

```

        : datos( y )
{
    // cuerpo vacío
} // fin del constructor de Ejemplo
int getDatosIncrementados() const
{
    return datos++;
} // fin de la función getDatosIncrementados
static int getCuenta()
{
    cout << "Los datos son " << datos << endl;
    return cuenta;
} // fin de la función getCuenta
private:
    int datos;
    static int cuenta;
}; // fin de la clase Ejemplo

```

Respuestas a los ejercicios de autoevaluación

10.1 a) inicializadores de miembros. b) friend. c) new, apuntador. d) inicializarse. e) static. f) this. g) const. h) constructor predeterminado. i) no static. j) antes. k) delete.

10.2 *Error:* la definición de clase para Ejemplo tiene dos errores. El primero ocurre en la función getDatosIncrementados. La función se declara const, pero modifica el objeto.

Corrección: para corregir el primer error, elimine la palabra clave const de la definición de getDatosIncrementados.

Error: el segundo error ocurre en la función getCuenta. Esta función se declara como static, por lo que no puede acceder a ningún miembro no static (es decir, datos) de la clase.

Corrección: para corregir el segundo error, elimine la línea de salida de la definición getCuenta.

Ejercicios

10.3 Compare y contraste la asignación dinámica de memoria con los operadores de desasignación new, new [], delete y delete [].

10.4 Explique la noción de amistad. Explique los aspectos negativos de la amistad, como se describe en el texto.

10.5 ¿Puede una definición correcta de la clase Tiempo incluir los dos constructores que se muestran a continuación? Si no es así, explique por qué.

```

Tiempo( int h = 0, int m = 0, int s = 0 );
Tiempo();

```

10.6 ¿Qué ocurre cuando se especifica un tipo de retorno, incluso void, para un constructor o destructor?

10.7 Modifique la clase Fecha en la figura 10.10 para que tenga las siguientes herramientas:

a) Imprimir la fecha en varios formatos, como

```

DDD AAAA
MM/DD/AA
Junio 14, 1992

```

- b) Usar constructores sobrecargados para crear objetos Fecha inicializados con fechas de los formatos en la parte (a).
- c) Crear un constructor de Fecha que lea la fecha del sistema utilizando las funciones de la biblioteca estándar del encabezado <ctime>, y que establezca los miembros de Fecha. (Consulte la documentación de referencia de su compilador, o el sitio wwwcplusplus.com/ref/ctime/index.html para obtener información sobre las funciones en el encabezado <ctime>).

En el capítulo 11, podremos crear operadores para evaluar la igualdad de dos fechas y para comparar fechas y determinar si una fecha es anterior, o posterior, a otra.

10.8 Cree una clase llamada CuentaAhorros. Use un miembro de datos static llamado tasaInteresAnual para almacenar la tasa de interés anual para cada uno de los ahorradores. Cada miembro de la clase debe contener un miembro de datos

private llamado `saldoAhorros`, que indique el monto que tiene el ahorrador actualmente en depósito. Proporcione la función miembro `calcularInteresMensual` que calcule el interés mensual multiplicando el `saldo` por `tasaInteresAnual` dividido entre 12; este interés debe sumarse a `saldoAhorros`. Proporcione una función miembro `static modificarTasaInteres` que establezca el miembro de datos `static tasaInteresAnual` a un nuevo valor. Escriba un programa controlador para probar la clase `CuentaAhorros`. Cree instancias de dos objetos distintos de la clase `CuentaAhorros` llamados `ahorrador1` y `ahorrador2`, con saldos de \$2000.00 y \$3000.00, respectivamente. Establezca la `tasaInteresAnual` al 3 por ciento. Después calcule el interés mensual e imprima los nuevos saldos para cada uno de los ahorradores. Después establezca la `tasaInteresAnual` al 4 por ciento, calcule el interés del siguiente mes e imprima los nuevos saldos para cada uno de los ahorradores.

10.9 Cree la clase `ConjuntoEnteros` para la que cada objeto pueda contener enteros en el rango de 0 a 100. Un conjunto se representa en forma interna como un arreglo de unos y ceros. El elemento `a[i]` del arreglo es 1 si el entero `i` se encuentra en el conjunto. El elemento `a[j]` del arreglo es 0 si el entero `j` no se encuentra en el conjunto. El constructor predeterminado inicializa un conjunto que se denomina “conjunto vacío”; es decir, un conjunto cuya representación de arreglo contiene sólo ceros.

Proporcione funciones miembro para las operaciones comunes de los conjuntos. Por ejemplo, proporcione una función miembro llamada `unionDeConjuntos` que cree un tercer conjunto que sea la unión teórica de dos conjuntos existentes (es decir, un elemento del arreglo del tercer conjunto se establece en 1 si ese elemento es 1 en cualquiera, o en ambos de los conjuntos existentes, y un elemento del arreglo del tercer conjunto se establece en 0 si ese elemento es 0 en cada uno de los conjuntos existentes).

Proporcione una función miembro llamada `interseccionDeConjuntos`, para crear un tercer conjunto que sea la intersección teórica de dos conjuntos existentes (es decir, un elemento del arreglo del tercer conjunto se establece en 1 si ese elemento es 1 en uno o ambos de los conjuntos existentes, y un elemento del arreglo del tercer conjunto se establece en 0 si ese elemento es 0 en cada uno de los conjuntos existentes).

Proporcione una función miembro llamada `insertarElemento`, que inserte un nuevo entero `k` en un conjunto (estableciendo `a[k]` en 1). Proporcione una función miembro llamada `eliminarElemento` que elimine el entero `m` (estableciendo `a[m]` en 0).

Proporcione una función miembro llamada `imprimirConjunto` que imprima un conjunto como una lista de números separados por espacios. Imprima sólo los elementos que estén presentes en el conjunto (es decir, que su posición en el arreglo tenga un valor de 1). Imprima --- para un conjunto vacío.

Proporcione una función miembro llamada `esIgualA` que determine si dos conjuntos son iguales.

Proporcione un constructor adicional que reciba un arreglo de enteros y el tamaño de ese arreglo, y que utilice el arreglo para inicializar un objeto conjunto.

Ahora escriba un programa controlador para probar su clase `ConjuntoEnteros`. Cree instancias de varios objetos `ConjuntoEnteros`. Pruebe que todas sus funciones miembro trabajen en forma apropiada.

10.10 Sería perfectamente razonable que la clase `Tiempo` de las figuras 10.18 y 10.19 representaran el tiempo en forma interna como el número de segundos transcurridos desde medianoche, en vez de hacerlo con los tres valores enteros `hora`, `minuto` y `segundo`. Los clientes podrían usar los mismos métodos `public` y obtener los mismos resultados. Modifique la clase `Tiempo` de la figura 10.18 para implementar el tiempo como el número de segundos transcurridos desde medianoche y mostrar que no hay un cambio visible en la funcionalidad para los clientes de esa clase. [Nota: este ejercicio demuestra apropiadamente las virtudes del ocultamiento de información].



La diferencia completa entre construcción y creación es exactamente ésta: que un objeto construido sólo puede ser amado después de construirlo; pero un objeto creado se ama incluso antes de que exista.

—Gilbert Keith Chesterton

La suerte está echada.

—Julio César

Nuestro doctor nunca operaría, a menos que fuera necesario. Él era así. Si no necesitara el dinero, no te pondría una mano.

—Herb Shriner



Sobrecarga de operadores: objetos String y Array

OBJETIVOS

En este capítulo aprenderá a:

- Conocer la sobrecarga de operadores y cómo puede facilitar la legibilidad de los programas, además de hacer más conveniente la programación.
- Redefinir (sobrecargar) los operadores para trabajar con objetos de clases definidas por el usuario.
- Diferenciar entre sobrecargar operadores unarios y binarios.
- Convertir objetos de una clase a otra.
- Saber cuándo (y cuándo no) sobrecargar operadores.
- Crear las clases `NumeroTelefonico`, `Array`, `String` y `Fecha` que demuestren la sobrecarga de operadores.
- Usar los operadores sobrecargados y otras funciones miembro de la clase `string` de la biblioteca estándar.
- Usar la palabra clave `explicit` para evitar que el compilador utilice constructores con un solo argumento para realizar conversiones implícitas.

- 11.1** Introducción
- 11.2** Fundamentos de la sobre carga de operadores
- 11.3** Restricciones acerca de la sobre carga de operadores
- 11.4** Las funciones de operadores como clase miembro vs. funciones globales
- 11.5** Sobre carga de los operadores de inserción de flujo y extracción de flujo
- 11.6** Sobre carga de operadores unarios
- 11.7** Sobre carga de operadores binarios
- 11.8** Ejemplo práctico: la clase `Array`
- 11.9** Conversión entre tipos
- 11.10** Ejemplo práctico: la clase `String`
- 11.11** Sobre carga de `++` y `--`
- 11.12** Ejemplo práctico: una clase `Fecha`
- 11.13** La clase `string` de la biblioteca estándar
- 11.14** Constructores `explicit`
- 11.15** Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)

11.1 Introducción

En los capítulos 9 a 10 se presentaron los fundamentos de las clases de C++. Se obtuvieron servicios de objetos mediante el envío de mensajes (en forma de llamadas a funciones miembro) a los objetos. Esta notación de llamadas a funciones es incómoda para ciertos tipos de clases (como las clases matemáticas). Además, muchas manipulaciones comunes se llevan a cabo con los operadores (por ejemplo, entrada y salida). Podemos usar el extenso conjunto de operadores integrados de C++ para especificar las manipulaciones de objetos comunes. En este capítulo mostraremos cómo permitir que los operadores de C++ trabajen con objetos; a este proceso se le conoce como **sobre carga de operadores**. Es un proceso simple y natural extender a C++ con estas nuevas herramientas, pero debe hacerse con cuidado.

Un ejemplo de un operador sobre cargado integrado en C++ es `<<`, el cual se usa como operador de inserción de flujo y como operador de desplazamiento a la izquierda a nivel de bits (que veremos en el capítulo 21, Bits, caracteres, cadenas y tipos `struct`). De manera similar, `>>` también está sobre cargado; se utiliza como operador de extracción de flujo y como operador de desplazamiento a la derecha a nivel de bits. Ambos operadores están sobre cargados en la Biblioteca estándar de C++.

Aunque la sobre carga de operadores suena como una herramienta exótica, la mayoría de los programadores utilizan operadores sobre cargados de manera implícita con frecuencia. Por ejemplo, el mismo lenguaje C++ sobre carga los operadores de suma (`+`) y de resta (`-`). Estos operadores tienen un desempeño distinto, dependiendo de su contexto en la aritmética de enteros, de punto flotante y de apuntadores.

C++ nos permite sobre cargar la mayoría de los operadores, para que sean sensibles al contexto en el que se utilizan; el compilador genera el código apropiado con base en el contexto (en especial, los tipos de los operandos). Algunos operadores se sobre cargan con frecuencia, en especial los operadores de asignación, relacionales y varios operadores aritméticos como `+` y `-`. Los trabajos que desempeñan los operadores sobre cargados también se pueden llevar a cabo mediante llamadas explícitas a funciones, pero la notación de los operadores es comúnmente más clara y familiar para los programadores.

Vamos a hablar acerca de cuándo (y cuándo no) usar la sobre carga de operadores. Implementaremos las clases definidas por el usuario `NúmeroTelefónico`, `Array`, `String` y `Fecha` para demostrar cómo sobre cargar operadores, incluyendo los operadores de inserción de flujo, extracción de flujo, asignación, igualdad, relacionales, de subíndice, negación lógica, paréntesis y de incremento. El capítulo termina con un ejemplo de la clase `string` de la Biblioteca estándar de C++, la cual proporciona muchos operadores sobre cargados que son similares a los de nuestra clase `String` que presentamos antes en este capítulo. En los ejercicios le pediremos que implemente varias clases con operadores sobre cargados. Los ejercicios también usan las clases `Complejo` (para números complejos) y `EnteroEnorme` (para enteros mayores de los que puede representar una computadora con el tipo `long`) para demostrar los operadores aritméticos sobre cargados `+` y `-`, y le pediremos que mejore esas clases, sobre cargando otros operadores aritméticos.

11.2 Fundamentos de la sobrecarga de operadores

La programación en C++ es un proceso sensible y enfocado a los tipos. Los programadores pueden usar los tipos fundamentales y definir nuevos tipos. Los tipos fundamentales se pueden utilizar con la extensa colección de operadores de C++. Los operadores proporcionan a los programadores una notación concisa para expresar manipulaciones de datos de tipos fundamentales.

Los programadores pueden usar operadores con tipos definidos por el usuario también. Aunque C++ no permite crear nuevos operadores, si permite sobrecargar la mayoría de los operadores existentes para que, cuando éstos se utilicen con objetos, tengan un significado apropiado. Ésa es una poderosa herramienta.



Observación de Ingeniería de Software 11.1

La sobrecarga de operadores contribuye a la extensibilidad de C++; uno de los atributos más atractivos del lenguaje.



Buena práctica de programación 11.1

Use la sobrecarga de operadores cuando un programa se haga más claro que lograr las mismas operaciones con llamadas a funciones.



Buena práctica de programación 11.2

Los operadores sobrecargados deben imitar la funcionalidad de sus contrapartes integrados; por ejemplo, el operador + debe sobrecargarse para realizar la suma, no la resta. Evite el uso excesivo o inconsistente de la sobrecarga de operadores, ya que esto puede ocasionar que el programa se haga críptico y difícil de leer.

Para sobrecargar un operador, se escribe la definición de una función miembro no `static` o la definición de una función global como se hace normalmente, excepto que el nombre de la función se convierte ahora en la palabra clave `operator`, seguida del símbolo del operador que se va a sobrecargar. Por ejemplo, el nombre de función `operator+` se utilizaría para sobrecargar el operador de suma (+). Cuando los operadores se sobrecargan como funciones miembro, deben ser no `static`, debido a que se deben llamar en un objeto de la clase y deben operar en ese objeto.

Para usar un operador en objetos de clases, éste *debe* sobrecargarse; hay tres excepciones a esto. El operador de asignación (=) se puede usar con cualquier clase para realizar la asignación a nivel de bits de los miembros de datos de la clase; cada miembro de datos se asigna del objeto “origen” al objeto “destino” de la asignación. Pronto veremos que dicha asignación predeterminada a nivel de bits es peligrosa para las clases con miembros apuntadores; sobrecargaremos de manera explícita el operador de asignación para dichas clases. Los operadores dirección (&) y coma (,) también pueden usarse con objetos de cualquier clase sin sobrecargarse. El operador dirección devuelve la dirección del objeto en memoria. El operador coma evalúa la expresión a su izquierda, y después la expresión a su derecha. Ambos operadores también se pueden sobrecargar.

La sobrecarga es en especial apropiada para las clases matemáticas. A menudo, estas clases requieren que se sobre cargue un conjunto sustancial de operadores para asegurar la consistencia con la forma en que se manejan estas clases matemáticas en el mundo real. Por ejemplo, sería inusual sobrecargar sólo la suma para una clase de números complejos, ya que también hay otros operadores aritméticos que se utilizan comúnmente con los números complejos.

La sobrecarga de operadores proporciona las mismas expresiones concisas y familiares para los tipos definidos por el usuario que C++ proporciona con su vasta colección de operadores para los tipos fundamentales. La sobrecarga de operadores no es automática; el programador debe escribir funciones de sobrecarga de operadores para realizar las operaciones deseadas. Algunas veces, estas funciones se desempeñan mejor como funciones miembro; otras veces son mejores como funciones `friend`; en algunas ocasiones se pueden hacer funciones no `friend` globales. Más adelante presentaremos ejemplos de estas posibilidades.

11.3 Restricciones acerca de la sobrecarga de operadores

La mayoría de los operadores de C++ se pueden sobrecargar. Éstos se muestran en la figura 11.1. En la figura 11.2 se muestran los operadores que no se pueden sobrecargar.



Error común de programación 11.1

Tratar de sobrecargar un operador no sobrecargable es un error de sintaxis.

Precedencia, asociatividad y número de operandos

La precedencia de un operador no se puede cambiar mediante la sobre carga. Esto puede provocar situaciones extrañas en las que un operador se sobre carga de una forma para la que su precedencia fija es inapropiada. Sin embargo, se pueden utilizar paréntesis para forzar el orden de evaluación de los operadores sobre cargados en una expresión.

La asociatividad de un operador (es decir, si el operador se aplica de derecha a izquierda o de izquierda a derecha) no se puede cambiar mediante la sobre carga.

No es posible modificar la “aridad” de un operador (es decir, el número de operandos que recibe): los operadores unarios sobre cargados siguen siendo operadores unarios; los operadores binarios sobre cargados siguen siendo operadores binarios. El único operador ternario (`? :`) no se puede sobre cargar. Los operadores `&`, `*`, `+` y `-` tienen versiones unarias y binarias; cada una de estas versiones unarias y binarias se pueden sobre cargar.



Error común de programación 11.2

Tratar de cambiar la “aridad” de un operador a través de la sobre carga de operadores es un error de compilación.

Creación de nuevos operadores

No es posible crear nuevos operadores; sólo los operadores existentes se pueden sobre cargar. Por desgracia, esto evita que podamos usar notaciones populares como el operador `**` que se utiliza en algunos otros lenguajes de programación para la exponentiación. [Nota: podríamos sobre cargar un operador existente para realizar la exponentiación].



Error común de programación 11.3

Tratar de crear nuevos operadores a través de la sobre carga de operadores es un error de sintaxis.

Operadores para los tipos fundamentales

El significado de la forma en que trabaja un operador con objetos de tipos fundamentales no se puede modificar mediante la sobre carga de operadores. Por ejemplo, no podemos modificar el significado de la forma en como `+` suma dos enteros. La sobre carga de operadores funciona sólo con objetos de los tipos definidos por el usuario, o con una mezcla de un objeto de un tipo definido por el usuario y un objeto de un tipo fundamental.



Observación de Ingeniería de Software 11.2

Por lo menos un argumento de una función de operador debe ser un objeto o referencia de un tipo definido por el usuario. Esto evita que los programadores modifiquen la forma en que funcionan los operadores con los tipos fundamentales.

Operadores que se pueden sobre cargar

<code>+</code>	<code>-</code>	<code>*</code>	<code>/</code>	<code>%</code>	<code>^</code>	<code>&</code>	<code> </code>
<code>~</code>	<code>!</code>	<code>=</code>	<code><</code>	<code>></code>	<code>+=</code>	<code>-=</code>	<code>*=</code>
<code>/=</code>	<code>%=</code>	<code>^=</code>	<code>&=</code>	<code> =</code>	<code><<</code>	<code>>></code>	<code>>>=</code>
<code><<=</code>	<code>==</code>	<code>!=</code>	<code><=</code>	<code>>=</code>	<code>&&</code>	<code> </code>	<code>++</code>
<code>--</code>	<code>->*</code>	<code>,</code>	<code>-></code>	<code>[]</code>	<code>()</code>	<code>new</code>	<code>delete</code>
<code>new[]</code>	<code>delete[]</code>						

Figura 11.1 | Operadores que se pueden sobre cargar.

Operadores que no se pueden sobre cargar

<code>.</code>	<code>*</code>	<code>::</code>	<code>?:</code>
----------------	----------------	-----------------	-----------------

Figura 11.2 | Operadores que no se pueden sobre cargar.



Error común de programación 11.4

Tratar de modificar la forma en que funciona un operador con objetos de los tipos fundamentales es un error de compilación.

Operadores relacionados

La sobrecarga de un operador de asignación y un operador de suma para permitir instrucciones como

```
objeto2 = objeto2 + objeto1;
```

no implica que el operador `+=` también se sobrecarga para permitir instrucciones tales como

```
objeto2 += objeto1;
```

Dicho comportamiento se puede obtener sólo mediante la sobrecarga explícita del operador `+=` para esa clase.



Error común de programación 11.5

Suponer que al sobrecargar un operador como `+` se sobrecargan los operadores relacionados tales como `+=`, o que al sobrecargar `==` se sobrecarga un operador relacionado tal como `!=`, puede provocar errores. Los operadores se pueden sobrecargar sólo de manera explícita; no hay sobrecarga implícita.

11.4 Las funciones de operadores como clase miembro vs. funciones globales

Las funciones de operadores pueden ser funciones miembro o funciones globales; por lo general, las funciones globales se hacen `friend` por cuestiones de rendimiento. Las funciones miembro utilizan el apuntador `this` de manera implícita para obtener uno de los argumentos de los objetos de su clase (el operando izquierdo para operadores binarios). Los argumentos para ambos operandos de un operador binario deben listarse de manera explícita en la llamada a una función global.

Operadores que deben sobrecargarse como funciones miembro

Al sobrecargar `()`, `[]`, `->` o cualquiera de los operadores de asignación, la función de sobrecarga de operadores debe declararse como una clase miembro. Para los otros operadores, las funciones de sobrecarga de operadores pueden ser clases miembro o funciones globales.

Operadores como funciones miembro y funciones globales

Ya sea que una función de operador se implemente como función miembro o como función global, el operador se sigue utilizando de la misma forma en las expresiones. Entonces, ¿cuál implementación es mejor?

Cuando una función de operador se implementa como función miembro, el operando de más a la izquierda (o el único) debe ser un objeto (o una referencia a un objeto) de la clase del operador. Si el operando izquierdo debe ser un objeto de una clase distinta o de un tipo fundamental, esta función de operador se debe implementar como una función global (como haremos en la sección 11.5, al sobrecargar `<< y >>` como los operadores de inserción de flujo y de extracción de flujo, respectivamente). Una función de operador global puede convertirse en `friend` de una clase, si esa función debe aceptar directamente miembros `private` o `protected` de esa clase.

Las funciones miembro de operador de una clase específica se llaman (de manera implícita por el compilador) sólo cuando el operando izquierdo de un operador binario es específicamente un objeto de esa clase, o cuando el único operando de un operador unario es un objeto de esa clase.

Por qué los operadores de inserción de flujo y de extracción de flujo sobrecargados se sobrecargan como funciones globales

El operador de inserción de flujo (`<<`) sobrecargado se utiliza en una expresión en la que el operando izquierdo tiene el tipo `ostream &`, como en `cout << objetoClase`. Para usar el operador de esta manera en donde el operando `derecho` es un objeto de una clase definida por el usuario, debe sobrecargarse como una función global. Para ser una función miembro, el operador `<<` tendría que ser una clase miembro `ostream`. Esto no es posible para las clases definidas por el usuario, ya que no se nos permite modificar las clases de la Biblioteca estándar de C++. De manera similar, el operador de extracción de flujo sobrecargado (`>>`) se utiliza en una expresión en la que el operando izquierdo tiene el tipo `istream &`, como en `cin >> objetoClase`, y el operando `derecho` es un objeto de una clase definida por el usuario, por lo que también debe ser una función global. Además, cada una de estas funciones de operador sobrecargadas pueden requerir acceso a los miembros de datos `private` del objeto de la clase que se va a enviar o recibir, por lo que estas funciones de operador sobrecargadas se pueden convertir en funciones `friend` de la clase, por cuestiones de rendimiento.



Tip de rendimiento 11.1

Es posible sobre cargar un operador como una función no `friend` global, pero dicha función que requiera acceso a los datos `private` o `protected` de una clase tendría que usar las funciones `set` o `get` que se proporcionen en la interfaz `public` de esa clase. La sobre carga de llamar a estas funciones podría provocar un rendimiento pobre, por lo que estas funciones se pueden poner en línea para mejorar el rendimiento.

Operadores conmutativos

Otra razón por la que podríamos elegir una función global para sobre cargar un operador sería para permitir que el operador fuera conmutativo. Por ejemplo, suponga que tenemos un objeto llamado `numero` de tipo `long int`, y un objeto llamado `enteroGrande1` de la clase `EnteroEnorme` (una clase en la que los enteros pueden ser arbitrariamente grandes, en vez de estar limitados por el tamaño de palabra del hardware subyacente del equipo; desarrollaremos la clase `EnteroEnorme` en los ejercicios del capítulo). El operador de suma (+) produce un objeto `EnteroEnorme` temporal como la suma de un `EnteroEnorme` y un `long int` (como en la expresión `enteroGrande1 + numero`), o como la suma de un `long int` y un `EnteroEnorme` (como en la expresión `numero + enteroGrande1`). Por ende, requerimos que el operador de suma sea conmutativo (exactamente como es con dos operandos de los tipos fundamentales). El problema es que el objeto de la clase debe aparecer del lado *izquierdo* del operador de suma, si ese operador se va a sobre cargar como función miembro. Por lo tanto, sobre cargamos el operador como una función global para permitir que el `EnteroEnorme` aparezca del lado *derecho* de la suma. La función `operator+`, que lidiá con el `EnteroEnorme` a la izquierda, puede seguir siendo una función miembro. La función global simplemente intercambia sus argumentos y llama a la función miembro.

11.5 Sobre carga de los operadores de inserción de flujo y extracción de flujo

C++ puede recibir y enviar los tipos fundamentales mediante el uso del operador de extracción de flujo `>>` y del operador de inserción de flujo `<<`. Las bibliotecas de clases que se proporcionan con los compiladores de C++ sobre cargan estos operadores para procesar cada tipo fundamental, incluyendo apuntadores y cadenas `char *` estilo C. Los operadores de inserción de flujo y extracción de flujo también se pueden sobre cargar para realizar operaciones de entrada y salida para los tipos definidos por el usuario. El programa de las figuras 11.3 a 11.5 demuestra cómo sobre cargar estos operadores para manejar datos de una clase de número telefónico definida por el usuario, llamada `NumeroTelefonico`. Este programa supone que los números telefónicos se introducen de manera correcta.

```

1 // Fig. 11.3: NumeroTelefonico.h
2 // Definición de la clase NumeroTelefonico
3 #ifndef NUMEROTELEFONICO_H
4 #define NUMEROTELEFONICO_H
5
6 #include <iostream>
7 using std::ostream;
8 using std::istream;
9
10 #include <string>
11 using std::string;
12
13 class NumeroTelefonico
14 {
15     friend ostream &operator<<( ostream &, const NumeroTelefonico & );
16     friend istream &operator>>( istream &, NumeroTelefonico & );
17 private:
18     string codigoArea; // código de área de 3 dígitos
19     string intercambio; // intercambio de 3 dígitos
20     string linea; // línea de 4 dígitos
21 }; // fin de la clase NumeroTelefonico
22
23 #endif

```

Figura 11.3 | Clase `NumeroTelefonico` con los operadores de inserción de flujo y de extracción de flujo sobre cargados como funciones `friend`.

```

1 // Fig. 11.4: NumeroTelefonico.cpp
2 // Operadores de inserción de flujo y de extracción de flujo sobrecargados
3 // para la clase NumeroTelefonico.
4 #include <iomanip>
5 using std::setw;
6
7 #include "NumeroTelefonico.h"
8
9 // operador de inserción de flujo sobrecargado; no puede ser
10 // una función miembro si deseamos invocarlo con
11 // cout << unNumeroTelefonico;
12 ostream &operator<<( ostream &salida, const NumeroTelefonico &numero )
13 {
14     salida << "(" << numero.codigoArea << ") "
15         << numero.intercambio << "-" << numero.linea;
16     return salida; // permite cout << a << b << c;
17 } // fin de la función operator<<
18
19 // operador de extracción de flujo sobrecargado; no puede ser
20 // una función miembro si deseamos invocarlo con
21 // cin >> unNumeroTelefonico;
22 istream &operator>>( istream &input, NumeroTelefonico &numero )
23 {
24     input.ignore(); // omite (
25     input >> setw( 3 ) >> numero.codigoArea; // recibe el código de área
26     input.ignore( 2 ); // omite ) y espacio
27     input >> setw( 3 ) >> numero.intercambio; // recibe intercambio
28     input.ignore(); // omite el guión corto (-)
29     input >> setw( 4 ) >> numero.linea; // recibe linea
30     return input; // permite cin >> a >> b >> c;
31 } // fin de la función operator>>

```

Figura 11.4 | Operadores de inserción de flujo y de extracción de flujo sobrecargados para la clase `NumeroTelefonico`.

```

1 // Fig. 11.5: fig11_05.cpp
2 // Demostración de los operadores de inserción de flujo y de extracción
3 // de flujo sobrecargados de la clase NumeroTelefonico
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include "NumeroTelefonico.h"
10
11 int main()
12 {
13     NumeroTelefonico telefono; // crea el objeto telefono
14
15     cout << "Escriba el numero telefonico en la forma (123) 456-7890:" << endl;
16
17     // cin >> telefono invoca a operator>> generando de manera implícita
18     // la llamada a la función global operator>>( cin, telefono )
19     cin >> telefono;
20
21     cout << "El numero telefonico introducido fue: ";
22
23     // cout << telefono invoca a operator<< generando de manera implícita
24     // la llamada a la función global operator<<( cout, telefono )
25     cout << telefono << endl;
26     return 0;
27 } // fin de main

```

Figura 11.5 | Operadores de inserción de flujo y de extracción de flujo sobrecargados. (Parte I de 2).

```
Escriba el numero telefonico en la forma (123) 456-7890:  

(800) 555-1212  

El numero telefonico introducido fue: (800) 555-1212
```

Figura 11.5 | Operadores de inserción de flujo y de extracción de flujo sobre cargados. (Parte 2 de 2).

La función del operador de extracción de flujo `operator>>` (figura 11.4, líneas 22 a 31) recibe la referencia `istream` llamada `entrada` y la referencia `NumeroTelefonico` llamada `num` como argumentos, y devuelve una referencia `istream`. La función de operador `operator>>` introduce números telefónicos de la forma

```
(800) 555-1212
```

en objetos de la clase `NumeroTelefonico`. Cuando el compilador ve la expresión

```
cin >> teléfono
```

en la línea 19 de la figura 11.5, el compilador genera la llamada a la función global

```
operator>>( cin, teléfono );
```

Cuando se ejecute esta llamada, el parámetro de referencia `entrada` (figura 11.4, línea 22) se convierte en un alias para `cin` y el parámetro de referencia `numero` se convierte en un alias para `teléfono`. La función de operador lee como objetos `string` las tres partes del número telefónico y las coloca en los miembros `codigoArea` (línea 25), `intercambio` (línea 27) y `linea` (línea 29) del objeto `NumeroTelefonico` al que hace referencia el parámetro `numero`. El manipulador de flujo `setw` limita el número de caracteres que se lean y se colocan en cada arreglo de caracteres. Cuando se utiliza con `cin` y objetos `string`, `setw` restringe el número de caracteres leídos al número de caracteres especificados por su argumento (es decir, `setw(3)` permite que se lean tres caracteres). Los paréntesis, espacios y guiones cortos se omiten mediante una llamada a la función miembro `ignore` de `istream` (figura 11.4, líneas 24, 26 y 28), la cual descarta el número especificado de caracteres en el flujo de entrada (un carácter de manera predeterminada). La función `operator>>` devuelve la referencia `istream` llamada `entrada` (es decir, `cin`). Esto permite que las operaciones de entrada en los objetos `NumeroTelefonico` se realicen en cascada con las operaciones de entrada en otros objetos `NumeroTelefonico`, o en objetos de otros tipos de datos. Por ejemplo, un programa puede recibir dos objetos `NumeroTelefonico` en una instrucción de la siguiente manera:

```
cin >> teléfono1 >> teléfono2;
```

Primero se ejecuta la expresión `cin >> teléfono1`, mediante una llamada a la función global

```
operator>>( cin, teléfono1 );
```

Después, esta llamada devuelve una referencia a `cin` como el valor de `cin >> teléfono1`, por lo que la porción restante de la expresión se interpreta simplemente como `cin >> teléfono2`. Esto se ejecuta haciendo una llamada a la función global

```
operator>>( cin, teléfono2 );
```

La función del operador de inserción de flujo (figura 11.4, líneas 12 a 17) recibe una referencia `ostream` (`salida`) y una referencia `const NumeroTelefonico` (`numero`) como argumentos, y devuelve una referencia `ostream`. La función `operator<<` muestra objetos de tipo `NumeroTelefonico`. Cuando el compilador ve la expresión

```
cout << teléfono
```

en la línea 25 de la figura 11.5, genera una llamada a la función global

```
operator<<( cout, teléfono );
```

La función `operator<<` muestra las partes del número telefónico como objetos `string`, ya que se almacenan como objetos `string`.



Tip para prevenir errores 11.1

Por lo general, la acción de devolver una referencia de una función del operador << o >> sobre cargado es exitosa, debido a que cout, cin y la mayoría de los objetos de flujo son globales, o por lo menos de larga duración. Es peligroso devolver una referencia a una variable automática u otro objeto temporal; esto puede crear “referencias suspendidas” a objetos inexistentes.

Observe que las funciones `operator>>` y `operator<<` se declaran en `NumeroTelefonico` como funciones `friend` globales (figura 11.3, líneas 15 y 16). Son funciones globales debido a que el objeto de la clase `NumeroTelefonico` aparece en cada caso como el operando derecho del operador. Recuerde que las funciones de operadores sobre cargados para los operadores binarios pueden ser funciones miembro sólo cuando el operando izquierdo es un objeto de la clase en la que la función es miembro. Los operadores de entrada y salida sobre cargados se declaran como `friend` si necesitan acceder a los miembros no `public` de las clases miembro directamente por cuestiones de rendimiento, o debido a que la clase tal vez no ofrezca funciones `get` apropiadas. Observe además que la referencia `NumeroTelefonico` en la lista de parámetros de la función `operator<<` (figura 11.4, línea 12) es `const`, ya que el objeto `NumeroTelefonico` simplemente se enviará a la salida, y la referencia `NumeroTelefonico` en la lista de parámetros de la función `operator>>` (línea 22) no es `const`, ya que el objeto `NumeroTelefonico` debe modificarse para almacenar el número telefónico de entrada en el objeto.



Observación de Ingeniería de Software 11.3

Las nuevas herramientas de entrada/salida para los tipos definidos por el usuario se agregan a C++ sin modificar las clases de la biblioteca de entrada/salida estándar. Éste es otro ejemplo de la extensibilidad de C++.

11.6 Sobre carga de operadores unarios

Un operador unario para una clase se puede sobre cargar como función miembro no `static` sin argumentos, o como función global con un argumento; ese argumento debe ser un objeto de la clase, o una referencia a un objeto de la misma. Las funciones miembro que implementan operadores sobre cargados deben ser no `static`, de manera que puedan acceder a los datos no `static` en cada objeto de la clase. Recuerde que las funciones miembro `static` sólo pueden acceder a los miembros `static` de la clase.

Más adelante en el capítulo sobre cargaremos el operador unario `!` para evaluar si un objeto de la clase `String` que creamos (sección 11.10) está vacío, y devolver un resultado `bool`. Considere la expresión `!s`, en la que `s` es un objeto de la clase `String`. Cuando un operador unario tal como `!` se sobre carga como función miembro sin argumentos y el compilador ve la expresión `!s`, genera la llamada a la función `s.operator!()`. El operando `s` es el objeto de la clase para el que se está invocando la función miembro `operator!` de la clase `String`. La función se declara en la definición de la clase de la siguiente manera:

```
class String
{
public:
    bool operator!() const;
    ...
}; // fin de la clase String
```

Un operador unario tal como `!` se puede sobre cargar como función global con un parámetro, de dos maneras distintas: con un parámetro que sea un objeto (para ello se requiere una copia del objeto, de manera que los efectos secundarios de la función no se apliquen al objeto original), o con un parámetro que sea una referencia a un objeto (no se hace una copia del objeto original, por lo que todos los efectos secundarios de esta función se aplican al objeto original). Si `s` es un objeto de la clase `String` (o una referencia a un objeto de la clase `String`), entonces `!s` se trata como si se hubiera escrito la llamada `operator!(s)`, invocando a la función global `operator!` que se declara de la siguiente manera:

```
bool operator!( const String & );
```

11.7 Sobre carga de operadores binarios

Un operador binario se puede sobre cargar como una función miembro no `static` con un parámetro, o como una función global con dos parámetros (uno de esos parámetros debe ser el objeto de una clase o una referencia al objeto de una clase).

Más adelante en este capítulo, sobre cargaremos el operador `<` para comparar dos objetos `String`. Al sobre cargar el operador binario `<` como una función miembro no `static` de una clase `String` con un argumento, si `y` y `z` son objetos de clases `String`, entonces `y < z` se considera como si se hubiera escrito `y.operator<(z)`, invocando a la función miembro `operator<` que se declara a continuación:

```
class String
{
public:
    bool operator<( const String & ) const;
    ...
}; // fin de la clase String
```

Si el operador binario `<` se va a sobre cargar como una función global, debe recibir dos argumentos (uno de los cuales debe ser el objeto de una clase, o una referencia al objeto de una clase). Si `y` y `z` son objetos de la clase `String` o referencias a objetos de esa clase, entonces `y < z` se considera como si se hubiera escrito la llamada `operator<(y, z)` en el programa, invocando a la función global `operator<`, que se declara a continuación:

```
bool operator<( const String &, const String & );
```

11.8 Ejemplo práctico: la clase Array

Los arreglos basados en apuntadores tienen varios problemas. Por ejemplo, un programa puede “salirse” fácilmente de cualquier extremo de un arreglo, ya que C++ no comprueba si los subíndices están fuera del rango de un arreglo (de todas formas se puede hacer esto explícitamente). Los arreglos de tamaño n deben enumerar sus elementos así: $0, \dots, n - 1$; no se permiten rangos de subíndices alternados. Un arreglo que no sea `char` completo no puede recibirse ni enviarse todo a la vez; se debe leer o escribir cada elemento del arreglo de manera individual. Dos arreglos no pueden compararse significativamente con los operadores de igualdad o relacionales (debido a que los nombres de los arreglos son simplemente apuntadores a la dirección en la que empiezan los arreglos en memoria y, desde luego, dos arreglos siempre estarán en distintas ubicaciones de memoria). Cuando se pasa un arreglo a una función de propósito especial diseñada para manejar arreglos de cualquier tamaño, el tamaño del arreglo debe pasarse como argumento adicional. Un arreglo no puede asignarse a otro con el(es) operador(es) de asignación (debido a que los nombres de los arreglos son apuntadores `const`, y un apuntador constante no se puede utilizar del lado izquierdo de un operador de asignación). Éstas y otras herramientas parecen sin duda la “opción natural” para lidiar con los arreglos, pero los arreglos basados en apuntador no proporcionan dichas herramientas. Sin embargo, C++ proporciona los medios para implementar dichas herramientas de los arreglos, a través del uso de las clases y la sobre carga de operadores.

En este ejemplo, vamos a crear una poderosa clase tipo arreglo que realiza comprobación de rangos para asegurar que los subíndices permanezcan dentro de los límites del objeto `Array`. La clase permite asignar un objeto `Array` a otro mediante el operador de asignación. Los objetos de la clase `Array` conocen su tamaño, por lo que éste no se necesita pasar por separado como argumento, al pasar un objeto `Array` a una función. Pueden recibirse o enviarse objetos `Array` completos mediante los operadores de extracción de flujo e inserción de flujo, respectivamente. Pueden realizarse comparaciones entre objetos `Array` mediante los operadores de igualdad `==` y `!=`.

En el siguiente ejemplo, el lector agudizará su apreciación acerca de la abstracción de datos. Probablemente quiera sugerir otras mejoras a esta clase `Array`. El desarrollo de clases es una actividad interesante, creativa e intelectualmente retadora; siempre con el objetivo de “creación de clases valiosas”.

El programa de las figuras 11.6 a 11.8 demuestra la clase `Array` y sus operadores sobre cargados. Primero recorremos `main` (figura 11.8). Despues consideraremos la definición de la clase (figura 11.6) y cada una de las definiciones de las funciones miembro y funciones `friend` de la clase (figura 11.7).

```

1 // Fig. 11.6: Array.h
2 // Definición de la clase Array con operadores sobre cargados .
3 #ifndef ARRAY_H
4 #define ARRAY_H
5
6 #include <iostream>
7 using std::ostream;
8 using std::istream;
9
10 class Array
11 {
12     friend ostream &operator<<( ostream &, const Array & );
13     friend istream &operator>>( istream &, Array & );
14 public:
15     Array( int = 10 ); // constructor predeterminado
16     Array( const Array & ); // constructor de copia
17     ~Array(); // destructor
18     int getTamanio() const; // devuelve el tamaño
19
20     const Array &operator=( const Array & ); // operador de asignación
21     bool operator==( const Array & ) const; // operador de igualdad

```

Figura 11.6 | Definición de la clase `Array` con operadores sobre cargados. (Parte 1 de 2).

```

22 // operador de desigualdad; devuelve el opuesto del operador ==
23 bool operator!=( const Array &derecho ) const
24 {
25     return ! ( *this == derecho ); // invoca a Array::operator==
26 } // fin de la función operator!=
27
28 // el operador de subíndice para los objetos no const devuelve un lvalue modificable
29 int &operator[]( int );
30
31 // el operador de subíndice para los objetos const devuelve rvalue
32 int operator[]( int ) const;
33
34 private:
35     int tamano; // arreglo tamaño basado en apuntador
36     int *ptr; // apuntador al primer elemento del arreglo basado en apuntador
37 }; // fin de la clase Arreglo
38
39 #endif

```

Figura 11.6 | Definición de la clase `Array` con operadores sobrecargados. (Parte 2 de 2).

```

1 // Fig 11.7: Array.cpp
2 // Definiciones de las funciones miembro y friend de la clase Array.
3 #include <iostream>
4 using std::cerr;
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include <iomanip>
10 using std::setw;
11
12 #include <cstdlib> // prototipo de la función exit
13 using std::exit;
14
15 #include "Array.h" // Definición de la clase Array
16
17 // constructor predeterminado para la clase Array (tamaño predeterminado 10)
18 Array::Array( int tamanoArreglo )
19 {
20     tamano = ( tamanoArreglo > 0 ? tamanoArreglo : 10 ); // valida tamanoArreglo
21     ptr = new int[ tamano ]; // crea espacio para el arreglo basado en apuntador
22
23     for ( int i = 0; i < tamano; i++ )
24         ptr[ i ] = 0; // establece elemento de arreglo basado en apuntador
25 } // fin del constructor predeterminado de Array
26
27 // constructor de copia para la clase Array;
28 // debe recibir una referencia para evitar la recursividad infinita
29 Array::Array( const Array &arregloACopiar )
30     : tamano( arregloACopiar.tamano )
31 {
32     ptr = new int[ tamano ]; // crea espacio para el arreglo basado en apuntador
33
34     for ( int i = 0; i < tamano; i++ )
35         ptr[ i ] = arregloACopiar.ptr[ i ]; // lo copia en el objeto
36 } // fin del constructor de copia de Array
37
38 // destructor para la clase Array
39 Array::~Array()
40 {

```

Figura 11.7 | Definiciones de las funciones miembro y funciones friend de la clase `Array`. (Parte 1 de 3).

```

41     delete [] ptr; // libera el espacio del arreglo basado en apuntador
42 } // fin del destructor
43
44 // devuelve el número de elementos del objeto Array
45 int Array::getTamanio() const
46 {
47     return tamanio; // número de elementos en el objeto Array
48 } // fin de la función getTamanio
49
50 // operador de asignación sobre cargado;
51 // devolución de const evita: ( a1 = a2 ) = a3
52 const Array &Array::operator=( const Array &derecho )
53 {
54     if ( &derecho != this ) // evita la auto-asignación
55     {
56         // para los objetos Array de distintos tamaños, desasigna el arreglo
57         // original del lado izquierdo, después asigna el nuevo arreglo del lado izquierdo
58         if ( tamanio != derecho.tamanio )
59         {
60             delete [] ptr; // libera espacio
61             tamanio = derecho.tamanio; // cambia el tamaño de este objeto
62             ptr = new int[ tamanio ]; // crea espacio para la copia del arreglo
63         } // fin del if interior
64
65         for ( int i = 0; i < tamanio; i++ )
66             ptr[ i ] = derecho.ptr[ i ]; // copia el arreglo en el objeto
67     } // fin del if exterior
68
69     return *this; // permite x = y = z, por ejemplo
70 } // fin de la función operator=
71
72 // determina si dos objetos Array son iguales y
73 // devuelve true, en caso contrario devuelve false
74 bool Array::operator==( const Array &derecho ) const
75 {
76     if ( tamanio != derecho.tamanio )
77         return false; // arreglos con distinto número de elementos
78
79     for ( int i = 0; i < tamanio; i++ )
80         if ( ptr[ i ] != derecho.ptr[ i ] )
81             return false; // el contenido de los objetos Array no es igual
82
83     return true; // los objetos Array son iguales
84 } // fin de la función operator==
85
86 // operador de subíndice sobre cargado para objetos Array no const;
87 // la devolución de una referencia crea un lvalue modificable
88 int &Array::operator[]( int subindice )
89 {
90     // comprueba error de subíndice fuera de rango
91     if ( subindice < 0 || subindice >= tamanio )
92     {
93         cerr << "\nError: subíndice " << subindice
94         << " fuera de rango" << endl;
95         exit( 1 ); // termina el programa; subíndice fuera de rango
96     } // fin de if
97
98     return ptr[ subindice ]; // devuelve una referencia
99 } // fin de la función operator[]
100
101 // operador de subíndice sobre cargado para objetos Array const
102 // devolución de referencia const crea un rvalue

```

Figura 11.7 | Definiciones de las funciones miembro y funciones friend de la clase Array. (Parte 2 de 3).

```

103 int Array::operator[]( int subindice ) const
104 {
105     // comprueba error de subíndice fuera de rango
106     if ( subindice < 0 || subindice >= tamano )
107     {
108         cerr << "\nError: subíndice " << subindice
109             << " fuera de rango" << endl;
110         exit( 1 ); // termina el programa; subíndice fuera de rango
111     } // fin de if
112
113     return ptr[ subindice ]; // devuelve una copia de este elemento
114 } // fin de la función operator[]
115
116 // operador de entrada sobrecargado para la clase Array;
117 // recibe valores para el objeto Array completo
118 istream &operator>>( istream &entrada, Array &a )
119 {
120     for ( int i = 0; i < a.tamano; i++ )
121         entrada >> a.ptr[ i ];
122
123     return entrada; // permite cin >> x >> y;
124 } // fin de la función
125
126 // operador de salida sobrecargado para la clase Array
127 ostream &operator<<( ostream &salida, const Array &a )
128 {
129     int i;
130
131     // imprime arreglo private basado en ptr
132     for ( i = 0; i < a.tamano; i++ )
133     {
134         salida << setw( 12 ) << a.ptr[ i ];
135
136         if ( ( i + 1 ) % 4 == 0 ) // 4 números por fila de salida
137             salida << endl;
138     } // fin de for
139
140     if ( i % 4 != 0 ) // fin de la última línea de salida
141         salida << endl;
142
143     return salida; // permite cout << x << y;
144 } // fin de la función operator<<

```

Figura 11.7 | Definiciones de las funciones miembro y funciones friend de la clase Array. (Parte 3 de 3).

```

1 // Fig. 11.8: fig11_08.cpp
2 // Programa de prueba de la clase Array.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include "Array.h"
9
10 int main()
11 {
12     Array enteros1( 7 ); // objeto Array de 7 elementos
13     Array enteros2; // objeto Array de 10 elementos de manera predeterminada
14
15     // imprime el tamaño y contenido de enteros1
16     cout << "El tamaño del objeto Array enteros1 es "

```

Figura 11.8 | Programa de prueba de la clase Array. (Parte 1 de 3).

```

17     << enteros1.getTamanio()
18     << "\nEl objeto Array despues de la inicializacion es:\n" << enteros1;
19
20 // imprime el tamaño y el contenido de enteros2
21 cout << "\nEl tamano del objeto Array enteros2 es "
22     << enteros2.getTamanio()
23     << "\nEl objeto Array despues de la inicializacion es:\n" << enteros2;
24
25 // recibe e imprime enteros1 y enteros2
26 cout << "\nIntroduzca 17 enteros:" << endl;
27 cin >> enteros1 >> enteros2;
28
29 cout << "\nDespues de la entrada, los objetos Array contienen:\n"
30     << "enteros1:\n" << enteros1
31     << "enteros2:\n" << enteros2;
32
33 // usa el operador de desigualdad (!=) sobre cargado
34 cout << "\nEvaluando: enteros1 != enteros2" << endl;
35
36 if ( enteros1 != enteros2 )
37     cout << "enteros1 y enteros2 no son iguales" << endl;
38
39 // crea el objeto Array enteros3, usando enteros1 como
40 // inicializador; imprime el tamaño y el contenido
41 Array enteros3( enteros1 ); // invoca el constructor de copia
42
43 cout << "\nEl tamano del objeto Array enteros3 es "
44     << enteros3.getTamanio()
45     << "\nObjeto Array despues de la inicializacion:\n" << enteros3;
46
47 // usa el operador de asignación (=) sobre cargado
48 cout << "\nAsignando enteros2 a enteros1:" << endl;
49 enteros1 = enteros2; // observe que el objeto Array de destino es más pequeño
50
51 cout << "enteros1:\n" << enteros1
52     << "enteros2:\n" << enteros2;
53
54 // usa el operador de igualdad (==) sobre cargado
55 cout << "\nEvaluando: enteros1 == enteros2" << endl;
56
57 if ( enteros1 == enteros2 )
58     cout << "enteros1 y enteros2 son iguales" << endl;
59
60 // usa el operador de subíndice sobre cargado para crear rvalue
61 cout << "\nEnteros1[5] es " << enteros1[ 5 ];
62
63 // usa el operador de subíndice sobre cargado para crear lvalue
64 cout << "\n\nAsignando 1000 a enteros1[5]" << endl;
65 enteros1[ 5 ] = 1000;
66 cout << "enteros1:\n" << enteros1;
67
68 // trata de usar un subíndice fuera de rango
69 cout << "\nTrata de asignar 1000 a enteros1[15]" << endl;
70 enteros1[ 15 ] = 1000; // ERROR: fuera de rango
71
72 return 0;
73 } // fin de main

```

El tamano del objeto Array enteros1 es 7

El objeto Array despues de la inicializacion es:

0	0	0	0
0	0	0	

Figura 11.8 | Programa de prueba de la clase Array. (Parte 2 de 3).

```
El tamaño del objeto Array enteros2 es 10
El objeto Array después de la inicialización es:
    0      0      0      0
    0      0      0      0
    0      0
```

Introduzca 17 enteros:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

Después de la entrada, los objetos Array contienen:
enteros1:

1	2	3	4
5	6	7	

enteros2:

8	9	10	11
12	13	14	15
16	17		

Evaluando: **enteros1 != enteros2**
enteros1 y **enteros2** no son iguales

El tamaño del objeto Array enteros3 es 7
Objeto Array después de la inicialización:

1	2	3	4
5	6	7	

Asignando **enteros2** a **enteros1**:
enteros1:

8	9	10	11
12	13	14	15
16	17		

enteros2:

8	9	10	11
12	13	14	15
16	17		

Evaluando: **enteros1 == enteros2**
enteros1 y **enteros2** son iguales

enteros1[5] es 13

Asignando 1000 a **enteros1[5]**
enteros1:

8	9	10	11
12	1000	14	15
16	17		

Trata de asignar 1000 a **enteros1[15]**

Error: subíndice 15 fuera de rango

Figura 11.8 | Programa de prueba de la clase Array. (Parte 3 de 3).

Cómo crear objetos Array, imprimir su tamaño y mostrar su contenido

El programa empieza por instanciar dos objetos de la clase Array: **enteros1** (figura 11.8, línea 12) con siete elementos, y **enteros2** (figura 11.8, línea 13) con el tamaño predeterminado de Array, 10 elementos (especificado por el prototipo del constructor predeterminado de **Array** en la figura 11.6, línea 15). En las líneas 16 a 18 se utiliza la función miembro **getTamaño** para determinar el tamaño de **enteros1** e imprimir **enteros1**, usando el operador de inserción de flujo sobrecargado de **Array**. La salida del ejemplo confirma que los elementos del objeto **Array** se establecieron correctamente en cero mediante el constructor. A continuación, en las líneas 21 a 23 se imprime el tamaño del objeto **Array** **enteros2** y se imprime **enteros2**, usando el operador de inserción de flujo sobrecargado de **Array**.

Uso del operador de inserción de flujo sobre cargado para llenar un objeto Array

En la línea 26 se pide al usuario que introduzca 17 enteros. En la línea 27 se utiliza el operador de extracción de flujo sobre cargado de `Array` para leer estos valores y colocarlos en ambos arreglos. Los primeros siete valores se almacenan en `enteros1` y los 10 valores restantes en `enteros2`. En las líneas 29 a 31 se imprimen los dos arreglos con el operador de inserción de flujo sobre cargado de `Array` para confirmar que la entrada se haya realizado de manera correcta.

Uso del operador de desigualdad sobre cargado

En la línea 36 se prueba el operador de desigualdad sobre cargado, evaluando la condición

```
enteros1 != enteros2
```

La salida del programa muestra que los objetos `Array` no son iguales.

Cómo inicializar un nuevo objeto Array con una copia del contenido de un objeto Array existente

En la línea 41 se crea una instancia de un tercer objeto `Array` llamado `entero3`, y se inicializa con una copia del objeto `Array` `enteros1`. Esto invoca al **constructor de copia** de `Array` para copiar los elementos de `enteros1` a `entero3`. En breve hablaremos sobre los detalles del constructor de copia. Observe que este constructor de copia también se puede invocar si escribimos la línea 41 de la siguiente manera:

```
Array enteros3 = enteros1;
```

El signo de igual en la instrucción anterior *no* es el operador de asignación. Cuando aparece un signo de igual en la declaración de un objeto, se invoca a un constructor para ese objeto. Esta forma se puede utilizar para pasar sólo un argumento a un constructor.

En las líneas 43 a 45 se imprime el tamaño de `enteros3` y se imprime el contenido de `enteros3`, usando el operador de inserción de flujo sobre cargado para confirmar que los elementos del objeto `Array` se hayan establecido correctamente mediante el constructor de copia.

Uso del operador de asignación sobre cargado

A continuación, en la línea 49 se prueba el operador de asignación sobre cargado (`=`) mediante la asignación de `enteros2` a `enteros1`. En las líneas 51 y 52 se imprimen ambos objetos `Array` para confirmar que la asignación haya tenido éxito. Observe que, en un principio, `enteros1` contenía 7 enteros y se cambió su tamaño para que pudiera contener una copia de los 10 elementos en `enteros2`. Como veremos más adelante, el operador de asignación sobre cargado realiza esta operación de ajuste de tamaño de una forma que sea transparente para el código cliente.

Uso del operador de igualdad sobre cargado

Después, en la línea 57 se utiliza el operador de igualdad sobre cargado (`==`) para confirmar que los objetos `enteros1` y `enteros2` sean sin duda idénticos después de la asignación.

Uso del operador de subíndice sobre cargado

En la línea 61 se utiliza el operador de subíndice sobre cargado para hacer referencia a `enteros1[5]`: un elemento de `enteros1` dentro del rango. Este nombre con subíndice se utiliza como *rvalue* para imprimir el valor asignado a `enteros1[5]`. En la línea 65 se utiliza `enteros1[5]` como un *lvalue* modificable del lado izquierdo de una instrucción de asignación para asignar un nuevo valor, 1000, al elemento 5 de `enteros1`. Más adelante veremos que `operator[]` devuelve una referencia para usarla como el *lvalue* modificable, una vez que el operador confirma que 5 es un subíndice válido para `enteros1`.

En la línea 70 se trata de asignar el valor 1000 a `enteros1[15]`; un elemento fuera de rango. En este ejemplo, `operator[]` determina que el subíndice está fuera de rango, imprime un mensaje y termina el programa. Observe que resaltamos la línea 70 del programa para enfatizar que es un error acceder a un elemento que está fuera de rango. Éste es un error lógico en tiempo de ejecución.

Es interesante observar que el operador de subíndice de arreglo `[]` no está restringido para usarlo sólo con arreglos; por ejemplo, también se puede utilizar para seleccionar elementos de otros tipos de clases contenedoras, como listas enlazadas, cadenas y diccionarios. Además, cuando se definen las funciones `operator[]`, los subíndices ya no tienen que ser enteros; también se pueden usar caracteres, cadenas, números de punto flotante o incluso objetos de clases definidas por el usuario. En el capítulo 22, Biblioteca de plantillas estándar (STL), hablaremos sobre la clase `map` de la STL que permite subíndices no enteros.

Definición de la clase `Array`

Ahora que hemos visto cómo opera este programa, vamos a recorrer el encabezado de la clase (figura 11.6). A medida que hagamos referencia a cada función miembro en el encabezado, hablaremos sobre la implementación de esa función en la figura 11.7. En la figura 11.6, en las líneas 35 y 36 se representan los miembros de datos `private` de la clase `Array`. Cada objeto `Array` consiste en un miembro `tamano` que indica el número de elementos en el objeto `Array` y un apuntador `int (ptr)` que apunta al arreglo de enteros basado en apuntador y asignado en forma dinámica que maneja el objeto `Array`.

Cómo sobrecargar los operadores de inserción de flujo y extracción de flujo como funciones `friend`

En las líneas 12 y 13 de la figura 11.6 se declaran el operador de inserción de flujo sobrecargado y el operador de extracción de flujo sobrecargado como funciones `friend` de la clase `Array`. Cuando el compilador ve una expresión como `cout << objetoArray`, invoca a la función global `operator<<` con la llamada

```
operator<<( cout, objetoArreglo )
```

Cuando el compilador ve una expresión como `cin >> objetoArray`, invoca a la función global `operator>>` con la llamada

```
operator>>( cin, objetoArray )
```

Observamos de nuevo que estas funciones de operador de inserción de flujo y operador de extracción de flujo no pueden ser miembros de la clase `Array`, ya que el objeto `Array` siempre se menciona del lado derecho del operador de inserción de flujo y del operador de extracción de flujo. Si estas funciones de operador fueran a ser miembros de la clase `Array`, tendrían que utilizarse las siguientes instrucciones extrañas para enviar y recibir un objeto `Array`:

```
objetoArray << cout;
objetoArray >> cin;
```

Dichas instrucciones serían confusas para la mayoría de los programadores de C++, que están familiarizados con el hecho de que `cout` y `cin` aparezcan como los operandos izquierdos de `<<` y `>>`, respectivamente.

La función `operator<<` (definida en la figura 11.7, líneas 127 a 144) imprime el número de elementos indicados por `tamano` a partir del arreglo de enteros al que apunta `ptr`. La función `operator>>` (definida en la figura 11.7, líneas 118 a 124) introduce los datos directamente en el arreglo al que apunta `ptr`. Cada una de estas funciones de operador devuelve una referencia apropiada para permitir instrucciones de salida o entrada en cascada, respectivamente. Observe que cada una de estas funciones tiene acceso a los datos `private` de un objeto `Array`, debido a que estas funciones se declaran como funciones `friend` de la clase `Array`. Observe además que las funciones `getTamanio` y `operator[]` de la clase `Array` podrían ser utilizadas por `operator<<` y `operator>>`, en cuyo caso estas funciones de operador no tendrían que ser funciones `friend` de la clase `Array`. Sin embargo, las llamadas a funciones adicionales podrían incrementar la sobrecarga en tiempo de ejecución.

Constructor predeterminado de `Array`

En la línea 15 de la figura 11.6 se declara el constructor predeterminado para la clase y se especifica un tamaño predeterminado de 10 elementos. Cuando el compilador ve una declaración como la línea 13 de la figura 11.8, invoca al constructor predeterminado de `Array` (recuerde que el constructor predeterminado en este ejemplo recibe en realidad un solo argumento `int`, que tiene un valor predeterminado de 10). El constructor predeterminado (definido en la figura 11.7, líneas 18 a 25) valida y asigna el argumento al miembro de datos `tamano`, utiliza `new` para obtener la memoria para la representación interna basada en apuntador de este arreglo y asigna el apuntador devuelto por `new` al miembro de datos `ptr`. Después, el constructor utiliza una instrucción `for` para establecer todos los elementos del arreglo en cero. Es posible tener una clase `Array` que no inicialice sus miembros si, por ejemplo, estos miembros se van a leer más adelante en cierto momento; pero esto se considera como mala práctica de programación. Los objetos `Array` (y los objetos en general) deben inicializarse de manera apropiada y mantenerse en un estado consistente.

Constructor de copia de `Array`

En la línea 16 de la figura 11.6 se declara un **constructor de copia** (definido en la figura 11.7, líneas 29 a 36) que inicializa un objeto `Array`, para lo cual crea una copia de un objeto `Array` existente. Dicha copia debe realizarse con cuidado, para evitar el problema de dejar ambos objetos `Array` apuntando a la misma memoria asignada en forma dinámica. Esto es exactamente el problema que ocurriría con la copia predeterminada a nivel de miembros, si el compilador tiene permitido definir un constructor de copia predeterminado para esta clase. Los constructores de copia se invocan cada vez que se necesita una copia de un objeto, como cuando se pasa un objeto por valor a una función, se devuelve un objeto por

valor de una función, o se inicializa un objeto con una copia de otro objeto de la misma clase. El constructor de copia se llama en una declaración cuando se instancia un objeto de la clase `Array` y se inicializa con otro objeto de la clase `Array`, como en la declaración en la línea 41 de la figura 11.8.



Observación de Ingeniería de Software 11.4

El argumento para un constructor de copia debe ser una referencia const para permitir que se copie un objeto const.



Error común de programación 11.6

Observe que un constructor de copia debe recibir su argumento por referencia, no por valor. En caso contrario, la llamada al constructor de copia produce recursividad infinita (un error lógico fatal), ya que para recibir un objeto por valor, el constructor de copia tiene que realizar una copia del objeto que se usa como argumento. Recuerde que cualquier vez que se requiere una copia de un objeto, se hace una llamada al constructor de copia de la clase. Si el constructor de copia recibió su argumento por valor, ¡se llamaría a sí mismo de manera recursiva para realizar una copia de su argumento!

El constructor de copia para `Array` utiliza un inicializador de miembros (figura 11.7, línea 30) para copiar el `tamaño` del objeto `Array` inicializador en el miembro de datos `tamano`, usa `new` (línea 32) para obtener la memoria para la representación interna basada en apuntador de este objeto `Array` y asigna el apuntador devuelto por `new` al datos miembro `ptr`.¹ Después, el constructor de copia utiliza una instrucción `for` para copiar todos los elementos del objeto `Array` inicializador en el nuevo objeto `Array`. Observe que un objeto de una clase puede mirar los datos `private` de cualquier objeto de esa clase (usando un manejador que indique a cuál objeto acceder).



Error común de programación 11.7

Si el constructor de copia simplemente copió el apuntador del objeto de origen al apuntador del objeto de destino, entonces ambos objetos apuntarían a la misma memoria asignada en forma dinámica. El primer destructor en ejecutarse eliminaría entonces la memoria asignada en forma dinámica, y el ptr del otro objeto quedaría indefinido, a lo cual se le conoce como apuntador suspendido; probablemente esto produciría un grave error en tiempo de ejecución (como la terminación anticipada del programa) a la hora de utilizar el apuntador.

Destructor de Array

En la línea 17 de la figura 11.6 se declara el destructor para la clase (definido en la figura 11.7, líneas 39 a 42). El destructor se invoca cuando un objeto de la clase `Array` queda fuera de alcance. El destructor usa `delete []` para liberar la memoria asignada en forma dinámica por `new` en el constructor.

Función miembro `getTamanio`

En la línea 18 de la figura 11.6 se declara la función `getTamanio` (definida en la figura 11.7, líneas 45 a 48), que devuelve el número de elementos en el objeto `Array`.

Operador de asignación sobrecargado

En la línea 20 de la figura 11.6 se declara la función del operador de asignación sobrecargado para la clase. Cuando el compilador ve la expresión `enteros1 = enteros2` en la línea 49 de la figura 11.8, invoca a la función miembro `operator=` con la llamada

```
enteros1.operator=( enteros2 )
```

La implementación de la función miembro `operator=` (figura 11.7, líneas 52 a 70) prueba la **auto-asignación** (línea 54), en la que un objeto de la clase `Array` se asigna a sí mismo. Cuando `this` es igual a la dirección del operando `derecho`, se intenta una auto-asignación, por lo que se omite la asignación (es decir, el objeto ya es en sí mismo; en un momento veremos por qué la auto-asignación es peligrosa). Si no es una auto-asignación, entonces la función miembro determina si los tamaños de los dos arreglos son idénticos (línea 58); en ese caso, el arreglo original de `enteros` en el objeto `Array` del lado izquierdo no se reasigna. En caso contrario, `operator=` utiliza `delete` (línea 60) para liberar la memoria originalmente asignada al arreglo de destino, copia el `tamano` del arreglo de origen al `tamano` del arreglo de destino (línea 61), usa `new` para asignar memoria para el arreglo de destino y coloca el apuntador devuelto por `new` en el miembro `ptr` del arreglo.² Después, la instrucción `for` en las líneas 65 y 66 copia los elementos del arreglo de origen al arreglo de destino.

1. Observe que `new` podría fallar al obtener la memoria necesaria. En el capítulo 16, Manejo de excepciones, trataremos con las fallas de `new`.
 2. Una vez más, `new` podría fallar. En el capítulo 16 hablaremos sobre las fallas de `new`.

Sin importar que ésta sea o no una auto-asignación, la función miembro devuelve el objeto actual (es decir, `*this` en la línea 69) como una referencia constante; esto permite asignaciones de objetos `Array` en cascada, como `x = y = z`. Si ocurre la auto-asignación, y la función `operator=` no probó este caso, `operator=` eliminaría la memoria dinámica asignada con el objeto `Array` antes de completar la asignación. Esto dejaría a `ptr` apuntando a memoria que fue previamente desasignada, lo cual podría producir errores fatales en tiempo de ejecución.



Observación de Ingeniería de Software 11.5

Un constructor de copia, un destructor y un operador de asignación sobrecargado se proporcionan generalmente como un grupo para cualquier clase que utilice memoria asignada en forma dinámica.



Error común de programación 11.8

Si no se proporcionan un operador de asignación sobrecargado y un constructor de copia para una clase cuando los objetos de esa clase contienen apuntadores a memoria asignada en forma dinámica, se produce un error lógico.



Observación de Ingeniería de Software 11.6

Es posible evitar que un objeto de una clase se asigne a otro. Para ello, se declara el operador de asignación como un miembro private de la clase.



Observación de Ingeniería de Software 11.7

Es posible evitar que se copien los objetos de una clase; para ello, simplemente se hacen private tanto el operador de asignación sobrecargado como el constructor de copia de esa clase.

Operadores de igualdad y desigualdad sobrecargados

En la línea 21 de la figura 11.6 se declara el operador de igualdad sobrecargado (`==`) para la clase. Cuando el compilador ve la expresión `enteros1 == enteros2` en la línea 57 de la figura 11.8, el compilador invoca a la función miembro `operator==` con la llamada

```
enteros1.operator==( enteros2 )
```

La función miembro `operator==` (definida en la figura 11.7, líneas 74 a 84) devuelve inmediatamente `false`, si los miembros `tamano` de los arreglos no son iguales. En caso contrario, `operator==` compara cada par de elementos. Si todos son iguales, la función devuelve `true`. El primer par de elementos que difieren provoca que la función devuelva `false` de inmediato.

En las líneas 24 a 27 del archivo de encabezado se define el operador de desigualdad sobrecargado (`!=`) para la clase. La función miembro `operator!=` utiliza la función `operator==` sobrecargada para determinar si un objeto `Array` es igual a otro, y después devuelve el valor opuesto de ese resultado. Al escribir `operator!=` de esta forma, podemos reutilizar `operator==`, lo cual reduce la cantidad de código que debemos escribir en la clase. Observe además que la definición completa de la función para `operator!=` está en el archivo de encabezado de `Array`. Esto permite al compilador poner en línea la definición de `operator!=` para eliminar la sobrecarga de la llamada adicional a la función.

Operadores de subíndice sobrecargados

En las líneas 30 y 33 de la figura 11.6 se declaran dos operadores de subíndice sobrecargados (definidos en la figura 11.7, en las líneas 88 a 89 y 103 a 114, respectivamente). Cuando el compilador ve la expresión `enteros1[5]` (figura 11.8, línea 61), invoca a la función miembro `operator[]` sobrecargada, para lo cual genera la llamada

```
enteros1.operator[]( 5 )
```

El compilador crea una llamada a la versión `const` de `operator[]` (figura 11.7, líneas 103 a 114) cuando se utiliza el operador de subíndice en un objeto `const Array`. Por ejemplo, si el objeto `const z` se instancia con la instrucción

```
const Array z( 5 );
```

entonces se requiere la versión `const` de `operator[]` para ejecutar una instrucción tal como

```
cout << z[ 3 ] << endl;
```

Recuerde que un programa sólo puede invocar a las funciones miembro `const` de un objeto `const`.

Cada definición de `operator[]` determina si el subíndice que recibe como argumento está dentro del rango. Si no es así, cada función imprime un mensaje de error y termina el programa con una llamada a la función `exit` (encabezado

`<stdlib>`.³ Si el subíndice está en el rango, la versión no `const` de `operator[]` devuelve el elemento apropiado del arreglo como una referencia, para que se pueda utilizar como un *lvalue* modificable (por ejemplo, del lado izquierdo de una instrucción de asignación). Si el subíndice está en el rango, la versión `const` de `operator[]` devuelve una copia del elemento apropiado del arreglo. El carácter devuelto es un *rvalue*.

11.9 Conversión entre tipos

La mayoría de los programas procesan información de muchos tipos. Algunas veces, todas las operaciones “permanecen dentro de un tipo”. Por ejemplo, al sumar un `int` con un `int` se produce un `int` (siempre y cuando el valor no sea demasiado grande como para representarlo con un `int`). Sin embargo, con frecuencia es necesario convertir datos de un tipo a datos de otro tipo. Esto puede ocurrir en asignaciones, en cálculos, en el paso de valores a funciones y en la devolución de valores de las funciones. El compilador sabe cómo llevar a cabo ciertas conversiones entre los tipos fundamentales (como vimos en el capítulo 6). Podemos usar operadores de conversión de tipos para forzar las conversiones entre los tipos fundamentales.

Pero ¿qué hay acerca de los tipos definidos por el usuario? El compilador no puede saber de antemano cómo convertir entre tipos definidos por el usuario, y entre tipos definidos por el usuario y tipos fundamentales, por lo que debemos especificar cómo hacer esto. Dichas conversiones se pueden llevar a cabo mediante **constructores de conversión**: constructores con un solo argumento que convierten objetos de otros tipos (incluyendo los tipos fundamentales) en objetos de una clase específica. En la sección 11.10 usaremos un constructor de conversión para convertir cadenas `char *` ordinarias en objetos de la clase `String`.

Un **operador de conversión** (también llamado **operador cast**) se puede utilizar para convertir un objeto de una clase en un objeto de otra clase, o en un objeto de un tipo fundamental. Dicho operador de conversión debe ser una función miembro no `static`. El prototipo de función

```
A::operator char *() const;
```

declara una función de operador de conversión de tipos sobrecargada para convertir un objeto del tipo definido por el usuario A en un objeto `char *` temporal. La función de operador se declara `const` dado que no modifica el objeto original. Una función de operador de conversión sobrecargada no especifica un tipo de valor de retorno; éste viene siendo el tipo al que se va a convertir el objeto. Si s es un objeto de la clase, cuando el compilador ve la expresión `static_cast<char *>(s)`, genera la llamada

```
s.operator char *()
```

El operando s es el objeto s de la clase para el que se va a invocar la función miembro `operator char *`.

Se pueden definir funciones de operador de conversión de tipos sobrecargadas para convertir objetos de tipos definidos por el usuario en tipos fundamentales, o en objetos de otros tipos definidos por el usuario. Los prototipos

```
A::operator int() const;
A::operator OtraClase() const;
```

declaran funciones de operador de conversión de tipos sobrecargadas, que pueden convertir un objeto del tipo definido por el usuario A en un entero, o en un objeto del tipo definido por el usuario OtraClase, respectivamente.

Una de las características agradables de los operadores de conversión de tipos y los constructores de conversión es que, cuando es necesario, el compilador puede llamar a estas funciones de manera implícita para crear objetos temporales. Por ejemplo, si un objeto s de una clase `String` definida por el usuario aparece en un programa, en una ubicación en la que se espera un valor `char *` ordinario, como en

```
cout << s;
```

el compilador puede llamar a la función de operador de conversión de tipos `operator char *` para convertir el objeto en un `char *` y usar el `char *` resultante en la expresión. Al proporcionar este operador de conversión de tipos en nuestra clase `String`, el operador de inserción de flujo no tiene que sobrecargarse para imprimir un objeto `String` mediante `cout`.

11.10 Ejemplo práctico: la clase String

Como ejercicio final para nuestro estudio acerca de la sobre carga, construiremos nuestra propia clase `String` para manejar la creación y manipulación de cadenas (figuras 11.9 a 11.11). La biblioteca estándar de C++ proporciona también una

3. Observe que es más apropiado “lanzar una excepción” cuando un subíndice está fuera de rango, para indicar el subíndice fuera de rango. Después el programa puede “atrapsar” esa excepción, procesarla y posiblemente continuar su ejecución. En el capítulo 16 encontrará más información acerca de las excepciones.

clase `string` similar y más robusta. Presentaremos un ejemplo de la clase `string` estándar en la sección 11.13 y estudiaremos la clase `string` con detalle en el capítulo 18. Por ahora, utilizaremos extensivamente la sobrecarga de operadores para crear nuestra propia clase `String`.

Primero vamos a presentar el archivo de encabezado para la clase `String`. Hablaremos sobre los datos `private` que se utilizan para representar objetos `String`. Después vamos a recorrer la interfaz `public` de la clase, y hablaremos sobre cada uno de los servicios que proporciona la clase. Describiremos las definiciones de las funciones miembro para la clase `String`. Para cada una de las funciones de operador sobrecargadas, mostraremos el código en el programa que invoca a la función de operador sobrecargada, y proporcionaremos una explicación acerca de cómo trabaja la función de operador sobrecargada.

Definición de la clase `String`

Ahora vamos a recorrer el archivo de encabezado de la clase `String` de la figura 11.9. Empezaremos con la representación interna basada en apuntador de un objeto `String`. En las líneas 55 y 56 se declaran los miembros de datos `private` de la clase. Nuestra clase `String` tiene un campo llamado `longitud`, el cual representa el número de caracteres en la cadena, sin incluir el carácter nulo al final, y tiene un apuntador llamado `sPtr` que apunta a la memoria asignada en forma dinámica que representa la cadena de caracteres.

```

1 // Fig. 11.9: String.h
2 // Definición de la clase String mediante la sobrecarga de operadores.
3 #ifndef STRING_H
4 #define STRING_H
5
6 #include <iostream>
7 using std::ostream;
8 using std::istream;
9
10 class String
11 {
12     friend ostream &operator<<( ostream &, const String & );
13     friend istream &operator>>( istream &, String & );
14 public:
15     String( const char * = "" ); // constructor de conversión/predeterminado
16     String( const String & ); // constructor de copia
17     ~String(); // destructor
18
19     const String &operator=( const String & ); // operador de asignación
20     const String &operator+=( const String & ); // operador de concatenación
21
22     bool operator!() const; // ¿el objeto String está vacío?
23     bool operator==( const String & ) const; // evalúa s1 == s2
24     bool operator<( const String & ) const; // evalúa s1 < s2
25
26     // evalúa s1 != s2
27     bool operator!=( const String &derecho ) const
28     {
29         return !( *this == derecho );
30     } // fin de la función operator!=
31
32     // evalúa s1 > s2
33     bool operator>( const String &derecho ) const
34     {
35         return derecho < *this;
36     } // fin de la función operator>
37
38     // evalúa s1 <= s2
39     bool operator<=( const String &derecho ) const
40     {
41         return !( derecho < *this );
42     } // end function operator<=

```

Figura 11.9 | Definición de la clase `String` con sobrecarga de operadores. (Parte I de 2).

```

43 // test s1 >= s2
44 bool operator>=( const String &derecho ) const
45 {
46     return !( *this < derecho );
47 } // fin de la función operator>=
48
49 char &operator[]( int ); // operador de subíndice (lvalue modificable)
50 char operator[]( int ) const; // operador de subíndice (rvalue)
51 String operator()( int, int = 0 ) const; // devuelve una subcadena
52 int getLongitud() const; // devuelve la longitud de la cadena
53
54 private:
55     int longitud; // longitud de la cadena (sin contar el terminador nulo)
56     char *sPtr; // apuntador al inicio de la cadena basada en apuntador
57
58     void setString( const char * ); // función utilitaria
59 }; // fin de la clase String
60
61 #endif

```

Figura 11.9 | Definición de la clase String con sobre carga de operadores. (Parte 2 de 2).

```

1 // Fig. 11.10: String.cpp
2 // Definiciones de las funciones miembro y funciones friend de la clase String.
3 #include <iostream>
4 using std::cerr;
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 #include <cstring> // prototipos de strcpy y strcat
12 using std::strcmp;
13 using std::strcpy;
14 using std::strcat;
15
16 #include <cstdlib> // prototipo de exit
17 using std::exit;
18
19 #include "String.h" // definición de la clase String
20
21 // el constructor de conversión (y predeterminado) convierte char * a String
22 String::String( const char *s )
23     : longitud( ( s != 0 ) ? strlen( s ) : 0 )
24 {
25     cout << "Constructor de conversión (y predeterminado): " << s << endl;
26     setString( s ); // llama a la función utilitaria
27 } // fin del constructor de conversión de String
28
29 // constructor de copia
30 String::String( const String &copia )
31     : longitud( copia.longitud )
32 {
33     cout << "Constructor de copia: " << copia.sPtr << endl;
34     setString( copia.sPtr ); // llama a la función utilitaria
35 } // fin del constructor de copia de String
36
37 // Destructor
38 String::~String()

```

Figura 11.10 | Definiciones de las funciones miembro y funciones friend de la clase String. (Parte 1 de 4).

```

39  {
40      cout << "Destructor: " << sPtr << endl;
41      delete [] sPtr; // libera la memoria de la cadena basada en apuntador
42  } // fin del destructor ~String
43
44 // operador = sobrecargado; evita la auto-asignación
45 const String &String::operator=( const String &derecho )
46 {
47     cout << "se llama a operator=" << endl;
48
49     if ( &derecho != this ) // evita la auto-asignación
50     {
51         delete [] sPtr; // evita la fuga de memoria
52         longitud = derecho.longitud; // nueva longitud de String
53         setString( derecho.sPtr ); // llama a la función utilitaria
54     } // fin de if
55     else
56         cout << "Intento de asignar un objeto String a si mismo" << endl;
57
58     return *this; // permite las asignaciones en cascada
59 } // fin de la función operator=
60
61 // concatena el operando derecho a este objeto y lo almacena en este objeto
62 const String &String::operator+=( const String &derecho )
63 {
64     size_t nuevaLongitud = longitud + derecho.longitud; // nueva longitud
65     char *tempPtr = new char[ nuevaLongitud + 1 ]; // crea la memoria
66
67     strcpy( tempPtr, sPtr ); // copia sPtr
68     strcpy( tempPtr + longitud, derecho.sPtr ); // copia derecho.sPtr
69
70     delete [] sPtr; // reclama el espacio anterior
71     sPtr = tempPtr; // asigna nuevo arreglo a sPtr
72     longitud = nuevaLongitud; // asigna nueva longitud a longitud
73     return *this; // permite las llamadas en cascada
74 } // fin de la función operator+=
75
76 // ¿este objeto String está vacío?
77 bool String::operator!() const
78 {
79     return longitud == 0;
80 } // fin de la función operator!
81
82 // ¿Este objeto String es igual al objeto String derecho?
83 bool String::operator==( const String &derecho ) const
84 {
85     return strcmp( sPtr, derecho.sPtr ) == 0;
86 } // fin de la función operator==
87
88 // ¿Este objeto String es menor que el objeto String derecho?
89 bool String::operator<( const String &derecho ) const
90 {
91     return strcmp( sPtr, derecho.sPtr ) < 0;
92 } // fin de la función operator<
93
94 // devuelve referencia al carácter en el objeto String como un lvalue modificable
95 char &String::operator[]( int subindice )
96 {
97     // evalúa si el subíndice está fuera de rango
98     if ( subindice < 0 || subindice >= longitud )
99     {
100        cerr << "Error: Subíndice " << subindice

```

Figura 11.10 | Definiciones de las funciones miembro y funciones friend de la clase String. (Parte 2 de 4).

```

101         << " fuera de rango" << endl;
102     exit( 1 ); // termina el programa
103 } // fin de if
104
105 return sPtr[ subindice ]; // devuelve un valor no const; lvalue modifiable
106 } // fin de la función operator[]
107
108 // devuelve referencia al carácter en el objeto String como rvalue
109 char String::operator[]( int subindice ) const
110 {
111     // evalúa si el subíndice está fuera de rango
112     if ( subindice < 0 || subindice >= longitud )
113     {
114         cerr << "Error: Subíndice "
115             << " fuera de rango" << endl;
116         exit( 1 ); // termina el programa
117     } // fin de if
118
119     return sPtr[ subindice ]; // devuelve una copia de este elemento
120 } // fin de la función operator[]
121
122 // devuelve una subcadena que empieza en indice y tiene longitud subLongitud
123 String String::operator()( int indice, int subLongitud ) const
124 {
125     // si indice está fuera de rango o la longitud de la subcadena < 0,
126     // devuelve un objeto String vacío
127     if ( indice < 0 || indice >= longitud || subLongitud < 0 )
128         return ""; // se convirtió en objeto String de manera automática
129
130     // determina la longitud de la subcadena
131     int lon;
132
133     if ( ( subLongitud == 0 ) || ( indice + subLongitud > longitud ) )
134         lon = longitud - indice;
135     else
136         lon = subLongitud;
137
138     // asigna un arreglo temporal para la subcadena y
139     // el carácter nulo de terminación
140     char *tempPtr = new char[ lon + 1 ];
141
142     // copia subcadena en arreglo char y termina la cadena
143     strncpy( tempPtr, &sPtr[ indice ], lon );
144     tempPtr[ lon ] = '\0';
145
146     // crea objeto String temporal que contiene la subcadena
147     String tempString( tempPtr );
148     delete [] tempPtr; // elimina arreglo temporal
149     return tempString; // devuelve copia del objeto String temporal
150 } // fin de la función operator()
151
152 // devuelve la longitud de la cadena
153 int String::getLongitud() const
154 {
155     return longitud;
156 } // fin de la función getLongitud
157
158 // función utilitaria llamada por los constructores y operator=
159 void String::setString( const char *string2 )
160 {
161     sPtr = new char[ longitud + 1 ]; // asigna memoria
162

```

Figura 11.10 | Definiciones de las funciones miembro y funciones friend de la clase String. (Parte 3 de 4).

```

163     if ( string2 != 0 ) // si string2 no es apuntador nulo, copia el contenido
164         strcpy( sPtr, string2 ); // copia literal a objeto
165     else // si string2 es un apuntador nulo, lo hace cadena vacía
166         sPtr[ 0 ] = '\0'; // cadena vacía
167     } // fin de la función setString
168
169 // operador de salida sobrecargado
170 ostream &operator<<( ostream &salida, const String &s )
171 {
172     salida << s.sPtr;
173     return salida; // permite las asignaciones en cascada
174 } // fin de la función operator<<
175
176 // operador de entrada sobrecargado
177 istream &operator>>( istream &entrada, String &s )
178 {
179     char temp[ 100 ]; // búfer para almacenar la entrada
180     entrada >> setw( 100 ) >> temp;
181     s = temp; // usa el operador de asignación de la clase String
182     return entrada; // permite las asignaciones en cascada
183 } // fin de la función operator>>

```

Figura 11.10 | Definiciones de las funciones miembro y funciones friend de la clase String. (Parte 4 de 4).

```

1 // Fig. 11.11: fig11_11.cpp
2 // Programa de prueba de la clase String.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::boolalpha;
7
8 #include "String.h"
9
10 int main()
11 {
12     String s1( "feliz" );
13     String s2( " cumpleanios" );
14     String s3;
15
16     // evalúa los operadores de igualdad y relacionales sobrecargados
17     cout << "s1 es \"" << s1 << "\"; s2 es \"" << s2
18     << "\"; s3 es \"" << s3 << '\"'
19     << boolalpha << "\n\nLos resultados de comparar s2 y s1:"
20     << "\ns2 == s1 produce " << ( s2 == s1 )
21     << "\ns2 != s1 produce " << ( s2 != s1 )
22     << "\ns2 > s1 produce " << ( s2 > s1 )
23     << "\ns2 < s1 produce " << ( s2 < s1 )
24     << "\ns2 >= s1 produce " << ( s2 >= s1 )
25     << "\ns2 <= s1 produce " << ( s2 <= s1 );
26
27
28     // evalúa el operador vacío (!) sobrecargado de String
29     cout << "\n\nEvaluando !s3:" << endl;
30
31     if ( !s3 )
32     {
33         cout << "s3 esta vacio; se asigno s1 a s3;" << endl;
34         s3 = s1; // evalúa operador de asignación sobrecargado
35         cout << "s3 es \"" << s3 << "\"";
36     } // fin de if

```

Figura 11.11 | Programa de prueba de la clase String. (Parte 1 de 3).

```

37 // evalúa el operador de concatenación sobre cargado de String
38 cout << "\n\ns1 += s2 produce s1 = ";
39 s1 += s2; // evalúa el operador de concatenación sobre cargado
40 cout << s1;
41
42 // evalúa el constructor de conversión
43 cout << "\n\ns1 += \" a ti\" produce" << endl;
44 s1 += " a ti"; // evalúa el constructor de conversión
45 cout << "s1 = " << s1 << "\n\n";
46
47 // evalúa el operador () de llamada a función sobre cargado para la subcadena
48 cout << "La subcadena de s1 empezando en\n"
49     << "la ubicacion 0 para 14 caracteres, s1(0, 17), es:\n"
50     << s1( 0, 17 ) << "\n\n";
51
52 // evalúa la opción "hasta el final de String" de la subcadena
53 cout << "La subcadena de s1 empezando en\n"
54     << "la ubicacion 18, s1(18), es: "
55     << s1( 18 ) << "\n\n";
56
57 // evalúa el constructor de copia
58 String *s4Ptr = new String( s1 );
59 cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";
60
61 // evalúa el operador de asignación (=) con la auto-asignación
62 cout << "asignando *s4Ptr a *s4Ptr" << endl;
63 *s4Ptr = *s4Ptr; // evalúa el operador de asignación sobre cargado
64 cout << "*s4Ptr = " << *s4Ptr << endl;
65
66 // evalúa el destructor
67 delete s4Ptr;
68
69 // evalúa usando operador de subíndice para crear un lvalue modificable
70 s1[ 0 ] = 'F';
71 s1[ 6 ] = 'C';
72 cout << "\ns1 después de s1[0] = 'F' y s1[6] = 'C' es: "
73     << s1 << "\n\n";
74
75 // evalúa subíndice fuera de rango
76 cout << "Intento de asignar 'd' a s1[30] produce:" << endl;
77 s1[ 30 ] = 'd'; // ERROR: subíndice fuera de rango
78 return 0;
79
80 } // fin de main

```

Constructor de conversion (y predeterminado): feliz
 Constructor de conversion (y predeterminado): cumpleanios
 Constructor de conversion (y predeterminado):
 s1 es "feliz"; s2 es " cumpleanios"; s3 es ""

Los resultados de comparar s2 y s1:

s2 == s1 produce false
 s2 != s1 produce true
 s2 > s1 produce false
 s2 < s1 produce true
 s2 >= s1 produce false
 s2 <= s1 produce true

Evaluando !s3:

s3 esta vacío; se asignó s1 a s3;
 se llama a operator=
 s3 es "feliz"

(continúa...)

Figura 11.11 | Programa de prueba de la clase String. (Parte 2 de 3).

```

s1 += s2 produce s1 = feliz cumpleanios

s1 += " a ti" produce
Constructor de conversion (y predeterminado): a ti
Destructor: a ti
s1 = feliz cumpleanios a ti

Constructor de conversion (y predeterminado): feliz cumpleanios
Constructor de copia: feliz cumpleanios
Destructor: feliz cumpleanios
La subcadena de s1 empezando en
la ubicacion 0 para 17 caracteres, s1(0, 17), es:
feliz cumpleanios

Destructor: feliz cumpleanios
Constructor de conversion (y predeterminado): a ti
Constructor de copia: a ti
Destructor: a ti
La subcadena de s1 empezando en
la ubicacion 18, s1(18), es: a ti

Destructor: a ti
Constructor de copia: feliz cumpleanios a ti

*s4Ptr = feliz cumpleanios a ti

asignando *s4Ptr a *s4Ptr
se llama a operator=
Intento de asignar un objeto String a si mismo

*s4Ptr = feliz cumpleanios a ti
Destructor: feliz cumpleanios a ti

s1 despues de s1[0] = 'F' y s1[6] = 'C' es: Feliz Cumpleanios a ti

Intento de asignar 'd' a s1[30] produce:
Error: Subindice 30 fuera de rango

```

Figura 11.11 | Programa de prueba de la clase *String*. (Parte 3 de 3).

Sobrecarga de los operadores de inserción de flujo y extracción de flujo como funciones friend

En las líneas 12 y 13 (figura 11.9) se declara la función de operador de inserción de flujo sobrecargado `operator<<` (definida en la figura 11.10, líneas 170 a 174) y la función del operador de extracción de flujo sobrecargado `operator>>` (definida en la figura 11.10, líneas 177 a 183) como funciones `friend` de la clase. La implementación de `operator<<` es simple y directa. Observe que `operator>>` restringe el número total de caracteres que se pueden leer en el arreglo `temp` a 99 con `setw` (línea 180); la posición 100 se reserva para el carácter nulo de terminación de la cadena. [Nota: no tenemos esta restricción para `operator>>` en la clase `Array` (figuras 11.6 y 11.7), debido a que el `operator>>` de esa clase lee un elemento del arreglo a la vez, y deja de leer valores cuando se llega al final del arreglo. El objeto `cin` no sabe cómo hacer esto de manera predeterminada para la entrada de los arreglos de caracteres]. Observe además el uso de `operator=` (línea 181) para asignar la cadena estilo C `temp` al objeto `String` al que se refiere `s`. Esta instrucción invoca el constructor de conversión para crear un objeto `String` temporal que contiene la cadena estilo C; después, el objeto `String` temporal se asigna a `s`. Podríamos eliminar la sobrecarga de crear el objeto `String` temporal aquí, al proporcionar otro operador de asignación sobrecargado que reciba un parámetro de tipo `const char *`.

Constructor de conversión de String

En la línea 15 (figura 11.9) se declara un constructor de conversión. Este constructor (definido en la figura 11.10, líneas 22 a 27) recibe un argumento `const char *` (cuyo valor predeterminado es la cadena vacía; figura 11.9, línea 15) e inicializa un objeto `String` que contiene esa misma cadena de caracteres. Cualquier **constructor con un solo argumento** se puede considerar como un constructor de conversión. Como veremos más adelante, dichos constructores son útiles

cuando realizamos cualquier operación con `String` mediante argumentos `char *`. El constructor de conversión puede convertir una cadena `char *` en un objeto `String`, el cual a su vez se puede asignar al objeto `String` de destino. La disponibilidad de este constructor de conversión significa que no es necesario suministrar un operador de asignación sobrecargado para asignar específicamente cadenas de caracteres a objetos `String`. El compilador invoca al constructor de conversión para crear un objeto `String` temporal que contiene la cadena de caracteres; después el operador de asignación sobrecargado se invoca para asignar el objeto `String` temporal a otro objeto `String`.



Observación de Ingeniería de Software 11.8

Cuando se utiliza un constructor de conversión para realizar una conversión implícita, C++ sólo puede aplicar una llamada implícita al constructor (es decir, una sola conversión definida por el usuario) para tratar de coincidir con las necesidades de otro operador sobrecargado. El compilador no coincidirá con las necesidades de un operador sobrecargado realizando una serie de conversiones implícitas definidas por el usuario.

El constructor de conversión de `String` podría invocarse en dicha declaración como `String s1("feliz")`. El constructor de conversión calcula la longitud de su argumento de cadena de caracteres y lo asigna al datos miembro `longitud` en la lista de inicializadores de miembros. Después, en la línea 26 se hace una llamada a la función utilitaria `getString` (definida en la figura 11.10, líneas 159 a 167), la cual usa `new` para asignar una cantidad suficiente de memoria al datos miembro `private sPtr`, y utiliza a `strcpy` para copiar la cadena de caracteres en la memoria a la que `sPtr` apunta.⁴

Constructor de copia de `String`

En la línea 16 de la figura 11.9 se declara un constructor de copia (definido en la figura 11.10, líneas 30 a 35) que inicializa un objeto `String`; para ello realiza una copia de un objeto `String` existente. Al igual que con nuestra clase `Array` (figuras 11.6 y 11.7), dicha copia debe realizarse con cuidado para evitar el problema en el que ambos objetos `String` apuntan a la misma memoria asignada en forma dinámica. El constructor de copia opera de manera similar al constructor de conversión, excepto que simplemente copia el miembro `longitud` del objeto `String` de origen al objeto `String` de destino. Observe que el constructor de copia llama a `setString` para crear nuevo espacio para la cadena de caracteres interna del objeto de destino. Si sólo copiara el apuntador `sPtr` en el objeto de origen al apuntador `sPtr` en el objeto de destino, entonces ambos objetos apuntarían a la misma memoria asignada en forma dinámica. El primer destructor en ejecutarse eliminaría entonces la memoria asignada en forma dinámica, y el apuntador `sPtr` del otro objeto quedaría indefinido (es decir, `sPtr` sería un apuntador suspendido), una situación que probablemente produzca un error grave en tiempo de ejecución.

Destructor de `String`

En la línea 17 de la figura 11.9 se declara el destructor de `String` (definido en la figura 11.10, líneas 38 a 42). El destructor utiliza `delete []` para liberar la memoria dinámica a la que `sPtr` apunta.

Operador de asignación sobrecargado

En la línea 19 (figura 11.9) se declara la función del operador de asignación sobrecargado `operator=` (definida en la figura 11.10, líneas 45 a 59). Cuando el compilador ve una expresión como `string1 = string2`, genera la llamada a la función

```
string1.operator=( string2 );
```

La función del operador de asignación sobrecargado `operator=` evalúa la autoasignación. Si ésta es una autoasignación, la función no necesita modificar el objeto. Si se omitiera esta prueba, la función eliminaría de inmediato el espacio en el objeto de destino y, por ende, perdería la cadena de caracteres, de forma tal que el apuntador ya no estaría apuntando a datos válidos; un clásico ejemplo de un apuntador suspendido. Si no hay autoasignación, la función elimina la memoria y copia el campo `length` del objeto origen del objeto de destino. Después, `operator=` llama a `setString` para crear nuevo espacio para el objeto de destino y copia la cadena de caracteres del objeto de origen al objeto de destino. Ya sea una auto-asignación o no, `operator=` devuelve `*this` para permitir las asignaciones en cascada.

4. Hay una ligera cuestión en cuanto a la implementación de este constructor de conversión. En la forma en la que está implementado, si se pasa un apuntador nulo (es decir, 0) al constructor, el programa fallará. La manera apropiada de implementar este constructor sería detectar si su argumento es un apuntador nulo, y después “lanzar una excepción”. En el capítulo 16 veremos cómo podemos hacer las clases más robustas en este sentido. Observe además que un apuntador nulo (0) no es lo mismo que la cadena vacía (""). Un apuntador nulo es un apuntador que no apunta a nada. Una cadena vacía es en sí una cadena que sólo contiene un carácter nulo ('\0').

Operador de asignación de suma sobrecargado

En la línea 20 de la figura 11.9 se declara el operador de concatenación de cadenas sobrecargado `+=` (definido en la figura 11.10, líneas 62 a 74). Cuando el compilador ve la expresión `s1 += s2` (línea 40 en la figura 11.11), genera la llamada a la función miembro

```
s1.operator+=( s2 )
```

La función `operator+=` calcula la longitud combinada de la cadena concatenada y la almacena en la variable local `nuevaLongitud`, después crea un apuntador temporal (`tempPtr`) y asigna un nuevo arreglo de caracteres en el que se almacenará la cadena concatenada. A continuación, `operator+=` usa `strcpy` para copiar las cadenas de caracteres originales de `sPtr` y `derecho.sPtr` a la memoria a la que `tempPtr` apunta. Observe que la ubicación en la que `strcpy` copiará el primer carácter de `derecho.sPtr` se determina con base en el cálculo de aritmética de apuntadores `tempPtr + longitud`. Este cálculo indica que el primer carácter de `derecho.sPtr` debe colocarse en la ubicación `longitud` en el arreglo al que `tempPtr` apunta. Luego, `operator+=` usa `delete[]` para liberar el espacio ocupado por la cadena de caracteres original de este objeto, asigna `tempPtr` a `sPtr` de manera que este objeto `String` apunte a la nueva cadena de caracteres, asigna `nuevaLongitud` a `longitud` para que este objeto `String` contenga la nueva longitud de la cadena y devuelva `*this` como un valor `const String &` para permitir los operadores `+=` en cascada.

¿Necesitamos un segundo operador de concatenación sobrecargado para permitir la concatenación de un `String` y un `char *`? No. El constructor de conversión `const char *` convierte una cadena estilo C en un objeto `String` temporal, el cual a su vez coincide con el operador de concatenación sobrecargado existente. Esto es exactamente lo que hace el compilador cuando encuentra la línea 45 en la figura 11.11. De nuevo, C++ puede realizar tales conversiones únicamente a un nivel de profundidad para facilitar las coincidencias, C++ también puede realizar una conversión implícita definida por el compilador entre los tipos fundamentales, antes de realizar la conversión entre un tipo fundamental y una clase. Observe que, cuando se crea un objeto `String` temporal en este caso, se hacen llamadas al constructor y al destructor (vea la salida que se produce de la línea 45, `s1 += " a ti"`, en la figura 11.11). Éste es un ejemplo de sobrecarga de llamadas a funciones que se oculta al cliente de la clase cuando se crean y destruyen objetos temporales de ésta durante las conversiones implícitas. Los constructores de copia generan una sobrecarga similar en el paso de parámetros de llamada por valor y en la devolución de objetos de la clase por valor.

Tip de rendimiento 11.2



Sobrecargar el operador de concatenación `+=` con una versión adicional que reciba un solo argumento de tipo `const char *` se ejecuta con más eficiencia que tener sólo una versión que reciba un argumento `String`. Sin la versión `const char *` del operador `+=`, un argumento `const char *` se convertiría primero en un objeto `String` con el constructor de conversión de la clase `String`, y después el operador `+=` que recibe un argumento `String` se llamaría para realizar la concatenación.

Observación de Ingeniería de Software 11.9



A menudo, el uso de conversiones implícitas con operadores sobrecargados, en vez de sobrecargar operadores para muchos tipos de operandos distintos, requiere menos código, lo cual facilita la modificación, mantenimiento y depuración de una clase.

Operador de negación sobrecargado

En la línea 22 de la figura 11.9 se declara el operador de negación sobrecargado (definido en la figura 11.10, líneas 77 a 80). Este operador determina si un objeto de nuestra clase `String` está vacío. Por ejemplo, cuando el compilador ve la expresión `!string1`, genera la llamada a la función

```
string1.operator!()
```

Esta función simplemente devuelve el resultado de evaluar si `longitud` es igual a cero.

Operadores de igualdad y relacionales sobrecargados

En las líneas 23 y 24 de la figura 11.9 se declaran el operador de igualdad relacionado (definido en la figura 11.10, líneas 83 a 86) y el operador menor que sobrecargado (definido en la figura 11.10, líneas 89 a 92) para la clase `String`. Estos operadores son similares, por lo que sólo hablaremos de un ejemplo, a saber, la sobrecarga del operador `==`. Cuando el compilador ve la expresión `string1 == string2`, genera la llamada a la función miembro

```
string1.operator==( string2 )
```

la cual devuelve `true` si `string1` es igual a `string2`. Cada uno de estos operadores utiliza la función `strcmp` (de `<cstring>`) para comparar las cadenas de caracteres en los objetos `String`. Muchos programadores de C++ apoyan el

uso de ciertas funciones de operadores sobre cargados para implementar otras. Por lo tanto, los operadores `!=`, `>`, `<=` y `>=` se implementan (figura 11.9, líneas 27 a 48) en términos de `operator==` y `operator<`. Por ejemplo, la función sobre cargada `operator>=` (implementada en las líneas 45 a 48 en el archivo de encabezado) usa el operador `<` sobre cargado para determinar si un objeto `String` es mayor o igual que otro. Observe que las funciones de operador para `!=`, `>`, `<=` y `>=` se definen en el archivo de encabezado. El compilador pone en línea estas definiciones para eliminar la sobre carga de las llamadas a funciones adicionales.



Observación de Ingeniería de Software 11.10

Al implementar funciones miembro mediante el uso de funciones miembro definidas previamente, se reutiliza código para reducir la cantidad de éste que se debe escribir y mantener.

Operadores de subíndice sobre cargados

En las líneas 50 y 51 del archivo de encabezado se declaran dos operadores de subíndice sobre cargados (definidos en la figura 11.10, líneas 95 a 106 y 109 a 120, respectivamente): uno para los objetos `String` no `const` y otro para los objetos `String const`. Cuando el compilador ve una expresión como `string1[0]`, genera la llamada a la función miembro

```
string1.operator[]( 0 )
```

(usando la versión apropiada de `operator[]`, dependiendo de si el objeto `String` es `const`). Cada implementación de `operator[]` valida primero el subíndice, para asegurar que esté dentro del rango. Si el subíndice está fuera del rango, cada función imprime un mensaje de error y termina el programa con una llamada a `exit`.⁵ Si el subíndice está dentro del rango, la versión `no const` de `operator[]` devuelve un `char &` al carácter apropiado del objeto `String`; este `char &` se puede utilizar como un *lvalue* para modificar el carácter designado del objeto `String`. La versión `const` de `operator[]` devuelve el carácter apropiado del objeto `String`; éste se puede usar sólo como un *rvalue* para leer el valor del carácter.



Tip para prevenir errores 11.2

Devolver una referencia `char no const` de un operador de subíndice sobre cargado en una clase `String` es peligroso. Por ejemplo, el cliente podría usar esta referencia para insertar un carácter nulo ('\0') en cualquier parte de la cadena.

Operador de llamada a función sobre cargado

En la línea 52 de la figura 11.9 se declara el **operador de llamada a función sobre cargado** (definido en la figura 11.10, líneas 123 a 150). Sobre cargamos este operador para seleccionar una subcadena de un objeto `String`. Los dos parámetros enteros especifican la ubicación inicial y la longitud de la subcadena que se va a seleccionar del objeto `String`. Si la ubicación inicial está fuera de rango o si la longitud de la subcadena es negativa, el operador simplemente devuelve un objeto `String` vacío. Si la longitud de la subcadena es 0, entonces ésta se selecciona hasta el final del objeto `String`. Por ejemplo, suponga que `string1` es un objeto `String` que contiene la cadena "AEIOU". Para la expresión `string1(2, 2)`, el compilador genera la llamada a la función miembro

```
string1.operator()( 2, 2 )
```

Cuando se ejecuta esta llamada, produce un objeto `String` que contiene la cadena "IO" y devuelve una copia de ese objeto.

Es poderoso sobre cargar el operador `()` de llamadas a funciones, ya que éstas pueden recibir listas de parámetros arbitrariamente extensas y complejas. Por lo tanto, podemos usar esta herramienta para muchos fines interesantes. Uno de tales usos del operador de llamadas a funciones es una notación de subíndice de arreglo alterna: en vez de usar la incómoda notación de doble corchete de C para los arreglos bidimensionales basados en apuntador, como en `a[b][c]`, algunos programadores prefieren sobre cargar el operador de llamadas a funciones para permitir la notación `a(b, c)`. El operador de llamadas a funciones sobre cargado debe ser una función miembro no `static`. Este operador se utiliza sólo cuando el "nombre de la función" es un objeto de la clase `String`.

Función miembro `getLongitud` de `String`

En la línea 53 de la figura 11.9 se declara la función `getLongitud` (definida en la figura 11.10, líneas 153 a 156), la cual devuelve la longitud de un objeto `String`.

5. De nuevo, cuando un subíndice está fuera de rango, es más apropiado "lanzar una excepción" indicando el subíndice fuera de rango.

Notas acerca de nuestra clase String

En este punto, el lector debe recorrer el código en `main`, examinar la ventana de resultados y comprobar cada uso de un operador sobre cargado. Al estudiar los resultados, ponga especial atención a las llamadas implícitas al constructor que se generan para crear objetos `String` temporales a lo largo del programa. Muchas de estas llamadas presentan la sobre carga adicional en el programa, la cual se puede evitar si la clase proporciona operadores sobre cargados que reciban argumentos `char *`. Sin embargo, las funciones de operador adicionales pueden hacer que la clase sea más difícil de mantener, modificar y depurar.

11.11 Sobre carga de ++ y --

Todas las versiones prefijo y postfijo de los operadores de incremento y decremento se pueden sobre cargar. A continuación veremos cómo el compilador diferencia entre la versión prefijo y la versión postfijo de un operador de incremento o decremento.

Para sobre cargar el operador de incremento y permitir el uso del incremento prefijo y postfijo, cada función de operador sobre cargado debe tener una firma distinta, de manera que el compilador pueda determinar cuál es la versión de `++` que se desea. Las versiones prefijo se sobre cargarán de la misma forma que cualquier otro operador unario prefijo.

Sobre carga del operador prefijo de incremento

Suponga, por ejemplo, que deseamos sumar 1 al día en el objeto `Fecha` llamado `d1`. Cuando el compilador ve la expresión de preincremento `++d1`, genera la llamada a la función miembro

```
d1.operator++()
```

El prototipo para esta función de operador sería

```
Fecha &operator++();
```

Si el operador de incremento prefijo se implementa como una función global, entonces cuando el compilador vea la expresión `++d1`, generará la llamada a la función

```
operator++( d1 )
```

El prototipo para esta función de operador se declararía en la clase `Fecha` de la siguiente manera:

```
Fecha &operator++( Fecha & );
```

Sobre carga del operador postfijo de incremento

La sobre carga del operador postfijo de incremento representa un reto, ya que el compilador debe tener la capacidad de diferenciar entre las firmas de las funciones de los operadores prefijo y postfijo de incremento. La convención que se ha adoptado en C++ es que, cuando el compilador ve la expresión de postincremento `d1++`, genera la llamada a la función miembro

```
d1.operator++( 0 )
```

El prototipo para esta función es

```
Fecha operator++( int )
```

El argumento 0 es estrictamente un “valor de muestra” que permite al compilador diferenciar entre las funciones de los operadores de prefijo y postfijo de incremento

Si el operador prefijo de incremento se implementa como una función global, entonces cuando ve la expresión `d1++`, genera la llamada a la función

```
operator++( d1, 0 )
```

El prototipo para esta función sería

```
Fecha operator++( Fecha &, int );
```

Una vez más, el compilador utiliza el argumento 0 para diferenciar entre los operadores de prefijo y postfijo de incremento que se implementan como funciones globales. Observe que el operador postfijo de incremento devuelve objetos `Fecha` por valor, mientras que el operador prefijo de incremento devuelve objetos `Fecha` por referencia, ya que por lo general el operador postfijo de incremento devuelve un objeto temporal que contiene el valor original del objeto antes de que ocurra el incremento. C++ trata a dichos objetos como *rvalues*, los cuales no se pueden utilizar del lado izquierdo de una asignación. El operador prefijo de incremento devuelve el objeto real incrementado con su nuevo valor. Dicho objeto se puede utilizar como un *lvalue* en una expresión de continuación.



Tip de rendimiento 11.3

El objeto adicional que se crea mediante el operador postfix de incremento (o postfix de decremento) puede producir un considerable problema de rendimiento; en especial cuando el operador se utiliza en un ciclo. Por esta razón, el programador debe usar el operador postfix de incremento (o postfix de decremento) sólo cuando la lógica del programa requiera un postfix de incremento (o postfix de decremento).

Todo lo declarado en esta sección para sobre cargar los operadores prefijo de incremento y postfix de decremento se aplica a la sobre carga de los operadores prefijo de decremento y postfix de decremento. A continuación analizaremos una clase Fecha con operadores prefijo de incremento y postfix de decremento sobre cargados.

11.12 Ejemplo práctico: una clase Fecha

El programa de las figuras 11.12 a 11.14 demuestra una clase Fecha. La clase usa operadores preincremento y postdecremento sobre cargados para sumar 1 al día en un objeto Fecha, mientras que produce incrementos apropiados en el mes y el año, en caso de ser necesario. El archivo de encabezado de Fecha (figura 11.12) especifica que la interfaz public de Fecha incluye un operador de inserción de flujo sobre cargado (línea 11), un constructor predeterminado (línea 13), una función setFecha (línea 14), un operador prefijo de incremento sobre cargado (línea 15), un operador postfix de incremento sobre cargado (línea 16), un operador de asignación de suma += sobre cargado (línea 17), una función para evaluar los años bisiestos (línea 18) y una función para determinar si un día es el último del mes (línea 19).

```

1 // Fig. 11.12: Fecha.h
2 // Definición de la clase Fecha con operadores de incremento sobre cargados.
3 #ifndef FECHA_H
4 #define FECHA_H
5
6 #include <iostream>
7 using std::ostream;
8
9 class Fecha
10 {
11     friend ostream &operator<<( ostream &, const Fecha & );
12 public:
13     Fecha( int m = 1, int d = 1, int a = 1900 ); // constructor predeterminado
14     void setFecha( int, int, int ); // establece mes, día, año
15     Fecha &operator++(); // operador preincremento
16     Fecha operator++( int ); // operador postincremento
17     const Fecha &operator+=( int ); // suma días, modifica el objeto
18     bool anioBisiesto( int ) const; // ¿está la fecha en un año bisiesto?
19     bool finDeMes( int ) const; // ¿está la fecha en el fin del mes?
20 private:
21     int mes;
22     int dia;
23     int anio;
24
25     static const int dias[]; // arreglo de días por mes
26     void ayudaIncremento(); // función utilitaria para incrementar la fecha
27 }; // fin de la clase Fecha
28
29 #endif

```

Figura 11.12 | Definición de la clase Fecha con operadores de incremento sobre cargados.

```

1 // Fig. 11.13: Fecha.cpp
2 // Definiciones de las funciones miembro y funciones friend de la clase Fecha.
3 #include <iostream>
4 #include "Fecha.h"

```

Figura 11.13 | Definiciones de las funciones miembro y funciones friend de la clase Fecha. (Parte I de 3).

```
5 // inicializa miembro estático en alcance de archivo; una copia a nivel de clase
6 const int Fecha::dias[] =
7     { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
8
9
10 // constructor de Fecha
11 Fecha::Fecha( int m, int d, int a )
12 {
13     setFecha( m, d, a );
14 } // fin del constructor de Fecha
15
16 // establece mes, día y año
17 void Fecha::setFecha( int mm, int dd, int aa )
18 {
19     mes = ( mm >= 1 && mm <= 12 ) ? mm : 1;
20     anio = ( aa >= 1900 && aa <= 2100 ) ? aa : 1900;
21
22     // prueba si es año bisiesto
23     if ( mes == 2 && anioBisiesto( anio ) )
24         dia = ( dd >= 1 && dd <= 29 ) ? dd : 1;
25     else
26         dia = ( dd >= 1 && dd <= dias[ mes ] ) ? dd : 1;
27 } // fin de la función setFecha
28
29 // operador preincremento sobrecargado
30 Fecha &Fecha::operator++()
31 {
32     ayudaIncremento(); // incrementa la fecha
33     return *this; // devuelve referencia para crear un lvalue
34 } // fin de la función operator++
35
36 // operador postincremento sobrecargado; observe que el parámetro
37 // entero de muestra no tiene un nombre de parámetro
38 Fecha Fecha::operator++( int )
39 {
40     Fecha temp = *this; // contiene el estado actual del objeto
41     ayudaIncremento();
42
43     // devuelve objeto temporal almacenado y sin incrementar
44     return temp; // devuelve un valor; no devuelve una referencia
45 } // fin de la función operator++
46
47 // suma el número especificado de días a la fecha
48 const Fecha &Fecha::operator+=( int diasAdicionales )
49 {
50     for ( int i = 0; i < diasAdicionales; i++ )
51         ayudaIncremento();
52
53     return *this; // permite la asignación en cascada
54 } // fin de la función operator+=
55
56 // si el año es bisiesto, devuelve true; en caso contrario, devuelve false
57 bool Fecha::anioBisiesto( int anioPrueba ) const
58 {
59     if ( anioPrueba % 400 == 0 ||
60         ( anioPrueba % 100 != 0 && anioPrueba % 4 == 0 ) )
61         return true; // un año bisiesto
62     else
63         return false; // no es un año bisiesto
64 } // fin de la función anioBisiesto
65
66 // determina si el día es el último del mes
```

Figura 11.13 | Definiciones de las funciones miembro y funciones friend de la clase Fecha. (Parte 2 de 3).

```

67  bool Fecha::finDeMes( int diaPrueba ) const
68  {
69      return diaPrueba == 29; // último día de Feb. en año bisiesto
70  }
71  else
72      return diaPrueba == dias[ mes ];
73 } // fin de la función finDeMes
74
75 // función para ayudar a incrementar la fecha
76 void Fecha::ayudaIncremento()
77 {
78     // dia no es fin de mes
79     if ( !finDeMes( dia ) )
80         dia++; // incrementa dia
81     else
82         if ( mes < 12 ) // día es fin de mes y mes < 12
83         {
84             mes++; // incrementa mes
85             dia = 1; // primer día del nuevo mes
86         } // fin de if
87     else // último día de año
88     {
89         anio++; // incrementa año
90         mes = 1; // primer mes del nuevo año
91         dia = 1; // primer día del nuevo mes
92     } // fin de else
93 } // fin de la función ayudaIncremento
94
95 // operador de salida sobre cargado
96 ostream &operator<<( ostream &salida, const Fecha &d )
97 {
98     static char *nombreMes[ 13 ] = { "", "Enero", "Febrero",
99         "Marzo", "Abril", "Mayo", "Junio", "Julio", "Agosto",
100        "Septiembre", "Octubre", "Noviembre", "Diciembre" };
101     salida << nombreMes[ d.mes ] << ' ' << d.dia << ", " << d.anio;
102     return salida; // permite la asignación en cascada
103 } // fin de la función operator<<

```

Figura 11.13 | Definiciones de las funciones miembro y funciones friend de la clase Fecha. (Parte 3 de 3).

```

1 // Fig. 11.14: fig11_14.cpp
2 // Programa de prueba de la clase Fecha.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Fecha.h" // definición de la clase Fecha
8
9 int main()
10 {
11     Fecha d1; // valor predeterminado: Enero 1, 1900
12     Fecha d2( 12, 27, 1992 ); // Diciembre 27, 1992
13     Fecha d3( 0, 99, 8045 ); // fecha inválida
14
15     cout << "d1 es " << d1 << "\nd2 es " << d2 << "\nd3 es " << d3;
16     cout << "\n\n" << d2 += 7 << ( d2 += 7 );
17
18     d3.setFecha( 2, 28, 1992 );
19     cout << "\n\n" << d3;
20     cout << "\n\n" << d3 << " (año bisiesto permite dia 29)";
21

```

Figura 11.14 | Programa de prueba de la clase Fecha. (Parte 1 de 2).

```

22     Fecha d4( 7, 13, 2002 );
23
24     cout << "\n\nPrueba del operador preincremento:\n"
25         << "    d4 es " << d4 << endl;
26     cout << "++d4 es " << ++d4 << endl;
27     cout << "    d4 es " << d4;
28
29     cout << "\n\nPrueba del operador postincremento:\n"
30         << "    d4 es " << d4 << endl;
31     cout << "d4++ es " << d4++ << endl;
32     cout << "    d4 es " << d4 << endl;
33     return 0;
34 } // fin de main

```

```

d1 es Enero 1, 1900
d2 es Diciembre 27, 1992
d3 es Enero 1, 1900

d2 += 7 es Enero 3, 1993

d3 es Febrero 28, 1992
++d3 es Febrero 29, 1992 (anio bisiesto permite dia 29)

Prueba del operador preincremento:
d4 es Julio 13, 2002
++d4 es Julio 14, 2002
d4 es Julio 14, 2002

Prueba del operador postincremento:
d4 es Julio 14, 2002
d4++ es Julio 14, 2002
d4 es Julio 15, 2002

```

Figura 11.14 | Programa de prueba de la clase Fecha. (Parte 2 de 2).

La función `main` (figura 11.14) crea tres objetos `Fecha` (líneas 11 a 13): `d1` se inicializa de manera predeterminada con Enero 1, 1900; `d2` se inicializa con Diciembre 27, 1992; y `d3` se inicializa con una fecha inválida. El constructor de `Fecha` (definido en la figura 11.13, líneas 11 a 14) llamada a `setFecha` para validar el mes, día y año especificados. Un mes inválido se establece en 1, un año inválido se establece en 1900 y un día inválido se establece en 1.

En las líneas 15 y 16 de `main` se imprime cada uno de los objetos `Fecha` construidos, usando el operador de inserción de flujo sobrecargado (definido en la figura 11.13, líneas 96 a 103). En la línea 16 de `main` se utiliza el operador sobrecargado `+=` para sumar siete días a `d2`. En la línea 18 se utiliza la función `setFecha` para establecer `d3` a Febrero 28, 1992, que es un año bisiesto. Despues, en la línea 20 se preincrementa `d3` para mostrar que la fecha se incrementa en forma apropiada a Febrero 29. A continuación, en la línea 22 se crea un objeto `Fecha` llamado `d4`, el cual se inicializa con la fecha Julio 13, 2002. Luego en la línea 26 se incrementa `d4` en 1 con el operador prefix incremento sobrecargado. En las líneas 24 a 27 se imprime `d4` antes y después de la operación de prefix incremento, para confirmar que haya funcionado en forma apropiada. Por último, en la línea 31 se incrementa `d4` con el operador prefix incremento sobrecargado. En las líneas 29 a 32 se imprime `d4` antes y después de la operación de postfix incremento, para confirmar que trabaje en forma apropiada.

La sobrecarga del operador preincremento es un proceso directo. El operador prefix incremento (definido en la figura 11.13, líneas 30 a 34) llama a la función utilitaria `ayudaIncremento` (definida en la figura 11.13, líneas 76 a 93) para incrementar la fecha. Esta función trata con “envolturas” o “acarreos” que ocurren cuando incrementamos el último día del mes. Estos acarreos requieren incrementar el mes. Si éste ya es 12, entonces el año también se debe incrementar y el mes se debe establecer en 1. La función `ayudaIncremento` usa la función `finDeMes` para incrementar el día en forma apropiada.

El operador de preincremento sobrecargado devuelve una referencia al objeto `Fecha` actual (es decir, el que se acaba de incrementar). Esto ocurre debido a que el objeto actual, `*this`, se devuelve como un objeto `Date &`. Esto permite que un objeto `Fecha` preincrementado se utilice como un *lvalue*, que es como trabaja el operador prefix incremento integrado para los tipos fundamentales.

La sobre carga del operador prefix incremento (definido en la figura 11.13, líneas 38 a 45) es más complicada. Para emular el efecto del postincremento, debemos devolver una copia del objeto `Fecha` sin incremento. Por ejemplo, si la variable `int x` tiene el valor 7, la instrucción

```
cout << x++ << endl;
```

imprime el valor original de la variable `x`. Por lo tanto, nos gustaría que el operador postfix incremento operara de la misma forma con un objeto `Fecha`. Al entrar a `operator++`, guardamos el objeto actual (`*this`) en `temp` (línea 40). A continuación, llamamos a `ayudaIncremento` para incrementar el objeto `Fecha` actual. Después, en la línea 44 se devuelve la copia sin incremento del objeto previamente almacenado en `temp`. Observe que esta función no puede devolver una referencia al objeto `Fecha` local llamado `temp`, debido a que una variable local se destruye cuando la función en la que se declara termina su ejecución. Así, al declarar el tipo de valor de retorno para esta función como `Fecha &` se devolvería una referencia a un objeto que ya no existe. Devolver una referencia (o un apuntador) a una variable local es un error común para el cual la mayoría de los compiladores generarán una advertencia.

11.13 La clase `string` de la Biblioteca estándar

En este capítulo, el lector aprendió que puede crear una clase `String` (figuras 11.9 a 11.11) que es mejor que las cadenas `char *` estilo C que C++ absorbió de C. También aprendió que puede crear una clase `Array` (figuras 11.6 a 11.8) que es mejor que los arreglos basados en apuntador estilo C que C++ absorbió de C.

Para crear clases útiles y reutilizables tales como `String` y `Array` se requiere de trabajo. Sin embargo, una vez que estas clases se prueban y depuran, pueden ser reutilizadas por usted, sus colegas, su empresa, muchas empresas, toda una industria o incluso muchas industrias (si se colocan en bibliotecas públicas o de venta al público). Los diseñadores de C++ hicieron exactamente eso, crear la clase `string` (que hemos estado utilizando desde el capítulo 3) y la plantilla de clase `vector` (que presentamos en el capítulo 7) en el lenguaje C++ estándar. Estas clases están disponibles para cualquiera que cree aplicaciones con C++. Como veremos en el capítulo 22, la Biblioteca estándar de C++ proporciona varias plantillas de clase predefinidas para usarlas en sus programas.

Para cerrar este capítulo, vamos a rehacer nuestro ejemplo con `String` (figuras 11.9 a 11.11) utilizando la clase `string` estándar de C++. Rediseñaremos nuestro ejemplo para demostrar la funcionalidad similar que proporciona la clase `string` estándar. También demostraremos tres funciones miembro de la clase `string` estándar —`empty`, `substr` y `at`— que no formaron parte de nuestro ejemplo con `String`. La función `empty` determina si un objeto `string` está vacío, la función `substr` devuelve un objeto `string` que representa una parte de un objeto `string` existente y la función `at` devuelve el carácter en el subíndice especificado en un objeto `string` (después de comprobar que el subíndice esté dentro del rango). En el capítulo 18 se presenta la clase `string` con detalle.

La clase `string` de la Biblioteca estándar

El programa de la figura 11.15 reimplementa el programa de la figura 11.11, usando la clase `string` estándar. Como veremos en este ejemplo, la clase `string` proporciona toda la funcionalidad de nuestra clase `String` presentada en las figuras 11.9 y 11.10. La clase `string` se define en el encabezado `<string>` (línea 7) y pertenece al espacio de nombres `std` (línea 8).

```

1 // Fig. 11.15: fig11_15.cpp
2 // Programa de prueba de la clase string de la Biblioteca estándar.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 int main()
11 {
12     string s1( "feliz" );
13     string s2( " cumpleaños" );
14     string s3;
15
16     // prueba los operadores de igualdad y relacionales sobre cargados
17     cout << "s1 es \"" << s1 << "\"; s2 es \"" << s2

```

Figura 11.15 | La clase `string` de la Biblioteca estándar. (Parte I de 3).

```
18     << "\"; s3 es \"" << s3 << "\""
19     << "\n\nLos resultados de comparar s2 y s1:"
20     << "\ns2 == s1 produce " << ( s2 == s1 ? "true" : "false" )
21     << "\ns2 != s1 produce " << ( s2 != s1 ? "true" : "false" )
22     << "\ns2 > s1 produce " << ( s2 > s1 ? "true" : "false" )
23     << "\ns2 < s1 produce " << ( s2 < s1 ? "true" : "false" )
24     << "\ns2 >= s1 produce " << ( s2 >= s1 ? "true" : "false" )
25     << "\ns2 <= s1 produce " << ( s2 <= s1 ? "true" : "false" );
26
27 // prueba la función miembro empty de string
28 cout << "\n\nPrueba de s3.empty():" << endl;
29
30 if ( s3.empty() )
31 {
32     cout << "s3 esta vacia; se asigno s1 a s3;" << endl;
33     s3 = s1; // asigna s1 a s3
34     cout << "s3 es \"" << s3 << "\"";
35 } // fin de if
36
37 // prueba el operador de concatenación sobrecargado de string
38 cout << "\n\ns1 += s2 produce s1 = ";
39 s1 += s2; // prueba el operador de concatenación sobrecargado
40 cout << s1;
41
42 // prueba el operador de concatenación sobrecargado de string con una cadena estilo C
43 cout << "\n\ns1 += \" a ti\" produce" << endl;
44 s1 += " a ti";
45 cout << "s1 = " << s1 << "\n\n";
46
47 // prueba la función miembro substr de string
48 cout << "La subcadena de s1 que empieza en la ubicacion 0 para\n"
49     << "17 caracteres, s1.substr(0, 17), es:\n"
50     << s1.substr( 0, 17 ) << "\n\n";
51
52 // prueba la opción "hasta el final de la cadena" de substr
53 cout << "La subcadena de s1 qe empieza en\n"
54     << "la ubicacion 18, s1.substr(18), es:\n"
55     << s1.substr( 18 ) << endl;
56
57 // prueba el constructor de copia
58 string *s4Ptr = new string( s1 );
59 cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";
60
61 // prueba el operador de asignación (=) con la auto-asignación
62 cout << "asignando *s4Ptr a *s4Ptr" << endl;
63 *s4Ptr = *s4Ptr;
64 cout << "*s4Ptr = " << *s4Ptr << endl;
65
66 // prueba el destructor
67 delete s4Ptr;
68
69 // prueba el uso del operador de subíndice para crear un lvalue
70 s1[ 0 ] = 'F';
71 s1[ 6 ] = 'C';
72 cout << "\ns1 despues de s1[0] = 'F' y s1[6] = 'C' es: "
73     << s1 << "\n\n";
74
75 // prueba el subíndice fuera de rango con la función miembro "at" de string
76 cout << "El intento de asignar 'd' a s1.at( 30 ) produce:" << endl;
77 s1.at( 30 ) = 'd'; // ERROR: subíndice fuera de rango
78 return 0;
79 } // fin de main
```

Figura 11.15 | La clase string de la Biblioteca estándar. (Parte 2 de 3).

```
s1 es "feliz"; s2 es " cumpleanios"; s3 es ""

Los resultados de comparar s2 y s1:
s2 == s1 produce false
s2 != s1 produce true
s2 > s1 produce false
s2 < s1 produce true
s2 >= s1 produce false
s2 <= s1 produce true

Prueba de s3.empty():
s3 esta vacia; se asigno s1 a s3;
s3 es "feliz"

s1 += s2 produce s1 = feliz cumpleanios

s1 += " a ti" produce
s1 = feliz cumpleanios a ti

La subcadena de s1 que empieza en la ubicacion 0 para
17 caracteres, s1.substr(0, 17), es:
feliz cumpleanios

La subcadena de s1 que empieza en
la ubicacion 18, s1.substr(18), es:
a ti

*s4Ptr = feliz cumpleanios a ti

asignando *s4Ptr a *s4Ptr
*s4Ptr = feliz cumpleanios a ti

s1 despues de s1[0] = 'F' y s1[6] = 'C' es: Feliz Cumpleaños a ti

El intento de asignar 'd' a s1.at( 30 ) produce:

Terminación anormal del programa
```

Figura 11.15 | La clase `string` de la Biblioteca estándar. (Parte 3 de 3).

En las líneas 12 a 14 se crean tres objetos `string`: `s1` se inicializa con la literal "feliz", `s2` se inicializa con la literal "cumpleaños" y `s3` utiliza el constructor de `string` predeterminado para crear un objeto `string` vacío. En las líneas 17 y 18 se imprimen estos tres objetos, usando `cout` y el operador `<<`, que los diseñadores de la clase `string` sobre cargaron para manejar objetos `string`. Después, en las líneas 19 a 25 se muestran los resultados de comparar `s2` con `s1`, usando los operadores de igualdad y relacionales sobre cargados de la clase `string`.

Nuestra clase `String` (figuras 11.9 y 11.10) contiene un `operator!` sobre cargado que prueba un objeto `String` para determinar si está vacío. La clase `string` estándar no proporciona esta funcionalidad como un operador sobre cargado; en vez de ello, proporciona la función miembro `empty`, que demostramos en la línea 30. La función miembro `empty` devuelve `true` si el objeto `string` está vacío; en caso contrario, devuelve `false`.

En la línea 33 se demuestra el operador de asignación sobre cargado de la clase `string` mediante la asignación de `s1` a `s3`. En la línea 34 se imprime `s3` para demostrar que la asignación funcionó de manera correcta.

En la línea 39 se demuestra el operador `+=` sobre cargado de la clase `string` para la concatenación de cadenas. En este caso, el contenido de `s2` se adjunta a `s1`. Después, en la línea 40 se imprime la cadena resultante que se almacena en `s1`. En la línea 44 se demuestra que una literal de cadena estilo C se puede adjuntar a un objeto `string` mediante el uso del operador `+=`. En la línea 45 se muestra el resultado.

Nuestra clase `String` (figuras 11.9 y 11.10) contiene el `operator()` sobre cargado para obtener subcadenas. La clase `string` estándar no proporciona esta funcionalidad como un operador sobre cargado; en vez de ello, proporciona la función miembro `substr` (líneas 50 y 55). La llamada a `substr` en la línea 50 obtiene una subcadena de 14 caracteres (especificada por el segundo argumento) de `s1`, empezando en la posición 0 (especificada por el primer argumento). La llamada a `substr`

en la línea 55 obtiene una subcadena que empieza desde la posición 15 de `s1`. Cuando el segundo argumento no se especifica, `substr` devuelve el resto del objeto `string` en el que se llamó.

En la línea 58 se asigna en forma dinámica un objeto `string` y se inicializa con una copia de `s1`. Esto produce una llamada al constructor de copia de la clase `string`. En la línea 63 se utiliza el operador `=` sobrecargado de la clase `string` para demostrar que maneja la auto-asignación en forma apropiada.

En las líneas 70 y 71 se utiliza el operador `[]` sobrecargado de la clase `string` para crear `lvalues` que permitan que nuevos caracteres reemplacen los caracteres existentes en `s1`. En la línea 73 se imprime el nuevo valor de `s1`. En nuestra clase `String` (figuras 11.9 y 11.10), el operador `[]` sobrecargado realizaba la comprobación de límites para determinar si el subíndice que recibía como argumento era un subíndice válido en la cadena. Si el subíndice era inválido, el operador imprimía un mensaje de error y terminaba el programa. El operador `[]` sobrecargado de la clase `string` estándar no realiza comprobación de límites. Por lo tanto, debe asegurar que las operaciones que utilizan el operador `[]` sobrecargado de la clase `string` estándar no manipula accidentalmente los elementos fuera de los límites del objeto `string`. La clase `string` estándar proporciona la comprobación de límites en su función miembro `at`, la cual “lanza una excepción” si su argumento es un subíndice inválido. De manera predeterminada, esto hace que un programa de C++ termine.⁶ Si el subíndice es válido, la función `at` devuelve el carácter en la ubicación especificada como un `lvalue` modificable o un `lvalue` no modificable (es decir, una referencia `const`), dependiendo del contexto en el que aparece la llamada. En la línea 77 se demuestra una llamada a la función `at` con un subíndice inválido.

11.14 Constructores explicit

En las secciones 11.8 y 11.9, vimos que el compilador puede utilizar cualquier constructor con un solo argumento para realizar una conversión implícita; el tipo recibido por el constructor se convierte en un objeto de la clase en la que éste se define. La conversión es automática y no hay que usar un operador de conversión de tipos. En ciertas situaciones, las conversiones implícitas son indeseables o propensas a errores. Por ejemplo, nuestra clase `Array` en la figura 11.6 define un constructor que recibe un solo argumento `int`. La intención de este constructor es crear un objeto `Array` que contenga el número de elementos especificados por el argumento `int`. Sin embargo, este constructor puede ser mal utilizado por el compilador para realizar una conversión implícita.



Error común de programación 11.9

Por desgracia, el compilador podría usar las conversiones implícitas en casos en los que no se espera, lo cual produce expresiones ambiguas que generan errores de compilación, o errores lógicos en tiempo de ejecución.

Uso accidental de un constructor con un solo argumento como un constructor de conversión

El programa (figura 11.16) utiliza la clase `Array` de las figuras 11.6 y 11.7 para demostrar una conversión implícita inapropiada.

En la línea 13 de `main` se instancia el objeto `Array` llamado `enteros1` y se llama al constructor con un solo argumento, con el valor `int` 7 para especificar el número de elementos en el objeto `Array`. En la figura 11.7 vimos que el constructor de `Array` que recibe un argumento `int` inicializa todos los elementos del arreglo con 0. En la línea 14 se hace una llamada a la función `imprimirArray` (definida en las líneas 20 a 24), la cual recibe como argumento un `const Array &` a un objeto `Array`. La función imprime el número de elementos en su argumento `Array` y el contenido del mismo. En este caso, el tamaño del objeto `Array` es 7, por lo que se imprimen siete 0s.

```

1 // Fig. 11.16: Fig11_16.cpp
2 // Controlador para la clase simple Array.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Array.h"
8
9 void imprimirArray( const Array & ); // prototipo

```

Figura 11.16 | Constructores con un solo argumento y conversiones implícitas. (Parte 1 de 2).

6. De nuevo, en el capítulo 16, Manejo de excepciones, se demuestra cómo “atravar” y manejar dichas excepciones.

```

10
11 int main()
12 {
13     Array enteros1( 7 ); // arreglo con 7 elementos
14     imprimirArray( enteros1 ); // imprime el objeto Array enteros1
15     imprimirArray( 3 ); // convierte 3 en un objeto Array e imprime su contenido
16     return 0;
17 } // fin de main
18
19 // imprime el contenido de un objeto Array
20 void imprimirArray( const Array &arregloAImprimir )
21 {
22     cout << "El objeto Array recibido tiene " << arregloAImprimir.getTamanio()
23         << " elementos. Su contenido es:\n" << arregloAImprimir << endl;
24 } // fin de imprimirArray

```

El objeto Array recibido tiene 7 elementos. Su contenido es:

0	0	0	0
0	0	0	

El objeto Array recibido tiene 3 elementos. Su contenido es:

0	0	0
---	---	---

Figura 11.16 | Constructores con un solo argumento y conversiones implícitas. (Parte 2 de 2).

En la línea 15 se hace una llamada a la función `imprimirArray` con el valor `int 3` como argumento. Sin embargo, este programa no contiene una función llamada `imprimirArray` que reciba un argumento `int`. Por lo tanto, el compilador determina si la clase `Array` proporciona un constructor de conversión que pueda convertir un `int` en un `Array`. Como cualquier constructor que recibe un solo argumento se considera como constructor de conversión, el compilador asume que el constructor de `Array` que recibe un solo `int` es un constructor de conversión y lo utiliza para convertir el argumento `3` en un objeto `Array` temporal que contiene tres elementos. Después, el compilador pasa el objeto `Array` temporal a la función `imprimirArray` para imprimir el contenido del objeto `Array`. Por ende, aun y cuando no proporcionamos de manera explícita una función `imprimirArray` que reciba un argumento `int`, el compilador puede compilar la línea 15. Los resultados muestran que el objeto `Array` de tres elementos contiene 0s.

Cómo evitar un uso accidental de un constructor con un solo argumento como un constructor de conversión

C++ proporciona la palabra clave `explicit` para suprimir las conversiones implícitas a través de los constructores de conversión, cuando no deben permitirse dichas conversiones. Un constructor que se declara `explicit` no puede usarse en una conversión implícita. La figura 11.17 declara un constructor `explicit` en la clase `Array`. La única modificación a `Array.h` fue que se agregó la palabra clave `explicit` a la declaración del constructor con un solo argumento en la línea 15. No se requieren modificaciones al archivo de código fuente que contiene las definiciones de las funciones miembro de la clase `Array`.

```

1 // Fig. 11.17: Array.h
2 // Definición de la clase Array para almacenar arreglos de enteros.
3 #ifndef ARRAY_H
4 #define ARRAY_H
5
6 #include <iostream>
7 using std::ostream;
8 using std::istream;
9
10 class Array
11 {
12     friend ostream &operator<<( ostream &, const Array & );
13     friend istream &operator>>( istream &, Array & );
14 public:
15     explicit Array( int = 10 ); // constructor predeterminado

```

Figura 11.17 | Definición de la clase `Array` con constructor `explicit`. (Parte 1 de 2).

```

16     Array( const Array & ); // constructor de copia
17     ~Array(); // destructor
18     int obtenerTamanio() const; // devuelve el tamaño
19
20     const Array &operator=( const Array & ); // operador de asignación
21     bool operator==( const Array & ) const; // operador de igualdad
22
23     // operador de desigualdad; devuelve el opuesto del operador ==
24     bool operator!=( const Array &derecho ) const
25     {
26         return ! ( *this == derecho ); // invoca a Array::operator==
27     } // fin de la función operator!=
28
29     // el operador de subíndice para los objetos no const devuelve un lvalue modificable
30     int &operator[]( int );
31
32     // el operador de subíndice para los objetos const devuelve rvalue
33     int operator[]( int ) const;
34 private:
35     int tamanio; // arreglo tamaño basado en apuntador
36     int *ptr; // apuntador al primer elemento del arreglo basado en apuntador
37 }; // fin de la clase Arreglo
38
39 #endif

```

Figura 11.17 | Definición de la clase `Array` con constructor `explicit`. (Parte 2 de 2).

La figura 11.18 presenta una versión ligeramente modificada del programa de la figura 11.16. Cuando se compila este programa, el compilador produce un mensaje de error indicando que el valor de entero pasado a `imprimirArray` en la línea 15 no se puede convertir a `const Array &`. El mensaje de error emitido por el compilador se muestra en la ventana de salida. La línea 16 demuestra como el constructor explícito se puede usar para crear un `Array` temporal de 3 elementos y pasarlo a la función `imprimirArray`.

Error común de programación 11.10



Tratar de invocar el constructor `explicit` para una conversión implícita es un error de compilación.

Error común de programación 11.11



Utilizando la palabra clave `explicit` en datos miembro o funciones miembro diferente a un constructor de un solo argumento es un error de compilación.

```

1 // Fig. 11.18: Fig11_18.cpp
2 // Controlador para la clase simple Array.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Array.h"
8
9 void imprimirArray( const Array & ); // prototipo
10
11 int main()
12 {
13     Array enteros1( 7 ); // arreglo con 7 elementos
14     imprimirArray( enteros1 ); // imprime el objeto Array enteros1
15     imprimirArray( 3 ); // convierte 3 en un objeto Array e imprime su contenido
16     imprimirArray( Array( 3 ) ); // llama al constructor explícito con un solo argumento
17     return 0;

```

Figura 11.18 | Demostración de un constructor `explicit`. (Parte 1 de 2).

```

18 } // fin de main
19
20 // imprime el contenido del arreglo
21 void imprimirArray( const Array &arregloAImprimir )
22 {
23     cout << "El objeto Array recibido tiene " << arregloAImprimir.getTamanio()
24         << " elementos. Su contenido es:\n" << arregloAImprimir << endl;
25 } // fin de imprimirArray

```

```
c:\cpphtp6_ejemplos\cap11\fig11_17_18\fig11_18.cpp(15) : error C2664:
'imirimirArray' : no se puede convertir el parámetro 1 de 'int' a 'const Array &
Razón: no se puede realizar la conversión de 'int' a 'const Array'
El constructor de class 'Array' está declarado como 'explicit'
```

Figura 11.18 | Demostración de un constructor *explicit*. (Parte 2 de 2).



Tip para prevenir errores 11.3

Use la palabra clave `explicit` en los constructores con un solo argumento que no deban ser utilizados por el compilador para llevar a cabo conversiones implícitas.

11.15 Repaso

En este capítulo aprendió a crear clases más robustas mediante la definición de operadores sobre cargados que permitan a los programadores tratar a los objetos de sus clases como si fueran tipos de datos fundamentales de C++. Presentamos los conceptos básicos de la sobre carga de operadores, así como varias restricciones que el estándar de C++ coloca sobre los operadores sobre cargados. Aprendió las razones de implementar operadores sobre cargados como funciones miembro o como funciones globales. Describimos las diferencias entre la sobre carga de operadores unarios y binarios como funciones miembro y funciones globales. Con las funciones globales, le mostramos cómo recibir y enviar objetos de nuestras clases mediante los operadores de extracción de flujo y de inserción de flujo, respectivamente. Mostramos una sintaxis especial que se requiere para diferenciar entre las versiones prefijo y postfijo del operador de incremento (++). También demostramos la clase `string` estándar de C++, que hace un uso extensivo de los operadores sobre cargados para crear una clase reutilizable y robusta que pueda reemplazar las cadenas basadas en apuntador estilo C. Por último, aprendió a utilizar la palabra clave `explicit` para evitar que el compilador utilice un constructor con un solo argumento para realizar conversiones implícitas. En el siguiente capítulo continuaremos con nuestra discusión sobre las clases, en donde presentaremos una forma de reutilización de software conocida como herencia. Ahí veremos que cuando las clases comparten atributos y comportamientos comunes, es posible definir esos atributos y comportamientos en una clase “base” común, y “heredar” esas capacidades en nuevas definiciones de clases.

Resumen

Sección 11.1 Introducción

- C++ nos permite sobre cargar la mayoría de los operadores, para que sean sensibles al contexto en el que se utilizan; el compilador genera el código apropiado con base en el contexto (en especial, en los tipos de los operandos).
- Muchos de los operadores de C++ se pueden sobre cargar para trabajar con tipos definidos por el usuario.
- Un ejemplo de un operador sobre cargado integrado en C++ es <<, el cual se usa como operador de inserción de flujo y como operador de desplazamiento a la izquierda a nivel de bits. De manera similar, >> también está sobre cargado; se utiliza como operador de extracción de flujo y como operador de desplazamiento a la derecha a nivel de bits. Ambos operadores están sobre cargados en la Biblioteca estándar de C++.
- El mismo lenguaje C++ sobre carga los operadores de suma (+) y de resta (-). Estos operadores tienen un desempeño distinto, dependiendo de su contexto en la aritmética de enteros, de punto flotante y de apuntadores.
- Los trabajos realizados por los operadores sobre cargados también se pueden llevar a cabo mediante llamadas a funciones, pero la notación de operador es a menudo más clara y conocida para los programadores.

Sección 11.2 Fundamentos de la sobrecarga de operadores

- Para sobrecargar un operador, se escribe la definición de una función miembro no `static` o la definición de una función global en donde el nombre de la función es la palabra clave `operator`, seguida del símbolo del operador que se va a sobre cargar.
- Cuando los operadores se sobrecargan como funciones miembro, deben ser no `static` debido a que se deben llamar en un objeto de la clase y deben operar en ese objeto.
- Para usar un operador en objetos de clases, éste *debe* sobrecargarse, con tres excepciones: el operador de asignación (`=`), el operador de dirección (`&`) y el operador de coma (`,`).

Sección 11.3 Restricciones acerca de la sobrecarga de operadores

- No se puede cambiar la precedencia y asociatividad de un operador mediante la sobrecarga.
- No se puede cambiar la “aridad” de un operador (es decir, el número de operandos que recibe un operador).
- No se pueden crear nuevos operadores; sólo se pueden sobrecargar los ya existentes.
- No se puede cambiar el significado de la forma en que un operador trabaja sobre los objetos de tipos fundamentales.
- La sobrecarga de un operador de asignación y un operador de suma para una clase no implica que el operador `+=` también se sobrecarga. Dicho comportamiento sólo se puede lograr sobrecargando de manera explícita el operador `+=` para esa clase.

Sección 11.4 Las funciones de operadores como clase miembro vs. funciones globales

- Las funciones de operadores pueden ser funciones miembro o funciones globales; por lo general, las funciones globales se hacen `friend` por cuestiones de rendimiento. Las funciones miembro utilizan el apuntador `this` de manera implícita para obtener uno de los argumentos de los objetos de su clase (el operando izquierdo para operadores binarios). Los argumentos para ambos operandos de un operador binario deben listarse de manera explícita en la llamada a una función global.
- Al sobrecargar `O`, `[]`, `->` o cualquiera de los operadores de asignación, la función de sobrecarga de operadores debe declararse como miembro de la clase. Para los otros operadores, las funciones de sobrecarga de operadores pueden ser miembros de la clase o funciones globales.
- Cuando una función de operador se implementa como función miembro, el operando de más a la izquierda (o el único operando) debe ser un objeto (o una referencia a un objeto) de la clase del operador.
- Si el operando izquierdo debe ser un objeto de una clase distinta o de un tipo fundamental, esta función de operador se debe implementar como una función global.
- Una función de operador global puede convertirse en `friend` de una clase, si esa función debe aceptar directamente miembros `private` o `protected` de esa clase.

Sección 11.5 Sobre carga de los operadores de inserción de flujo y extracción de flujo

- El operador de inserción de flujo sobrecargado (`<<`) se utiliza en una expresión en la que el operando izquierdo tiene el tipo `ostream &`. Por esta razón, se debe sobrecargar como una función global. Para ser una función miembro, el operador `<<` tendría que ser un miembro de la clase `ostream`, pero esto no es posible, ya que no tenemos permitido modificar las clases de la Biblioteca estándar de C++. De manera similar, el operador de extracción de flujo sobrecargado (`>>`) debe ser una función global.
- Otra razón por la que se debe elegir una función global para sobrecargar un operador es para permitir que éste sea commutativo.
- Cuando se utiliza con `cin` y objetos `string`, la función `setw` restringe el número de caracteres leídos al número de caracteres especificados por su argumento.
- La función miembro `ignore` de `istream` descarta el número especificado de caracteres en el flujo de entrada (un carácter de manera predeterminada).
- Los operadores de entrada y salida sobrecargados se declaran como funciones `friend` si necesitan acceder a los miembros `no public` de la clase directamente, por cuestiones de rendimiento.

Sección 11.6 Sobre carga de operadores unarios

- Un operador unario para una clase se puede sobrecargar como función miembro no `static` sin argumentos, o como función global con un argumento; ese argumento debe ser un objeto de la clase, o una referencia a un objeto de la misma.
- Las funciones miembro que implementan operadores sobrecargados deben ser no `static`, de manera que puedan acceder a los datos no `static` en cada objeto de la clase.

Sección 11.7 Sobre carga de operadores binarios

- Un operador binario se puede sobrecargar como una función miembro no `static` con un parámetro, o como una función global con dos parámetros (uno de esos parámetros debe ser el objeto de una clase o una referencia al objeto de una clase).

Sección 11.8 Ejemplo práctico: la clase `Array`

- Para inicializar un nuevo objeto de una clase, un constructor de copia realiza una copia de los miembros de un objeto existente de esa clase. Cuando los objetos de una clase contienen memoria asignada en forma dinámica, la clase debe proporcionar un constructor de copia para asegurar que cada copia de un objeto tenga su propia copia separada de la memoria asignada en forma dinámica. Por lo general, dicha clase también debe proporcionar un destructor y un operador de asignación sobre cargado.
- La implementación de la función miembro `operator=` debe evaluar la auto-asignación, en la cual un objeto se asigna a sí mismo.
- El compilador llama a la versión `const` de `operator[]` cuando el operador de subíndice se utiliza en un objeto `const`, y llama a la versión no `const` del operador cuando se utiliza en un objeto no `const`.
- El operador de subíndice de arreglo (`[]`) no tiene su uso restringido a los arreglos. Puede usarse para seleccionar elementos de otros tipos de clases contenedoras. Además, con la sobre carga los valores de los subíndices no necesitan ser enteros; pueden usarse caracteres o cadenas, por ejemplo.

Sección 11.9 Conversión entre tipos

- El compilador no puede saber de antemano cómo convertir entre tipos definidos por el usuario, y entre tipos definidos por el usuario y tipos fundamentales, por lo que debemos especificar cómo hacer esto. Dichas conversiones se pueden llevar a cabo mediante constructores de conversión: constructores con un solo argumento que convierten objetos de otros tipos (incluyendo los tipos fundamentales) en objetos de una clase específica.
- Un operador de conversión (también llamado operador de conversión de tipos) se puede utilizar para convertir un objeto de una clase en un objeto de otra clase, o en un objeto de un tipo fundamental. Dicho operador de conversión debe ser una función miembro no `static`. Pueden definirse funciones de operadores de conversión sobre cargados para convertir objetos de tipos definidos por el usuario en tipos fundamentales o en objetos de otros tipos definidos por el usuario.
- Una función de operador de conversión sobre cargada no especifica un tipo de valor de retorno; éste viene siendo el tipo al que se va a convertir el objeto.
- Una de las características agradables de los operadores de conversión de tipos y los constructores de conversión es que, cuando es necesario, el compilador puede llamar a estas funciones de manera implícita para crear objetos temporales.

Sección 11.10 Ejemplo práctico: la clase `String`

- Cualquier constructor con un solo argumento se puede considerar como un constructor de conversión.
- Es poderoso sobre cargar el operador () de llamadas a funciones, ya que las funciones pueden recibir listas de parámetros arbitrariamente extensas y complejas.

Sección 11.11 Sobre carga de `++` y `--`

- Todas las versiones prefijo y postfixo de los operadores de incremento y decremento se pueden sobre cargar.
- Para sobre cargar el operador de incremento y permitir el uso del incremento prefijo y postfixo, cada función de operador sobre cargado debe tener una firma distinta, de manera que el compilador pueda determinar cuál es la versión de `++` que se desea. Las versiones prefijo se sobre cargarán de la misma forma que cualquier otro operador unario prefijo. Para proporcionar una firma única al operador postfixo incremento se utiliza un segundo argumento, que debe ser de tipo `int`. Este argumento no se proporciona en el código cliente. El compilador lo utiliza de manera implícita para diferenciar entre las versiones prefijo y postfixo del operador de incremento.

Sección 11.13 La clase `string` de la biblioteca estándar

- La clase `string` estándar se define en el encabezado `<string>` y pertenece al espacio de nombres `std`.
- La clase `string` proporciona muchos operadores sobre cargados, incluyendo los operadores de igualdad, relacionales, de asignación, de asignación de suma (para la concatenación) y de subíndice.
- La clase `string` proporciona la función miembro `empty`, que devuelve `true` si el objeto `string` está vacío; en caso contrario devuelve `false`.
- La función miembro `substr` de la clase `string` estándar obtiene una subcadena de una longitud especificada por el segundo argumento, empezando en la posición especificada por el primer argumento. Cuando no se especifica el segundo argumento, `substr` devuelve el resto del objeto `string` en el que se llamó.
- El operador `[]` sobre cargado de la clase `string` no realiza comprobación de límites. Por lo tanto, el programador se debe asegurar que las operaciones que utilizan el operador `[]` sobre cargado de la clase `string` estándar no manipule accidentalmente los elementos fuera de los límites del objeto `string`.
- La clase `string` estándar proporciona la comprobación de límites en su función miembro `at`, la cual “lanza una excepción” si su argumento es un subíndice inválido. De manera predeterminada, esto hace que un programa de C++ termine. Si el subíndice es válido, la función `at` devuelve el carácter en la ubicación especificada como un `lvalue` modificable o un `lvalue` no modificable (es decir, una referencia `const`), dependiendo del contexto en el que aparece la llamada.

Sección 11.14 Constructores *explicit*

- C++ proporciona la palabra clave *explicit* para suprimir las conversiones implícitas a través de los constructores de conversión cuando no deben permitirse dichas conversiones. Un constructor que se declara como *explicit* no se puede utilizar en una conversión implícita.

Terminología

“aridad” de un operador	operador == sobrecargado
Array, clase	operador > sobrecargado
auto-asignación	operador >= sobrecargado
concatenación de cadenas	operador >> sobrecargado
constructor de conversión	operador de asignación (=) sobrecargado
constructor de copia	operador de conversión
constructor <i>explicit</i>	operador de extracción de flujo sobrecargado
conversión definida por el usuario	operador de inserción de flujo sobrecargado
conversión entre tipos fundamentales y tipos de clases	operadores sobrecargables
conversiones implícitas definidas por el usuario	operator!
empty, función miembro de <i>string</i>	operator!=
función de operador	operator()
función del operador de conversión de tipos	operator, palabra clave
función global para sobrecargar un operador	operator[]
funciones del operador de asignación	operator+ operator++ operator+(int)
ignore, función miembro de <i>istream</i>	operator< operator<<
la asociatividad no se cambia mediante la sobrecarga	operator=
<i>lvalue</i> (“valor izquierdo”)	operator==
operación conmutativa	operator>=
operador ! sobrecargado	operator>>
operador != sobrecargado	sobrecarga de operadores
operador () de llamadas a funciones	sobrecarga de un operador binario
operador () sobrecargado	sobrecarga de un operador unario
operador [] sobrecargado	<i>string</i> (clase estándar de C++)
operador + sobrecargado	subcadena
operador ++ sobrecargado	substr, función miembro de <i>string</i>
operador +(int) sobrecargado	tipo definido por el usuario
operador += sobrecargado	versión const de operator[]
operador < sobrecargado	
operador << sobrecargado	
operador <= sobrecargado	

Ejercicios de autoevaluación

11.1 Complete los siguientes enunciados:

- Suponga que a y b son variables enteras y que formamos la suma a + b. Ahora suponga que c y d son variables de punto flotante y que formamos la suma c + d. Los dos operadores + de aquí se están utilizando claramente para distintos fines. Éste es un ejemplo de _____.
- La palabra clave _____ introduce la definición de una función de operador sobrecargado.
- Para usar operadores en objetos de clases, éstos deben sobrecargarse con la excepción de los operadores _____, _____ y _____.
- La _____, _____ y _____ de un operador no se puede modificar al sobrecargar este operador.

11.2 Explique los múltiples significados de los operadores << y >>.

11.3 ¿En qué contexto se podría utilizar el nombre operator/?

11.4 (Verdadero/falso) Sólo los operadores existentes se pueden sobrecargar.

11.5 ¿Cómo se compara la precedencia de un operador sobrecargado con la precedencia del operador original?

Respuestas a los ejercicios de autoevaluación

11.1 a) sobre carga de operadores. b) `operator`. c) asignación (`=`), dirección (`&`), coma (`,`). d) precedencia, asociatividad, “aridad”.

11.2 El operador `>>` es tanto el operador de desplazamiento a la derecha como el operador de extracción de flujo, dependiendo de su contexto. El operador `<<` es tanto el operador de desplazamiento a la izquierda como el operador de inserción de flujo, dependiendo de su contexto.

11.3 Para la sobre carga de operadores: sería el nombre de una función que proporcionara una versión sobre cargada del operador / para una clase específica.

11.4 Verdadero.

11.5 La precedencia es idéntica.

Ejercicios

11.6 Proporcione todos los ejemplos que pueda acerca de la sobre carga de operadores implícita en C++. Dé un ejemplo razonable de una situación en la que se podría sobre cargar un operador explícitamente en C++.

11.7 Los operadores que no se pueden sobre cargar son _____, _____, _____ y _____.

11.8 La concatenación de cadenas requiere dos operandos: las dos cadenas que se van a concatenar. En el texto mostramos al lector cómo implementar un operador de concatenación sobre cargado que concatena el segundo objeto `String` a la derecha del primer objeto `String`, con lo cual se modifica el primer objeto `String`. En ciertas aplicaciones, es conveniente producir un objeto `String` concatenado sin modificar los argumentos `String`. Implemente la función `operator+` para permitir operaciones tales como:

```
string1 = string2 + string3;
```

11.9 (*Ejercicio final de sobre carga de operadores*) Para apreciar el cuidado que se debe poner al seleccionar operadores para sobre cargar, liste cada uno de los operadores que se pueden sobre cargar y, para cada uno de ellos, liste un posible significado (o varios, según sea apropiado) para cada una de varias de las clases que ha estudiado en este texto. Le sugerimos que pruebe con:

- a) Array
- b) Pila
- c) String

Después de hacer esto, comente cuáles operadores parecen tener significado para una amplia variedad de clases. ¿Cuáles operadores parecen tener poco valor para la sobre carga? ¿Cuáles operadores parecen ambiguos?

11.10 Ahora realice el proceso descrito en el ejercicio 11.9 a la inversa. Liste cada uno de los operadores de C++ que pueden sobre cargarse. Para cada uno, liste lo que usted cree que sea tal vez la “máxima operación” que debería representar el operador. Si hay varias operaciones excelentes, lístelas todas.

11.11 Un buen ejemplo de cómo sobre cargar el operador `()` de llamadas a funciones es permitir otra forma de subíndices dobles de arreglos comunes en ciertos lenguajes de programación. En vez de decir

```
tableroAjedrez[ fila ][ columna ]
```

para un arreglo de objetos, sobre cargue el operador de llamadas a funciones para permitir la siguiente forma alterna:

```
tableroAjedrez( fila, columna )
```

Cree una clase llamada `ArregloSubindiceDoble` que tenga características similares a la clase `Array` de las figuras 11.6 y 11.7. Al momento de la construcción, la clase debe poder crear un arreglo de cualquier número de filas y columnas. La clase debe proporcionar el `operator()` para realizar operaciones con doble subíndice. Por ejemplo, en un objeto `ArregloSubindiceDoble` de 3 por 5 llamado `a`, el usuario podría escribir `a(1, 3)` para acceder al elemento en la fila 1 y la columna 3. Recuerde que `operator()` puede recibir cualquier número de argumentos (en la clase `String` en las figuras 11.9 y 11.10 podrá ver un ejemplo de `operator()`). La representación subyacente del arreglo con doble subíndice debe ser un arreglo de enteros con un subíndice, con un número de elementos equivalente a `filas * columnas`. La función `operator()` debe realizar la aritmética de apuntadores apropiada para acceder a cada elemento del arreglo. Debe haber dos versiones de `operator()`: una que devuelva `int &` (de manera que un elemento de un objeto `ArregloSubindiceDoble` pueda usarse como *lvalue*) y otra que devuelva `const int &` (de manera que un elemento de un objeto `constArregloSubindiceDoble` pueda usarse sólo como *rvalue*). La clase debe también proporcionar los siguientes operadores: `==`, `!=`, `=`, `<<` (para imprimir el arreglo en formato de fila y columna) y `>>` (para recibir todo el contenido completo del arreglo).

11.12 Sobre cargue el operador de subíndice para devolver el elemento más grande de una colección, el segundo más grande, el tercero, y así en lo sucesivo.

11.13 Considere la clase **Complejo** que se muestra en las figuras 11.19 a 11.21. Esta clase permite operaciones con *números complejos*. Éstos son números de la forma **parteReal + parteImaginaria * i**, en donde *i* tiene el valor

$$\sqrt{-1}$$

- a) Modifique la clase para permitir la entrada y salida de números complejos a través de los operadores **>>** y **<<**, respectivamente (debe eliminar la función **imprimir** de la clase).
- b) Sobrecargue el operador de multiplicación para permitir la multiplicación de dos números complejos, como en álgebra.
- c) Sobrecargue los operadores **==** y **!=** para permitir comparaciones de números complejos.

```

1 // Fig. 11.19: Complejo.h
2 // Definición de la clase Complejo.
3 #ifndef COMPLEJO_H
4 #define COMPLEJO_H
5
6 class Complejo
7 {
8 public:
9     Complejo( double = 0.0, double = 0.0 ); // constructor
10    Complejo operator+( const Complejo & ) const; // suma
11    Complejo operator-( const Complejo & ) const; // resta
12    void imprimir() const; // salida
13 private:
14     double real; // parte real
15     double imaginaria; // parte imaginaria
16 }; // fin de la clase Complejo
17
18 #endif

```

Figura 11.19 | Definición de la clase **Complejo**.

```

1 // Fig. 11.20: Complejo.cpp
2 // Definiciones de las funciones miembro de la clase Complejo.
3 #include <iostream>
4 using std::cout;
5
6 #include "Complejo.h" // definición de la clase Complejo
7
8 // Constructor
9 Complejo::Complejo( double parteReal, double parteImaginaria )
10   : real( parteReal ),
11     imaginaria( parteImaginaria )
12 {
13     // cuerpo vacío
14 } // fin del constructor de Complejo
15
16 // operador de suma
17 Complejo Complejo::operator+( const Complejo &operando2 ) const
18 {
19     return Complejo( real + operando2.real,
20                      imaginaria + operando2.imaginaria );
21 } // fin de la función operator+
22
23 // operador de resta
24 Complejo Complejo::operator-( const Complejo &operando2 ) const
25 {
26     return Complejo( real - operando2.real,
27                      imaginaria - operando2.imaginaria );
28 } // fin de la función operator-
29
30 // muestra un objeto Complejo en la forma: (a, b)
31 void Complejo::imprimir() const
32 {
33     cout << '(' << real << ", " << imaginaria << ')';
34 } // fin de la función imprimir

```

Figura 11.20 | Definiciones de las funciones miembro de la clase **Complejo**.

```

1 // Fig. 11.21: fig11_21.cpp
2 // Programa de prueba de la clase Complejo.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Complejo.h"
8
9 int main()
10 {
11     Complejo x;
12     Complejo y( 4.3, 8.2 );
13     Complejo z( 3.3, 1.1 );
14
15     cout << "x: ";
16     x.imprimir();
17     cout << "\ny: ";
18     y.imprimir();
19     cout << "\nz: ";
20     z.imprimir();
21
22     x = y + z;
23     cout << "\nx = y + z:" << endl;
24     x.imprimir();
25     cout << " = ";
26     y.imprimir();
27     cout << " + ";
28     z.imprimir();
29
30     x = y - z;
31     cout << "\nx = y - z:" << endl;
32     x.imprimir();
33     cout << " = ";
34     y.imprimir();
35     cout << " - ";
36     z.imprimir();
37     cout << endl;
38     return 0;
39 } // fin de main

```

```

x: (0, 0)
y: (4.3, 8.2)
z: (3.3, 1.1)

x = y + z:
(7.6, 9.3) = (4.3, 8.2) + (3.3, 1.1)

x = y - z:
(1, 7.1) = (4.3, 8.2) - (3.3, 1.1)

```

Figura 11.21 | Números complejos.

11.14 Una máquina con enteros de 32 bits puede representar enteros en el rango aproximado de -2 mil millones a +2 mil millones. Esta restricción de tamaño fijo raras veces presenta problemas, pero hay aplicaciones en las que sería conveniente poder usar un rango de enteros mucho más amplio. Esto es para lo que se creó C++, a saber, para crear nuevos y poderosos tipos de datos. Considere la clase `EnteroEnorme` de las figuras 11.22 a 11.24. Estudie la clase con cuidado y después responda a lo siguiente:

- Describa la forma en que opera con precisión.
- ¿Qué restricciones tiene la clase?
- Sobre cargue el operador de multiplicación *.
- Sobre cargue el operador de división /.
- Sobre cargue todos los operadores relacionales y de igualdad.

[Nota: no mostramos un operador de asignación o constructor de copia para la clase `EnteroEnorme`, debido a que el operador de asignación y el constructor de copia que proporciona el compilador son capaces de copiar el dato miembro tipo arreglo completo de manera apropiada].

```

1 // Fig. 11.22: EnteroEnorme.h
2 // Definición de la clase EnteroEnorme.
3 #ifndef ENTEROENORME_H
4 #define ENTEROENORME_H
5
6 #include <iostream>
7 using std::ostream;
8
9 class EnteroEnorme
10 {
11     friend ostream &operator<<( ostream &, const EnteroEnorme & );
12 public:
13     EnteroEnorme( long = 0 ); // constructor de conversión/predeterminado
14     EnteroEnorme( const char * ); // constructor de conversión
15
16     // operador de suma; EnteroEnorme + EnteroEnorme
17     EnteroEnorme operator+( const EnteroEnorme & ) const;
18
19     // operador de suma; EnteroEnorme + int
20     EnteroEnorme operator+( int ) const;
21
22     // operador de suma;
23     // EnteroEnorme + string que representa un valor entero mayor
24     EnteroEnorme operator+( const char * ) const;
25 private:
26     short entero[ 30 ];
27 }; // fin de la clase EnteroEnorme
28
29 #endif

```

Figura 11.22 | Definición de la clase EnteroEnorme.

```

1 // Fig. 11.23: EnteroEnorme.cpp
2 // Definiciones de las funciones miembro y funciones friend de EnteroEnorme.
3 #include <cctype> // prototipo de la función isdigit
4 using std::isdigit;
5
6 #include <cstring> // prototipo de la función strlen
7 using std::strlen;
8
9 #include "EnteroEnorme.h" // definición de la clase EnteroEnorme
10
11 // constructor predeterminado; constructor de conversión que convierte
12 // un entero long en un objeto EnteroEnorme
13 EnteroEnorme::EnteroEnorme( long valor )
14 {
15     // inicializa el arreglo con cero
16     for ( int i = 0; i <= 29; i++ )
17         entero[ i ] = 0;
18
19     // coloca los dígitos del argumento en el arreglo
20     for ( int j = 29; valor != 0 && j >= 0; j-- )
21     {
22         entero[ j ] = valor % 10;
23         valor /= 10;
24     } // fin de for
25 } // fin del constructor predeterminado/de conversión de EnteroEnorme
26
27 // constructor de conversión que convierte una cadena de caracteres
28 // que representa a un entero grande en un objeto EnteroEnorme
29 EnteroEnorme::EnteroEnorme( const char *string )
30 {
31     // inicializa el arreglo con cero
32     for ( int i = 0; i <= 29; i++ )
33         entero[ i ] = 0;
34
35     // coloca los dígitos del argumento en el arreglo

```

Figura 11.23 | Definiciones de las funciones miembro y funciones friend de la clase EnteroEnorme. (Parte I de 2).

```

36     int longitud = strlen( string );
37
38     for ( int j = 30 - longitud, k = 0; j <= 29; j++, k++ )
39
40         if ( isdigit( string[ k ] ) )
41             entero[ j ] = string[ k ] - '0';
42 } // fin del constructor de conversión de EnteroEnorme
43
44 // operador de suma; EnteroEnorme + EnteroEnorme
45 EnteroEnorme EnteroEnorme::operator+( const EnteroEnorme &op2 ) const
46 {
47     EnteroEnorme temp; // resultado temporal
48     int acarreo = 0;
49
50     for ( int i = 29; i >= 0; i-- )
51     {
52         temp.entero[ i ] =
53             entero[ i ] + op2.entero[ i ] + acarreo;
54
55         // determina si se acarrea un 1
56         if ( temp.entero[ i ] > 9 )
57         {
58             temp.entero[ i ] %= 10; // lo reduce a 0-9
59             acarreo = 1;
60         } // fin de if
61         else // no hay acarreo
62             acarreo = 0;
63     } // fin de for
64
65     return temp; // devuelve una copia del objeto temporal
66 } // fin de la función operator+
67
68 // operador de suma; EnteroEnorme + int
69 EnteroEnorme EnteroEnorme::operator+( int op2 ) const
70 {
71     // convierte op2 en un objeto EnteroEnorme, después invoca
72     // a operator+ para dos objetos EnteroEnorme
73     return *this + EnteroEnorme( op2 );
74 } // fin de la función operator+
75
76 // operador de suma;
77 // EnteroEnorme + cadena que representa un valor entero grande
78 EnteroEnorme EnteroEnorme::operator+( const char *op2 ) const
79 {
80     // convierte op2 en un objeto EnteroEnorme, después invoca
81     // a operator+ para dos objetos EnteroEnorme
82     return *this + EnteroEnorme( op2 );
83 } // fin de operator+
84
85 // operador de salida sobre cargado
86 ostream& operator<<( ostream &salida, const EnteroEnorme &num )
87 {
88     int i;
89
90     for ( i = 0; ( num.entero[ i ] == 0 ) && ( i <= 29 ); i++ )
91         ; // omite ceros a la izquierda
92
93     if ( i == 30 )
94         salida << 0;
95     else
96
97         for ( ; i <= 29; i++ )
98             salida << num.entero[ i ];
99
100    return salida;
101 } // fin de la función operator<<

```

Figura 11.23 | Definiciones de las funciones miembro y funciones friend de la clase EnteroEnorme. (Parte 2 de 2).

```

1 // Fig. 11.24: fig11_24.cpp
2 // Programa de prueba de EnteroEnorme.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "EnteroEnorme.h"
8
9 int main()
10 {
11     EnteroEnorme n1( 7654321 );
12     EnteroEnorme n2( 7891234 );
13     EnteroEnorme n3( "99999999999999999999999999999999" );
14     EnteroEnorme n4( "1" );
15     EnteroEnorme n5;
16
17     cout << "n1 es " << n1 << "\nn2 es " << n2
18         << "\nn3 es " << n3 << "\nn4 es " << n4
19         << "\nn5 es " << n5 << "\n\n";
20
21     n5 = n1 + n2;
22     cout << n1 << " + " << n2 << " = " << n5 << "\n\n";
23
24     cout << n3 << " + " << n4 << "\n= " << ( n3 + n4 ) << "\n\n";
25
26     n5 = n1 + 9;
27     cout << n1 << " + " << 9 << " = " << n5 << "\n\n";
28
29     n5 = n2 + "10000";
30     cout << n2 << " + " << "10000" << " = " << n5 << endl;
31     return 0;
32 } // fin de main

```

Figura 11.24 | Enteros enormes.

11.15 Cree una clase llamada `NumeroRacional` (fracciones) con las siguientes capacidades:

- a) Cree un constructor que evite un denominador 0 en una fracción, que reduzca o simplifique fracciones que no estén en forma reducida y que evite los denominadores negativos.
 - b) Sobrecargue los operadores de suma, resta, multiplicación y división para esta clase.
 - c) Sobrecargue los operadores relacionales y de igualdad para esta clase.

11.16 Estudie las funciones de la biblioteca para manejo de cadenas estilo C (www.cplusplus.com/reference/clibrary/cstring/) e implemente cada una de las funciones como parte de la clase *String* (figuras 11.9 y 11.10). Después utilice estas funciones para realizar manipulaciones de texto.

11.17 Desarrolle la clase Polinomio. La representación interna de un Polinomio es un arreglo de términos. Cada término contiene un coeficiente y un exponente, por ejemplo el término

$$2x^4$$

tiene el coeficiente 2 y el exponente 4. Desarrolle una clase completa que contenga las funciones apropiadas del constructor y destructor, así como funciones *set* y *get*. La clase también debe proporcionar las siguientes herramientas de operadores sobre cargados:

- a) Sobrecargue el operador de suma (+) para sumar dos objetos Polinomio.
 - b) Sobrecargue el operador de resta (-) para restar dos objetos Polinomio.
 - c) Sobrecargue el operador de asignación para asignar un Polinomio a otro.
 - d) Sobrecargue el operador de multiplicación (*) para multiplicar dos objetos Polinomio.
 - e) Sobrecargue el operador de asignación de suma (+=), el operador de asignación de resta (-=) y el operador de asignación de multiplicación (*=).
- 11.18** En el programa de las figuras 11.3 a 11.5, la figura 11.4 contiene el comentario “operador de inserción de flujo sobre cargado; no puede ser una función miembro si queremos invocarlo con cout << unNumeroTelefonico;”. En realidad, el operador de inserción de flujo podría ser una función miembro de la clase NumeroTelefonico si deseamos invocarlo como `unNumeroTelefonico.operator<<(cout)`; o como `unNumeroTelefonico << cout;`. Vuelva a escribir el programa de la figura 11.5 con el `operator<<` de inserción de flujo sobre cargado como función miembro, y pruebe las dos instrucciones anteriores en el programa para demostrar que funcionan.



*No digas que conoces
a otro por completo,
sino hasta que hayas dividido
una herencia con él.*

—Johann Kaspar Lavater

*Este método es definir como
el número de una clase, la
clase de todas las clases
similares
a la clase dada.*

—Bertrand Russell

*Es bueno heredar
una biblioteca, pero es mejor
colecciónar una.*

—Augustine Birrell

*Preserva la autoridad base
de los libros de otros.*

—William Shakespeare

Programación orientada a objetos: herencia

OBJETIVOS

En este capítulo aprenderá a:

- Crear clases heredando de clases existentes.
- Conocer cómo la herencia promueve la reutilización de código.
- Conocer los conceptos de clases bases y clases derivadas, y las relaciones entre éstas.
- Utilizar el especificador de acceso a miembros `protected`.
- Usar constructores y destructores en las jerarquías de herencias de clases.
- Conocer el orden en el que se llama a los constructores y destructores en las jerarquías de herencia de clases.
- Conocer las diferencias entre `public`, `protected` y `private`.
- Usar la herencia para personalizar el software existente.

- 12.1** Introducción
- 12.2** Clases base y clases derivadas
- 12.3** Miembros **protected**
- 12.4** Relación entre las clases base y las clases derivadas
 - 12.4.1** Creación y uso de una clase **EmpleadoPorComision**
 - 12.4.2** Creación de una clase **EmpleadoBaseMasComision** sin usar la herencia
 - 12.4.3** Creación de una jerarquía de herencia **EmpleadoPorComision-EmpleadoBaseMasComision**
 - 12.4.4** La jerarquía de herencia **EmpleadoPorComision-EmpleadoBaseMasComision** mediante el uso de datos **protected**
 - 12.4.5** La jerarquía de herencia **EmpleadoPorComision-EmpleadoBaseMasComision** mediante el uso de datos **private**
- 12.5** Los constructores y destructores en las clases derivadas
- 12.6** Herencia **public**, **protected** y **private**
- 12.7** Ingeniería de software mediante la herencia
- 12.8** Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)

12.1 Introducción

En este capítulo continuaremos nuestra discusión acerca de la programación orientada a objetos (POO), mediante la introducción de una de sus características principales: la **herencia**, que es una forma de reutilización de software, en la cual para crear una nueva clase se absorben los datos y comportamientos de una clase existente y se mejoran con capacidades nuevas. La reutilización de software ahorra tiempo durante el desarrollo de programas. También fomenta la utilización de código probado y depurado de alta calidad, lo cual incrementa la probabilidad de que el sistema se implemente en forma efectiva.

Al crear una clase, en lugar de escribir datos miembro y funciones miembro completamente nuevos, podemos designar que la nueva clase **herede** los miembros de una ya existente. La clase existente se llama **clase base**, y la clase nueva es la **clase derivada**. (Otros lenguajes de programación, como Java, se refieren a la clase base como la **superclase** y a la nueva clase como la **subclase**). Una clase derivada representa a un grupo más especializado de objetos. Por lo general, una clase derivada contiene los comportamientos heredados de su clase base además de comportamientos adicionales. Como veremos, una clase derivada también puede personalizar los comportamientos que hereda de la clase base. Una **clase base directa** es la clase base a partir de la cual una clase derivada hereda en forma explícita. Una **clase base indirecta** es la que se hereda de dos o más niveles hacia arriba en la **jerarquía de clases**. En el caso de la **herencia simple**, una clase se deriva de una sola clase base. C++ también soporta la **herencia múltiple**, en la cual una clase derivada hereda de varias clases base (posiblemente no relacionadas). La herencia simple es directa; mostraremos varios ejemplos que permitirán al lector obtener habilidad rápidamente. La herencia múltiple puede ser compleja y propensa a errores. En el capítulo 25, Otros temas, hablaremos sobre la herencia múltiple.

C++ ofrece herencia **public**, **protected** y **private**. En este capítulo nos concentraremos en la herencia **public** y explicaremos brevemente los otros dos tipos. En el capítulo 20, Estructuras de datos, le mostraremos cómo puede utilizarse la herencia **private** como alternativa para la composición. La tercera forma, herencia **protected**, se utiliza raras veces. Con la herencia **public**, todo objeto de una clase derivada es también un objeto de la clase base de esa clase derivada. Sin embargo, los objetos de la clase base no son objetos de sus clases derivadas. Por ejemplo, si tenemos **vehículo** como clase base y **auto** como clase derivada, entonces todos los autos son vehículos, pero no todos los vehículos son autos. A medida que continuemos con nuestro estudio acerca de la programación orientada a objetos en este capítulo y en el capítulo 13, aprovecharemos esta relación para realizar ciertas manipulaciones interesantes.

La experiencia en la creación de sistemas de software indica que cantidades considerables de código tratan con casos especiales muy relacionados. Cuando los programadores se preocupan por los casos especiales, los detalles pueden oscurecer el panorama general. Con la programación orientada a objetos, los programadores pueden enfocarse en las características comunes entre los objetos en el sistema, en vez de los casos especiales.

Hay una diferencia entre la relación “*es un*” y la relación “*tiene un*”. La relación “*es un*” representa la herencia. En una relación del tipo “*es un*”, un objeto de una clase derivada también puede tratarse como un objeto de su clase base; por

ejemplo, un auto *es un* vehículo, por lo que cualquier atributo y comportamiento de un vehículo es también atributo y comportamiento de un auto. En contraste, la relación “*tiene un*” representa la composición. (En el capítulo 10 hablamos sobre la composición). En una relación del tipo “*tiene un*”, un objeto contiene uno o más objetos de otras clases como miembros. Por ejemplo, un auto incluye muchos componentes: *tiene un volante*, *tiene un pedal acelerador*, *tiene una transmisión* y *tiene* muchos otros componentes.

Las funciones miembro de las clases derivadas pueden llegar a requerir acceso a los datos miembro y funciones miembro de la clase base. Una clase derivada puede acceder a los miembros no **private** de su clase base. Los miembros de la clase base que no deben ser accesibles para las funciones miembro de las clases derivadas deben declararse como **private** en la clase base. Una clase derivada *puede* efectuar cambios de estado en los miembros **private** de la clase base, pero sólo a través de funciones miembro no **private** que se proporcionen en la clase base y se hereden en la clase derivada.



Observación de Ingeniería de Software 12.1

Las funciones miembro de una clase derivada no pueden acceder directamente a los miembros private de la clase base.



Observación de Ingeniería de Software 12.2

Si una clase derivada pudiera acceder a los miembros private de su clase base, las clases que heredan de esa clase derivada podrían acceder a esos datos también. Esto propagaría el acceso a lo que deben ser datos private, y se perderían los beneficios del ocultamiento de la información.

Un problema con la herencia es que una clase derivada puede heredar los datos miembro y las funciones miembro que no necesita o no debe tener. Es responsabilidad del diseñador de la clase asegurar que las herramientas proporcionadas por una clase sean apropiadas para las futuras clases derivadas. Aun y cuando la función miembro de una clase base sea apropiada para una clase derivada, por lo general la clase derivada requiere que la función miembro se comporte de una manera específica. En tales casos, la función miembro de la clase base se puede redefinir en la clase derivada con una implementación apropiada.

12.2 Clases base y clases derivadas

A menudo, un objeto de una clase *es un* objeto de otra clase también. Por ejemplo, en geometría, un rectángulo *es un* cuadrilátero (así como los cuadrados, los paralelogramos y los trapezoides). Por ende, en C++ se puede decir que la clase **Rectangulo** *hereda* de la clase **Cuadrilatero**. En este contexto, **Cuadrilatero** es una clase base y **Rectangulo** es una clase derivada. Un rectángulo *es un* tipo específico de cuadrilátero, pero es incorrecto decir que un cuadrilátero *es un* rectángulo; el cuadrilátero podría ser un paralelogramo o cualquier otra figura. En la figura 12.1 se listan varios ejemplos simples de las clases base y las clases derivadas.

Como todo objeto de una clase derivada *es un* objeto de su clase base, y una clase base puede tener muchas clases derivadas, el conjunto de objetos representados por una clase base es por lo general mayor que el conjunto de objetos representado por cualquiera de sus clases derivadas. Por ejemplo, la clase base **Vehiculo** representa a todos los vehículos, incluyendo autos, camiones, barcos, aviones, bicicletas, etcétera. En contraste, la clase derivada **Auto** representa un subconjunto más pequeño y específico de todos los vehículos.

Las relaciones de herencia forman estructuras jerárquicas similares a los árboles. Una clase base existe en una relación jerárquica con sus clases derivadas. Aunque las clases pueden existir de manera independiente, una vez que se emplean en relaciones de herencia, se afilian con otras clases. Una clase se convierte ya sea en una clase base (suministrando miembros a las otras clases), en una clase derivada (heredando sus miembros de otra clase), o en ambas.

Clase base	Clases derivadas
Estudiante	EstudianteGraduado, EstudianteNoGraduado
Figura	Circulo, Triangulo, Rectangulo, Esfera, Cubo
Prestamo	PrestamoAuto, PrestamoMejoraHogar, PrestamoHipotecario
Empleado	Docente, Administrativo
CuentaBanco	CuentaCheques, CuentaAhorros

Figura 12.1 | Ejemplos de herencia.

Desarrollaremos una jerarquía de herencia simple con cinco niveles (representada por el diagrama de clases de UML de la figura 12.2). Una comunidad universitaria que tiene miles de miembros.

Estos miembros consisten en empleados, estudiantes y exalumnos. Los empleados pueden ser docentes o administrativos. Los miembros docentes pueden ser administradores (como los decanos y los directores de departamento) o maestros. Sin embargo, algunos administradores también imparten clases. Observe que hemos usado la herencia múltiple para formar la clase **MaestroAdministrador**. Observe además que la jerarquía de herencia podría contener muchas otras clases. Por ejemplo, los estudiantes pueden ser graduados o no graduados. Los estudiantes no graduados pueden ser de primer año, de segundo, de tercero o de cuarto año.

Cada flecha en la jerarquía (figura 12.2) representa una relación “*es un*”. Por ejemplo, al seguir las flechas en esta jerarquía de clases podemos decir que “un **Empleado** es un **MiembroDeLaComunidad**” y que “un **Maestro** es un miembro **Docente**”. **MiembroDeLaComunidad** es la clase base directa de **Empleado**, **Estudiante** y **ExAlumno**. Además, **MiembroDeLaComunidad** es una clase base indirecta de todas las demás clases en el diagrama. Empezando desde la parte inferior del diagrama, usted puede seguir las flechas y aplicar la relación “*es un*” hasta la clase base superior. Por ejemplo, un **MaestroAdministrador** es un **Administrador**, es un miembro **Docente**, es un **Empleado** y es un **MiembroDeLaComunidad**.

Ahora considere la jerarquía de herencia para **Figura** en la figura 12.3. Esta jerarquía comienza con la clase base **Figura**. Las clases **FiguraBidimensional** y **FiguraTridimensional** se derivan de la clase base **Figura** (un objeto **Figura** puede ser **FiguraBidimensional** o **FiguraTridimensional**). El tercer nivel de esta jerarquía contiene algunos tipos más específicos de objetos **FiguraBidimensional** y **FiguraTridimensional**. Al igual que en la figura 12.2, podemos seguir las flechas desde la parte inferior del diagrama de clases hasta la clase base superior en esta jerarquía de clases, para identificar varias relaciones del tipo *es un*. Por ejemplo, un **Triangulo** es una **FiguraBidimensional** y es una **Figura**, mientras que una **Esfera** es una **FiguraTridimensional** y es una **Figura**. Observe que esta jerarquía podría contener muchas otras clases, como **Rectangulo**, **Elipse** y **Trapezoide**, que son objetos **FiguraBidimensional**.

Para especificar que la clase **FiguraBidimensional** (figura 12.3) se deriva de (o hereda de) la clase **Figura**, la definición de la clase **FiguraBidimensional** podría empezar de la siguiente manera:

```
class FiguraBidimensional : public Figura
```

Éste es un ejemplo de **herencia public**, la forma de uso más común. También hablaremos sobre la **herencia private** y la **herencia protected** (sección 12.6). Con todas las formas de herencia, los miembros **private** de una clase base no pueden utilizarse directamente desde las clases derivadas de esa clase, pero estos miembros **private** de la clase base de todas formas se heredan (es decir, se consideran parte de las clases derivadas). Con la herencia **public**, todos los demás miembros de la clase base retienen su acceso original a los miembros cuando se convierten en miembros de la clase derivada (por ejemplo, los miembros **public** de la clase base se convierten en miembros **public** de la clase derivada y, como veremos pronto, los miembros **protected** de la clase base se convierten en miembros **protected** de la clase derivada). A través de estos miembros heredados de la clase base, la clase derivada puede manipular los miembros **private** de la clase base (si estos miembros heredados proporcionan dicha funcionalidad en la clase base). Observe que las funciones **friend** no se heredan.

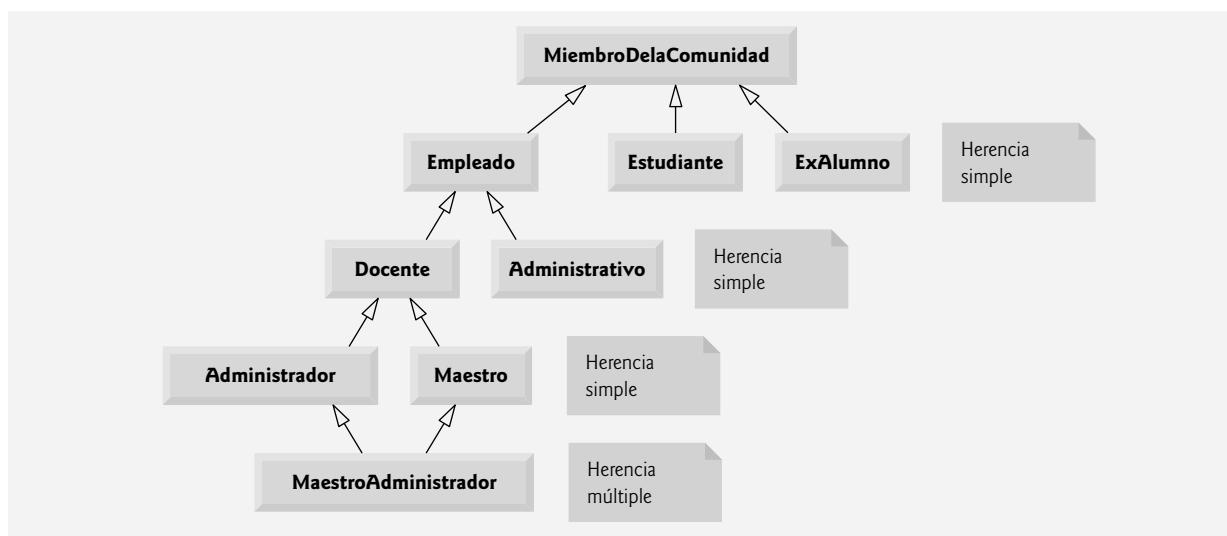


Figura 12.2 | Jerarquía de herencia para cada **MiembroDeLaComunidad** de una universidad.

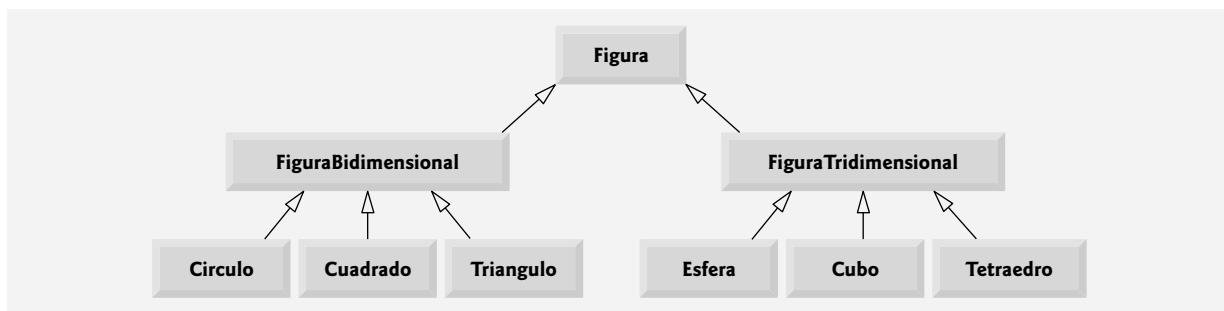


Figura 12.3 | Jerarquía de herencia para objetos Figura.

La herencia no es apropiada para todas las relaciones de las clases. En el capítulo 10 hablamos sobre la relación *tiene un*, en la cual las clases tienen miembros que son objetos de otras clases. Dichas relaciones crean clases mediante la composición de clases existentes. Por ejemplo, dadas las clases `Empleado`, `FechaNacimiento` y `NumeroTelefonico`, es impropio decir que un `Empleado` es una `FechaNacimiento` o que un `Empleado` es un `NumeroTelefonico`. Sin embargo, es apropiado decir que un `Empleado` tiene una `FechaNacimiento` y que un `Empleado` tiene un `NumeroTelefonico`.

Es posible tratar a los objetos de la clase base y a los objetos de las clases derivadas de manera similar; sus características comunes se expresan en los miembros de la clase base. Los objetos de todas las clases derivadas a partir de una clase base común se pueden tratar como objetos de esa clase base (es decir, dichos objetos tienen una relación *es un* con la clase base). En el capítulo 13, consideraremos muchos ejemplos que aprovechan esta relación.

12.3 Miembros protected

En el capítulo 3 se presentaron los especificadores de acceso `public` y `private`. Los miembros `public` de una clase base son accesibles dentro del cuerpo de esa clase base, y en cualquier parte que el programa tenga un manejador (es decir, un nombre, referencia o apuntador) para un objeto de esa clase base, o una de sus clases derivadas. Los miembros `private` de una clase base son accesibles sólo dentro de la misma clase. Los miembros `private` de una clase base son accesibles sólo dentro del cuerpo de esa clase base y de las funciones `friend` de esa clase base. En esta sección presentaremos un especificador de acceso adicional: `protected`.

El uso del acceso `protected` ofrece un nivel intermedio de protección entre el acceso `public` y `private`. Se puede acceder a los miembros `protected` de una clase base dentro del cuerpo de esa clase base, mediante los miembros y funciones `friend` de esa clase base y mediante los miembros y funciones `friend` de cualquier base que se derive de esa clase base.

Las funciones miembro de la clase derivada pueden hacer referencia a los miembros `public` y `protected` de la clase base, con sólo utilizar los nombres de los miembros. Cuando la función miembro de una clase derivada redefine a la función miembro de una clase base, se puede acceder al miembro de la clase base desde la clase derivada anteponiendo el nombre de la clase base y el operador de resolución de ámbito binario (`::`) al nombre del miembro de la clase base. En la sección 12.4 hablaremos sobre el acceso a los miembros redefinidos de la clase base, y en la sección 12.4.4 hablaremos sobre el uso de datos `protected`.

12.4 Relación entre las clases base y las clases derivadas

En esta sección usaremos una jerarquía de herencia que contiene tipos de empleados en la aplicación de nómina de una compañía, para hablar sobre la relación entre una clase base y una clase derivada. A los empleados por comisión (que se representan como objetos de una clase base) se les paga un porcentaje de sus ventas, mientras que los empleados por comisión con salario base (que se representan como objetos de una clase derivada) reciben un salario base, más un porcentaje de sus ventas. Dividiremos nuestra discusión sobre la relación entre los empleados por comisión y los empleados por comisión con salario base en una serie cuidadosamente pautada de cinco ejemplos:

1. En el primer ejemplo creamos la clase `EmpleadoPorComision`, que contiene como datos miembro `private` un primer nombre, apellido paterno, número de seguro social, tarifa de comisión (porcentaje) y monto de ventas en bruto (es decir, total).
2. El segundo ejemplo declara la clase `EmpleadoBaseMasComision`, que contiene como datos miembro `private` un primer nombre, apellido paterno, número de seguro social, tarifa de comisión, monto de ventas en bruto y salario base. Para crear esta última clase, escribiremos cada línea de código que ésta requiera; pronto veremos que es mucho más eficiente crear esta clase haciendo que herede de la clase `EmpleadoPorComision`.

3. El tercer ejemplo define una nueva versión de la clase `EmpleadoBaseMasComision` que hereda directamente de la clase `EmpleadoPorComision` (es decir, un `EmpleadoBaseMasComision` es un `EmpleadoPorComision` que también tiene un salario base) y trata de acceder a los miembros `private` de la clase `EmpleadoPorComision`; esto produce errores de compilación, ya que la clase derivada no tiene acceso a los datos `private` de la clase base.
4. El cuarto ejemplo muestra que si los datos de `EmpleadoPorComision` se declaran como `protected`, una nueva versión de la clase `EmpleadoBaseMasComision` que hereda de la clase `EmpleadoPorComision` *puede* acceder a los datos de manera directa. Para este fin, definimos una nueva versión de la clase `EmpleadoPorComision` con datos `protected`. Tanto las clases heredadas como no heredadas de `EmpleadoBasePorComision` contienen una funcionalidad idéntica, pero le mostraremos que la versión de `EmpleadoBaseMasComision` que hereda de la clase `EmpleadoPorComision` es más fácil de crear y manipular.
5. Una vez que hablamos sobre la conveniencia de utilizar datos `protected`, crearemos el quinto ejemplo, que establece los datos miembro de `EmpleadoPorComision` de vuelta a `private`, para hacer cumplir las buenas prácticas de ingeniería de software. Este ejemplo demuestra que la clase derivada `EmpleadoBaseMasComision` puede usar las funciones miembro `public` de la clase base `EmpleadoPorComision` para manipular los datos `private` de `EmpleadoPorComision`.

12.4.1 Creación y uso de una clase `EmpleadoPorComision`

Vamos a analizar la definición de la clase `EmpleadoPorComision` (figuras 12.4 a 12.5). El archivo de encabezado de `EmpleadoPorComision` (figura 12.4) especifica los servicios `public` de la clase `EmpleadoPorComision`, que incluyen un constructor (líneas 12 y 13) y las funciones miembro `ingresos` (línea 30) e `imprimir` (línea 31). En las líneas 15 a 28 se declaran funciones `public set` y `get` que manipulan los datos miembro de la clase (declarados en las líneas 33 a 37) `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision`. El archivo de encabezado de `EmpleadoPorComision` especifica que estos datos miembro son `private`, por lo que los objetos de otras clases no pueden acceder directamente a estos datos. Al declarar los datos miembro como `private` y proporcionar funciones miembro `get` y `set` no `private` para manipular y validar los datos miembros, hacemos cumplir las buenas prácticas de ingeniería de software. Por ejemplo, las funciones miembro `setVentasBrutas` (definida en las líneas 57 a 60 de la figura 12.5) y `setTarifaComision` (definida en las líneas 69 a 72 de la figura 12.5) validan sus argumentos antes de asignar los valores a los datos miembro `ventasBrutas` y `tarifaComision`, respectivamente.

La definición del constructor de `EmpleadoPorComision` no utiliza a propósito la sintaxis de inicialización de miembros en los primeros ejemplos de esta sección, para que podamos demostrar cómo afectan los especificadores `private` y `protected` al acceso a los miembros en las clases derivadas. Como se muestra en la figura 12.5, en las líneas 13 a 15 asignamos valores a los datos miembro `primerNombre`, `apellidoPaterno` y `numeroSeguroSocial` en el cuerpo del constructor. Más adelante en esta sección, volveremos a usar listas de inicialización de miembros en los constructores.

Observe que no validamos los valores de los argumentos `nombre`, `apellido` y `numeroSeguroSocial` del constructor antes de asignarlos a los correspondientes miembros de datos. Sin duda, hubiéramos podido validar el nombre y el apellido;

```

1 // Fig. 12.4: EmpleadoPorComision.h
2 // La definición de la clase EmpleadoPorComision representa a un empleado por comisión.
3 #ifndef COMISION_H
4 #define COMISION_H
5
6 #include <string> // la clase string estándar de C++
7 using std::string;
8
9 class EmpleadoPorComision
10 {
11 public:
12     EmpleadoPorComision( const string &, const string &, const string &,
13                          double = 0.0, double = 0.0 );
14
15     void setPrimerNombre( const string & ); // establece el primer nombre
16     string getPrimerNombre() const; // devuelve el primer nombre
17 }
```

Figura 12.4 | Archivo de encabezado de la clase `EmpleadoPorComision`. (Parte I de 2).

```

18 void setApellidoPaterno( const string & ); // establece el apellido paterno
19 string getApellidoPaterno() const; // devuelve el apellido paterno
20
21 void setNumeroSeguroSocial( const string & ); // establece el NSS
22 string getNumeroSeguroSocial() const; // devuelve el NSS
23
24 void setVentasBrutas( double ); // establece el monto de ventas brutas
25 double getVentasBrutas() const; // devuelve el monto de ventas brutas
26
27 void setTarifaComision( double ); // establece la tarifa de comisión (porcentaje)
28 double getTarifaComision() const; // devuelve la tarifa de comisión
29
30 double ingresos() const; // calcula los ingresos
31 void imprimir() const; // imprime el objeto EmpleadoPorComision
32 private:
33     string primerNombre;
34     string apellidoPaterno;
35     string numeroSeguroSocial;
36     double ventasBrutas; // ventas brutas por semana
37     double tarifaComision; // porcentaje de comisión
38 }; // fin de la clase EmpleadoPorComision
39
40 #endif

```

Figura 12.4 | Archivo de encabezado de la clase `EmpleadoPorComision`. (Parte 2 de 2).

```

1 // Fig. 12.5: EmpleadoPorComision.cpp
2 // Definiciones de las funciones miembro de EmpleadoPorComision.
3 #include <iostream>
4 using std::cout;
5
6 #include "EmpleadoPorComision.h" // definición de la clase EmpleadoPorComision
7
8 // constructor
9 EmpleadoPorComision::EmpleadoPorComision(
10     const string &nombre, const string &apellido, const string &nss,
11     double ventas, double tarifa )
12 {
13     primerNombre = nombre; // debe validar
14     apellidoPaterno = apellido; // debe validar
15     numeroSeguroSocial = nss; // debe validar
16     setVentasBrutas( ventas ); // valida y almacena las ventas brutas
17     setTarifaComision( tarifa ); // valida y almacena la tarifa de comisión
18 } // fin del constructor de EmpleadoPorComision
19
20 // establece el primer nombre
21 void EmpleadoPorComision::setPrimerNombre( const string &nombre )
22 {
23     primerNombre = nombre; // debe validar
24 } // fin de la función setPrimerNombre
25
26 // devuelve el primer nombre
27 string EmpleadoPorComision::getPrimerNombre() const
28 {
29     return primerNombre;
30 } // fin de la función getPrimerNombre
31
32 // establece el apellido paterno
33 void EmpleadoPorComision::setApellidoPaterno( const string &apellido )

```

Figura 12.5 | Archivo de implementación para la clase `EmpleadoPorComision` que representa a un empleado que recibe un porcentaje de las ventas brutas. (Parte 1 de 2).

```

34 {
35     apellidoPaterno = apellido; // debe validar
36 } // fin de la función setApellidoPaterno
37
38 // devuelve el apellido paterno
39 string EmpleadoPorComision::getApellidoPaterno() const
40 {
41     return apellidoPaterno;
42 } // fin de la función getApellidoPaterno
43
44 // establece el número de seguro social
45 void EmpleadoPorComision::setNumeroSeguroSocial( const string &nss )
46 {
47     numeroSeguroSocial = nss; // debe validar
48 } // fin de la función setNumeroSeguroSocial
49
50 // devuelve el número de seguro social
51 string EmpleadoPorComision::getNumeroSeguroSocial() const
52 {
53     return numeroSeguroSocial;
54 } // fin de la función getNumeroSeguroSocial
55
56 // establece el monto de ventas brutas
57 void EmpleadoPorComision::setVentasBrutas( double ventas )
58 {
59     ventasBrutas = ( ventas < 0.0 ) ? 0.0 : ventas;
60 } // fin de la función setVentasBrutas
61
62 // devuelve el monto de ventas brutas
63 double EmpleadoPorComision::getVentasBrutas() const
64 {
65     return ventasBrutas;
66 } // fin de la función getVentasBrutas
67
68 // establece la tarifa de comisión
69 void EmpleadoPorComision::setTarifaComision( double tarifa )
70 {
71     tarifaComision = ( tarifa > 0.0 && tarifa < 1.0 ) ? tarifa : 0.0;
72 } // fin de la función setTarifaComision
73
74 // devuelve la tarifa de comisión
75 double EmpleadoPorComision::getTarifaComision() const
76 {
77     return tarifaComision;
78 } // fin de la función getTarifaComision
79
80 // calcula los ingresos
81 double EmpleadoPorComision::ingresos() const
82 {
83     return tarifaComision * ventasBrutas;
84 } // fin de la función ingresos
85
86 // imprime el objeto EmpleadoPorComision
87 void EmpleadoPorComision::imprimir() const
88 {
89     cout << "empleado por comision: " << primerNombre << ' ' << apellidoPaterno
90     << "\nnumero de seguro social: " << numeroSeguroSocial
91     << "\nventas brutas: " << ventasBrutas
92     << "\ntarifa de comision: " << tarifaComision;
93 } // fin de la función imprimir

```

Figura 12.5 | Archivo de implementación para la clase `EmpleadoPorComision` que representa a un empleado que recibe un porcentaje de las ventas brutas. (Parte 2 de 2).

tal vez asegurándonos que sean de una longitud razonable. De manera similar, se podría validar un número de seguro social para asegurar que contenga nueve dígitos, con o sin guiones cortos (por ejemplo, 123-45-6789 o 123456789).

La función miembro `ingresos` (líneas 81 a 84) calcula los ingresos de un `EmpleadoPorComision`. En la línea 83 se multiplica la `tarifaComision` por las `ventasBrutas` y se devuelve el resultado. La función miembro `imprimir` (líneas 87 a 93) muestra los valores de los datos miembro de un objeto `EmpleadoPorComision`.

La figura 12.6 prueba la clase `EmpleadoPorComision`. En las líneas 16 y 17 se instancia el objeto `empleado` de la clase `EmpleadoPorComision` y se invoca su constructor para inicializar el objeto con "Sue" como primer nombre, "Jones" como apellido paterno, "222-22-2222" como número de seguro social, 10000 como el monto de ventas brutas y .06 como la tarifa de comisión. En las líneas 23 a 29 se utilizan las funciones `get` de `empleado` para mostrar los valores de sus miembros de datos. En las líneas 31 y 32 se invocan las funciones miembro del objeto `setVentasBrutas` y `setTarifaComision` para modificar los valores de los datos miembro `ventasBrutas` y `tarifaComision`, respectivamente. Después, en la línea 36 se llama a la función miembro `imprimir` de `empleado` para imprimir la información actualizada del `EmpleadoPorComision`. Por último, en la línea 39 se muestran los ingresos del `EmpleadoPorComision`, calculados por la función miembro `ingresos` del objeto mediante el uso de los valores actualizados de los datos miembro `ventasBrutas` y `tarifaComision`.

```
1 // Fig. 12.6: fig12_06.cpp
2 // Prueba de la clase EmpleadoPorComision.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 #include "EmpleadoPorComision.h" // definición de la clase EmpleadoPorComision
12
13 int main()
14 {
15     // instancia un objeto EmpleadoPorComision
16     EmpleadoPorComision empleado(
17         "Sue", "Jones", "222-22-2222", 10000, .06 );
18
19     // establece el formato de salida de punto flotante
20     cout << fixed << setprecision( 2 );
21
22     // obtiene los datos del empleado por comisión
23     cout << "Informacion del empleado obtenida por las funciones get: \n"
24         << "\nEl primer nombre es " << empleado.getPrimerNombre()
25         << "\nEl apellido paterno es " << empleado.getApellidoPaterno()
26         << "\nEl numero de seguro social es "
27         << empleado.getNumeroSeguroSocial()
28         << "\nLas ventas brutas son " << empleado.getVentasBrutas()
29         << "\nLa tarifa de comision es " << empleado.getTarifaComision() << endl;
30
31     empleado.setVentasBrutas( 8000 ); // establece las ventas brutas
32     empleado.setTarifaComision( .1 ); // establece la tarifa de comisión
33
34     cout << "\nInformacion actualizada del empleado, mostrada por la funcion imprimir: \n"
35         << endl;
36     empleado.imprimir(); // muestra la nueva informacion del empleado
37
38     // muestra los ingresos del empleado
39     cout << "\n\nIngresos del empleado: $" << empleado.ingresos() << endl;
40
41     return 0;
42 } // fin de main
```

Figura 12.6 | Programa de prueba de la clase `EmpleadoPorComision`. (Parte 1 de 2).

Información del empleado obtenida por las funciones `get`:

```
El primer nombre es Sue
El apellido paterno es Jones
El numero de seguro social es 222-22-2222
Las ventas brutas son 10000.00
La tarifa de comision es 0.06
```

Información actualizada del empleado, mostrada por la función `imprimir`:

```
empleado por comision: Sue Jones
numero de seguro social: 222-22-2222
ventas brutas: 8000.00
tarifa de comision: 0.10
```

Ingresos del empleado: \$800.00

Figura 12.6 | Programa de prueba de la clase `EmpleadoPorComision`. (Parte 2 de 2).

12.4.2 Creación de una clase `EmpleadoBaseMasComision` sin usar la herencia

Ahora veremos la segunda parte de nuestra introducción a la herencia, para lo cual vamos a crear y probar una clase (completamente nueva e independiente) llamada `EmpleadoBaseMasComision` (figuras 12.7 a 12.8), la cual contiene un primer nombre, apellido paterno, número de seguro social, monto de ventas brutas, tarifa de comisión y salario base.

Definición de la clase `EmpleadoBaseMasComision`

El archivo de encabezado de `EmpleadoBaseMasComision` (figura 12.7) especifica los servicios `public` de la clase `EmpleadoBaseMasComision`, que incluyen el constructor de `EmpleadoBaseMasComision` (líneas 13 y 14) y las funciones miembro `ingresos` (línea 34) e `imprimir` (línea 35). En las líneas 16 a 32 se declaran funciones `public get` y `set` para los datos miembro `private` de la clase (que se declaran en las líneas 37 a 42): `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas`, `tarifaComision` y `salarioBase`. Estas variables y las funciones miembro encapsulan todas las características necesarias de un empleado por comisión con salario base. Observe la similitud entre esta clase y la clase `EmpleadoPorComision` (figuras 12.4 y 12.5); en este ejemplo, no explotaremos todavía esa similitud.

```

1 // Fig. 12.7: EmpleadoBaseMasComision.h
2 // Definición de la clase EmpleadoBaseMasComision que representa
3 // a un empleado que recibe un salario base además de la comisión.
4 #ifndef BASEMAS_H
5 #define BASEMAS_H
6
7 #include <string> // clase string estándar de C++
8 using std::string;
9
10 class EmpleadoBaseMasComision
11 {
12 public:
13     EmpleadoBaseMasComision( const string &, const string &,
14                             const string &, double = 0.0, double = 0.0, double = 0.0 );
15
16     void setPrimerNombre( const string & ); // establece el primer nombre
17     string getPrimerNombre() const; // devuelve el primer nombre
18
19     void setApellidoPaterno( const string & ); // establece el apellido paterno
20     string getApellidoPaterno() const; // devuelve el apellido paterno
21
22     void setNumeroSeguroSocial( const string & ); // establece el NSS
23     string getNumeroSeguroSocial() const; // devuelve el NSS
24
```

Figura 12.7 | Archivo de encabezado de la clase `EmpleadoBaseMasComision`. (Parte 1 de 2).

```

25     void setVentasBrutas( double ); // establece el monto de ventas brutas
26     double getVentasBrutas() const; // devuelve el monto de ventas brutas
27
28     void setTarifaComision( double ); // establece la tarifa de comisión
29     double getTarifaComision() const; // devuelve la tarifa de comisión
30
31     void setSalarioBase( double ); // establece el salario base
32     double getSalarioBase() const; // devuelve el salario base
33
34     double ingresos() const; // calcula los ingresos
35     void imprimir() const; // imprime el objeto EmpleadoBaseMasComision
36 private:
37     string primerNombre;
38     string apellidoPaterno;
39     string numeroSeguroSocial;
40     double ventasBrutas; // ventas brutas por semana
41     double tarifaComision; // porcentaje de comisión
42     double salarioBase; // salario base
43 }; // fin de la clase EmpleadoBaseMasComision
44
45 #endif

```

Figura 12.7 | Archivo de encabezado de la clase `EmpleadoBaseMasComision`. (Parte 2 de 2).

La función miembro `ingresos` de la clase `EmpleadoBaseMasComision` (definida en las líneas 96 a 99 de la figura 12.8) calcula los ingresos de un empleado por comisión con salario base. En la línea 98 se devuelve el resultado de sumar el salario base del empleado al producto de la tarifa de comisión y las ventas brutas del empleado.

Prueba de la clase `EmpleadoBaseMasComision`

La figura 12.9 prueba la clase `EmpleadoBaseMasComision`. En las líneas 17 y 18 se instancia el objeto `empleado` de la clase `EmpleadoBaseMasComision`, y se pasan los datos "Bob", "Lewis", "333-33-3333", 5000, .04 y 300 al constructor como primer nombre, apellido paterno, número de seguro social, ventas brutas, tarifa de comisión y salario base, respectivamente. En las líneas 24 a 31 se utilizan las funciones `get` de `EmpleadoBaseMasComision` para obtener los valores de los datos miembro del objeto para la salida. En la línea 33 se invoca la función miembro del `setSalarioBase` del objeto para modificar el salario base. La función miembro `setSalarioBase` (figura 12.8, líneas 84 a 87) asegura que al miembro de datos `salarioBase` no se le asigne un valor negativo, ya que el salario base de un empleado no puede ser negativo. En la línea 37 de la figura 12.9 se invoca la función miembro `imprimir` del objeto para imprimir la información actualizada del `EmpleadoBaseMasComision`, y en la línea 40 se llama a la función miembro `ingresos` para mostrar los ingresos de `EmpleadoBaseMasComision`.

```

1 // Fig. 12.8: EmpleadoBaseMasComision.cpp
2 // Definiciones de las funciones miembro de la clase EmpleadoBaseMasComision.
3 #include <iostream>
4 using std::cout;
5
6 // definición de la clase EmpleadoBaseMasComision
7 #include "EmpleadoBaseMasComision.h"
8
9 // constructor
10 EmpleadoBaseMasComision::EmpleadoBaseMasComision(
11     const string &nombre, const string &apellido, const string &nss,
12     double ventas, double tarifa, double salario )
13 {
14     primerNombre = nombre; // debe validar
15     apellidoPaterno = apellido; // debe validar
16     numeroSeguroSocial = nss; // debe validar

```

Figura 12.8 | La clase `EmpleadoBaseMasComision` representa a un empleado que recibe un salario base, además de una comisión. (Parte 1 de 3).

```

17     setVentasBrutas( ventas ); // valida y almacena las ventas brutas
18     setTarifaComision( tarifa ); // valida y almacena la tarifa de comisión
19     setSalarioBase( salario ); // valida y almacena el salario base
20 } // fin del constructor de EmpleadoBaseMasComision
21
22 // establece el primer nombre
23 void EmpleadoBaseMasComision::setPrimerNombre( const string &nombre )
24 {
25     primerNombre = nombre; // debe validar
26 } // fin de la función setPrimerNombre
27
28 // devuelve el primer nombre
29 string EmpleadoBaseMasComision::getPrimerNombre() const
30 {
31     return primerNombre;
32 } // fin de la función getPrimerNombre
33
34 // establece el apellido paterno
35 void EmpleadoBaseMasComision::setApellidoPaterno( const string &apellido )
36 {
37     apellidoPaterno = apellido; // debe validar
38 } // fin de la función setApellidoPaterno
39
40 // devuelve el apellido paterno
41 string EmpleadoBaseMasComision::getApellidoPaterno() const
42 {
43     return apellidoPaterno;
44 } // fin de la función getApellidoPaterno
45
46 // establece el número de seguro social
47 void EmpleadoBaseMasComision::setNumeroSeguroSocial(
48     const string &nss )
49 {
50     numeroSeguroSocial = nss; // debe validar
51 } // fin de la función setNumeroSeguroSocial
52
53 // devuelve el número de seguro social
54 string EmpleadoBaseMasComision::getNumeroSeguroSocial() const
55 {
56     return numeroSeguroSocial;
57 } // fin de la función getNumeroSeguroSocial
58
59 // establece el monto de ventas brutas
60 void EmpleadoBaseMasComision::setVentasBrutas( double ventas )
61 {
62     ventasBrutas = ( ventas < 0.0 ) ? 0.0 : ventas;
63 } // fin de la función setVentasBrutas
64
65 // devuelve el monto de ventas brutas
66 double EmpleadoBaseMasComision::getVentasBrutas() const
67 {
68     return ventasBrutas;
69 } // fin de la función getVentasBrutas
70
71 // establece la tarifa de comisión
72 void EmpleadoBaseMasComision::setTarifaComision( double tarifa )
73 {
74     tarifaComision = ( tarifa > 0.0 && tarifa < 1.0 ) ? tarifa : 0.0;
75 } // fin de la función setTarifaComision
76

```

Figura 12.8 | La clase `EmpleadoBaseMasComision` representa a un empleado que recibe un salario base, además de una comisión. (Parte 2 de 3).

```

77 // devuelve la tarifa de comisión
78 double EmpleadoBaseMasComision::getTarifaComision() const
79 {
80     return tarifaComision;
81 } // fin de la función getTarifaComision
82
83 // establece el salario base
84 void EmpleadoBaseMasComision::setSalarioBase( double salario )
85 {
86     salarioBase = ( salario < 0.0 ) ? 0.0 : salario;
87 } // fin de la función setSalarioBase
88
89 // devuelve el salario base
90 double EmpleadoBaseMasComision::getSalarioBase() const
91 {
92     return salarioBase;
93 } // fin de la función getSalarioBase
94
95 // calcula los ingresos
96 double EmpleadoBaseMasComision::ingresos() const
97 {
98     return salarioBase + ( tarifaComision * ventasBrutas );
99 } // fin de la función ingresos
100
101 // imprime el objeto EmpleadoBaseMasComision
102 void EmpleadoBaseMasComision::imprimir() const
103 {
104     cout << "empleado por comision con salario base: " << primerNombre << ' '
105             << apellidoPaterno << "\nnumero de seguro social: " << numeroSeguroSocial
106             << "\nventas brutas: " << ventasBrutas
107             << "\ntarifa de comision: " << tarifaComision
108             << "\nsalario base: " << salarioBase;
109 } // fin de la función imprimir

```

Figura 12.8 | La clase `EmpleadoBaseMasComision` representa a un empleado que recibe un salario base, además de una comisión. (Parte 3 de 3).

```

1 // Fig. 12.9: fig12_09.cpp
2 // Prueba de la clase EmpleadoBaseMasComision.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 // definición de la clase EmpleadoBaseMasComision
12 #include "EmpleadoBaseMasComision.h"
13
14 int main()
15 {
16     // instancia un objeto EmpleadoBaseMasComision
17     EmpleadoBaseMasComision
18         empleado( "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
19
20     // establece el formato de salida de punto flotante
21     cout << fixed << setprecision( 2 );
22

```

Figura 12.9 | Programa de prueba de la clase `EmpleadoBaseMasComision`. (Parte 1 de 2).

```

23 // obtiene los datos del empleado por comisión
24 cout << "Informacion del empleado obtenida por las funciones get: \n"
25     << "\nEl primer nombre es " << empleado.getPrimerNombre()
26     << "\nEl apellido paterno es " << empleado.getApellidoPaterno()
27     << "\nEl numero de seguro social es "
28     << empleado.getNumeroSeguroSocial()
29     << "\nLas ventas brutas son " << empleado.getVentasBrutas()
30     << "\nLa tarifa de comision es " << empleado.getTarifaComision()
31     << "\nEl salario base es " << empleado.getSalarioBase() << endl;
32
33 empleado.setSalarioBase( 1000 ); // establece el salario base
34
35 cout << "\nInformacion actualizada del empleado, impresa por la funcion imprimir: \n"
36     << endl;
37 empleado.imprimir(); // muestra la nueva informacion del empleado
38
39 // muestra los ingresos del empleado
40 cout << "\n\nIngresos del empleado: $" << empleado.ingresos() << endl;
41
42 return 0;
43 } // fin de main

```

Informacion del empleado obtenida por las funciones get:

```

El primer nombre es Bob
El apellido paterno es Lewis
El numero de seguro social es 333-33-3333
Las ventas brutas son 5000.00
La tarifa de comision es 0.04
El salario base es 300.00

```

Informacion actualizada del empleado, impresa por la funcion imprimir:

```

empleado por comision con salario base: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa de comision: 0.04
salario base: 1000.00

```

Ingresos del empleado: \$1200.00

Figura 12.9 | Programa de prueba de la clase `EmpleadoBaseMasComision`. (Parte 2 de 2).

Exploración de las similitudes entre la clase `EmpleadoBaseMasComision` y la clase `EmpleadoPorComision`

Observe que la mayoría del código para la clase `EmpleadoBaseMasComision` (figuras 12.7 y 12.8) es similar (si no es que idéntico) al código de la clase `EmpleadoPorComision` (figuras 12.4 y 12.5). Por ejemplo, en la clase `EmpleadoBasePorComision`, los datos miembro `private primerNombre` y `apellidoPaterno`, y las funciones miembro `setPrimerNombre`, `getPrimerNombre`, `setApellidoPaterno` y `getApellidoPaterno` son idénticos a los de la clase `EmpleadoPorComision`. Las clases `EmpleadoPorComision` y `EmpleadoBaseMasComision` también contienen los datos miembro `private numeroSeguroSocial`, `tarifaComision` y `ventasBrutas`, así como funciones `get` y `set` para manipular esos miembros. Además, el constructor de `EmpleadoBaseMasComision` es casi idéntico al de la clase `EmpleadoPorComision`, excepto que el constructor de `EmpleadoBaseMasComision` también establece el `salarioBase`. Las demás adiciones a la clase `EmpleadoBaseMasComision` son el miembro de datos `private salarioBase`, y las funciones miembro `setSalario` y `getSalarioBase`. La función miembro `imprimir` de la clase `EmpleadoBaseMasComision` es casi idéntica a la de la clase `EmpleadoPorComision`, excepto que la función `imprimir` de `EmpleadoBaseMasComision` también imprime el valor del miembro de datos `salarioBase`.

Literalmente copiamos el código de la clase `EmpleadoPorComision` y lo pegamos en la clase `EmpleadoBaseMasComision`, después modificamos la clase `EmpleadoBaseMasComision` para incluir un salario base y las funciones miembro que lo manipulan. Esta metodología de “copiar y pegar” es propensa a errores y consume mucho tiempo. Peor aún, puede esparcir muchas copias físicas del mismo código a través de un sistema, creando una pesadilla de mantenimiento.

de código. ¿Existe una forma de “absorber” los datos miembro y las funciones miembro de una clase de una manera que los convierta en parte de otra clase, sin duplicar código? En los siguientes ejemplos haremos exactamente esto, por medio de la herencia.



Observación de Ingeniería de Software 12.3

Copiar y pegar código de una clase a otra puede esparcir errores a través de varios archivos de código fuente. Para evitar duplicar código (y posiblemente los errores), use la herencia en vez de la metodología de “copiar y pegar” en situaciones en las que una clase debe “absorber” los datos miembro y las funciones miembro de otra clase.



Observación de Ingeniería de Software 12.4

Con la herencia, los datos miembro y las funciones miembro comunes de todas las clases en la jerarquía se declaran en una clase base. Cuando se requieren cambios para esas características comunes, es necesario realizar los cambios sólo en la clase base; así, las clases derivadas pueden heredar los cambios. Sin la herencia, los cambios tendrían que realizarse en todos los archivos de código fuente que contengan una copia del código en cuestión.

12.4.3 Creación de una jerarquía de herencia EmpleadoPorComision-Employee-BaseMasComision

Ahora vamos a crear y probar una nueva clase `EmpleadoBaseMasComision` (figuras 12.10 y 12.11) que se deriva de la clase `EmpleadoPorComision` (figuras 12.4 y 12.5). En este ejemplo, un objeto `EmpleadoBaseMasComision` es un `EmpleadoPorComision` (debido a que la herencia transfiere las capacidades de la clase `EmpleadoPorComision`), pero la clase `EmpleadoBaseMasComision` también tiene el miembro de datos `salarioBase` (figura 12.10, línea 24). El signo de dos puntos (:) en la línea 12 de la definición de clase indica la herencia. La palabra clave `public` indica el tipo de herencia. Como clase derivada (que se forma con herencia `public`), `EmpleadoBaseMasComision` hereda todos los miembros de la clase `EmpleadoPorComision`, excepto el constructor; cada clase proporciona sus propios constructores específicos para ésta. [Observe que los destructores tampoco se heredan]. Por ende, los servicios `public` de `EmpleadoBaseMasComision` incluyen a su constructor (líneas 15 y 16) y las funciones miembro `public` heredadas de la clase `EmpleadoPorComision`; aunque no podemos ver estas funciones miembro heredadas en el código fuente de `EmpleadoBaseMasComision`, forman sin duda parte de esta clase derivada. Los servicios `public` de la clase derivada también incluyen las funciones miembro `setSalarioBase`, `getSalarioBase`, `ingresos` e `imprimir` (líneas 18 a 22).

La figura 12.11 muestra las implementaciones de las funciones miembro de `EmpleadoBaseMasComision`. El constructor (líneas 10 a 17) introduce la sintaxis de inicialización de clase base (línea 14), la cual utiliza un inicializador de miembros para pasar argumentos al constructor de la clase base (`EmpleadoPorComision`). C++ requiere que el constructor de una clase derivada llame al constructor de su clase base para inicializar los datos miembro de la clase base que se heredan en la clase derivada. En la línea 14 se realiza esta tarea, invocando al constructor de `EmpleadoPorComision` por su nombre, y pasando los parámetros del constructor `nombre`, `apellido`, `nss`, `ventas` y `tarifa` como argumentos para inicializar los datos miembro de la clase base `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision`. Si el constructor de `EmpleadoBaseMasComision` no invocara al constructor de la clase `EmpleadoPorComision` de manera explícita, C++ trataría de invocar al constructor predeterminado de la clase `EmpleadoPorComision`; pero esta clase no tiene un constructor de este tipo, por lo que el compilador generaría un error. En el capítulo 3 vimos que el compilador proporciona un constructor predeterminado sin parámetros en cualquier clase que no incluya de manera explícita un constructor. Sin embargo, `EmpleadoPorComision` sí incluye de manera explícita un constructor, por lo que no se proporciona un constructor predeterminado, y cualquier intento de llamar implícitamente al constructor predeterminado de `EmpleadoPorComision` generaría errores de compilación.

```

1 // Fig. 12.10: EmpleadoBaseMasComision.h
2 // Clase EmpleadoBaseMasComision derivada de la clase
3 // EmpleadoPorComision.
4 #ifndef BASEMAS_H
5 #define BASEMAS_H
6
7 #include <string> // clase string estándar de C++

```

Figura 12.10 | Definición de la clase `EmpleadoBaseMasComision` que indica la relación de herencia con la clase `EmpleadoPorComision`. (Parte 1 de 2).

```

8  using std::string;
9
10 #include "EmpleadoPorComision.h" // declaración de la clase EmpleadoPorComision
11
12 class EmpleadoBaseMasComision : public EmpleadoPorComision
13 {
14 public:
15     EmpleadoBaseMasComision( const string &, const string &,
16                             const string &, double = 0.0, double = 0.0 );
17
18     void setSalarioBase( double ); // establece el salario base
19     double getSalarioBase() const; // devuelve el salario base
20
21     double ingresos() const; // calcula los ingresos
22     void imprimir() const; // imprime el objeto EmpleadoBaseMasComision
23 private:
24     double salarioBase; // salario base
25 }; // fin de la clase EmpleadoBaseMasComision
26
27 #endif

```

Figura 12.10 | Definición de la clase `EmpleadoBaseMasComision` que indica la relación de herencia con la clase `EmpleadoPorComision`. (Parte 2 de 2).

```

1 // Fig. 12.11: EmpleadoBaseMasComision.cpp
2 // Definiciones de las funciones miembro de la clase EmpleadoBaseMasComision.
3 #include <iostream>
4 using std::cout;
5
6 // Definición de la clase EmpleadoBaseMasComision
7 #include "EmpleadoBaseMasComision.h"
8
9 // constructor
10 EmpleadoBaseMasComision::EmpleadoBaseMasComision(
11     const string &nomb, const string &apellido, const string &nss,
12     double ventas, double tarifa, double salario )
13     // llama explícitamente al constructor de la clase base
14     : EmpleadoPorComision( nombre, apellido, nss, ventas, tarifa )
15 {
16     setSalarioBase( salario ); // valida y almacena el salario base
17 } // fin del constructor de EmpleadoBaseMasComision
18
19 // establece el salario base
20 void EmpleadoBaseMasComision::setSalarioBase( double salario )
21 {
22     salarioBase = ( salario < 0.0 ) ? 0.0 : salario;
23 } // fin de la función setSalarioBase
24
25 // devuelve el salario base
26 double EmpleadoBaseMasComision::getSalarioBase() const
27 {
28     return salarioBase;
29 } // fin de la función getSalarioBase
30
31 // calcula los ingresos
32 double EmpleadoBaseMasComision::ingresos() const
33 {
34     // la clase derivada no puede acceder a los datos privados de la clase base
35     return salarioBase + ( tarifaComision * ventasBrutas );

```

Figura 12.11 | Archivo de implementación de `EmpleadoBaseMasComision`: la clase derivada no puede acceder a los datos `private` de la clase base. (Parte 1 de 3).

```

36 } // fin de la función ingresos
37
38 // imprime el objeto EmpleadoBaseMasComision
39 void EmpleadoBaseMasComision::imprimir() const
40 {
41     // la clase derivada no puede acceder a los datos privados de la clase base
42     cout << "empleado por comision con salario base: " << primerNombre << ' '
43     << lastName << "\nnumero de seguro social: " << numeroSeguroSocial
44     << "\nventas brutas: " << ventasBrutas
45     << "\ntarifa de comision: " << tarifaComision
46     << "\nsalario base: " << salarioBase;
47 } // fin de la función imprimir

```

```

C:\cpphtp6_ejemplos\cap12\fig12_10_11\empleadobasemascomision.cpp(35) :
error C2248: 'EmpleadoPorComision::tarifaComision' :
no se puede obtener acceso al miembro private miembro declarado en la clase 'EmpleadoPorComision'
C:\cpphtp6_ejemplos\cap12\fig12_10_11\empleadoporcomision.h(37) :
vea la declaración de 'EmpleadoPorComision::tarifaComision'
C:\cpphtp6_ejemplos\cap12\fig12_10_11\empleadoporcomision.h(10) :
vea la declaración de 'EmpleadoPorComision'

C:\cpphtp6_ejemplos\cap12\fig12_10_11\empleadobasemascomision.cpp(35) :
error C2248: 'EmpleadoPorComision::ventasBrutas' :
no se puede obtener acceso al miembro private miembro declarado en la clase 'EmpleadoPorComision'
C:\cpphtp6_ejemplos\cap12\fig12_10_11\empleadoporcomision.h(36) :
vea la declaración de 'EmpleadoPorComision::ventasBrutas'
C:\cpphtp6_ejemplos\cap12\fig12_10_11\empleadoporcomision.h(10) :
vea la declaración de 'EmpleadoPorComision'

C:\cpphtp6_ejemplos\cap12\fig12_10_11\empleadobasemascomision.cpp(42) :
error C2248: 'EmpleadoPorComision::primerNombre' :
no se puede obtener acceso al miembro private miembro declarado en la clase 'EmpleadoPorComision'
C:\cpphtp6_ejemplos\cap12\fig12_10_11\empleadoporcomision.h(33) :
vea la declaración de 'EmpleadoPorComision::primerNombre'
C:\cpphtp6_ejemplos\cap12\fig12_10_11\empleadoporcomision.h(10) :
vea la declaración de 'EmpleadoPorComision'

C:\cpphtp6_ejemplos\cap12\fig12_10_11\empleadobasemascomision.cpp(43) :
error C2248: 'EmpleadoPorComision::apellidoPaterno' :
no se puede obtener acceso al miembro private miembro declarado en la clase 'EmpleadoPorComision'
C:\cpphtp6_ejemplos\cap12\fig12_10_11\empleadoporcomision.h(34) :
vea la declaración de 'EmpleadoPorComision::apellidoPaterno'
C:\cpphtp6_ejemplos\cap12\fig12_10_11\empleadoporcomision.h(10) :
vea la declaración de 'EmpleadoPorComision'

C:\cpphtp6_ejemplos\cap12\fig12_10_11\empleadobasemascomision.cpp(43) :
error C2248: 'EmpleadoPorComision::numeroSeguroSocial' :
no se puede obtener acceso al miembro private miembro declarado en la clase 'EmpleadoPorComision'
C:\cpphtp6_ejemplos\cap12\fig12_10_11\empleadoporcomision.h(35) :
vea la declaración de 'EmpleadoPorComision::numeroSeguroSocial'
C:\cpphtp6_ejemplos\cap12\fig12_10_11\empleadoporcomision.h(10) :
vea la declaración de 'EmpleadoPorComision'

C:\cpphtp6_ejemplos\cap12\fig12_10_11\empleadobasemascomision.cpp(44) :
error C2248: 'EmpleadoPorComision::ventasBrutas' :
no se puede obtener acceso al miembro private miembro declarado en la clase 'EmpleadoPorComision'
C:\cpphtp6_ejemplos\cap12\fig12_10_11\empleadoporcomision.h(36) :
vea la declaración de 'EmpleadoPorComision::ventasBrutas'
C:\cpphtp6_ejemplos\cap12\fig12_10_11\empleadoporcomision.h(10) :
vea la declaración de 'EmpleadoPorComision'

```

Figura 12.11 | Archivo de implementación de EmpleadoBaseMasComision: la clase derivada no puede acceder a los datos private de la clase base. (Parte 2 de 3).

```
C:\cpphttp6_ejemplos\cap12\fig12_10_11\empleadobasemascomision.cpp(45) :
error C2248: 'EmpleadoPorComision::tarifaComision' :
no se puede obtener acceso al miembro private miembro declarado en la clase 'EmpleadoPorComision'
C:\cpphttp6_ejemplos\cap12\fig12_10_11\empleadoporcomision.h(37) :
vea la declaración de 'EmpleadoPorComision::tarifaComision'
C:\cpphttp6_ejemplos\cap12\fig12_10_11\empleadoporcomision.h(10) :
vea la declaración de 'EmpleadoPorComision'
```

Figura 12.11 | Archivo de implementación de `EmpleadoBaseMasComision`: la clase derivada no puede acceder a los datos `private` de la clase base. (Parte 2 de 3).

Error común de programación 12.1



Si el constructor de una clase derivada llama a uno de los constructores de su clase base con argumentos que sean inconsistentes con el número y tipos de los parámetros especificados en una de las definiciones del constructor de la clase base, se produce un error de compilación.

Tip de rendimiento 12.1



En el constructor de una clase derivada, al inicializar los objetos miembro e invocar a los constructores de la clase base de manera explícita en la lista de inicializadores de miembros, se evita duplicar la inicialización en la que se hace la llamada a un constructor predeterminado, y después los datos miembro se modifican de nuevo en el cuerpo del constructor de la clase derivada.

El compilador genera errores para la línea 35 de la figura 12.11 debido a que los datos miembro `tarifaComision` y `ventasBrutas` de la clase base `EmpleadoPorComision` son `private`; las funciones miembro de la clase derivada `EmpleadoBaseMasComision` no pueden acceder a los datos `private` de la clase base `EmpleadoPorComision`. Observe que utilizamos texto en color gris en la figura 12.11 (líneas 34-35, 41-45) para indicar el código erróneo. El compilador genera errores adicionales en las líneas 42 a 45 de la función miembro `imprimir` de `EmpleadoBaseMasComision` por la misma razón. Como podemos ver, C++ hace cumplir rígidamente las restricciones en cuanto al acceso a los datos miembro `private`, de tal forma que hasta una clase derivada (que está íntimamente relacionada con su clase base) no puede acceder a los datos `private` de la clase base. [Nota: para ahorrar espacio sólo mostramos los mensajes de error de Visual C++ 2005 en este ejemplo. Los mensajes de error producidos por su compilador podrían ser distintos de los que se muestran aquí. Observe además que resaltamos en negrita las porciones clave de los extensos mensajes de error].

Incluimos a propósito el código erróneo en la figura 12.11 para enfatizar que las funciones miembro de una clase derivada no pueden acceder a los datos `private` de su clase base. Los errores en `EmpleadoBaseMasComision` se podían haber evitado mediante el uso de las funciones miembro `get` heredadas de la clase `EmpleadoPorComision`. Por ejemplo, en la línea 35 se podía haber invocado a `getTarifaComision` y a `setVentasBrutas` para acceder a los datos miembro `private tarifaComision` y `ventasBrutas` de `EmpleadoPorComision`, respectivamente. De manera similar, en las líneas 42 a 45 se pudieron haber utilizado funciones miembro `get` apropiadas para obtener los valores de los datos miembro de la clase base. En el siguiente ejemplo le mostraremos cómo el uso de datos `protected` también nos permite evitar los errores que encontramos en este ejemplo.

Cómo incluir el archivo de encabezado de la clase base en el archivo de encabezado de la clase derivada mediante `#include`

Observe que incluimos el archivo de encabezado de la clase base en el archivo de encabezado de la clase derivada (línea 10 de la figura 12.10). Esto es necesario por tres razones. En primer lugar, para que la clase derivada utilice el nombre de la clase base en la línea 12, debemos indicar al compilador que la clase base existe; la definición de clase en `EmpleadoPorComision.h` hace exactamente eso.

La segunda razón es que el compilador utiliza una definición de clase para determinar el tamaño de un objeto de esa clase (como vimos en la sección 3.8). Un programa cliente que crea un objeto de una clase debe incluir (mediante `#include`) la definición de clase para permitir que el compilador reserve la cantidad apropiada de memoria para el objeto. Al usar herencia, el tamaño de un objeto de una clase derivada depende de los datos miembro declarados explícitamente en su definición de clase, y de los datos miembro heredados de sus clases base directa e indirecta. Al incluir la definición de la clase base en la línea 10, permitimos que el compilador determine los requerimientos de memoria para los datos miembro de la clase base que se conviertan en parte del objeto de una clase derivada, y por ende contribuyen al tamaño total del objeto de la clase derivada.

La última razón de incluir la línea 10 es para permitir al compilador determinar si la clase derivada utiliza los miembros heredados de la clase base en forma apropiada. Por ejemplo, en el programa de las figuras 12.10 y 12.11, el compilador utiliza el archivo de encabezado de la clase base para determinar que los datos miembro que está usando la clase derivada son `private` en la clase base. Como éstos son inaccesibles para la clase derivada, el compilador genera errores. El compilador también utiliza los prototipos de las funciones de la clase base para validar las llamadas a funciones realizadas por la clase derivada a las funciones heredadas de la clase base; en la figura 12.16 veremos un ejemplo de dicha llamada a función.

El proceso de enlace en una jerarquía de herencia

En la sección 3.9, hablamos sobre el proceso de enlace para crear una aplicación `LibroCalificaciones` ejecutable. En ese ejemplo, vimos que el código objeto del cliente se enlazó con el código objeto para la clase `LibroCalificaciones`, así como con el código objeto para cualquier clase de la Biblioteca estándar de C++ utilizada en el código cliente o en la clase `LibroCalificaciones`.

El proceso de enlace es similar para un programa que utiliza clases en una jerarquía de herencia. El proceso requiere el código objeto para todas las clases utilizadas en el programa, y el código objeto para las clases base directas e indirectas de cualquier clase derivada utilizada por el programa. Suponga que un cliente desea crear una aplicación que utilice la clase `EmpleadoBaseMasComision`, la cual es una clase derivada de `EmpleadoPorComision` (en la sección 12.4.4 veremos un ejemplo de esto). Al compilar la aplicación cliente, el código objeto del cliente debe enlazarse con el código objeto para las clases `EmpleadoBaseMasComision` y `EmpleadoPorComision`, ya que la clase `EmpleadoBaseMasComision` hereda las funciones miembro de su clase base `EmpleadoPorComision`. El código también se enlaza con el código objeto para cualquier clase de la Biblioteca estándar de C++ que se utilice en las clases `EmpleadoPorComision`, `EmpleadoBaseMasComision` o en el código cliente. Esto proporciona al programa acceso a las implementaciones de toda la funcionalidad que el programa puede utilizar.

12.4.4 La jerarquía de herencia `EmpleadoPorComision-`

`EmpleadoBaseMasComision` mediante el uso de datos `protected`

Para permitir que la clase `EmpleadoBaseMasComision` acceda directamente a los datos miembro `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` de `EmpleadoPorComision`, podemos declarar esos miembros como `protected` en la clase base. Como vimos en la sección 12.3, los miembros `protected` de una clase base pueden ser utilizados por los miembros y funciones `friend` de la clase base, y por los miembros y funciones `friend` de cualquier clase derivada de esa clase base.



Buena práctica de programación 12.1

Declare los miembros `public` primero, los miembros `protected` en segundo lugar y los miembros `private` al último.

Definición de la clase base `EmpleadoPorComision` con datos `protected`

La clase `EmpleadoPorComision` (figuras 12.12 y 12.13) declara ahora los datos miembro `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` como `protected` (figura 12.12, líneas 33 a 37) en vez de `private`. Las implementaciones de las funciones miembro en la figura 12.13 son idénticas a las de la figura 12.5.

```

1 // Fig. 12.12: EmpleadoPorComision.h
2 // Definición de la clase EmpleadoPorComision con datos protected.
3 #ifndef COMISION_H
4 #define COMISION_H
5
6 #include <string> // clase string estándar de C++
7 using std::string;
8
9 class EmpleadoPorComision
10 {
11 public:
```

Figura 12.12 | Definición de la clase `EmpleadoPorComision` que declara datos `protected`, para permitir que las clases derivadas accedan a éstos. (Parte 1 de 2)

```

12     EmpleadoPorComision( const string &, const string &, const string &,
13         double = 0.0, double = 0.0 );
14
15     void setPrimerNombre( const string & ); // establece el primer nombre
16     string getPrimerNombre() const; // devuelve el primer nombre
17
18     void setApellidoPaterno( const string & ); // establece el apellido paterno
19     string getApellidoPaterno() const; // devuelve el apellido paterno
20
21     void setNumeroSeguroSocial( const string & ); // establece el NSS
22     string getNumeroSeguroSocial() const; // devuelve el NSS
23
24     void setVentasBrutas( double ); // establece el monto de ventas brutas
25     double getVentasBrutas() const; // devuelve el monto de ventas brutas
26
27     void setTarifaComision( double ); // establece la tarifa de comisión
28     double getTarifaComision() const; // devuelve la tarifa de comisión
29
30     double ingresos() const; // calcula los ingresos
31     void imprimir() const; // imprime el objeto EmpleadoPorComision
32 protected:
33     string primerNombre;
34     string apellidoPaterno;
35     string numeroSeguroSocial;
36     double ventasBrutas; // ventas brutas por semana
37     double tarifaComision; // porcentaje de comisión
38 }; // fin de la clase EmpleadoPorComision
39
40 #endif

```

Figura 12.12 | Definición de la clase `EmpleadoPorComision` que declara datos `protected`, para permitir que las clases derivadas accedan a éstos. (Parte 2 de 2).

```

1 // Fig. 12.13: EmpleadoPorComision.cpp
2 // Definiciones de las funciones miembro de EmpleadoPorComision.
3 #include <iostream>
4 using std::cout;
5
6 #include "EmpleadoPorComision.h" // definición de la clase EmpleadoPorComision
7
8 // constructor
9 EmpleadoPorComision::EmpleadoPorComision(
10     const string &nombre, const string &apellido, const string &nss,
11     double ventas, double tarifa )
12 {
13     primerNombre = nombre; // debe validar
14     apellidoPaterno = apellido; // debe validar
15     numeroSeguroSocial = nss; // debe validar
16     setVentasBrutas( ventas ); // valida y almacena las ventas brutas
17     setTarifaComision( tarifa ); // valida y almacena la tarifa de comisión
18 } // fin del constructor de EmpleadoPorComision
19
20 // establece el primer nombre
21 void EmpleadoPorComision::setPrimerNombre( const string &nombre )
22 {
23     primerNombre = nombre; // debe validar
24 } // fin de la función setPrimerNombre
25
26 // devuelve el primer nombre
27 string EmpleadoPorComision::getPrimerNombre() const

```

Figura 12.13 | Clase `EmpleadoPorComision` con datos `protected`. (Parte 1 de 3).

```
28 {
29     return primerNombre;
30 } // fin de la función getPrimerNombre
31
32 // establece el apellido paterno
33 void EmpleadoPorComision::setApellidoPaterno( const string &apellido )
34 {
35     apellidoPaterno = apellido; // debe validar
36 } // fin de la función setApellidoPaterno
37
38 // devuelve el apellido paterno
39 string EmpleadoPorComision::getApellidoPaterno() const
40 {
41     return apellidoPaterno;
42 } // fin de la función getApellidoPaterno
43
44 // establece el número de seguro social
45 void EmpleadoPorComision::setNumeroSeguroSocial( const string &nss )
46 {
47     numeroSeguroSocial = nss; // debe validar
48 } // fin de la función setNumeroSeguroSocial
49
50 // devuelve el número de seguro social
51 string EmpleadoPorComision::getNumeroSeguroSocial() const
52 {
53     return numeroSeguroSocial;
54 } // fin de la función getNumeroSeguroSocial
55
56 // establece el monto de ventas brutas
57 void EmpleadoPorComision::setVentasBrutas( double ventas )
58 {
59     ventasBrutas = ( ventas < 0.0 ) ? 0.0 : ventas;
60 } // fin de la función setVentasBrutas
61
62 // devuelve el monto de las ventas brutas
63 double EmpleadoPorComision::getVentasBrutas() const
64 {
65     return ventasBrutas;
66 } // fin de la función getVentasBrutas
67
68 // establece la tarifa de comisión
69 void EmpleadoPorComision::setTarifaComision( double tarifa )
70 {
71     tarifaComision = ( tarifa > 0.0 && tarifa < 1.0 ) ? tarifa : 0.0;
72 } // fin de la función setTarifaComision
73
74 // devuelve la tarifa de comisión
75 double EmpleadoPorComision::getTarifaComision() const
76 {
77     return tarifaComision;
78 } // fin de la función getTarifaComision
79
80 // calcula los ingresos
81 double EmpleadoPorComision::ingresos() const
82 {
83     return tarifaComision * ventasBrutas;
84 } // fin de la función ingresos
85
86 // imprime el objeto EmpleadoPorComision
87 void EmpleadoPorComision::imprimir() const
88 {
89     cout << "empleado por comision: " << primerNombre << ' ' << apellidoPaterno
```

Figura 12.13 | Clase EmpleadoPorComision con datos protected. (Parte 2 de 3).

```

90      << "\nnumero de seguro social: " << numeroSeguroSocial
91      << "\nventas brutas: " << ventasBrutas
92      << "\ntarifa de comision: " << tarifaComision;
93 } // fin de la función imprimir

```

Figura 12.13 | Clase EmpleadoPorComision con datos `protected`. (Parte 3 de 3).

Modificación de la clase derivada `EmpleadoBaseMasComision`

Ahora vamos a modificar la clase `EmpleadoBaseMasComision` (figuras 12.14 y 12.15), de manera que herede de la clase `EmpleadoPorComision` de las figuras 12.12 y 12.13. Como la clase `EmpleadoBaseMasComision` hereda de esta versión de la clase `EmpleadoPorComision`, los objetos de la clase `EmpleadoBaseMasComision` pueden acceder a los datos miembro heredados que se declaran `protected` en la clase `EmpleadoPorComision` (es decir, los datos miembro `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision`). Como resultado, el compilador no genera errores al compilar las definiciones de las funciones miembro `ingresos` e `imprimir` de `EmpleadoPorComision` en la figura 12.15 (líneas 32 a 36 y 39 a 47, respectivamente). Esto muestra los privilegios especiales que se otorgan a una clase derivada para acceder a los datos miembro `protected` de su clase base. Los objetos de una clase derivada también pueden acceder a los miembros `protected` en cualquiera de las clases base indirecta de la clase derivada.

```

1 // Fig. 12.14: EmpleadoBaseMasComision.h
2 // Clase EmpleadoBaseMasComision derivada de la clase
3 // EmpleadoPorComision.
4 #ifndef BASEMAS_H
5 #define BASEMAS_H
6
7 #include <string> // clase string estándar de C++
8 using std::string;
9
10 #include "EmpleadoPorComision.h" // declaración de la clase EmpleadoPorComision
11
12 class EmpleadoBaseMasComision : public EmpleadoPorComision
13 {
14 public:
15     EmpleadoBaseMasComision( const string &, const string &,
16                             const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18     void setSalarioBase( double ); // establece el salario base
19     double getSalarioBase() const; // devuelve el salario base
20
21     double ingresos() const; // calcula los ingresos
22     void imprimir() const; // imprime el objeto EmpleadoBaseMasComision
23 private:
24     double salarioBase; // salario base
25 }; // fin de la clase EmpleadoBaseMasComision
26
27 #endif

```

Figura 12.14 | Archivo de encabezado de la clase `EmpleadoBaseMasComision`.

```

1 // Fig. 12.15: EmpleadoBaseMasComision.cpp
2 // Definiciones de las funciones miembro de la clase EmpleadoBaseMasComision.
3 #include <iostream>
4 using std::cout;
5
6 // definición de la clase EmpleadoBaseMasComision
7 #include "EmpleadoBaseMasComision.h"

```

Figura 12.15 | Archivo de implementación de `EmpleadoBaseMasComision` para la clase `EmpleadoBaseMasComision`, que hereda los datos `protected` de `EmpleadoPorComision`. (Parte 1 de 2).

```

8 // constructor
9 EmpleadoBaseMasComision::EmpleadoBaseMasComision(
10    const string &nombre, const string &apellido, const string &nss,
11    double ventas, double tarifa, double salario )
12    // llama explícitamente al constructor de la clase base
13    : EmpleadoPorComision( nombre, apellido, nss, ventas, tarifa )
14 {
15     setSalarioBase( salario ); // valida y almacena el salario base
16 } // fin del constructor de EmpleadoBaseMasComision
17
18
19 // establece el salario base
20 void EmpleadoBaseMasComision::setSalarioBase( double salario )
21 {
22     salarioBase = ( salario < 0.0 ) ? 0.0 : salario;
23 } // fin de la función setSalarioBase
24
25 // devuelve el salario base
26 double EmpleadoBaseMasComision::getSalarioBase() const
27 {
28     return salarioBase;
29 } // fin de la función getSalarioBase
30
31 // calcula los ingresos
32 double EmpleadoBaseMasComision::ingresos() const
33 {
34     // puede acceder a los datos protected de la clase base
35     return salarioBase + ( tarifaComision * ventasBrutas );
36 } // fin de la función ingresos
37
38 // imprime el objeto EmpleadoBaseMasComision
39 void EmpleadoBaseMasComision::imprimir() const
40 {
41     // puede acceder a los datos protected de la clase base
42     cout << "empleado por comision con salario base: " << primerNombre << ' '
43         << apellidoPaterno << "\nnumero de seguro social: " << numeroSeguroSocial
44         << "\nventas brutas: " << ventasBrutas
45         << "\ntarifa de comision: " << tarifaComision
46         << "\nsalario base: " << salarioBase;
47 } // fin de la función imprimir

```

Figura 12.15 | Archivo de implementación de `EmpleadoBaseMasComision` para la clase `EmpleadoBaseMasComision`, que hereda los datos `protected` de `EmpleadoPorComision`. (Parte 2 de 2).

La clase `EmpleadoBaseMasComision` no hereda el constructor de la clase `EmpleadoPorComision`. Sin embargo, el constructor de la clase `EmpleadoBaseMasComision` (figura 12.15, líneas 10 a 17) llama al constructor de la clase `EmpleadoPorComision` de manera explícita, mediante la sintaxis de inicializador de miembros (línea 14). Recuerde que el constructor de `EmpleadoBaseMasComision` debe llamar en forma explícita al constructor de la clase `EmpleadoPorComision`, ya que `EmpleadoPorComision` no contiene un constructor predeterminado que pueda invocarse en forma implícita.

Prueba de la clase `EmpleadoBaseMasComision` modificada

La figura 12.16 utiliza un objeto `EmpleadoBaseMasComision` para realizar las mismas tareas que realizó la figura 12.9 en un objeto de la primera versión de la clase `EmpleadoBaseMasComision` (figuras 12.7 y 12.8). Observe que los resultados de los dos programas son idénticos. Creamos la primera clase `EmpleadoBaseMasComision` sin usar la herencia, y creamos esta versión de `EmpleadoBaseMasComision` usando la herencia; sin embargo, ambas clases proporcionan la misma funcionalidad. Observe que el código para la clase `EmpleadoBaseMasComision` (es decir, los archivos de encabezado y de implementación), que es de 74 líneas, es considerablemente más corto que el código para la versión no heredada de la clase, que es de 154 líneas, debido a que la versión heredada absorbe parte de su funcionalidad de `EmpleadoPorComision`, mientras que la versión no heredada no absorbe ninguna funcionalidad. Además, ahora sólo hay una copia de la funcionalidad de `EmpleadoPorComision` declarada y definida en la clase `EmpleadoPorComision`. Esto facilita el

```

1 // Fig. 12.16: fig12_16.cpp
2 // Prueba de la clase EmpleadoBaseMasComision.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 // definición de la clase EmpleadoBaseMasComision
12 #include "EmpleadoBaseMasComision.h"
13
14 int main()
15 {
16     // instancia un objeto EmpleadoBaseMasComision
17     EmpleadoBaseMasComision
18         empleado( "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
19
20     // establece el formato de salida de punto flotante
21     cout << fixed << setprecision( 2 );
22
23     // obtiene los datos del empleado por comisión
24     cout << "Informacion del empleado obtenida por las funciones get: \n"
25         << "\nEl primer nombre es " << empleado.getPrimerNombre()
26         << "\nEl apellido paterno es " << empleado.getApellidoPaterno()
27         << "\nEl numero de seguro social es "
28         << empleado.getNumeroSeguroSocial()
29         << "\nLas ventas brutas son " << empleado.getVentasBrutas()
30         << "\nLa tarifa de comision es " << empleado.getTarifaComision()
31         << "\nEl salario base es " << empleado.getSalarioBase() << endl;
32
33     empleado.setSalarioBase( 1000 ); // establece el salario base
34
35     cout << "\nInformacion actualizada del empleado, impresa por la funcion imprimir: \n"
36         << endl;
37     empleado.imprimir(); // muestra la nueva información del empleado
38
39     // muestra los ingresos del empleado
40     cout << "\n\nIngresos del empleado: $" << empleado.ingresos() << endl;
41
42     return 0;
43 } // fin de main

```

Informacion del empleado obtenida por las funciones get:

El primer nombre es Bob
 El apellido paterno es Lewis
 El numero de seguro social es 333-33-3333
 Las ventas brutas son 5000.00
 La tarifa de comision es 0.04
 El salario base es 300.00

Informacion actualizada del empleado, impresa por la funcion imprimir:

empleado por comision con salario base: Bob Lewis
 numero de seguro social: 333-33-3333
 ventas brutas: 5000.00
 tarifa de comision: 0.04
 salario base: 1000.00

Ingresos del empleado: \$1200.00

Figura 12.16 | Los datos **protected** de la clase base se pueden utilizar desde la clase derivada.

mantenimiento, la modificación y depuración del código fuente, ya que el código fuente relacionado a un `EmpleadoPorComision` sólo existe en los archivos de las figuras 12.12 y 12.13.

Observaciones acerca de los datos `protected`

En este ejemplo, declaramos los datos miembro de la clase base como `protected`, de manera que las clases derivadas puedan modificar los datos directamente. Al heredar los datos miembro `protected` se incrementa ligeramente el rendimiento, ya que podemos acceder directamente a los miembros sin incurrir en la sobrecarga de las llamadas a funciones `set` o `get`. Sin embargo, en la mayoría de los casos es mejor usar datos miembro `private` para fomentar la ingeniería de software apropiada, y dejar las cuestiones de optimización de código al compilador. El código del programador será más fácil de mantener, modificar y depurar.

El uso de datos miembro `protected` crea dos problemas graves. En primer lugar, el objeto de la clase derivada no tiene que usar una función miembro para establecer el valor del miembro de datos `protected` de la clase base. Por lo tanto, el objeto de una clase derivada puede asignar fácilmente un valor inválido al miembro de datos `protected`, con lo cual se deja el objeto en un estado inconsistente. Por ejemplo, con el miembro de datos `ventasBrutas` de `EmpleadoPorComision` declarado como `protected`, el objeto de una clase derivada (por ejemplo, `EmpleadoBaseMasComision`) puede asignar un valor negativo a `ventasBrutas`. El segundo problema con el uso de datos miembro `protected` es que es más probable que las funciones miembro de la clase derivada se escriban de manera que dependan en la implementación de la clase base. En la práctica, las clases derivadas sólo deben depender de los servicios de la clase base (es decir, las funciones miembro no `private`) y no de su implementación. Con los datos miembro `protected` en la clase base, si se modifica la implementación de la clase base, tal vez haya que modificar todas las clases derivadas de esa clase base. Por ejemplo, si por alguna razón tuviéramos que modificar los nombres de los datos miembro `primerNombre` y `apellidoPaterno` a `nombre` y `apellido`, entonces tendríamos que hacerlo para todas las ocurrencias en las que una clase derivada haga referencia a estos datos miembro de la clase base directamente. En tal caso, se dice que el software es **frágil** o **quebradizo**, ya que una pequeña modificación en la clase base puede “quebrantar” la implementación de la clase derivada. El programador debe tener la capacidad de modificar la implementación de la clase base y al mismo tiempo debe poder seguir proporcionando los mismos servicios a las clases derivadas. (Desde luego, si los servicios de la clase base cambian, debemos reimplementar nuestras clases derivadas; el buen diseño orientado a objetos trata de evitar esto).



Observación de Ingeniería de Software 12.5

Es apropiado usar el especificador de acceso `protected` cuando una clase base debe proporcionar un servicio (es decir, una función miembro) sólo a sus clases derivadas (y funciones `friend`), no a otros clientes.



Observación de Ingeniería de Software 12.6

Declarar los datos miembro de la clase base como `private` (en vez de declararlos `protected`) permite a los programadores modificar la implementación de la clase base, sin tener que modificar la implementación de la clase derivada.



Tip para prevenir errores 12.1

Siempre que sea posible, evite incluir datos miembro `protected` en una clase base. En vez de ello, incluya funciones miembro no `private` que accedan a los datos miembro `private`, asegurando que el objeto mantenga un estado consistente.

12.4.5 La jerarquía de herencia `EmpleadoPorComision`–

`EmpleadoBaseMasComision` mediante el uso de datos `private`

Ahora vamos a reexaminar nuestra jerarquía una vez más, pero en esta ocasión incluiremos las mejores prácticas de ingeniería de software. La clase `EmpleadoPorComision` (figuras 12.17 y 12.18) ahora declara los datos miembro `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` como `private` (figura 12.17, líneas 33 a 37) y proporciona las funciones miembro `public` `setPrimerNombre`, `getPrimerNombre`, `setApellidoPaterno`, `getApellidoPaterno`, `setNumeroSeguroSocial`, `getNumeroSeguroSocial`, `setVentasBrutas`, `getVentasBrutas`, `setTarifaComision`, `getTarifaComision`, `ingresos` e `imprimir` para manipular estos valores. Si decidimos modificar los nombres de los miembros de datos, las definiciones de `ingresos` e `imprimir` no requerirán modificación; sólo las definiciones de las funciones `get` y `set` que manipulen directamente los datos miembro tendrán que modificarse. Observe que estas modificaciones ocurren sólo dentro de la clase base; no se necesitan modificaciones en la clase derivada. Localizar los efectos de modificaciones como éstas es una buena práctica de ingeniería de software. La clase derivada `EmpleadoBaseMasComision` (figuras 12.19 y 12.20) hereda las funciones miembro no `private` de `EmpleadoPorComision` y puede acceder a los miembros `private` de la clase base a través de esas funciones miembro.

```

1 // Fig. 12.17: EmpleadoPorComision.h
2 // Definición de la clase EmpleadoPorComision con buena ingeniería de software.
3 #ifndef COMISION_H
4 #define COMISION_H
5
6 #include <string> // clase string estándar de C++
7 using std::string;
8
9 class EmpleadoPorComision
10 {
11 public:
12     EmpleadoPorComision( const string &, const string &, const string &,
13                          double = 0.0, double = 0.0 );
14
15     void setPrimerNombre( const string & ); // establece el primer nombre
16     string getPrimerNombre() const; // devuelve el primer nombre
17
18     void setApellidoPaterno( const string & ); // establece el apellido paterno
19     string getApellidoPaterno() const; // devuelve el apellido paterno
20
21     void setNumeroSeguroSocial( const string & ); // establece el NSS
22     string getNumeroSeguroSocial() const; // devuelve el NSS
23
24     void setVentasBrutas( double ); // establece el monto de ventas brutas
25     double getVentasBrutas() const; // devuelve el monto de ventas brutas
26
27     void setTarifaComision( double ); // establece la tarifa de comisión
28     double getTarifaComision() const; // devuelve la tarifa de comisión
29
30     double ingresos() const; // calcula los ingresos
31     void imprimir() const; // imprime el objeto EmpleadoPorComision
32 private:
33     string primerNombre;
34     string apellidoPaterno;
35     string numeroSeguroSocial;
36     double ventasBrutas; // ventas brutas por semana
37     double tarifaComision; // porcentaje de comisión
38 }; // fin de la clase EmpleadoPorComision
39
40 #endif

```

Figura 12.17 | Clase EmpleadoPorComision definida mediante el uso de buenas prácticas de ingeniería de software.

```

1 // Fig. 12.18: EmpleadoPorComision.cpp
2 // Definiciones de las funciones miembro de EmpleadoPorComision.
3 #include <iostream>
4 using std::cout;
5
6 #include "EmpleadoPorComision.h" // definición de la clase EmpleadoPorComision
7
8 // constructor
9 EmpleadoPorComision::EmpleadoPorComision(
10     const string &nombre, const string &apellido, const string &nss,
11     double ventas, double tarifa )
12     : primerNombre( nombre ), apellidoPaterno( apellido ), numeroSeguroSocial( nss )
13 {
14     setVentasBrutas( ventas ); // valida y almacena las ventas brutas
15     setTarifaComision( tarifa ); // valida y almacena la tarifa de comisión
16 } // fin del constructor de EmpleadoPorComision

```

Figura 12.18 | Archivo de implementación de la clase EmpleadoPorComision: la clase EmpleadoPorComision utiliza funciones miembro para manipular sus datos `private`. (Parte 1 de 3).

```
17 // establece el primer nombre
18 void EmpleadoPorComision::setPrimerNombre( const string &nombre )
19 {
20     primerNombre = nombre; // debe validar
21 } // fin de la función setPrimerNombre
22
23 // devuelve el primer nombre
24 string EmpleadoPorComision::getPrimerNombre() const
25 {
26     return primerNombre;
27 } // fin de la función getPrimerNombre
28
29 // establece el apellido paterno
30 void EmpleadoPorComision::setApellidoPaterno( const string &apellido )
31 {
32     apellidoPaterno = apellido; // debe validar
33 } // fin de la función setApellidoPaterno
34
35 // devuelve el apellido paterno
36 string EmpleadoPorComision::getApellidoPaterno() const
37 {
38     return apellidoPaterno;
39 } // fin de la función getApellidoPaterno
40
41 // establece el número de seguro social
42 void EmpleadoPorComision::setNumeroSeguroSocial( const string &nss )
43 {
44     numeroSeguroSocial = nss; // debe validar
45 } // fin de la función setNumeroSeguroSocial
46
47 // devuelve el número de seguro social
48 string EmpleadoPorComision::getNumeroSeguroSocial() const
49 {
50     return numeroSeguroSocial;
51 } // fin de la función getNumeroSeguroSocial
52
53 // establece el monto de ventas brutas
54 void EmpleadoPorComision::setVentasBrutas( double ventas )
55 {
56     ventasBrutas = ( ventas < 0.0 ) ? 0.0 : ventas;
57 } // fin de la función setVentasBrutas
58
59 // devuelve el monto de ventas brutas
60 double EmpleadoPorComision::getVentasBrutas() const
61 {
62     return ventasBrutas;
63 } // fin de la función getVentasBrutas
64
65 // establece la tarifa de comisión
66 void EmpleadoPorComision::setTarifaComision( double tarifa )
67 {
68     tarifaComision = ( tarifa > 0.0 && tarifa < 1.0 ) ? tarifa : 0.0;
69 } // fin de la función setTarifaComision
70
71 // devuelve la tarifa de comisión
72 double EmpleadoPorComision::getTarifaComision() const
73 {
74     return tarifaComision;
75 } // fin de la función getTarifaComision
```

Figura 12.18 | Archivo de implementación de la clase EmpleadoPorComision: la clase EmpleadoPorComision utiliza funciones miembro para manipular sus datos **private**. (Parte 2 de 3).

```

77 // calcula los ingresos
78 double EmpleadoPorComision::ingresos() const
79 {
80     return getTarifaComision() * getVentasBrutas();
81 } // fin de la función ingresos
82
83 // imprime el objeto EmpleadoPorComision
84 void EmpleadoPorComision::imprimir() const
85 {
86     cout << "empleado por comision: "
87     << getPrimerNombre() << ' ' << getApellidoPaterno()
88     << "\nnumero de seguro social: " << getNumeroSeguroSocial()
89     << "\nventas brutas: " << getVentasBrutas()
90     << "\ntarifa de comision: " << getTarifaComision();
91 } // fin de la función imprimir
92

```

Figura 12.18 | Archivo de implementación de la clase `EmpleadoPorComision`: la clase `EmpleadoPorComision` utiliza funciones miembro para manipular sus datos `private`. (Parte 3 de 3).

En la implementación del constructor de `EmpleadoPorComision` (figura 12.18, líneas 9 a 16), observe que usamos inicializadores de miembros (línea 12) para establecer los valores de los miembros `primerNombre`, `apellidoPaterno` y `numeroSeguroSocial`. Vamos a mostrar cómo la clase derivada `EmpleadoBaseMasComision` (figuras 12.19 y 12.20) puede invocar a las funciones miembro no `private` de la clase base (`setPrimerNombre`, `getPrimerNombre`, `setApellidoPaterno`, `getApellidoPaterno`, `setNumeroSeguroSocial` y `getNumeroSeguroSocial`) para manipular estos miembros de datos.



Tip de rendimiento 12.2

Usar una función miembro para acceder al valor de un miembro de datos puede ser un poco más lento que acceder a los datos directamente. Sin embargo, los compiladores optimizadores de hoy en día están diseñados cuidadosamente para realizar muchas optimizaciones de manera implícita (como poner en línea las llamadas a funciones set y get). Como resultado, los programadores deben escribir código que se adhiera a los principios de la ingeniería de software apropiada, y dejar las cuestiones de optimización al compilador. Una buena regla es, “No dudar del compilador”.

La clase `EmpleadoBaseMasComision` (figuras 12.19 y 12.20) tiene varias modificaciones en las implementaciones de sus funciones miembro (figura 12.20) que la diferencian de la versión anterior (figuras 12.14 y 12.15). Las funciones miembro `ingresos` (figura 12.20, líneas 32 a 35) e `imprimir` (líneas 38 a 46) invocan a la función miembro `getSalarioBase` para obtener el valor del salario base, en vez de acceder directamente a `salarioBase`. Esto aísla a `ingresos` e `imprimir` de las potenciales modificaciones a la implementación del miembro de datos `salarioBase`. Por ejemplo, si decidimos renombrar el miembro de datos `salarioBase` o modificar su tipo, sólo las funciones miembro `setSalarioBase` y `getSalarioBase` tendrán que modificarse.

```

1 // Fig. 12.19: EmpleadoBaseMasComision.h
2 // Clase EmpleadoBaseMasComision derivada de la clase
3 // EmpleadoPorComision.
4 #ifndef BASEMAS_H
5 #define BASEMAS_H
6
7 #include <string> // clase string estándar de C++
8 using std::string;
9
10 #include "EmpleadoPorComision.h" // declaración de la clase EmpleadoPorComision
11
12 class EmpleadoBaseMasComision : public EmpleadoPorComision
13 {

```

Figura 12.19 | Archivo de encabezado de la clase `EmpleadoBaseMasComision`. (Parte 1 de 2).

```

14 public:
15     EmpleadoBaseMasComision( const string &, const string &,
16         const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18     void setSalarioBase( double ); // establece el salario base
19     double getSalarioBase() const; // devuelve el salario base
20
21     double ingresos() const; // calcula los ingresos
22     void imprimir() const; // imprime el objeto EmpleadoBaseMasComision
23 private:
24     double salarioBase; // salario base
25 }; // fin de la clase EmpleadoBaseMasComision
26
27 #endif

```

Figura 12.19 | Archivo de encabezado de la clase EmpleadoBaseMasComision. (Parte 2 de 2).

```

1 // Fig. 12.20: EmpleadoBaseMasComision.cpp
2 // Definiciones de las funciones miembro de la clase EmpleadoBaseMasComision.
3 #include <iostream>
4 using std::cout;
5
6 // definición de la clase EmpleadoBaseMasComision
7 #include "EmpleadoBaseMasComision.h"
8
9 // constructor
10 EmpleadoBaseMasComision::EmpleadoBaseMasComision(
11     const string &nombre, const string &apellido, const string &nss,
12     double ventas, double tarifa, double salario )
13     // llama explícitamente al constructor de la clase base
14     : EmpleadoPorComision( nombre, apellido, nss, ventas, tarifa )
15 {
16     setSalarioBase( salario ); // valida y almacena el salario base
17 } // fin del constructor de EmpleadoBaseMasComision
18
19 // establece el salario base
20 void EmpleadoBaseMasComision::setSalarioBase( double salario )
21 {
22     salarioBase = ( salario < 0.0 ) ? 0.0 : salario;
23 } // fin de la función setSalarioBase
24
25 // devuelve el salario base
26 double EmpleadoBaseMasComision::getSalarioBase() const
27 {
28     return salarioBase;
29 } // fin de la función getSalarioBase
30
31 // calcula los ingresos
32 double EmpleadoBaseMasComision::ingresos() const
33 {
34     return getSalarioBase() + EmpleadoPorComision::ingresos();
35 } // fin de la función ingresos
36
37 // imprime el objeto EmpleadoBaseMasComision
38 void EmpleadoBaseMasComision::imprimir() const
39 {
40     cout << "con salario base ";
41

```

Figura 12.20 | Clase EmpleadoBaseMasComision que hereda de la clase EmpleadoPorComision pero no puede acceder directamente a los datos **private** de la clase. (Parte 1 de 2).

```

42 // invoca a la función imprimir de EmpleadoPorComision
43 EmpleadoPorComision::imprimir();
44
45 cout << "\nsalario base: " << getSalarioBase();
46 } // fin de la función imprimir

```

Figura 12.20 | Clase `EmpleadoBaseMasComision` que hereda de la clase `EmpleadoPorComision` pero no puede acceder directamente a los datos `private` de la clase. (Parte 2 de 2).

La función `ingresos` de la clase `EmpleadoBaseMasComision` (figura 12.20, líneas 32 a 35) redefine la función miembro `ingresos` de la clase `EmpleadoPorComision` (figura 12.18, líneas 79 a 82) para calcular los ingresos de un empleado por comisión con salario base. La versión de `ingresos` de la clase `EmpleadoBaseMasComision` obtiene la porción de los ingresos del empleado únicamente con base en la comisión, llamando a la función `ingresos` de la clase base `EmpleadoPorComision` con la expresión `EmpleadoPorComision::ingresos()` (figura 12.20, línea 34). Después, la función `ingresos` de `EmpleadoBaseMasComision` suma el salario base a este valor para calcular los ingresos totales del empleado. Observe la sintaxis utilizada para invocar a una función miembro de la clase base redefinida desde una clase derivada; se coloca el nombre de la clase base y el operador de resolución de ámbito binario (`::`) antes del nombre de la función miembro de la clase base. Esta invocación a la función miembro es una buena práctica de ingeniería de software: en la *Observación de Ingeniería de Software 9.9* vimos que, si la función miembro de un objeto realiza las acciones que necesita otro objeto, debemos llamar a esa función miembro en vez de duplicar su cuerpo de código. Al hacer que la función `ingresos` de `EmpleadoBaseMasComision` invoque a la función `ingresos` de `EmpleadoPorComision` para calcular parte de los ingresos de un objeto `EmpleadoBaseMasComision`, evitamos duplicar el código y reducimos los problemas de mantenimiento de código.

Error común de programación 12.2



Cuando se redefine una función miembro de la clase base en una clase derivada, a menudo la versión de la clase derivada llama a la versión de la clase base para realizar trabajo adicional. Si no se utiliza el nombre de la clase base y el operador `::` al hacer referencia a la función miembro de la clase base, se produce una recursividad infinita, ya que entonces la función miembro de la clase derivada se llamaría a sí misma.

Error común de programación 12.3



Al incluir una función miembro de una clase base con una firma distinta en la clase derivada, se oculta la versión de la función correspondiente a la clase base. Los intentos de llamar a la versión de la clase base a través de la interfaz `public` de un objeto de la clase derivada producen errores de compilación.

De manera similar, la función `imprimir` de `EmpleadoBaseMasComision` (figura 12.20, líneas 38 a 46) redefine la función miembro `imprimir` de `EmpleadoPorComision` (figura 12.18, líneas 85 a 92) para imprimir información apropiada para un empleado por comisión con salario base. La versión de la clase `EmpleadoBaseMasComision` muestra parte de la información de un objeto `EmpleadoBaseMasComision` (es decir, la cadena "empleado por comision" y los valores de los datos miembro `private` de la clase `EmpleadoPorComision`) llamando a la función miembro `imprimir` de `EmpleadoPorComision` con el nombre calificado `EmpleadoPorComision::imprimir()` (figura 12.20, línea 43). Después, la función `imprimir` de `EmpleadoBaseMasComision` imprime el resto de la información de un objeto `EmpleadoBaseMasComision` (es decir, el valor del salario base de la clase `EmpleadoBaseMasComision`).

La figura 12.21 realiza las mismas manipulaciones en un objeto `EmpleadoBaseMasComision` que se hicieron en las figuras 12.9 y 12.16 con objetos de las clases `EmpleadoPorComision` y `EmpleadoBaseMasComision`, respectivamente. Aunque cada clase "empleado por comisión con salario base" se comporta en forma idéntica, la clase `EmpleadoBaseMasComision` es la mejor diseñada. Al usar la herencia y llamar a las funciones miembro que ocultan los datos y aseguran la consistencia, hemos construido en forma eficiente y efectiva una clase bien diseñada.

```

1 // Fig. 12.21: fig12_21.cpp
2 // Prueba de la clase EmpleadoBaseMasComision.
3 #include <iostream>

```

Figura 12.21 | Los datos `private` de la clase base pueden ser utilizados por una clase derivada a través de una función miembro `public` o `protected` heredada por la clase derivada. (Parte 1 de 2).

```
4  using std::cout;
5  using std::endl;
6  using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 // definición de la clase EmpleadoBaseMasComision
12 #include "EmpleadoBaseMasComision.h"
13
14 int main()
15 {
16     // instancia un objeto EmpleadoBaseMasComision
17     EmpleadoBaseMasComision
18         empleado( "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
19
20     // establece el formato de salida de punto flotante
21     cout << fixed << setprecision( 2 );
22
23     // obtiene los datos del empleado por comisión
24     cout << "Informacion del empleado obtenida por las funciones get: \n"
25         << "\nEl primer nombre es " << empleado.getPrimerNombre()
26         << "\nEl apellido paterno es " << empleado.getApellidoPaterno()
27         << "\nEl numero de seguro social es "
28         << empleado.getNumeroSeguroSocial()
29         << "\nLas ventas brutas son " << empleado.getVentasBrutas()
30         << "\nLa tarifa de comision es " << empleado.getTarifaComision()
31         << "\nEl salario base es " << empleado.getSalarioBase() << endl;
32
33     empleado.setSalarioBase( 1000 ); // establece el salario base
34
35     cout << "\nInformacion actualizada del empleado, impresa por la funcion imprimir: \n"
36         << endl;
37     empleado.imprimir(); // muestra la nueva información del empleado
38
39     // muestra los ingresos del empleado
40     cout << "\n\nIngresos del empleado: $" << empleado.ingresos() << endl;
41
42     return 0;
43 } // end main
```

Informacion del empleado obtenida por las funciones get:

El primer nombre es Bob
El apellido paterno es Lewis
El numero de seguro social es 333-33-3333
Las ventas brutas son 5000.00
La tarifa de comision es 0.04
El salario base es 300.00

Informacion actualizada del empleado, impresa por la funcion imprimir:

con salario base empleado por comision: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa de comision: 0.04
salario base: 1000.00

Ingresos del empleado: \$1200.00

Figura 12.21 | Los datos **private** de la clase base pueden ser utilizados por una clase derivada a través de una función miembro **public** o **protected** heredada por la clase derivada. (Parte 2 de 2).

En esta sección vimos un conjunto evolutivo de ejemplos que se diseñó cuidadosamente para enseñar las herramientas clave para la buena ingeniería de software mediante la herencia. El lector aprendió a crear una clase derivada mediante el uso de la herencia, a utilizar datos miembro `protected` para permitir que una clase derivada acceda a los datos miembro heredados de la clase base, y cómo redefinir las funciones de la clase base para proporcionar versiones que sean más apropiadas para los objetos de la clase derivada. Además, aprendió a aplicar las técnicas de ingeniería de software de los capítulos 9 y 10, junto con las de este capítulo, para crear clases que sean fáciles de mantener, modificar y depurar.

12.5 Los constructores y destructores en las clases derivadas

Como explicamos en la sección anterior, al instanciar un objeto de una clase derivada se inicia una cadena de llamadas a constructores, en donde el constructor de la clase derivada, antes de realizar sus propias tareas, invoca al constructor de su clase base directa, ya sea de manera explícita (a través de un inicializador de miembros de la clase base) o implícita (llamando al constructor predeterminado de la clase base). De manera similar, si la clase base se deriva de otra clase, se requiere el constructor de la clase base para invocar al constructor de la siguiente clase hacia arriba de la jerarquía, y así en lo sucesivo. El último constructor llamado en esta cadena es el constructor de la clase en la base de la jerarquía, cuyo cuerpo en realidad termina primero de ejecutarse. El cuerpo del constructor de la clase derivada original termina de ejecutarse al último. El constructor de cada clase base inicializa los datos miembro de la clase base que hereda el objeto de la clase derivada. Por ejemplo, considere la jerarquía `EmpleadoPorComision/EmpleadoBaseMasComision` de las figuras 12.17 a 12.20. Cuando un programa crea un objeto de la clase `EmpleadoBaseMasComision`, se hace una llamada al constructor de `EmpleadoPorComision`. Como la clase `EmpleadoPorComision` se encuentra en la base de la jerarquía, se ejecuta su constructor y se inicializan los datos miembro `private` de `EmpleadoPorComision` que forman parte del objeto `EmpleadoBaseMasComision`. Cuando el constructor de `EmpleadoPorComision` completa su ejecución, devuelve el control al constructor de `EmpleadoBaseMasComision`, el cual inicializa al objeto `salarioBase` de `EmpleadoBaseMasComision`.



Observación de Ingeniería de Software 12.7

Cuando un programa crea un objeto de una clase derivada, el constructor de la clase derivada llama de inmediato al constructor de la clase base, se ejecuta el cuerpo del constructor de la clase base y después se ejecutan los inicializadores de miembros de la clase derivada; al último, se ejecuta el cuerpo del constructor de la clase derivada. Este proceso avanza en cascada hacia arriba por la jerarquía, si contiene más de dos niveles.

Cuando se destruye un objeto de una clase derivada, el programa llama al destructor de ese objeto. Esto empieza una cadena (o cascada) de llamadas a destructores en las que el destructor de la clase derivada y los destructores de las clases base directas e indirectas y los miembros de las clases se ejecutan en orden inverso al orden en el que se ejecutaron los constructores. Cuando se hace una llamada al destructor de un objeto de una clase derivada, el destructor realiza su tarea, y después invoca al destructor de la siguiente clase base hacia arriba por la jerarquía. Este proceso se repite hasta que se hace una llamada al destructor de la clase base final en la parte superior de la jerarquía. Después, el objeto se elimina de la memoria.



Observación de Ingeniería de Software 12.8

Supongamos que creamos un objeto de una clase derivada, en donde tanto la clase base como la clase derivada contienen (a través de la composición) objetos de otras clases. Cuando se crea un objeto de esa clase derivada, primero se ejecutan los constructores para los objetos miembro de la clase base, después se ejecuta el constructor de la clase base, luego se ejecutan los objetos miembro de la clase derivada, y después se ejecuta el constructor de la clase derivada. Los destructores para los objetos de una clase derivada se llaman en orden inverso al orden en el que se llamaron sus correspondientes constructores.

Las clases derivadas no heredan los constructores, destructores ni operadores de asignación sobrecargados (vea el capítulo 11, Sobre carga de operadores: objetos String y Array) de la clase base. Sin embargo, los constructores, destructores y operadores de asignación sobrecargados de la clase derivada pueden llamar a los constructores, destructores y operadores de asignación sobrecargados de la clase base.

Nuestro siguiente ejemplo define la clase `EmpleadoPorComision` (figuras 12.22 y 12.23) y la clase `EmpleadoBaseMasComision` (figuras 12.24 y 12.25) con constructores y destructores, cada uno de los cuales imprime un mensaje al ser invocado. Como veremos en los resultados de la figura 12.26, estos mensajes demuestran el orden en el que se llaman los constructores y destructores para los objetos en una jerarquía de herencia.

```

1 // Fig. 12.22: EmpleadoPorComision.h
2 // Definición de la clase EmpleadoPorComision que representa a un empleado por comisión.
3 #ifndef COMISION_H
4 #define COMISION_H
5
6 #include <string> // clase string estándar de C++
7 using std::string;
8
9 class EmpleadoPorComision
10 {
11 public:
12     EmpleadoPorComision( const string &, const string &, const string &,
13         double = 0.0, double = 0.0 );
14     ~EmpleadoPorComision(); // destructor
15
16     void setPrimerNombre( const string & ); // establece el primer nombre
17     string getPrimerNombre() const; // devuelve el primer nombre
18
19     void setApellidoPaterno( const string & ); // establece el apellido paterno
20     string getApellidoPaterno() const; // devuelve el apellido paterno
21
22     void setNumeroSeguroSocial( const string & ); // establece el NSS
23     string getNumeroSeguroSocial() const; // devuelve el NSS
24
25     void setVentasBrutas( double ); // establece el monto de ventas brutas
26     double getVentasBrutas() const; // devuelve el monto de ventas brutas
27
28     void setTarifaComision( double ); // establece la tarifa de comisión
29     double getTarifaComision() const; // devuelve la tarifa de comisión
30
31     double ingresos() const; // calcula los ingresos
32     void imprimir() const; // imprime el objeto EmpleadoPorComision
33 private:
34     string primerNombre;
35     string apellidoPaterno;
36     string numeroSeguroSocial;
37     double ventasBrutas; // ventas brutas por semana
38     double tarifaComision; // porcentaje de comisión
39 }; // fin de la clase EmpleadoPorComision
40
41 #endif

```

Figura 12.22 | Archivo de encabezado de `EmpleadoPorComision`.

En este ejemplo, modificamos el constructor de `EmpleadoPorComision` (líneas 10 a 21 de la figura 12.23) e incluimos un destructor de `EmpleadoPorComision` (líneas 24 a 29), cada uno de los cuales imprime una línea de texto al momento de su invocación. También modificamos el constructor de `EmpleadoBaseMasComision` (líneas 11 a 22 de la figura 12.25) e incluimos un destructor de `EmpleadoBaseMasComision` (líneas 25 a 30), cada uno de los cuales imprime una línea de texto al momento de su invocación.

```

1 // Fig. 12.23: EmpleadoPorComision.cpp
2 // Definiciones de las funciones miembro de EmpleadoPorComision.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "EmpleadoPorComision.h" // definición de la clase EmpleadoPorComision
8
9 // constructor
10 EmpleadoPorComision::EmpleadoPorComision(

```

Figura 12.23 | El constructor de `EmpleadoPorComision` imprime texto. (Parte I de 3).

```

11     const string &nombre, const string &apellido, const string &nss,
12     double ventas, double tarifa )
13 : primerNombre( nombre ), apellidoPaterno( apellido ), numeroSeguroSocial( nss )
14 {
15     setVentasBrutas( ventas ); // valida y almacena las ventas brutas
16     setTarifaComision( tarifa ); // valida y almacena la tarifa de comisión
17
18     cout << "Constructor de EmpleadoPorComision: " << endl;
19     imprimir();
20     cout << "\n\n";
21 } // fin del constructor de EmpleadoPorComision
22
23 // destructor
24 EmpleadoPorComision::~EmpleadoPorComision()
25 {
26     cout << "Destructor de EmpleadoPorComision: " << endl;
27     imprimir();
28     cout << "\n\n";
29 } // fin del destructor de EmpleadoPorComision
30
31 // establece el primer nombre
32 void EmpleadoPorComision::setPrimerNombre( const string &nombre )
33 {
34     primerNombre = nombre; // debe validar
35 } // fin de la función setPrimerNombre
36
37 // devuelve el primer nombre
38 string EmpleadoPorComision::getPrimerNombre() const
39 {
40     return primerNombre;
41 } // fin de la función getPrimerNombre
42
43 // establece el apellido paterno
44 void EmpleadoPorComision::setApellidoPaterno( const string &apellido )
45 {
46     apellidoPaterno = apellido; // debería validar
47 } // fin de la función setApellidoPaterno
48
49 // devuelve el apellido paterno
50 string EmpleadoPorComision::getApellidoPaterno() const
51 {
52     return apellidoPaterno;
53 } // fin de la función getApellidoPaterno
54
55 // establece el número de seguro social
56 void EmpleadoPorComision::setNumeroSeguroSocial( const string &nss )
57 {
58     numeroSeguroSocial = nss; // debería validar
59 } // fin de la función setNumeroSeguroSocial
60
61 // devuelve el número de seguro social
62 string EmpleadoPorComision::getNumeroSeguroSocial() const
63 {
64     return numeroSeguroSocial;
65 } // fin de la función getNumeroSeguroSocial
66
67 // establece el monto de ventas brutas
68 void EmpleadoPorComision::setVentasBrutas( double ventas )
69 {
70     ventasBrutas = ( ventas < 0.0 ) ? 0.0 : ventas;
71 } // fin de la función setVentasBrutas
72

```

Figura 12.23 | El constructor de EmpleadoPorComision imprime texto. (Parte 2 de 3).

```

73 // devuelve el monto de ventas brutas
74 double EmpleadoPorComision::getVentasBrutas() const
75 {
76     return ventasBrutas;
77 } // fin de la función getVentasBrutas
78
79 // establece la tarifa de comisión
80 void EmpleadoPorComision::setTarifaComision( double tarifa )
81 {
82     tarifaComision = ( tarifa > 0.0 && tarifa < 1.0 ) ? tarifa : 0.0;
83 } // fin de la función setTarifaComision
84
85 // devuelve la tarifa de comisión
86 double EmpleadoPorComision::getTarifaComision() const
87 {
88     return tarifaComision;
89 } // fin de la función getTarifaComision
90
91 // calcula los ingresos
92 double EmpleadoPorComision::ingresos() const
93 {
94     return getTarifaComision() * getVentasBrutas();
95 } // fin de la función ingresos
96
97 // imprime el objeto EmpleadoPorComision
98 void EmpleadoPorComision::imprimir() const
99 {
100    cout << "empleado por comision: "
101        << getPrimerNombre() << ' ' << getApellidoPaterno()
102        << "\nnumero de seguro social: " << getNumeroSeguroSocial()
103        << "\nventas brutas: " << getVentasBrutas()
104        << "\ntarifa de comision: " << getTarifaComision();
105 } // fin de la función imprimir

```

Figura 12.23 | El constructor de `EmpleadoPorComision` imprime texto. (Parte 3 de 3).

```

1 // Fig. 12.24: EmpleadoBaseMasComision.h
2 // Clase EmpleadoBaseMasComision derivada de la clase
3 // EmpleadoPorComision.
4 #ifndef BASEMAS_H
5 #define BASEMAS_H
6
7 #include <string> // clase string estándar de C++
8 using std::string;
9
10 #include "EmpleadoPorComision.h" // declaración de la clase EmpleadoPorComision
11
12 class EmpleadoBaseMasComision : public EmpleadoPorComision
13 {
14 public:
15     EmpleadoBaseMasComision( const string &, const string &,
16                             const string &, double = 0.0, double = 0.0, double = 0.0 );
17     ~EmpleadoBaseMasComision(); // destructor
18
19     void setSalarioBase( double ); // establece el salario base
20     double getSalarioBase() const; // devuelve el salario base
21
22     double ingresos() const; // calcula los ingresos
23     void imprimir() const; // imprime el objeto EmpleadoBaseMasComision
24 private:

```

Figura 12.24 | Archivo de encabezado de la clase `EmpleadoBaseMasComision`. (Parte 1 de 2).

```

25     double salarioBase; // salario base
26 } // fin de la clase EmpleadoBaseMasComision
27
28 #endif

```

Figura 12.24 | Archivo de encabezado de la clase EmpleadoBaseMasComision. (Parte 2 de 2).

```

1 // Fig. 12.25: EmpleadoBaseMasComision.cpp
2 // Definiciones de las funciones miembro de la clase EmpleadoBaseMasComision.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // definición de la clase EmpleadoBaseMasComision
8 #include "EmpleadoBaseMasComision.h"
9
10 // constructor
11 EmpleadoBaseMasComision::EmpleadoBaseMasComision(
12     const string &nombre, const string &apellido, const string &nss,
13     double ventas, double tarifa, double salario )
14     // llama explícitamente al constructor de la clase base
15     : EmpleadoPorComision( nombre, apellido, nss, ventas, tarifa )
16 {
17     setSalarioBase( salario ); // valida y almacena el salario base
18
19     cout << "Constructor de EmpleadoBaseMasComision: " << endl;
20     imprimir();
21     cout << "\n\n";
22 } // fin del constructor de EmpleadoBaseMasComision
23
24 // destructor
25 EmpleadoBaseMasComision::~EmpleadoBaseMasComision()
26 {
27     cout << "Destructor de EmpleadoBaseMasComision: " << endl;
28     imprimir();
29     cout << "\n\n";
30 } // fin del constructor de EmpleadoBaseMasComision
31
32 // establece el salario base
33 void EmpleadoBaseMasComision::setSalarioBase( double salario )
34 {
35     salarioBase = ( salario < 0.0 ) ? 0.0 : salario;
36 } // fin de la función setSalarioBase
37
38 // devuelve el salario base
39 double EmpleadoBaseMasComision::getSalarioBase() const
40 {
41     return salarioBase;
42 } // fin de la función getSalarioBase
43
44 // calcula los ingresos
45 double EmpleadoBaseMasComision::ingresos() const
46 {
47     return getSalarioBase() + EmpleadoPorComision::ingresos();
48 } // fin de la función ingresos
49
50 // imprime el objeto EmpleadoBaseMasComision
51 void EmpleadoBaseMasComision::imprimir() const
52 {
53     cout << "con salario base ";
54

```

Figura 12.25 | El constructor de EmpleadoBaseMasComision imprime texto. (Parte 1 de 2).

```

55     // invoca a la función imprimir de EmpleadoPorComision
56     EmpleadoPorComision::imprimir();
57
58     cout << "\nsalario base: " << getSalarioBase();
59 } // fin de la función imprimir

```

Figura 12.25 | El constructor de `EmpleadoBaseMasComision` imprime texto. (Parte 2 de 2).

La figura 12.26 demuestra el orden en el que se llaman los constructores y destructores para objetos de clases que forman parte de una jerarquía de herencia. La función `main` (líneas 15 a 34) empieza por instanciar el objeto `EmpleadoPorComision` llamado `empleado1` (líneas 21 y 22) en un bloque separado dentro de `main` (líneas 20 a 23). El objeto entra y sale de alcance inmediatamente (se llega al final del bloque justo después de crear el objeto), por lo que se llama tanto al constructor como al destructor de `EmpleadoPorComision`. A continuación, en las líneas 26 y 27 se instancia el objeto `empleado2` de la clase `EmpleadoBaseMasComision`. Esto invoca al constructor de `EmpleadoBaseMasComision` para mostrar los resultados con los valores que se pasan del constructor de `EmpleadoBaseMasComision`, y después se imprimen los resultados especificados en el constructor de `EmpleadoBaseMasComision`. Luego, en las líneas 30 y 31 se instancia el objeto `empleado3` de `EmpleadoBaseMasComision`. De nuevo, se hacen llamadas a los constructores de `EmpleadoPorComision` y `EmpleadoBaseMasComision`. Observe que, en cada caso, el cuerpo del constructor de `EmpleadoPorComision` se ejecuta antes del cuerpo del constructor de `EmpleadoBaseMasComision`. Cuando se llega al final de `main`, se hacen llamadas a los destructores para los objetos `empleado2` y `empleado3`. Pero, debido a que los destructores se llaman en el orden inverso al de sus correspondientes constructores, se hacen llamadas al destructor de `EmpleadoBaseMasComision` y al destructor de `EmpleadoPorComision` (en ese orden) para el objeto `empleado3`, y después se hacen llamadas a los destructores de `EmpleadoBaseMasComision` y `EmpleadoPorComision` (en ese orden) para el objeto `empleado2`.

```

1 // Fig. 12.26: fig12_26.cpp
2 // Muestra el orden en el que se llama a los constructores y destructores de
3 // la clase base y la clase derivada.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7 using std::fixed;
8
9 #include <iomanip>
10 using std::setprecision;
11
12 // definición de la clase EmpleadoBaseMasComision
13 #include "EmpleadoBaseMasComision.h"
14
15 int main()
16 {
17     // establece el formato de salida de punto flotante
18     cout << fixed << setprecision( 2 );
19
20     { // empieza nuevo alcance
21         EmpleadoPorComision empleado1(
22             "Bob", "Lewis", "333-33-3333", 5000, .04 );
23     } // fin del alcance
24
25     cout << endl;
26     EmpleadoBaseMasComision
27         empleado2( "Lisa", "Jones", "555-55-5555", 2000, .06, 800 );
28
29     cout << endl;
30     EmpleadoBaseMasComision
31         empleado3( "Mark", "Sands", "888-88-8888", 8000, .15, 2000 );
32     cout << endl;
33     return 0;
34 } // fin de main

```

Figura 12.26 | Orden de llamadas a los constructores y destructores. (Parte 1 de 2).

```
Constructor de EmpleadoPorComision:  
empleado por comision: Bob Lewis  
numero de seguro social: 333-33-3333  
ventas brutas: 5000.00  
tarifa de comision: 0.04  
  
Destructor de EmpleadoPorComision:  
empleado por comision: Bob Lewis  
numero de seguro social: 333-33-3333  
ventas brutas: 5000.00  
tarifa de comision: 0.04  
  
Constructor de EmpleadoPorComision:  
empleado por comision: Lisa Jones  
numero de seguro social: 555-55-5555  
ventas brutas: 2000.00  
tarifa de comision: 0.06  
  
Constructor de EmpleadoBaseMasComision:  
con salario base empleado por comision: Lisa Jones  
numero de seguro social: 555-55-5555  
ventas brutas: 2000.00  
tarifa de comision: 0.06  
salario base: 800.00  
  
Constructor de EmpleadoPorComision:  
empleado por comision: Mark Sands  
numero de seguro social: 888-88-8888  
ventas brutas: 8000.00  
tarifa de comision: 0.15  
  
Constructor de EmpleadoBaseMasComision:  
con salario base empleado por comision: Mark Sands  
numero de seguro social: 888-88-8888  
ventas brutas: 8000.00  
tarifa de comision: 0.15  
salario base: 2000.00  
  
Destructor de EmpleadoBaseMasComision:  
con salario base empleado por comision: Mark Sands  
numero de seguro social: 888-88-8888  
ventas brutas: 8000.00  
tarifa de comision: 0.15  
salario base: 2000.00  
  
Destructor de EmpleadoPorComision:  
empleado por comision: Mark Sands  
numero de seguro social: 888-88-8888  
ventas brutas: 8000.00  
tarifa de comision: 0.15  
  
Destructor de EmpleadoBaseMasComision:  
con salario base empleado por comision: Lisa Jones  
numero de seguro social: 555-55-5555  
ventas brutas: 2000.00  
tarifa de comision: 0.06  
salario base: 800.00  
  
Destructor de EmpleadoPorComision:  
empleado por comision: Lisa Jones  
numero de seguro social: 555-55-5555  
ventas brutas: 2000.00  
tarifa de comision: 0.06
```

Figura 12.26 | Orden de llamadas a los constructores y destructores. (Parte 2 de 2).

12.6 Herencia `public`, `protected` y `private`

Al derivar una clase de una clase base, ésta se puede heredar a través de la herencia `public`, `protected` o `private`. El uso de las herencias `protected` y `private` es poco común, además de que se deben utilizar con extremo cuidado; por lo general usaremos herencia `public` en este libro. (En el capítulo 20 se demuestra el uso de la herencia `private` como una alternativa para la composición). La figura 12.27 sintetiza, para cada tipo de herencia, la accesibilidad de los miembros de la clase base en una clase derivada. La primera columna contiene los especificadores de acceso de la clase base.

Al derivar una clase de una clase base `public`, los miembros `public` de la clase base se convierten en miembros `public` de la clase derivada, y los miembros `protected` de la clase base se convierten en miembros `protected` de la clase derivada. Los miembros `private` de la clase base nunca pueden utilizarse directamente desde una clase derivada, pero se puede acceder a ellos a través de llamadas a los miembros `public` y `protected` de la clase base.

Al derivar de una clase base `protected`, los miembros `public` y `protected` de la clase base se convierten en miembros `protected` de la clase derivada. Al derivar de una clase base `private`, los miembros `public` y `protected` de la clase base se convierten en miembros `private` (por ejemplo, las funciones se convierten en funciones utilitarias) de la clase derivada. Las relaciones de herencia `private` y `protected` no son relaciones del tipo “*es un*”.

12.7 Ingeniería de software mediante la herencia

En esta sección hablaremos sobre el uso de la herencia para personalizar el software existente. Al utilizar la herencia para crear una nueva clase a partir de una existente, la nueva clase hereda los datos miembro y las funciones miembro de la clase existente, como se describe en la figura 12.27. Podemos personalizar la nueva clase para satisfacer nuestras necesidades, para lo cual incluimos miembros adicionales y redefinimos los miembros de la clase base. El programador de la clase derivada hace esto en C++ sin acceder al código fuente de la clase base. La clase derivada también debe poder enlazarse con el código objeto de la clase base. Esta poderosa capacidad es atractiva para los distribuidores de software independientes (ISVs). Los ISVs pueden desarrollar clases propietarias para vender o licenciar, y ponen estas clases a disposición de los usuarios en formato de código objeto. Así, los usuarios pueden衍生 nuevas clases con rapidez y sin acceder al código fuente propietario de los ISVs. Todo lo que los ISVs necesitan suministrar con el código objeto son los archivos de encabezado.

Especificador de acceso a miembros de la clase base	Tipo de herencia		
	herencia <code>public</code>	herencia <code>protected</code>	herencia <code>private</code>
<code>public</code>	<code>public</code> en la clase derivada. Puede ser utilizado directamente por las funciones miembro, las funciones <code>friend</code> y las funciones no miembro.	<code>protected</code> en la clase derivada. Puede ser utilizado directamente por las funciones miembro y las funciones <code>friend</code> .	<code>private</code> en la clase derivada. Puede ser utilizado directamente por las funciones miembro y las funciones <code>friend</code> .
<code>protected</code>	<code>protected</code> en la clase derivada. Puede ser utilizado directamente por las funciones miembro y las funciones <code>friend</code> .	<code>protected</code> en la clase derivada. Puede ser utilizado directamente por las funciones miembro y las funciones <code>friend</code> .	<code>private</code> en la clase derivada. Puede ser utilizado directamente por las funciones miembro y las funciones <code>friend</code> .
<code>private</code>	Oculto en la clase derivada. Puede ser utilizado por las funciones miembro y las funciones <code>friend</code> a través de las funciones miembro <code>public</code> o <code>protected</code> de la clase base.	Oculto en la clase derivada. Puede ser utilizado por las funciones miembro y las funciones <code>friend</code> a través de funciones miembro <code>public</code> o <code>protected</code> de la clase base.	Oculto en la clase derivada. Puede ser utilizado por funciones miembro y funciones <code>friend</code> a través de funciones miembro <code>public</code> o <code>protected</code> de la clase base.

Figura 12.27 | Resumen de la accesibilidad de los miembros de una clase base en una clase derivada.

Algunas veces es difícil para los estudiantes apreciar el alcance de los problemas a los que se enfrentan los diseñadores que trabajan en proyectos de software de gran escala en la industria. La gente experimentada con dichos proyectos dice que la reutilización efectiva de software mejora el proceso de desarrollo de software. La programación orientada a objetos facilita la reutilización de software, con lo cual se reducen los tiempos de desarrollo y se mejora la calidad del software.

La disponibilidad de bibliotecas de clases extensas y útiles produce los máximos beneficios de la reutilización de software a través de la herencia. Así como el software “empaquetado” que producen los distribuidores de software independientes se convirtió en una industria con crecimiento explosivo al llegar la computadora personal, el interés en la creación y venta de las bibliotecas de clases está creciendo en forma exponencial. Los diseñadores de aplicaciones crean sus aplicaciones con estas bibliotecas, y los diseñadores de bibliotecas obtienen su recompensa cuando sus bibliotecas se incluyen con estas aplicaciones. Las bibliotecas estándar de C++ que se incluyen con los compiladores de C++ tienden a ser de propósito general y tienen su alcance limitado. Sin embargo, hay un compromiso masivo a nivel mundial en relación con el desarrollo de las bibliotecas de clases para una amplia variedad de áreas de aplicaciones.



Observación de Ingeniería de Software 12.9

En la etapa de diseño de un sistema orientado a objetos, a menudo el diseñador determina que ciertas clases están estrechamente relacionadas. El diseñador debe “factorizar” los atributos y comportamientos comunes y colocarlos en una clase base, y después usar la herencia para formar clases derivadas, otorgándoles capacidades más allá de las heredadas de la clase base.



Observación de Ingeniería de Software 12.10

La creación de una clase derivada no afecta al código fuente de su clase base. La herencia preserva la integridad de una clase base.



Observación de Ingeniería de Software 12.11

Así como los diseñadores de sistemas no orientados a objetos deben evitar la proliferación de funciones, los diseñadores de sistemas orientados a objetos deben evitar la proliferación de clases. Esto crea problemas administrativos y puede entorpecer la reutilización del software, debido a que al cliente se le dificulta localizar la clase más apropiada de una biblioteca de clases enorme. La alternativa es crear menos clases que proporcionen una funcionalidad más sustancial, pero dichas clases podrían proporcionar demasiada funcionalidad.



Tip de rendimiento 12.3

Si las clases producidas a través de la herencia son más grandes de lo necesario (es decir, si contienen demasiada funcionalidad), se podrían desperdiciar los recursos de procesamiento y la memoria. Hay que heredar de la clase cuya funcionalidad sea lo más “aproximado” a lo necesario.

Puede ser confuso leer las definiciones de clases derivadas, ya que los miembros heredados no se muestran físicamente en las clases derivadas, pero están presentes. Existe un problema similar al documentar los miembros de las clases derivadas.

12.8 Repaso

En este capítulo se introdujo la herencia: la habilidad de crear una clase al absorber los datos miembro y las funciones miembro de una clase, y adorlarlos con nuevas capacidades. A través de una serie de ejemplos mediante el uso de una jerarquía de herencia, el lector aprendió las nociones de las clases base y las clases derivadas, y utilizó la herencia `public` para crear una clase derivada que hereda miembros de una clase base. El capítulo introdujo el especificador de acceso `protected`; las funciones miembro de la clase derivada pueden acceder a los miembros `protected` de la clase base. El lector aprendió a acceder a los miembros redefinidos de la clase base, calificando sus nombres con el nombre de la clase base y el operador de resolución de ámbito binario (`::`). También vio el orden en el que se llaman los constructores y destructores para los objetos de clases que forman parte de una jerarquía de herencia. Por último, explicamos los tres tipos de herencia (`public`, `protected` y `private`) y la accesibilidad de los miembros de la clase base en una clase derivada, al usar cada uno de los tipos de herencia.

En el capítulo 13, Programación orientada a objetos: polimorfismo, continuaremos con nuestra discusión sobre la herencia al introducir el polimorfismo: un concepto orientado a objetos que nos permite escribir programas que manejen, de una forma más general, los objetos de una amplia variedad de clases relacionadas por la herencia. Al estudiar el capítulo 13, el lector estará familiarizado con las clases, objetos, encapsulación, herencia y polimorfismo: los conceptos esenciales de la programación orientada a objetos.

Resumen

Sección 12.1 Introducción

- La reutilización de software reduce el tiempo y el costo del desarrollo de programas.

Sección 12.2 Clases base y clases derivadas

- La herencia es una forma de reutilización de software en la que el programador crea una clase que absorbe los datos y comportamientos de una clase existente, y los mejora con nuevas capacidades. La clase existente se llama clase base, y la nueva clase se conoce como clase derivada.
- Una clase base directa es aquella de la que una clase derivada hereda de forma explícita (lo cual se especifica mediante el nombre de la clase a la derecha del signo : en la primera línea de una definición de clase). Una clase base indirecta se hereda de dos o más niveles hacia arriba en la jerarquía de clases.
- Con la herencia simple, una clase se deriva de una clase base. Con la herencia múltiple, una clase hereda de varias clases base (posiblemente no relacionadas).
- Una clase derivada representa a un grupo más especializado de objetos. Por lo general, una clase derivada contiene los comportamientos heredados de su clase base más ciertos comportamientos adicionales. Una clase derivada también puede personalizar los comportamientos heredados de la clase base.
- Cada objeto de una clase derivada es también un objeto de la clase base de esa clase. Sin embargo, un objeto de la clase base no es un objeto de las clases derivadas de esa clase.
- La relación “*es un*” representa la herencia. En una relación “*es un*”, un objeto de una clase derivada también se puede tratar como un objeto de su clase base.
- La relación “*tiene un*” representa la composición; un objeto contiene uno o más objetos de otras clases como miembros, pero no revela su comportamiento directamente en su interfaz.
- Una clase derivada no puede acceder a los miembros `private` de su clase base directamente; permitir esto violaría el encapsulamiento de la clase base. No obstante, una clase derivada puede acceder a los miembros `public` y `protected` de su clase base directamente.
- Una clase derivada puede efectuar cambios de estado en los miembros `private` de la clase base, pero sólo a través de las funciones miembro `no private` que se proporcionan en la clase base, y se heredan en la clase derivada.
- Cuando una función miembro de la clase base es inapropiada para una clase derivada, esa función miembro se puede redefinir en la clase derivada con una implementación apropiada.
- Las relaciones de herencia simple forman estructuras jerárquicas tipo árbol; una clase base existe en una relación jerárquica con sus clases derivadas.
- Es posible tratar a los objetos de la clase base y a los objetos de la clase derivada de manera similar; las características comunes compartidas entre los tipos de los objetos se expresan en los datos miembro y en las funciones miembro de la clase base.

Sección 12.3 Miembros `protected`

- Los miembros `public` de una clase base están accesibles en cualquier parte en donde el programa tenga un manejador a un objeto de esa clase base, o a un objeto de una de las clases derivadas de esa clase base; o, al usar el operador de resolución de ámbito binario, cada vez que el nombre de la clase esté dentro del alcance.
- Los miembros `private` de una clase base están accesibles sólo dentro de la definición de esa clase base, o desde las funciones `friend` de esa clase.
- Los miembros `protected` de una clase base tienen un nivel intermedio de protección entre el acceso `public` y `private`. Los miembros `protected` de una clase base pueden ser utilizados por los miembros y funciones `friend` de esa clase base, y por los miembros y funciones `friend` de cualquier clase que se derive de esa clase base.
- Por desgracia, los datos miembro `protected` presentan a menudo dos problemas serios. En primer lugar, el objeto de la clase derivada no tiene que usar una función `set` para modificar el valor de los datos `protected` de la clase base. En segundo lugar, es más probable que las funciones miembro de la clase derivada dependan de los detalles de implementación de la clase base.
- Cuando una función miembro de una clase derivada redefine a una función miembro de la clase base, la función miembro de la clase base puede ser utilizada por la clase derivada, calificando el nombre de la función miembro de la clase base con el nombre de la clase base y el operador de resolución de ámbito binario (:):.

Sección 12.5 Los constructores y destructores en las clases derivadas

- Cuando se instancia un objeto de una clase derivada, el constructor de la clase base se llama de inmediato (ya sea en forma explícita o implícita) para inicializar los datos miembro de la clase base en el objeto de la clase derivada (antes de inicializar los datos miembro de la clase derivada).

Sección 12.6 Herencia `public`, `protected` y `private`

- Al declarar los datos miembro `private`, proporcionando a la vez funciones miembro `no private` para manipular y realizar la comprobación de validez en estos datos, se hace cumplir la buena ingeniería de software.

- Cuando se destruye un objeto de una clase derivada, los destructores se llaman en el orden inverso al de los constructores; primero se llama el destructor de la clase derivada, y después se llama el destructor de la clase base.
- Al derivar una clase de una clase base, la clase base se puede declarar como **public**, **protected** o **private**.
- Al derivar una clase de una clase base **public**, los miembros **public** de la clase base se convierten en miembros **public** de la clase derivada, y los miembros **protected** de la clase base se convierten en miembros **protected** de la clase derivada.
- Al derivar una clase de una clase base **protected**, los miembros **public** y **protected** de la clase base se convierten en miembros **protected** de la clase derivada.
- Al derivar una clase de una clase base **private**, los miembros **public** y **protected** de la clase base se convierten en miembros **private** de la clase derivada.

Terminología

clase base	nombre calificado
clase base directa	personalizar software
clase base indirecta	private , clase base
clase derivada	private , herencia
composición	protected , clase base
constructor de la clase base	protected , herencia
constructor de la clase derivada	protected , miembro de una clase
constructor predeterminado de la clase base	protected , palabra clave
destructor de la clase base	public , clase base
destructor de la clase derivada	public , herencia
friend de una clase base	redefinir una función miembro de la clase base
friend de una clase derivada	relación “ <i>es un</i> ”
heredar los miembros de una clase existente	relación “ <i>tiene un</i> ”
herencia	relación jerárquica
herencia múltiple	software frágil
herencia simple	software quebradizo
inicializador de la clase base	subclase
jerarquía de clases	superclase

Ejercicios de autoevaluación

12.1 Complete los siguientes enunciados:

- _____ es una forma de reutilización de software, en la que nuevas clases absorben los datos y comportamientos de las clases existentes, y adornan estas clases con nuevas capacidades.
- Los miembros _____ de una clase base pueden utilizarse sólo en la definición de la clase base, o en las definiciones de la clase derivada.
- En una relación _____, un objeto de una clase derivada se puede tratar también como un objeto de su clase base.
- En una relación _____, el objeto de una clase tiene uno o más objetos de otras clases como miembros.
- En la herencia simple, una clase existe en una relación _____ con sus clases derivadas.
- Los miembros _____ de una clase base se pueden utilizar dentro de esa clase base, y en cualquier parte en donde el programa tenga un manejador a un objeto de esa clase, o a un objeto de una de sus clases derivadas.
- Los miembros de acceso **protected** de una clase base tienen un nivel de protección entre los de acceso **public** y _____.
- C++ cuenta con _____, la cual permite a una clase derivada heredar de muchas clases base, incluso aunque las clases base no estén relacionadas.
- Cuando se instancia un objeto de una clase derivada, el _____ constructor de la clase base se llama de manera implícita o explícita para realizar la inicialización necesaria de los datos miembro de la clase base en el objeto de la clase derivada.
- Al derivar una clase de una clase base con herencia **public**, los miembros **public** de la clase base se convierten en miembros _____ de la clase derivada, y los miembros **protected** de la clase base se convierten en miembros _____ de la clase derivada.
- Al derivar una clase de una clase base con herencia **protected**, los miembros **public** de la clase base se convierten en miembros _____ de la clase derivada, y los miembros **protected** de la clase base se convierten en miembros _____ de la clase derivada.

- 12.2** Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
- Los constructores de la clase base no son heredados por las clases derivadas.
 - Una relación “*tiene un*” se implementa mediante la herencia.
 - Una clase *Auto* tiene una relación “*es un*” con las clases *Volante*, *Direccion* y *Frenos*.
 - La herencia fomenta la reutilización de software comprobado, de alta calidad.
 - Cuando se destruye un objeto de una clase derivada, los destructores se llaman en el orden inverso al de los constructores.

Respuestas a los ejercicios de autoevaluación

12.1 a) Herencia. b) *protected*. c) “*es un*” o de herencia. d) “*tiene-un*”, o composición, o agregación. e) jerárquica. f) *public*. g) *private*. h) herencia múltiple. i) constructor. j) *public*, *protected*. k) *protected*, *protected*.

12.2 a) Verdadero. b) Falso. Una relación “*tiene un*” se implementa mediante la composición. Una relación “*es un*” se implementa mediante la herencia. c) Falso. Éste es un ejemplo de una relación “*tiene un*”. La clase *Auto* tiene una relación “*es-un*” con la clase *Vehículo*. d) Verdadero. e) Verdadero.

Ejercicios

12.3 Muchos programas escritos con herencia podrían escribirse mediante la composición, y viceversa. Vuelva a escribir la clase *EmpleadoBaseMasComision* de la jerarquía *EmpleadoPorComision-EmpleadoBaseMasComision* para usar la composición en vez de la herencia. Una vez que haga esto, valore los méritos relativos de las dos metodologías para diseñar las clases *EmpleadoPorComision* y *EmpleadoBaseMasComision*, así como también para los programas orientados a objetos en general. ¿Cuál metodología es más natural? ¿Por qué?

12.4 Describa las formas en las que la herencia promueve la reutilización de software, ahorra tiempo durante el desarrollo de los programas y ayuda a prevenir errores.

12.5 Algunos programadores prefieren no utilizar el acceso *protected*, ya que creen que quebranta el encapsulamiento de la clase base. Hable sobre los méritos relativos de utilizar el acceso *protected* en comparación con el acceso *private* en las clases base.

12.6 Dibuje una jerarquía de herencia para los estudiantes en una universidad, de manera similar a la jerarquía que se muestra en la figura 12.2. Use a *Estudiante* como la superclase de la jerarquía, y después incluya las clases *EstudianteNoGraduado* y *EstudianteGraduado*, que se deriven de *Estudiante*. Siga extendiendo la jerarquía con el mayor número de niveles que sea posible. Por ejemplo, *EstudiantePrimerAnio*, *EstudianteSegundoAnio*, *EstudianteTercerAnio* y *EstudianteCuartoAnio* podrían derivarse de *EstudianteNoGraduado*, y *EstudianteDoctorado* y *EstudianteMaestria* podrían derivarse de *EstudianteGraduado*. Después de dibujar la jerarquía, hable sobre las relaciones que existen entre las clases. [Nota: no necesita escribir código para este ejercicio].

12.7 El mundo de las figuras es más extenso que las figuras incluidas en la jerarquía de herencia de la figura 12.3. Anote todas las figuras en las que pueda pensar (tanto bidimensionales como tridimensionales) e intégrelas en una jerarquía *Figura* más completa, con todos los niveles que sea posible. Su jerarquía debe tener la clase base *Figura*, de la que se deriven las clases *FiguraBidimensional* y *FiguraTridimensional*. [Nota: no necesita escribir código para este ejercicio]. Utilizaremos esta jerarquía en los ejercicios del capítulo 13 para procesar un conjunto de figuras distintas como objetos de la clase base *Figura*. (Esta técnica, conocida como polimorfismo, es el tema del capítulo 13).

12.8 Dibuje una jerarquía de herencia para las clases *Cuadrilatero*, *Trapezoide*, *Paralelogramo*, *Rectangulo* y *Cuadrado*. Use *Cuadrilatero* como la clase base de la jerarquía. Agregue todos los niveles que sea posible a la jerarquía.

12.9 (Jerarquía de herencia Paquete) Los servicios de entrega de paquetes como FedEx®, DHL® y UPS® ofrecen una variedad de opciones de envío distintas, cada una con los costos específicos asociados. Cree una jerarquía de herencia para representar varios tipos de paquetes. Use *Paquete* como la clase base de la jerarquía y después incluya las clases *PaqueteDosDias* y *PaqueteNocturno* que se deriven de *Paquete*. La clase base *Paquete* debe incluir datos miembro que representen el nombre, dirección, ciudad, estado y código postal para el emisor y el destinatario del paquete, además de los datos miembro que almacenan el peso (en onzas) y el costo por onza para enviar el paquete. El constructor de *Paquete* debe inicializar estos miembros de datos. Asegúrese que el peso y costo por onza contengan valores positivos. *Paquete* debe proporcionar una función miembro *public* llamada *calcularCosto* que devuelva un valor *double* indicando el costo asociado con el envío del paquete. La función *calcularCosto* de *Paquete* debe determinar el costo al multiplicar el peso por el costo por onza. La clase derivada *PaqueteDosDias* debe heredar la funcionalidad de la clase base *Paquete*, pero también debe incluir un miembro de datos que represente una cuota fija que cobre la compañía de envío por el servicio de entrega de dos días. El constructor de *PaqueteDosDias* debe recibir un valor para inicializar este miembro de datos. *PaqueteDosDias* debe redefinir la función

miembro `calcularCosto`, de manera que calcule el costo sumando la cuota fija al costo basado en el peso, calculado por la función `calcularCosto` de la clase base `Paquete`. La clase `PaqueteNocturno` debe heredar directamente de la clase `Paquete` y debe contener un miembro de datos adicional que represente una cuota adicional por cada onza que se cobre por el servicio de entrega nocturna. `PaqueteNocturno` debe redefinir la función miembro `calcularCosto`, de manera que sume la cuota adicional por onza al costo estándar por onza, antes de calcular el costo de envío. Escriba un programa de prueba para crear objetos de cada tipo de `Paquete` y evaluar la función miembro `calcularCosto`.

12.10 (Jerarquía de herencia Cuenta) Cree una jerarquía de herencia que podría usar un banco para representar las cuentas bancarias de los clientes. Todos los clientes en este banco pueden depositar (es decir, abonar) dinero en sus cuentas, y retirar (es decir, cargar) dinero de ellas. También existen tipos más específicos de cuentas. Por ejemplo, las cuentas de ahorro obtienen intereses sobre el dinero que contienen. Por otro lado, las cuentas de cheques cobran una cuota por transacción (es decir, abono o cargo).

Cree una jerarquía de herencia que contenga la clase base `Cuenta`, junto con las clases derivadas `CuentaAhorros` y `CuentaCheques` que hereden de la clase `Cuenta`. La clase base `Cuenta` debe incluir un miembro de datos de tipo `double` para representar el saldo de la cuenta. La clase debe proporcionar un constructor que reciba un saldo inicial y lo utilice para inicializar el miembro de datos. El constructor debe validar el saldo inicial, para asegurar que sea mayor o igual a 0.0. De no ser así, el saldo debe establecerse en 0.0 y el constructor debe mostrar un mensaje de error, indicando que el saldo inicial es inválido. La clase debe proporcionar tres funciones miembro. La función miembro `abonar` debe sumar un monto al saldo actual. La función miembro `cargar` debe retirar dinero de la `Cuenta` y asegurar que el monto a cargar no exceda el saldo de la `Cuenta`. Si lo hace, el saldo debe permanecer sin cambio y la función debe imprimir el mensaje "El monto a cargar excedio el saldo de la cuenta." La función miembro `getSaldo` debe devolver el saldo actual.

La clase derivada `CuentaAhorros` debe heredar la funcionalidad de una `Cuenta`, pero también debe incluir un miembro de datos de tipo `double` que indique la tasa de interés (porcentaje) asignada a la `Cuenta`. El constructor de `CuentaAhorros` debe recibir el saldo inicial, así como un valor inicial para la tasa de interés de `CuentaAhorros`. `CuentaAhorros` debe proporcionar una función miembro `public` llamada `calcularInteres`, que devuelva un valor `double` que indique el monto de interés obtenido por una cuenta. La función miembro `calcularInteres` debe determinar este monto, multiplicando la tasa de interés por el saldo de la cuenta. [Nota: `CuentaAhorros` debe heredar las funciones miembro `abonar` y `cargar` como están, sin redefinirlas].

La clase derivada `CuentaCheques` debe heredar de la clase base `Cuenta` e incluir un miembro de datos adicional de tipo `double`, que represente la cuota que se cobra por transacción. El constructor de `CuentaCheques` debe recibir el saldo inicial, así como un parámetro que indique el monto de la cuota. La clase `CuentaCheques` debe redefinir las funciones miembro `abonar` y `cargar` de manera que resten la cuota del saldo de la cuenta, cada vez que se realice una de esas transacciones con éxito. Las versiones de `CuentaCheques` de estas funciones deben invocar la versión de la clase base `Cuenta` para realizar las actualizaciones en el saldo de una cuenta. La función `cargar` de `CuentaCheques` debe cobrar una cuota sólo si realmente se retiró dinero (es decir, que el monto a cargar no exceda el saldo de la cuenta). [Sugerencia: defina la función `cargar` de `Cuenta` de manera que devuelva un valor `bool` que indique si se retiró dinero. Después use el valor de retorno para determinar si se debe cobrar una cuota].

Después de definir las clases en esta jerarquía, escriba un programa para crear objetos de cada clase y evaluar sus funciones miembro. Agregue interés al objeto `CuentaAhorros`, primero invocando a su función `calcularInteres` y después pasando el monto de interés devuelto a la función `abonar` del objeto.



Un anillo para gobernarlos a todos, un anillo para encontrarlos, un anillo para traerlos a todos y en la oscuridad enlazarlos.

—John Ronald Reuel Tolkien

El silencio, a menudo de pura inocencia, persuade cuando el habla falla.

—William Shakespeare

Las proposiciones generales no deciden casos concretos.

—Oliver Wendell Holmes

Un filósofo de imponente estatura no piensa en un vacío. Incluso sus ideas más abstractas son, en cierta medida, condicionadas por lo que se conoce o no en el tiempo en que vive.

—Alfred North Whitehead

Programación orientada a objetos: polimorfismo

OBJETIVOS

En este capítulo aprenderá a:

- Comprender el concepto de polimorfismo, la forma en que hace la programación más conveniente y los sistemas más extensibles y fáciles de mantener.
- Declarar y usar funciones `virtual` para llevar a cabo el polimorfismo.
- Distinguir entre clases abstractas y concretas.
- Declarar funciones `virtual` puras para crear clases abstractas.
- Usar la información de tipos en tiempo de ejecución (RTTI) con la conversión descendente, `dynamic_cast`, `typeid` y `type_info`.
- Conocer la forma en que C++ implementa las funciones `virtual` y la vinculación dinámica “detrás de las cámaras”.
- Usar destructores `virtual` para asegurar que se ejecuten todos los destructores apropiados en un objeto.

- 13.1** Introducción
- 13.2** Ejemplos de polimorfismo
- 13.3** Relaciones entre los objetos en una jerarquía de herencia
 - 13.3.1** Invocación de funciones de la clase base desde objetos de una clase derivada
 - 13.3.2** Cómo orientar los apuntadores de una clase derivada a objetos de la clase base
 - 13.3.3** Llamadas a funciones miembro de una clase derivada a través de apuntadores de la clase base
 - 13.3.4** Funciones virtuales
 - 13.3.5** Resumen de las asignaciones permitidas entre objetos y apuntadores de la clase base y de la clase derivada
- 13.4** Tipos de campos e instrucciones `switch`
- 13.5** Clases abstractas y funciones `virtual` puras
- 13.6** Ejemplo práctico: sistema de nómina mediante el uso de polimorfismo
 - 13.6.1** Creación de la clase base abstracta `Empleado`
 - 13.6.2** Creación de la clase derivada concreta `EmpleadoAsalariado`
 - 13.6.3** Creación de la clase derivada concreta `EmpleadoPorHoras`
 - 13.6.4** Creación de la clase derivada concreta `EmpleadoPorComision`
 - 13.6.5** Creación de la clase derivada concreta indirecta `EmpleadoBaseMasComision`
 - 13.6.6** Demostración del procesamiento polimórfico
- 13.7** (Opcional) Polimorfismo, funciones virtuales y vinculación dinámica “detrás de las cámaras”
- 13.8** Ejemplo práctico: sistema de nómina mediante el uso de polimorfismo e información de tipos en tiempo de ejecución con conversión descendente, `dynamic_cast`, `typeid` y `type_info`
- 13.9** Destructores virtuales
- 13.10** (Opcional) Ejemplo práctico de Ingeniería de Software: incorporación de la herencia en el sistema ATM
- 13.11** Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)

13.1 Introducción

En los capítulos 9 a 12 hablamos sobre las tecnologías clave de programación orientada a objetos, incluyendo las clases, los objetos, el encapsulamiento, la sobrecarga de operadores y la herencia. Ahora continuaremos nuestro estudio de la POO al explicar y demostrar el **polimorfismo** con las jerarquías de herencia. El polimorfismo nos permite “programar en general” en vez de “programar de manera específica”. En especial, el polimorfismo nos permite escribir programas que procesen objetos de clases que formen parte de la misma jerarquía de clases, como si todos fueran objetos de la clase base de la jerarquía. Como veremos en breve, el polimorfismo trabaja con los manejadores de apuntadores de clase base y manejadores de referencias de clase base, pero no con los manejadores de nombres.

Considere el siguiente ejemplo de polimorfismo. Suponga que vamos a crear un programa que simula el movimiento de varios tipos de animales para un estudio biológico. Las clases `Pez`, `Rana` y `Ave` representan los tres tipos de animales bajo investigación. Imagine que cada una de estas clases hereda de la clase base `Animal`, la cual contiene una función llamada `mover` y mantiene la posición actual de un animal. Cada clase derivada implementa la función `mover`. Nuestro programa mantiene un vector de apuntadores a objetos de las diversas clases derivadas de `Animal`. Para simular los movimientos de los animales, el programa envía a cada objeto el mismo mensaje una vez por segundo; a saber, `mover`. No obstante, cada tipo específico de `Animal` responde a un mensaje `mover` de manera única; un `Pez` podría nadar dos pies, una `Rana` podría saltar tres pies y un `Ave` podría volar diez pies. El programa envía el mismo mensaje (es decir, `mover`) a cada objeto animal en forma genérica, pero cada objeto sabe cómo modificar sus ubicación en forma apropiada para su tipo específico de movimiento. Confiar en que cada objeto sepa cómo “hacer lo correcto” (es decir, lo que sea apropiado para ese tipo de objeto) en respuesta a la llamada a la misma función es el concepto clave del polimorfismo. El mismo mensaje (en este caso, `mover`) que se envía a una variedad de objetos tiene “muchas formas” de resultados; de aquí que se utilice el término polimorfismo.

Con el polimorfismo podemos diseñar e implementar sistemas que puedan extenderse con facilidad; pueden agregarse nuevas clases con sólo modificar un poco (o nada) las porciones generales del programa, siempre y cuando las nuevas clases sean parte de la jerarquía de herencia que el programa procesa en forma genérica. Las únicas partes de un programa que deben alterarse para dar cabida a las nuevas clases son las que requieren un conocimiento directo de las nuevas clases que el programador va a agregar a la jerarquía. Por ejemplo, si creamos la clase `Tortuga` que hereda de la clase `Animal` (que podría responder a un mensaje `mover` caminando una pulgada), necesitamos escribir sólo la clase `Tortuga` y la parte de la simulación que crea una instancia de un objeto `Tortuga`. Las porciones de la simulación que procesan a cada `Animal` en forma genérica pueden permanecer iguales.

Vamos a empezar con una secuencia de ejemplos pequeños y enfocados, que nos conducirán a una comprensión de las funciones `virtual` y la vinculación dinámica; las dos tecnologías subyacentes del polimorfismo. Después presentaremos un caso de estudio basado en la jerarquía `Empleado` del capítulo 12. En el caso de estudio definiremos una “interfaz” (conjunto de funcionalidad) común para todas las clases en la jerarquía. Esta funcionalidad común entre los empleados se define en una clase base abstracta llamada `Empleado`, a partir de la cual las clases `EmpleadoAsalariado`, `EmpleadoPorHoras` y `EmpleadoPorComision` heredan directamente, y la clase `EmpleadoBaseMasComision` hereda indirectamente. Pronto veremos qué es lo que hace a una clase “abstracta” o “concreta” (lo opuesto).

En esta jerarquía, cada empleado tiene una función `ingresos` para calcular el sueldo semanal del empleado. Estas funciones `ingresos` varían por tipo de empleado; por ejemplo, los objetos `EmpleadoAsalariado` reciben un salario semanal fijo sin importar el número de horas trabajadas, mientras que los objetos `EmpleadoPorHoras` reciben sueldo por hora y por las horas extra. Mostraremos al lector cómo procesar cada empleado “en general”; es decir, usando apuntadores de la clase base para llamar a la función `ingresos` de varios objetos de la clase derivada. De esta forma, sólo nos tenemos que preocupar con un tipo de llamada a función, la cual se puede utilizar para ejecutar varias funciones distintas con base en los objetos a los que hacen referencia los apuntadores de la clase base.

Una característica clave de este capítulo es su discusión detallada (opcional) acerca del polimorfismo, las funciones `virtuales` y la vinculación dinámica “detrás de las cámaras”, la cual utiliza un diagrama detallado para explicar cómo se puede implementar el polimorfismo en C++.

Ocasionalmente, cuando se lleva a cabo el procesamiento polimórfico, es necesario programar “en forma específica”, lo cual significa que las operaciones se necesitan realizar en un tipo específico de objeto en la jerarquía; la operación no se puede aplicar en forma general a varios tipos de objetos. Reutilizaremos nuestra jerarquía `Empleado` para demostrar las poderosas herramientas de la **información de tipos en tiempo de ejecución (RTTI)** y la **conversión dinámica de tipos**, que permiten a un programa determinar el tipo de un objeto en tiempo de ejecución y actuar sobre ese objeto de manera acorde. Utilizamos estas capacidades para determinar si cierto objeto empleado específico es un `EmpleadoBaseMasComision`, y después otorgamos a ese empleado un bono del 10 por ciento sobre su salario base.

13.2 Ejemplos de polimorfismo

En esta sección consideraremos varios ejemplos de polimorfismo. Con el polimorfismo, una función puede hacer que ocurran distintas acciones, dependiendo del tipo del objeto en el que se invoca la función. Esto nos proporciona una tremenda capacidad expresiva. Si la clase `Rectangulo` se deriva de la clase `Cuadrilatero`, entonces un objeto `Rectangulo` es una versión más específica de un objeto `Cuadrilatero`. Por lo tanto, cualquier operación (como calcular el perímetro o el área) que pueda realizarse en un objeto de la clase `Cuadrilatero` también puede realizarse en un objeto de la clase `Rectangulo`. Estas operaciones también pueden realizarse en otros tipos de objetos `Cuadrilatero`, tales como `Cuadrado`, `Paralelogramo` y `Trapezoid`. El polimorfismo ocurre cuando un programa invoca a una función `virtual` a través de un apuntador o referencia de la clase base (es decir, `Cuadrilatero`); C++ elige en forma dinámica (es decir, en tiempo de ejecución) la función correcta para la clase a partir de la cual se instanció el objeto. En la sección 13.3 veremos un ejemplo de código que ilustra este proceso.

Como otro ejemplo, suponga que vamos a diseñar un videojuego que manipule objetos de muchos tipos distintos, incluyendo objetos de las clases `Marciano`, `Venusino`, `Plutoniano`, `NaveEspacial` y `RayoLaser`. Imagine que cada clase hereda de la clase base común llamada `ObjetoEspacial`, la cual contiene el método `dibujar`. Cada clase derivada implementa a esta función de una manera apropiada para esa clase. Un programa administrador de la pantalla mantiene un contenedor (por ejemplo, un vector) que contiene apuntadores `ObjetoEspacial` a objetos de las distintas clases. Para actualizar la pantalla, el administrador de pantalla envía el mismo mensaje a cada objeto; a saber, `dibujar`. Cada tipo de objeto responde de una manera única. Por ejemplo, un objeto `Marciano` podría dibujarse a sí mismo en color rojo, con el número apropiado de antenas. Un objeto `NaveEspacial` podría dibujarse a sí mismo como un platillo volador de color plateado. Un objeto `RayoLaser` podría dibujarse a sí mismo como un rayo color rojo brillante a lo largo de la pantalla. De nuevo, el mismo mensaje (en este caso, `dibujar`) que se envía a una variedad de objetos tiene “muchas formas” de resultados.

Un administrador de pantalla polimórfico facilita el proceso de agregar nuevas clases a un sistema, con el mínimo de modificaciones a su código. Suponga que deseamos agregar objetos de la clase `Mercuriano` a nuestro videojuego. Para ello, debemos crear una clase `Mercuriano` que herede de `ObjetoEspacial`, pero proporcione su propia definición de la función miembro `dibujar`. Después, cuando aparezcan apuntadores a objetos de la clase `Mercuriano` en el contenedor, no será necesario modificar el código para el administrador de pantalla. El administrador de pantalla invocará a la función miembro `dibujar` en cada objeto en el contenedor, sin importar el tipo del objeto, por lo que los nuevos objetos `Mercuriano` simplemente se integran en forma automática. Así, sin modificar el sistema (más que para crear e incluir las mismas clases), los programadores pueden utilizar el polimorfismo para acomodar clases adicionales, incluyendo las que no se hayan considerado a la hora de crear el sistema.



Observación de Ingeniería de Software 13.1

Con las funciones `virtual` y el polimorfismo, podemos tratar con las generalidades y dejar que el entorno en tiempo de ejecución se encargue de los detalles específicos. Los programadores pueden ordenar a una variedad de objetos que se comporten en formas apropiadas para ellos, sin necesidad de conocer los tipos de los objetos (siempre y cuando éstos pertenezcan a la misma jerarquía de herencia y se utilicen a través de un apuntador común de la clase base).



Observación de Ingeniería de Software 13.2

El polimorfismo promueve la extensibilidad: el software escrito para invocar el comportamiento polimórfico se escribe de manera independiente de los tipos de los objetos a los cuales se envían los mensajes. Así, se pueden incorporar en un sistema nuevos tipos de objetos que pueden responder a los mensajes existentes, sin necesidad de modificar el sistema base. Sólo el código cliente que crea instancias de los nuevos objetos debe modificarse para dar cabida a los nuevos tipos.

13.3 Relaciones entre los objetos en una jerarquía de herencia

En la sección 12.4 se creó una jerarquía de clases de empleados, en la cual la clase `EmpleadoBaseMasComision` heredó de la clase `EmpleadoPorComision`. Los ejemplos del capítulo 12 manipularon objetos `EmpleadoPorComision` y objetos `EmpleadoBaseMasComision` mediante el uso de sus nombres para invocar a sus funciones miembro. Ahora examinaremos las relaciones entre las clases en una jerarquía con más detalle. Las siguientes secciones presentan una serie de ejemplos que demuestran cómo se pueden orientar apuntadores de la clase base y de la clase derivada a objetos de la clase base y de la clase derivada, y cómo se pueden utilizar esos apuntadores para invocar a funciones miembro que manipulen a esos objetos. En la sección 13.3.4, demostraremos cómo obtener un comportamiento polimórfico de los apuntadores de la clase base orientados a los objetos de la clase derivada.

En la sección 13.3.1 asignaremos la dirección de un objeto de la clase derivada a un apuntador de la clase base, y después mostraremos que al invocar una función a través del apuntador de la clase base se invoca a la funcionalidad de la clase base; es decir, el tipo del manejador determina cuál función se llama. En la sección 13.3.2 asignaremos la dirección de un objeto de la clase base a un apuntador de la clase derivada, lo cual produce un error de compilación. Hablaremos sobre el mensaje de error e investigaremos por qué el compilador no permite dicha asignación. En la sección 13.3.3 asignaremos la dirección de un objeto de la clase derivada a un apuntador de la clase base, y después examinaremos cómo puede usarse el apuntador de la clase base para invocar sólo la funcionalidad de la clase base; al tratar de invocar funciones miembro de la clase derivada a través del apuntador de la clase base, se producen errores de compilación. Por último, en la sección 13.3.4 introduciremos las funciones `virtual` y el polimorfismo, al declarar una función de la clase base como `virtual`. Después asignaremos la dirección de un objeto de la clase derivada al apuntador de la clase base y utilizaremos ese apuntador para invocar la funcionalidad de la clase derivada; precisamente la capacidad que necesitamos para lograr el comportamiento polimórfico.

Un concepto clave en estos ejemplos es demostrar que un objeto de una clase derivada puede tratarse como un objeto de su clase base. Esto permite varias manipulaciones interesantes. Por ejemplo, un programa puede crear un arreglo de apuntadores de la clase base que apunten a objetos de muchos tipos de clases derivadas. A pesar del hecho de que los objetos de las clases derivadas son de distintos tipos, el compilador lo permite debido a que cada objeto de una clase derivada *es un* objeto de su clase base. Sin embargo, no podemos tratar a un objeto de la clase base como un objeto de una de sus clases derivadas. Por ejemplo, un `EmpleadoPorComision` no es un `EmpleadoBaseMasComision` en la jerarquía definida en el capítulo 12; un `EmpleadoPorComision` no tiene un dato miembro `salarioBase` y no tiene las funciones miembro `setSalarioBase` y `getSalarioBase`. La relación “*es un*” se aplica sólo de una clase derivada a sus clases base directa e indirectas.

13.3.1 Invocación de funciones de la clase base desde objetos de una clase derivada

El ejemplo en las figuras 13.1 a 13.5 demuestra tres formas de orientar los apuntadores de la clase base y los apuntadores de la clase derivada a objetos de la clase base y objetos de la clase derivada. Las primeras dos son simples: orientamos un apuntador de la clase base a un objeto de la clase base (e invocamos la funcionalidad de la clase base), y orientamos un apuntador de la clase derivada a un objeto de la clase derivada (e invocamos la funcionalidad de la clase derivada). Después, demostramos la relación entre las clases derivadas y las clases base (es decir, la relación “*es un*” de herencia) al orientar un apuntador de la clase base a un objeto de la clase derivada (y mostrando que la funcionalidad de la clase base está evidentemente disponible en el objeto de la clase derivada).

La clase `EmpleadoPorComision` (figuras 13.1 y 13.2), que vimos en el capítulo 12, se utiliza para representar empleados que reciben un porcentaje de sus ventas. La clase `EmpleadoBaseMasComision` (figuras 13.3 y 13.4), que también vimos en el capítulo 12, se utiliza para representar empleados que reciben un salario base más un porcentaje de sus ventas. Cada objeto `EmpleadoBaseMasComision` *es un* `EmpleadoPorComision` que también tiene un salario base. La función miembro `ingresos` de la clase `EmpleadoBaseMasComision` (líneas 32 a 35 de la figura 13.4) redefine a la función miembro `ingresos` de la clase `EmpleadoPorComision` (líneas 79 a 82 de la figura 13.2) para incluir el salario base del objeto. La función miembro `imprimir` de la clase `EmpleadoBaseMasComision` (líneas 38 a 46 de la figura 13.4) redefine la función miembro `imprimir` de la clase `EmpleadoPorComision` (líneas 85 a 92 de la figura 13.2) para mostrar la misma información que la función `imprimir` en la clase `EmpleadoPorComision`, así como el salario base del empleado.

```

1 // Fig. 13.1: EmpleadoPorComision.h
2 // Definición de la clase EmpleadoPorComision que representa a un empleado por comisión.
3 #ifndef COMISION_H
4 #define COMISION_H
5
6 #include <string> // clase string estándar de C++
7 using std::string;
8
9 class EmpleadoPorComision
10 {
11 public:
12     EmpleadoPorComision( const string &, const string &, const string &,
13                          double = 0.0, double = 0.0 );
14
15     void setPrimerNombre( const string & ); // establece el primer nombre
16     string getPrimerNombre() const; // devuelve el primer nombre
17
18     void setApellidoPaterno( const string & ); // establece el apellido paterno
19     string getApellidoPaterno() const; // devuelve el apellido paterno
20
21     void setNumeroSeguroSocial( const string & ); // establece el NSS
22     string getNumeroSeguroSocial() const; // devuelve el NSS
23
24     void setVentasBrutas( double ); // establece el monto de ventas brutas
25     double getVentasBrutas() const; // devuelve el monto de ventas brutas
26
27     void setTarifaComision( double ); // establece la tarifa de comisión
28     double getTarifaComision() const; // devuelve la tarifa de comisión
29
30     double ingresos() const; // calcula los ingresos
31     void imprimir() const; // imprime el objeto EmpleadoPorComision
32 private:
33     string primerNombre;
34     string apellidoPaterno;
35     string numeroSeguroSocial;
36     double ventasBrutas; // ventas brutas por semana
37     double tarifaComision; // porcentaje de comisión
38 }; // fin de la clase EmpleadoPorComision
39
40 #endif

```

Figura 13.1 | Archivo de encabezado de la clase `EmpleadoPorComision`.

```

1 // Fig. 13.2: EmpleadoPorComision.cpp
2 // Definiciones de las funciones miembro de EmpleadoPorComision.
3 #include <iostream>
4 using std::cout;
5
6 #include "EmpleadoPorComision.h" // definición de la clase EmpleadoPorComision
7
8 // constructor
9 EmpleadoPorComision::EmpleadoPorComision(
10     const string &nombre, const string &apellido, const string &nss,
11     double ventas, double tarifa )
12 : primerNombre( nombre ), apellidoPaterno( apellido ), numeroSeguroSocial( nss )
13 {
14     setVentasBrutas( ventas ); // valida y almacena las ventas brutas
15     setTarifaComision( tarifa ); // valida y almacena la tarifa de comisión
16 } // fin del constructor de EmpleadoPorComision
17
18 // establece el primer nombre
19 void EmpleadoPorComision::setPrimerNombre( const string &nombre )
20 {
21     primerNombre = nombre; // debe validar
22 } // fin de la función setPrimerNombre
23
24 // devuelve el primer nombre
25 string EmpleadoPorComision::getPrimerNombre() const
26 {
27     return primerNombre;
28 } // fin de la función getPrimerNombre
29
30 // establece el apellido paterno
31 void EmpleadoPorComision::setApellidoPaterno( const string &apellido )
32 {
33     apellidoPaterno = apellido; // debe validar
34 } // fin de la función setApellidoPaterno
35
36 // devuelve el apellido paterno
37 string EmpleadoPorComision::getApellidoPaterno() const
38 {
39     return apellidoPaterno;
40 } // fin de la función getApellidoPaterno
41
42 // establece el número de seguro social
43 void EmpleadoPorComision::setNumeroSeguroSocial( const string &nss )
44 {
45     numeroSeguroSocial = nss; // debe validar
46 } // fin de la función setNumeroSeguroSocial
47
48 // devuelve el número de seguro social
49 string EmpleadoPorComision::getNumeroSeguroSocial() const
50 {
51     return numeroSeguroSocial;
52 } // fin de la función getNumeroSeguroSocial
53
54 // establece el monto de ventas brutas
55 void EmpleadoPorComision::setVentasBrutas( double ventas )
56 {
57     ventasBrutas = ( ventas < 0.0 ) ? 0.0 : ventas;
58 } // fin de la función setVentasBrutas
59

```

Figura 13.2 | Archivo de implementación de la clase EmpleadoPorComision. (Parte I de 2).

```

60 // devuelve el monto de ventas brutas
61 double EmpleadoPorComision::getVentasBrutas() const
62 {
63     return ventasBrutas;
64 } // fin de la función getVentasBrutas
65
66 // establece la tarifa de comisión
67 void EmpleadoPorComision::setTarifaComision( double tarifa )
68 {
69     tarifaComision = ( tarifa > 0.0 && tarifa < 1.0 ) ? tarifa : 0.0;
70 } // fin de la función setTarifaComision
71
72 // devuelve la tarifa de comisión
73 double EmpleadoPorComision::getTarifaComision() const
74 {
75     return tarifaComision;
76 } // fin de la función getTarifaComision
77
78 // calcula los ingresos
79 double EmpleadoPorComision::ingresos() const
80 {
81     return getTarifaComision() * getVentasBrutas();
82 } // fin de la función ingresos
83
84 // imprime el objeto EmpleadoPorComision
85 void EmpleadoPorComision::imprimir() const
86 {
87     cout << "empleado por comision: "
88         << getPrimerNombre() << ' ' << getApellidoPaterno()
89         << "\nnumero de seguro social: " << getNumeroSeguroSocial()
90         << "\nventas brutas: " << getVentasBrutas()
91         << "\ntarifa de comision: " << getTarifaComision();
92 } // fin de la función imprimir

```

Figura 13.2 | Archivo de implementación de la clase EmpleadoPorComision. (Parte 2 de 2).

```

1 // Fig. 13.3: EmpleadoBaseMasComision.h
2 // Clase EmpleadoBaseMasComision derivada de la clase
3 // EmpleadoPorComision.
4 #ifndef BASEMAS_H
5 #define BASEMAS_H
6
7 #include <string> // clase string estándar de C++
8 using std::string;
9
10 #include "EmpleadoPorComision.h" // declaración de la clase EmpleadoPorComision
11
12 class EmpleadoBaseMasComision : public EmpleadoPorComision
13 {
14 public:
15     EmpleadoBaseMasComision( const string &, const string &,
16                             const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18     void setSalarioBase( double ); // establece el salario base
19     double getSalarioBase() const; // devuelve el salario base
20
21     double ingresos() const; // calcula los ingresos

```

Figura 13.3 | Archivo de encabezado de la clase EmpleadoBaseMasComision. (Parte 1 de 2).

```

22     void imprimir() const; // imprime el objeto EmpleadoBaseMasComision
23 private:
24     double salarioBase; // salario base
25 }; // fin de la clase EmpleadoBaseMasComision
26
27 #endif

```

Figura 13.3 | Archivo de encabezado de la clase EmpleadoBaseMasComision. (Parte 2 de 2).

```

1 // Fig. 13.4: EmpleadoBaseMasComision.cpp
2 // Definiciones de las funciones miembro de la clase EmpleadoBaseMasComision.
3 #include <iostream>
4 using std::cout;
5
6 // definición de la clase EmpleadoBaseMasComision
7 #include "EmpleadoBaseMasComision.h"
8
9 // constructor
10 EmpleadoBaseMasComision::EmpleadoBaseMasComision(
11     const string &nombre, const string &apellido, const string &nss,
12     double ventas, double tarifa, double salario )
13     // llama en forma explícita al constructor de la clase base
14     : EmpleadoPorComision( nombre, apellido, nss, ventas, tarifa )
15 {
16     setSalarioBase( salario ); // valida y almacena el salario base
17 } // fin del constructor de EmpleadoBaseMasComision
18
19 // establece el salario base
20 void EmpleadoBaseMasComision::setSalarioBase( double salario )
21 {
22     salarioBase = ( salario < 0.0 ) ? 0.0 : salario;
23 } // fin de la función setSalarioBase
24
25 // devuelve el salario base
26 double EmpleadoBaseMasComision::getSalarioBase() const
27 {
28     return salarioBase;
29 } // fin de la función getSalarioBase
30
31 // calcula los ingresos
32 double EmpleadoBaseMasComision::ingresos() const
33 {
34     return getSalarioBase() + EmpleadoPorComision::ingresos();
35 } // fin de la función ingresos
36
37 // imprime el objeto EmpleadoBaseMasComision
38 void EmpleadoBaseMasComision::imprimir() const
39 {
40     cout << "con salario base ";
41
42     // invoca a la función imprimir de EmpleadoPorComision
43     EmpleadoPorComision::imprimir();
44
45     cout << "\nsalario base: " << getSalarioBase();
46 } // fin de la función imprimir

```

Figura 13.4 | Archivo de implementación de la clase EmpleadoBaseMasComision.

En la figura 13.5, en las líneas 19 y 20 se crea un objeto `EmpleadoPorComision` y en la línea 23 se crea un apuntador a un objeto `EmpleadoPorComision`; en las líneas 26 y 27 se crea un objeto `EmpleadoBaseMasComision` y en la línea 30 se crea un apuntador a un objeto `EmpleadoBaseMasComision`. En las líneas 37 y 39 se utiliza el nombre

de cada objeto (`empleadoPorComision` y `empleadoBaseMasComision`, respectivamente) para invocar a la función miembro `imprimir` de cada objeto. En la línea 42 se asigna la dirección del objeto `empleadoPorComision` de la clase base al apuntador `empleadoPorComisionPtr` de la clase base, que la línea 45 utiliza para invocar a la función miembro `imprimir` en ese objeto `EmpleadoPorComision`. Esto invoca a la versión de `imprimir` definida en la clase base `EmpleadoPorComision`. De manera similar, en la línea 48 se asigna la dirección del objeto `empleadoBaseMasComision` de la clase derivada al apuntador `empleadoBaseMasComisionPtr` de la clase derivada, que la línea 52 utiliza para invocar a la función miembro `imprimir` en ese objeto `EmpleadoBaseMasComision`. Esto invoca a la versión de `imprimir` definida en la clase derivada `EmpleadoBaseMasComision`. Después, en la línea 55 se asigna la dirección del objeto `empleadoBaseMasComision` de la clase derivada al apuntador `empleadoPorComisionPtr` de la clase base, que en la línea 59 se utiliza para invocar a la función miembro `imprimir`. Se permite esta “contradicción”, ya que un objeto de una clase derivada *es un* objeto de su clase base. Observe que, a pesar del hecho de que la clase base `EmpleadoPorComision` apunta a un objeto de la clase derivada `EmpleadoBaseMasComision`, se invoca a la función miembro `imprimir` de la clase base `EmpleadoPorComision` (en vez de la función `imprimir` de `EmpleadoBaseMasComision`). Los resultados de cada invocación a cada una de las funciones miembro `imprimir` en este programa revelan que *la funcionalidad invocada depende del tipo del manejador (es decir, el tipo de apuntador o de referencia) que se utiliza para invocar la función, no del tipo del objeto al que apunta el manejador*. En la sección 13.3.4, cuando presentemos las funciones `virtual`, demostraremos que es posible invocar la funcionalidad del tipo del objeto, en vez de invocar la funcionalidad del tipo del manejador. Veremos que esto es crucial para implementar el comportamiento polimórfico: el tema clave de este capítulo.

```

1 // Fig. 13.5: fig13_05.cpp
2 // Cómo orientar los apuntadores de la clase base y la clase derivada a los
3 // objetos de la clase base y la clase derivada, respectivamente.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7 using std::fixed;
8
9 #include <iomanip>
10 using std::setprecision;
11
12 // incluye las definiciones de las clases
13 #include "EmpleadoPorComision.h"
14 #include "EmpleadoBaseMasComision.h"
15
16 int main()
17 {
18     // crea el objeto de la clase base
19     EmpleadoPorComision empleadoPorComision(
20         "Sue", "Jones", "222-22-2222", 10000, .06 );
21
22     // crea un apuntador de la clase base
23     EmpleadoPorComision *empleadoPorComisionPtr = 0;
24
25     // crea un objeto de la clase derivada
26     EmpleadoBaseMasComision empleadoBaseMasComision(
27         "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
28
29     // crea un apuntador de la clase derivada
30     EmpleadoBaseMasComision *empleadoBaseMasComisionPtr = 0;
31
32     // establece el formato de salida de punto flotante
33     cout << fixed << setprecision( 2 );
34
35     // imprime los objetos empleadoPorComision y empleadoBaseMasComision
36     cout << "Impresión de los objetos de clase base y clase derivada:\n\n";
37     empleadoPorComision.imprimir(); // invoca a la función imprimir de la clase base

```

Figura 13.5 | Asignación de direcciones de objetos de la clase base y la clase derivada a apuntadores de la clase base y la clase derivada. (Parte I de 2).

```

38     cout << "\n\n";
39     empleadoBaseMasComision.imprimir(); // invoca a la función imprimir de la clase derivada
40
41 // orienta el apuntador de la clase base al objeto de la clase base e imprime
42 empleadoPorComisionPtr = &empleadoPorComision; // perfectamente natural
43 cout << "\n\n\nAl llamar a imprimir con el apuntador de clase base al "
44     << "\nobjeto de clase base se invoca la función imprimir de la clase base:\n\n";
45 empleadoPorComisionPtr->imprimir(); // invoca a la función imprimir de la clase base
46
47 // orienta el apuntador de clase derivada al objeto de clase derivada e imprime
48 empleadoBaseMasComisionPtr = &empleadoBaseMasComision; // natural
49 cout << "\n\n\nAl llamar a imprimir con el apuntador de clase derivada al "
50     << "\nobjeto de clase derivada se invoca a la función imprimir "
51     << "de la clase derivada:\n\n";
52 empleadoBaseMasComisionPtr->imprimir(); // invoca a la función imprimir de la clase derivada
53
54 // orienta el apuntador de clase base al objeto de clase derivada e imprime
55 empleadoPorComisionPtr = &empleadoBaseMasComision;
56 cout << "\n\n\nAl llamar a imprimir con el apuntador de clase base al "
57     << "objeto de clase derivada\nse invoca a la función imprimir de la "
58     << "clase base en ese objeto de la clase derivada:\n\n";
59 empleadoPorComisionPtr->imprimir(); // invoca a la función imprimir de la clase base
60 cout << endl;
61 return 0;
62 } // fin de main

```

Impresión de los objetos de clase base y clase derivada:

empleado por comision: Sue Jones
 numero de seguro social: 222-22-2222
 ventas brutas: 10000.00
 tarifa de comision: 0.06

con salario base empleado por comision: Bob Lewis
 numero de seguro social: 333-33-3333
 ventas brutas: 5000.00
 tarifa de comision: 0.04
 salario base: 300.00

Al llamar a imprimir con el apuntador de clase base al
 objeto de clase base se invoca la función imprimir de la clase base:

empleado por comision: Sue Jones
 numero de seguro social: 222-22-2222
 ventas brutas: 10000.00
 tarifa de comision: 0.06

Al llamar a imprimir con el apuntador de clase derivada al
 objeto de clase derivada se invoca a la función imprimir de la clase derivada:

con salario base empleado por comision: Bob Lewis
 numero de seguro social: 333-33-3333
 ventas brutas: 5000.00
 tarifa de comision: 0.04
 salario base: 300.00

Al llamar a imprimir con el apuntador de clase base al objeto de clase derivada
 se invoca a la función imprimir de la clase base en ese objeto de la clase derivada:

empleado por comision: Bob Lewis
 numero de seguro social: 333-33-3333
 ventas brutas: 5000.00
 tarifa de comision: 0.04

Figura 13.5 | Asignación de direcciones de objetos de la clase base y la clase derivada a apuntadores de la clase base y la clase derivada. (Parte 2 de 2).

13.3.2 Cómo orientar los apuntadores de una clase derivada a objetos de la clase base

En la sección 13.3.1, asignamos la dirección de un objeto de la clase derivada a un apuntador de la clase base y explicamos que el compilador de C++ permite esta asignación, debido a que un objeto de una clase derivada *es un* objeto de la clase base. Tomamos la metodología opuesta en la figura 13.6, al orientar un apuntador de la clase derivada a un objeto de la clase base. [Nota: este programa usa las clases `EmpleadoPorComision` y `EmpleadoBaseMasComision` de las figuras 13.1 a 13.4]. Las líneas 8 y 9 de la figura 13.6 crean un objeto `EmpleadoPorComision`, y en la línea 10 se crea un apuntador `EmpleadoBaseMasComision`. En la línea 14 se trata de asignar la dirección del objeto `empleadoPorComision` de la clase base al apuntador `empleadoBaseMasComisionPtr` de la clase derivada, pero el compilador de C++ genera un error. El compilador evita esta asignación, ya que un `EmpleadoPorComision` no es un `EmpleadoBaseMasComision`. Consideré las consecuencias si el compilador permitiera esta asignación. A través de un apuntador `EmpleadoBaseMasComision` podemos invocar cualquier función miembro de `EmpleadoBaseMasComision`, incluyendo `setSalarioBase`, para el objeto al que apunta el apuntador (es decir, el objeto `empleadoPorComision` de la clase base). Sin embargo, el objeto `EmpleadoPorComision` no proporciona una función miembro `setSalarioBase`, ni proporciona un dato miembro `salarioBase` para establecer su valor. Esto podría generar problemas, debido a que la función miembro `setSalarioBase` supondría que hay un dato miembro `salarioBase` que se debe establecer en su “ubicación usual” en un objeto `EmpleadoBaseMasComision`. Esta memoria no pertenece al objeto `EmpleadoPorComision`, por lo que la función miembro `setSalarioBase` podría sobrescribir otros datos importantes en la memoria, posiblemente datos que pertenezcan a un objeto distinto.

```

1 // Fig. 13.6: fig13_06.cpp
2 // Cómo orientar un apuntador de clase derivada a un objeto de clase base.
3 #include "EmpleadoPorComision.h"
4 #include "EmpleadoBaseMasComision.h"
5
6 int main()
7 {
8     EmpleadoPorComision empleadoPorComision(
9         "Sue", "Jones", "222-22-2222", 10000, .06 );
10    EmpleadoBaseMasComision *empleadoBaseMasComisionPtr = 0;
11
12    // orienta el apuntador de la clase derivada al objeto de la clase base
13    // Error: un EmpleadoPorComision no es un EmpleadoBaseMasComision
14    empleadoBaseMasComisionPtr = &empleadoPorComision;
15    return 0;
16 } // fin de main

```

Mensajes de error del compilador de líneas de comandos Borland C++:

```
Error E2034 Fig13_06\fig13_06.cpp 14: Cannot convert 'EmpleadoPorComision *'
to 'EmpleadoBaseMasComision *' in function main()
```

Mensajes de error del compilador GNU C++:

```
fig13_06.cpp:14: error: invalid conversion from 'EmpleadoPorComision*' to
'EmpleadoBaseMasComision*'
```

Mensajes de error del compilador Microsoft Visual C++ 2005:

```
C:\cpphtp6_ejemplos\cap13\fig13_06\fig13_06.cpp(14) : error C2440:
 '=' : cannot convert from 'EmpleadoPorComision *__w64' to
 'EmpleadoBaseMasComision *'
 Cast from base to derived requires dynamic_cast or static_cast
```

Figura 13.6 | Orientación de un apuntador de la clase derivada a un objeto de la clase base.

13.3.3 Llamadas a funciones miembro de una clase derivada a través de apuntadores de la clase base

Desde un apuntador de la clase base, el compilador nos permite invocar sólo a las funciones miembro de la clase base. Por ende, si un apuntador de la clase base se orienta a un objeto de la clase derivada, y se trata de acceder a una *función miembro que sólo pertenezca a la clase derivada*, se producirá un error de compilación.

En la figura 13.7 se muestran las consecuencias de tratar de invocar a una función miembro de la clase derivada desde un apuntador de la clase base. [Nota: estamos usando de nuevo las clases `EmpleadoPorComision` y `EmpleadoBaseMasComision` de las figuras 13.1 a 13.4]. En la línea 9 se crea `empleadoPorComisionPtr` (un apuntador a un objeto `EmpleadoPorComision`) y en las líneas 10 a 11 se crea un objeto `EmpleadoBaseMasComision`. En la línea 14 se orienta `empleadoPorComisionPtr` al objeto de la clase derivada llamado `empleadoBaseMasComision`. En la sección 13.3.1 vimos que esto está permitido, ya que un `EmpleadoBaseMasComision` es un `EmpleadoPorComision` (en el sentido en el que un objeto `EmpleadoBaseMasComision` contiene toda la funcionalidad de un objeto `EmpleadoPorComision`). En las líneas 18 a 22 se invocan las funciones miembro de la clase base `getPrimerNombre`, `getApellidoPaterno`, `getNumeroSeguroSocial`, `getVentasBrutas` y `getTarifaComision` desde el apuntador de la clase base. Todas estas llamadas son legítimas, ya que `EmpleadoBaseMasComision` hereda estas funciones miembro de `EmpleadoPorComision`. Sabemos que `empleadoPorComisionPtr` está orientado a un objeto `EmpleadoBaseMasComision`, por lo que en las líneas 26 y 27 tratamos de invocar a las funciones miembro `EmpleadoBaseMasComision` llamadas `getSalarioBase` y `setSalarioBase`. El compilador genera errores en ambas llamadas, debido a que no se hacen a las funciones miembro de la clase base `EmpleadoPorComision`. El manejador se puede utilizar para invocar sólo a las funciones que son miembros del tipo de clase asociado de ese manejador. (En este caso, desde un `EmpleadoPorComision *` sólo podemos invocar a las funciones miembro de `EmpleadoPorComision` llamadas `setPrimerNombre`, `getPrimerNombre`, `setApellidoPaterno`, `getApellidoPaterno`, `setNumeroSeguroSocial`, `getNumeroSeguroSocial`, `setVentasBrutas`, `getVentasBrutas`, `setTarifaComision`, `getTarifaComision`, `ingresos` e `imprimir`).

```

1 // Fig. 13.7: fig13_07.cpp
2 // Intento de invocar a las funciones miembro que sólo son de
3 // la clase derivada a través de un apuntador de la clase base.
4 #include "EmpleadoPorComision.h"
5 #include "EmpleadoBaseMasComision.h"
6
7 int main()
8 {
9     EmpleadoPorComision *empleadoPorComisionPtr = 0; // clase base
10    EmpleadoBaseMasComision empleadoBaseMasComision(
11        "Bob", "Lewis", "333-33-3333", 5000, .04, 300 ); // clase derivada
12
13    // orienta el apuntador de la clase base al objeto de la clase derivada
14    empleadoPorComisionPtr = &empleadoBaseMasComision;
15
16    // invoca a las funciones miembro de la clase base en el objeto de la
17    // clase derivada a través de un apuntador de la clase base (permitido)
18    string primerNombre = empleadoPorComisionPtr->getPrimerNombre();
19    string apellidoPaterno = empleadoPorComisionPtr->getApellidoPaterno();
20    string nss = empleadoPorComisionPtr->getNumeroSeguroSocial();
21    double ventasBrutas = empleadoPorComisionPtr->getVentasBrutas();
22    double tarifaComision = empleadoPorComisionPtr->getTarifaComision();
23
24    // intento de invocar a las funciones miembro que sólo son de la clase derivada
25    // en un objeto de la clase derivada a través de un apuntador de la clase base (no
26    // permitido)
27    double salarioBase = empleadoPorComisionPtr->getSalarioBase();
28    empleadoPorComisionPtr->setSalarioBase( 500 );
29
30 } // fin de main

```

Figura 13.7 | Intento de invocar a las funciones que sólo son de la clase derivada, a través de un apuntador de la clase base. (Parte 1 de 2).

Mensajes de error del compilador de línea de comandos Borland C++:

```
Error E2316 Fig13_07\fig13_07.cpp 26: 'getSalarioBase' is not a member of
'EmpleadoPorComision' in function main()
Error E2316 Fig13_07\fig13_07.cpp 27: 'setSalarioBase' is not a member of
'EmpleadoPorComision' in function main()
```

Mensajes de error del compilador Microsoft Visual C++ 2005:

```
C:\cpphtp6_ejemplos\cap13\Fig13_07\fig13_07.cpp(26) : error C2039:
  'getSalarioBase' : is not a member of 'EmpleadoPorComision'
    C:\cpphtp6_ejemplos\cap13\Fig13_07\EmpleadoPorComision.h(10) :
      see declaration of 'EmpleadoPorComision'
C:\cpphtp6_ejemplos\cap13\Fig13_07\fig13_07.cpp(27) : error C2039:
  'setSalarioBase' : is not a member of 'EmpleadoPorComision'
    C:\cpphtp6_ejemplos\cap13\Fig13_07\EmpleadoPorComision.h(10) :
      see declaration of 'EmpleadoPorComision'
```

Mensajes de error del compilador GNU C++:

```
fig13_07.cpp:26: error: 'getSalarioBase' undeclared (first use this function)
fig13_07.cpp:26: error: (Each undeclared identifier is reported only once for
each function it appears in.)
fig13_07.cpp:27: error: 'setSalarioBase' undeclared (first use this function)
```

Figura 13.7 | Intento de invocar a las funciones que sólo son de la clase derivada, a través de un apuntador de la clase base. (Parte 2 de 2).

El compilador permite el acceso a los miembros que sólo son de la clase derivada a través de un apuntador que esté orientado a un objeto de la clase derivada, si convertimos de manera explícita el apuntador de la clase base a un apuntador de la clase derivada; una técnica conocida como **conversión descendente**. Como vimos en la sección 13.3.1, es posible orientar un apuntador de la clase base a un objeto de la clase derivada. Sin embargo, como demostramos en la figura 13.7, un apuntador de la clase base se puede usar para invocar sólo las funciones declaradas en la clase base. La conversión descendente permite una operación específica de la clase derivada en un objeto de la clase derivada al que apunta un apuntador de la clase base. Después de una conversión descendente, el programa puede invocar las funciones de la clase derivada que no están en la clase base. En la sección 13.8 le mostraremos un ejemplo concreto de la conversión descendente.



Observación de Ingeniería de Software 13.3

Si la dirección de un objeto de la clase derivada se ha asignado a un apuntador de una de sus clases base directas o indirectas, es aceptable convertir ese apuntador de la clase base de vuelta a un apuntador del tipo de la clase derivada. De hecho, esto debe hacerse para enviar los mensajes de ese objeto de la clase derivada que no aparecen en la clase base.

13.3.4 Funciones virtuales

En la sección 13.3.1 orientamos un apuntador de la clase base `EmpleadoPorComision` a un objeto de la clase derivada `EmpleadoBaseMasComision`, y después invocamos a la función miembro `imprimir` a través de ese apuntador. Recuerde que el tipo del manejador determina cuál funcionalidad de la clase se va a invocar. En este caso, el apuntador `EmpleadoPorComision` invocó a la función miembro `imprimir` de `EmpleadoPorComision` en el objeto `EmpleadoBaseMasComision`, aun y cuando el apuntador estaba orientado a un objeto `EmpleadoBaseMasComision` que tiene su propia función `imprimir` personalizada. *Con las funciones virtual, el tipo del objeto al que se está apuntando, y no el tipo del manejador, es el que determina cuál versión de una función virtual se debe invocar.*

Primero vamos a considerar por qué son útiles las funciones `virtual`. Suponga que un conjunto de clases de figuras como `Circulo`, `Triangulo`, `Rectangulo` y `Cuadrado` se derivan de la clase base `Figura`. Cada una de estas clases podría estar dotada con la habilidad de dibujarse a sí misma a través de una función miembro llamada `dibujar`. Aunque cada clase tiene su propia función `dibujar`, la función para cada figura es bastante distinta. En un programa que dibuja un conjunto de figuras, podría ser útil poder tratar a todas las figuras en forma genérica como objetos de la clase base `Figura`.

Después, para dibujar cualquier figura podríamos simplemente usar un apuntador de la clase base `Figura` para invocar a la función `dibujar`, y dejar que el programa determine en forma **dinámica** (es decir, en tiempo de ejecución) de cuál clase derivada se va a utilizar la función `dibujar`, con base en el tipo del objeto al que apunta el apuntador de la clase base `Figura` en cualquier momento dado.

Para permitir este tipo de comportamiento, declaramos a `dibujar` en la clase base como una **función virtual**, y **sobrescribimos** a `dibujar` en cada una de las clases derivadas para dibujar la figura apropiada. Desde una perspectiva de implementación, sobrescribir una función no es algo distinto a redefinirla (que es la metodología que hemos estado usando hasta ahora). Una función sobrescrita en una clase derivada tiene la misma firma y el mismo tipo de valor de retorno (es decir, prototipo) que la función que sobrescribe en su clase base. Si no declaramos la función de la clase base como `virtual`, podemos redefinir esa función. En contraste, si declaramos la función de la clase base como `virtual`, podemos sobrescribir esa función para permitir el comportamiento polimórfico. Para declarar una función `virtual`, anteponemos al prototipo de la función la palabra clave `virtual` en la clase base. Por ejemplo,

```
virtual void dibujar() const;
```

aparecería en la clase base `Figura`. El prototipo anterior declara que la función `dibujar` es una función `virtual` que no recibe argumentos y no devuelve nada. Esta función se declara `const` debido a que, por lo general, una función `dibujar` no realizaría modificaciones al objeto `Figura` en el cual se invoca; las funciones virtuales no tienen que ser funciones `const`.



Observación de Ingeniería de Software 13.4

Una vez que una función se declara `virtual`, permanece `virtual` en todos los niveles hacia abajo de la jerarquía desde ese punto, aun si esa función no se declara explícitamente como `virtual` cuando una clase derivada la sobrescribe.



Buena práctica de programación 13.1

Cuando un programador explora una jerarquía de clases para localizar una clase y reutilizarla, es posible que una función en esa clase exhiba un comportamiento de función `virtual` aun y cuando no se declare explícitamente como `virtual`. Esto ocurre cuando la clase hereda una función `virtual` de su clase base, y puede producir ligeros errores sutiles. Dichos errores se pueden evitar al declarar explícitamente todas las funciones `virtual` como `virtual` a través de la jerarquía de herencia.



Observación de Ingeniería de Software 13.5

Cuando una clase derivada opta por no sobrescribir una función `virtual` de su clase base, la clase derivada simplemente hereda la implementación de la función `virtual` de su clase base.

Si un programa invoca a una función `virtual` a través de un apuntador de la clase base a un objeto de la clase derivada (por ejemplo, `figuraPtr->dibujar()`), el programa elije la función `dibujar` correcta de la clase derivada en forma dinámica (es decir, en tiempo de ejecución) con base en el tipo del objeto, no en el tipo del apuntador. Al proceso de elegir la función apropiada a llamar en tiempo de ejecución (en vez de hacerlo en tiempo de compilación) se le conoce como **vinculación dinámica** o **vinculación en tiempo de ejecución**.

Cuando una función `virtual` se llama al referenciar un objeto específico por su nombre y usando el operador punto de selección de miembros (por ejemplo, `objetoCuadrado.dibujar()`), la invocación de la función se resuelve en tiempo de compilación (a esto se le conoce como **vinculación estática**) y la función virtual que se llama es la que se define para (o se hereda por) la clase de ese objeto específico; esto no es comportamiento polimórfico. Por ende, la vinculación dinámica con las funciones `virtual` sólo ocurre a partir de manejadores tipo apuntador (y, como pronto veremos, de referencias).

Ahora veamos cómo las funciones `virtual` pueden permitir el comportamiento polimórfico en nuestra jerarquía de empleados. Las figuras 13.8 y 13.9 son los archivos de encabezado para las clases `EmpleadoPorComision` y `EmpleadoBaseMasComision`, respectivamente. Observe que la única diferencia entre estos archivos y los de las figuras 13.1 y 13.3 es que especificamos las funciones miembro `ingresos` e `imprimir` de cada clase como `virtual` (líneas 30 y 31 de la figura 13.8, y líneas 21 y 22 de la figura 13.9). Como las funciones `ingresos` e `imprimir` son `virtual` en la clase `EmpleadoPorComision`, las funciones `ingresos` e `imprimir` de la clase `EmpleadoBaseMasComision` sobrescriben a las de la clase `EmpleadoPorComision`. Ahora, si orientamos un apuntador de la clase base `EmpleadoPorComision` a un objeto de la clase derivada `EmpleadoBaseMasComision`, y el programa utiliza ese apuntador para llamar a la función `ingresos` o `imprimir`, se invocará la función correspondiente del objeto `EmpleadoBaseMasComision`. No hubo modificaciones a las implementaciones de las funciones miembro de las clases `EmpleadoPorComision` y `EmpleadoBaseMasComision`, por lo que reutilizamos las versiones de las figuras 13.2 y 13.4.

Modificamos la figura 13.5 para crear el programa de la figura 13.10. En las líneas 46 a 57 se demuestra otra vez que un apuntador `EmpleadoPorComision` orientado a un objeto `EmpleadoPorComision` puede utilizarse para invocar

la funcionalidad de `EmpleadoPorComision`, y que un apuntador `EmpleadoBaseMasComision` orientado a un objeto `EmpleadoBaseMasComision` puede utilizarse para invocar la funcionalidad de `EmpleadoBaseMasComision`. En la línea 60 se orienta el apuntador `empleadoPorComisionPtr` de la clase base al objeto `empleadoBaseMasComision` de la clase derivada. Observe que cuando en la línea 67 se invoca a la función miembro `imprimir` desde el apuntador de la clase base, se invoca a la función miembro `imprimir` de la clase derivada `EmpleadoBaseMasComision`, por lo que en la línea 67 se imprime un texto distinto al de la línea 59 en la figura 13.5 (cuando la función miembro `print` no se declaró `virtual`).

```

1 // Fig. 13.8: EmpleadoPorComision.h
2 // Definición de la clase EmpleadoPorComision que representa a un empleado por comisión.
3 #ifndef COMISION_H
4 #define COMISION_H
5
6 #include <string> // clase string estándar de C++
7 using std::string;
8
9 class EmpleadoPorComision
10 {
11 public:
12     EmpleadoPorComision( const string &, const string &, const string &,
13                          double = 0.0, double = 0.0 );
14
15     void setPrimerNombre( const string & ); // establece el primer nombre
16     string getPrimerNombre() const; // devuelve el primer nombre
17
18     void setApellidoPaterno( const string & ); // establece el apellido paterno
19     string getApellidoPaterno() const; // devuelve el apellido paterno
20
21     void setNumeroSeguroSocial( const string & ); // establece el NSS
22     string getNumeroSeguroSocial() const; // devuelve el NSS
23
24     void setVentasBrutas( double ); // establece el monto de ventas brutas
25     double getVentasBrutas() const; // devuelve el monto de ventas brutas
26
27     void setTarifaComision( double ); // establece la tarifa de comisión
28     double getTarifaComision() const; // devuelve la tarifa de comisión
29
30     virtual double ingresos() const; // calcula los ingresos
31     virtual void imprimir() const; // imprime el objeto EmpleadoPorComision
32 private:
33     string primerNombre;
34     string apellidoPaterno;
35     string numeroSeguroSocial;
36     double ventasBrutas; // ventas brutas por semana
37     double tarifaComision; // porcentaje de comisión
38 }; // fin de la clase EmpleadoPorComision
39
40 #endif

```

Figura 13.8 | Archivo de encabezado de la clase `EmpleadoPorComision`, que declara a las funciones `ingresos` e `imprimir` como `virtual`.

```

1 // Fig. 13.9: EmpleadoBaseMasComision.h
2 // Clase EmpleadoBaseMasComision derivada de la clase
3 // EmpleadoPorComision.
4 #ifndef BASEMAS_H
5 #define BASEMAS_H
6

```

Figura 13.9 | Archivo de encabezado de la clase `EmpleadoBaseMasComision` que declara a las funciones `ingresos` e `imprimir` como `virtual`. (Parte I de 2).

```

7 #include <string> // clase string estándar de C++
8 using std::string;
9
10 #include "EmpleadoPorComision.h" // declaración de la clase EmpleadoPorComision
11
12 class EmpleadoBaseMasComision : public EmpleadoPorComision
13 {
14 public:
15     EmpleadoBaseMasComision( const string &, const string &,
16                             const string &, double = 0.0, double = 0.0 );
17
18     void setSalarioBase( double ); // establece el salario base
19     double getSalarioBase() const; // devuelve el salario base
20
21     virtual double ingresos() const; // calcula los ingresos
22     virtual void imprimir() const; // imprime el objeto EmpleadoBaseMasComision
23 private:
24     double salarioBase; // salario base
25 }; // fin de la clase EmpleadoBaseMasComision
26
27 #endif

```

Figura 13.9 | Archivo de encabezado de la clase `EmpleadoBaseMasComision` que declara a las funciones `ingresos` e `imprimir` como `virtual`. (Parte 2 de 2).

Podemos ver que, al declarar una función miembro `virtual`, el programa determina en forma dinámica cuál función debe invocar con base en el tipo de objeto al que apunta el manejador, en vez de basarse en el tipo del manejador. Observe de nuevo que cuando `empleadoPorComisionPtr` apunta a un objeto `EmpleadoPorComision` (línea 46), se invoca a la función `imprimir` de la clase `EmpleadoPorComision`, y cuando `empleadoPorComisionPtr` apunta a un objeto `EmpleadoBaseMasComision`, se invoca a la función `imprimir` de `EmpleadoBaseMasComision`. Por ende, el mismo mensaje (`imprimir`, en este caso) que se envía (desde un apuntador de la clase base) a una variedad de objetos relacionados por la herencia a esa clase base, toma muchas formas; éste es el comportamiento polimórfico.

```

1 // Fig. 13.10: fig13_10.cpp
2 // Introducción al polimorfismo, las funciones virtuales y la vinculación postergada.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 // incluye las definiciones de las clases
12 #include "EmpleadoPorComision.h"
13 #include "EmpleadoBaseMasComision.h"
14
15 int main()
16 {
17     // crea un objeto de la clase base
18     EmpleadoPorComision empleadoPorComision(
19         "Sue", "Jones", "222-22-2222", 10000, .06 );
20
21     // crea un apuntador de la clase base
22     EmpleadoPorComision *empleadoPorComisionPtr = 0;
23
24     // crea un objeto de la clase derivada

```

Figura 13.10 | Demostración del polimorfismo al invocar una función `virtual` de la clase derivada a través de un apuntador de la clase base a un objeto de la clase derivada. (Parte 1 de 3).

```

25     EmpleadoBaseMasComision empleadoBaseMasComision(
26         "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
27
28     // crea un apuntador de la clase derivada
29     EmpleadoBaseMasComision *empleadoBaseMasComisionPtr = 0;
30
31     // establece el formato de salida de punto flotante
32     cout << fixed << setprecision( 2 );
33
34     // imprime los objetos usando la vinculación estática
35     cout << "Invocando a la funcion imprimir en objetos de la clase base "
36         << "\ny la clase derivada con vinculacion estatica\n\n";
37     empleadoPorComision.imprimir(); // vinculación estática
38     cout << "\n\n";
39     empleadoBaseMasComision.imprimir(); // vinculación estática
40
41     // imprime los objetos usando vinculación dinámica
42     cout << "\n\n\nInvocando a la funcion imprimir en objetos de la clase base "
43         << "y la \ncclase derivada con vinculacion dinamica";
44
45     // orienta el apuntador de la clase base al objeto de la clase base e imprime
46     empleadoPorComisionPtr = &empleadoPorComision;
47     cout << "\n\nAl llamar a la funcion virtual imprimir con un apuntador"
48         << "\nde la clase base a un objeto de la clase base se invoca a la "
49         << "funcion imprimir de la clase base:\n\n";
50     empleadoPorComisionPtr->imprimir(); // invoca a la función imprimir de la clase base
51
52     // orienta un apuntador de la clase derivada al objeto de la clase derivada e imprime
53     empleadoBaseMasComisionPtr = &empleadoBaseMasComision;
54     cout << "\n\nAl llamar a la funcion virtual imprimir con un apuntador "
55         << "de la clase derivada\ncna un objeto de la clase base se invoca a "
56         << "la funcion imprimir de la clase derivada:\n\n";
57     empleadoBaseMasComisionPtr->imprimir(); // invoca a la función imprimir de la clase
58     derivada
59
60     // orienta un apuntador de la clase base a un objeto de la clase derivada e imprime
61     empleadoPorComisionPtr = &empleadoBaseMasComision;
62     cout << "\n\nAl llamar a la funcion virtual imprimir con un apuntador de la clase base"
63         << "\ncna un objeto de la clase derivada se invoca a la funcion "
64         << "imprimir de la clase derivada:\n\n";
65
66     // polimorfismo; invoca a la función imprimir de EmpleadoBaseMasComision;
67     // apuntador de la clase base a un objeto de la clase derivada
68     empleadoPorComisionPtr->imprimir();
69     cout << endl;
70 } // fin de main

```

Invocando a la función imprimir en objetos de la clase base
y la clase derivada con vinculacion estatica

```

empleado por comision: Sue Jones
numero de seguro social: 222-22-2222
ventas brutas: 10000.00
tarifa de comision: 0.06

con salario base empleado por comision: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa de comision: 0.04
salario base: 300.00

```

Figura 13.10 | Demostración del polimorfismo al invocar una función `virtual` de la clase derivada a través de un apuntador de la clase base a un objeto de la clase derivada. (Parte 2 de 3).

Invocando a la función `imprimir` en objetos de la clase base y la clase derivada con vinculación dinámica

Al llamar a la función virtual `imprimir` con un apuntador de la clase base a un objeto de la clase base se invoca a la función `imprimir` de la clase base:

```
empleado por comision: Sue Jones
numero de seguro social: 222-22-2222
ventas brutas: 10000.00
tarifa de comision: 0.06
```

Al llamar a la función virtual `imprimir` con un apuntador de la clase derivada a un objeto de la clase base se invoca a la función `imprimir` de la clase derivada:

```
con salario base empleado por comision: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa de comision: 0.04
salario base: 300.00
```

Al llamar a la función virtual `imprimir` con un apuntador de la clase base a un objeto de la clase derivada se invoca a la función `imprimir` de la clase derivada:

```
con salario base empleado por comision: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa de comision: 0.04
salario base: 300.00
```

Figura 13.10 | Demostración del polimorfismo al invocar una función `virtual` de la clase derivada a través de un apuntador de la clase base a un objeto de la clase derivada. (Parte 3 de 3).

13.3.5 Resumen de las asignaciones permitidas entre objetos y apuntadores de la clase base y de la clase derivada

Ahora que hemos visto una aplicación completa que procesa diversos objetos en forma polimórfica, sintetizaremos lo que se puede hacer y lo que no se puede hacer con los objetos y apuntadores de la clase base y las clases derivadas. Aunque un objeto de la clase derivada también *es un* objeto de la clase base, los dos objetos son sin embargo distintos. Como vimos antes, los objetos de la clase derivada se pueden tratar como si fueran objetos de la clase base. Ésta es una relación lógica, ya que la clase derivada contiene todos los miembros de la clase base. Sin embargo, los objetos de la clase base no pueden tratarse como si fueran objetos de la clase derivada; la clase derivada puede tener miembros adicionales que sólo pertenezcan a ésta. Por esta razón, orientar un apuntador de la clase derivada a un objeto de la clase base no se permite sin una conversión explícita; dicha asignación dejaría a los miembros que sólo pertenecen a la clase derivada indefinidos en el objeto de la clase base. La conversión exime al compilador de la responsabilidad de generar un mensaje de error. En cierto sentido, al usar la conversión estamos diciendo “Sé que lo que estoy haciendo es peligroso y asumo toda la responsabilidad por mis acciones”.

En la sección actual y en el capítulo 12 hemos visto cuatro formas de orientar los apuntadores de la clase base y los apuntadores de la clase derivada a objetos de la clase base y objetos de la clase derivada:

1. Orientar un apuntador de la clase base a un objeto de la clase base es un proceso simple y directo: las llamadas realizadas desde el apuntador de la clase base simplemente invocan la funcionalidad de la clase base.
2. Orientar un apuntador de la clase derivada a un objeto de la clase derivada es un proceso simple y directo: las llamadas realizadas desde el apuntador de la clase derivada simplemente invocan la funcionalidad de la clase derivada.
3. Orientar un apuntador de la clase base a un objeto de la clase derivada es un proceso seguro, ya que el objeto de la clase derivada *es un* objeto de su clase base. Sin embargo, este apuntador se puede utilizar para invocar sólo a las funciones miembro de la clase base. Si tratamos de hacer referencia a un miembro que sólo pertenezca a la clase derivada a través del apuntador de la clase base, el compilador reporta un error. Para evitar este error, debemos convertir el apuntador de la clase base a un apuntador de la clase derivada. Así, el apuntador de la clase derivada se puede utilizar para invocar la funcionalidad completa del objeto de la clase derivada; en la sección 13.8 se demuestra cómo utilizar la conversión descendente con seguridad. Si se define una función `virtual` en las clases

base y derivada (ya sea por herencia o al sobreescribirla), y si esa función se invoca en un objeto de la clase derivada a través de un apuntador de la clase base, entonces se llama a la versión de esa función correspondiente a la clase derivada. Esto es un ejemplo del comportamiento polimórfico que ocurre sólo con las funciones `virtual`.

- Orientar un apuntador de la clase derivada a un objeto de la clase base es un proceso que genera un error de compilación. La relación “*es un*” se aplica sólo de una clase derivada a sus clases base directa e indirecta, y no al revés. Un objeto de la clase base no contiene los miembros que sólo pertenecen a la clase derivada, y que se pueden invocar desde un apuntador de la clase derivada.

Error común de programación 13.1

Después de orientar un apuntador de la clase base a un objeto de la clase derivada, tratar de hacer referencia a los miembros que sólo pertenecen a la clase derivada con el apuntador de la clase base es un error de compilación.

Error común de programación 13.2

Tratar a un objeto de la clase base como un objeto de la clase derivada puede producir errores.

13.4 Tipos de campos e instrucciones `switch`

Una manera de determinar el tipo de un objeto que se incorpora en un programa más grande es utilizar una instrucción `switch`. Esto nos permite diferenciar entre los tipos de los objetos, y después invocar una acción apropiada para un objeto específico. Por ejemplo, en una jerarquía de figuras en las que cada objeto figura tiene un atributo `tipoFigura`, una instrucción `switch` podría comprobar el `tipoFigura` para determinar cuál función `imprimir` debe llamar.

El uso de la lógica de `switch` expone a los programas a una variedad de problemas potenciales. Por ejemplo, el programador podría olvidar incluir una prueba de tipos cuando sea obligatorio hacerla, o podría olvidar evaluar todos los casos posibles en una instrucción `switch`. Al modificar un sistema basado en `switch` agregando nuevos tipos, el programador podría olvidar insertar los nuevos casos en todas las instrucciones `switch` relevantes. Cada adición o eliminación de una clase requiere la modificación de todas las instrucciones `switch` en el sistema; rastrear estas instrucciones puede ser un proceso que consuma mucho tiempo y esté propenso a errores.

Observación de Ingeniería de Software 13.6

La programación polimórfica puede eliminar la necesidad de la lógica de `switch`. Al utilizar el mecanismo polimórfico para realizar la lógica equivalente, los programadores pueden evitar el tipo de errores que se asocian generalmente con la lógica de `switch`.

Observación de Ingeniería de Software 13.7

Una consecuencia interesante de utilizar el polimorfismo es que los programas toman una apariencia simplificada. Contienen menos lógica de bifurcación y un código secuencial más simple. Esta simplificación facilita la prueba, depuración y mantenimiento de los programas.

13.5 Clases abstractas y funciones `virtual` puras

Cuando pensamos en una clase como un tipo, suponemos que los programas crearán objetos de ese tipo. Sin embargo, hay casos en los que es útil definir las clases de las cuales nunca se tendrá la intención de instanciar objetos. Dichas clases se conocen como **clases abstractas**. Como estas clases se utilizan comúnmente como clases base en las jerarquías de herencia, nos referimos a ellas como **clases base abstractas**. Estas clases no se pueden utilizar para instanciar objetos ya que, como veremos pronto, las clases abstractas están incompletas; las clases derivadas deben definir las “piezas faltantes”. En la sección 13.6 crearemos programas con clases abstractas.

El propósito de una clase abstracta es proporcionar una clase base apropiada, a partir de la cual otras clases puedan heredar. Las clases que se pueden utilizar para instanciar objetos se conocen como **clases concretas**. Dichas clases proporcionan implementaciones de todas las funciones miembro que definen. Podríamos tener una clase abstracta llamada `FiguraBidimensional` y derivar de ella las clases concretas tales como `Cuadrado`, `Círculo` y `Triángulo`. Podríamos tener también una clase base abstracta llamada `FiguraTridimensional` y derivar de ella las clases concretas tales como `Cubo`, `Esfera` y `Cilindro`. Las clases base abstractas son demasiado genéricas como para definir objetos reales; necesitamos ser más específicos antes de poder pensar en instanciar objetos. Por ejemplo, si alguien nos dice que “dibujemos la figura bidimensional”, ¿qué figura dibujaríamos? Las clases concretas proporcionan los detalles específicos que hacen que sea razonable instanciar objetos.

Una jerarquía de herencia no necesita contener clases abstractas pero, como veremos, muchos sistemas orientados a objetos tienen jerarquías de clases encabezadas por clases base abstractas. En ciertos casos, las clases abstractas constituyen unos cuantos niveles superiores de la jerarquía. Un buen ejemplo de ello es la jerarquía de figuras en la figura 12.3, la cual empieza con la clase base abstracta **Figura**. En el siguiente nivel de la jerarquía tenemos dos clases base abstractas más, a saber, **FiguraBidimensional** y **FiguraTridimensional**. El siguiente nivel de la jerarquía define clases concretas para las figuras bidimensionales (a saber, **Círculo**, **Cuadrado** y **Triángulo**) y para las figuras tridimensionales (a saber, **Esfera**, **Cubo** y **Tetraedro**).

Para hacer una clase abstracta, se declara una o más de sus funciones **virtual** como “puras”. Una **función virtual pura** se especifica colocando “=0” en su declaración, como en

```
virtual void dibujar() const = 0; // función virtual pura
```

El “=0” se conoce como un **especificador puro**. Las funciones **virtual** puras no proporcionan implementaciones. Cada clase derivada concreta *debe* sobrescribir a todas las funciones **virtual** puras de la clase base con implementaciones concretas de esas funciones. La diferencia entre una función **virtual** y una función **virtual** pura es que una función **virtual** tiene una implementación y proporciona a la clase derivada la *opción* de sobrescribir la función; en contraste, una función **virtual** pura no proporciona una implementación y *requiere* que la clase derivada sobrescriba la función para que esa clase derivada sea concreta; en caso contrario, la clase derivada permanece abstracta.

Las funciones **virtual** puras se utilizan cuando no tiene sentido para la clase base tener una implementación de una función, pero es conveniente que todas las clases derivadas concretas implementen la función. Regresando a nuestro ejemplo anterior de los objetos espaciales, no tiene sentido para la clase base **ObjetoEspacial** tener una implementación para la función **dibujar** (ya que no hay forma de dibujar un objeto espacial genérico sin tener más información acerca del tipo de objeto espacial que se va a dibujar). Un ejemplo de una función que se definiría como **virtual** (y no como **virtual** pura) sería una que devuelve un nombre para el objeto. Podemos nombrar a un **ObjetoEspacial** genérico (por ejemplo, como “**objeto espacial**”), por lo que se puede proporcionar una implementación predeterminada para esta función, y la función no necesita ser **virtual** pura. Sin embargo, la función se sigue declarando como **virtual**, ya que se espera que las clases derivadas sobrescriban esta función para proporcionar nombres más específicos para los objetos de la clase derivada.



Observación de Ingeniería de Software 13.8

Una clase abstracta define una interfaz pública común para las diversas clases en una jerarquía de clases. Una clase abstracta contiene una o más funciones virtual puras que las clases derivadas concretas deben sobrescribir.



Error común de programación 13.3

Tratar de instanciar un objeto de una clase abstracta produce un error de compilación.



Error común de programación 13.4

Si no se sobrescribe una función virtual pura en una clase derivada y después se trata de instanciar objetos de esa clase, se produce un error de compilación.



Observación de Ingeniería de Software 13.9

Una clase abstracta tiene por lo menos una función virtual pura. Una clase abstracta también puede tener datos miembro y funciones concretas (incluyendo los constructores y destructores), que están sujetos a las reglas normales de la herencia por las clases derivadas.

Aunque no podemos instanciar objetos de una clase base abstracta *sí podemos* usar la clase base abstracta para declarar apuntadores y referencias que puedan referirse a objetos de cualquier clase concreta derivada de la clase abstracta. Por lo general, los programas utilizan dichos apuntadores y referencias para manipular los objetos de la clase derivada mediante el polimorfismo.

Considere otra aplicación del polimorfismo. Un administrador de pantalla necesita mostrar una variedad de objetos, incluyendo nuevos tipos de objetos que se agregarán al sistema después de escribir el administrador de pantalla. El sistema podría necesitar mostrar varias figuras, como objetos **Círculo**, **Triángulo** o **Rectángulo**, que se derivan de la clase base abstracta **Figura**. El administrador de pantalla utiliza apuntadores **Figura** para administrar los objetos que se muestran. Para dibujar cualquier objeto (sin importar el nivel en el que aparece la clase del objeto en la jerarquía de herencia), el administrador de pantalla utiliza un apuntador de la clase base al objeto para invocar a la función **dibujar** del objeto,

que es una función `virtual` pura en la clase base `Figura`; por lo tanto, cada clase derivada concreta debe implementar la función `dibujar`. Cada objeto `Figura` en la jerarquía de herencia sabe cómo dibujarse a sí mismo. El administrador de pantalla no tiene que preocuparse por el tipo de cada objeto, o si el administrador de pantalla ha encontrado alguna vez objetos de ese tipo.

El polimorfismo es particularmente efectivo para implementar sistemas de software en niveles. Por ejemplo, en los sistemas operativos cada tipo de dispositivo físico podría operar en forma muy distinta de los otros. Aun así, los comandos para `leer` o `escribir` datos desde y hacia los dispositivos podrían tener cierta uniformidad. El mensaje `escribir` que se envía a un objeto controlador de dispositivos necesita interpretarse de manera específica en el contexto de ese controlador de dispositivo, y en la forma en que ese controlador de dispositivo manipula dispositivos de un tipo específico. Sin embargo, la llamada `escribir` en sí en realidad no es distinta de la `escritura` a cualquier otro dispositivo en el sistema; se coloca cierto número de bytes de memoria en ese dispositivo. Un sistema operativo orientado a objetos podría utilizar una clase base abstracta para proporcionar una interfaz apropiada para todos los controladores de dispositivos. Después, a través de la herencia de esa clase base abstracta, se forman clases derivadas que operen todas de manera similar. Las herramientas (es decir, funciones `public`) ofrecidas por los controladores de dispositivos se proporcionan como funciones `virtual` puras en la clase base abstracta. Las implementaciones de esas funciones `virtual` puras se proporcionan en las clases derivadas que corresponden a los tipos específicos de controladores de dispositivos. Esta arquitectura también permite agregar nuevos dispositivos a un sistema con facilidad, aun después de que se haya definido el sistema operativo. El usuario simplemente puede conectar el dispositivo e instalar el controlador de su nuevo dispositivo. El sistema operativo “habla” con este nuevo dispositivo a través de su controlador de dispositivos, que tiene las mismas funciones miembro `public` que todos los demás controladores de dispositivos; aquellas definidas en la clase base abstracta del controlador de dispositivos.

Es común en la programación orientada a objetos definir una **clase iteradora** que pueda recorrer todos los objetos en un contendor (como un arreglo). Por ejemplo, un programa puede imprimir una lista de objetos en un `vector`, para lo cual crea un objeto iterador y después utiliza el iterador para obtener el siguiente elemento de la lista cada vez que se llama al iterador. A menudo, los iteradores se utilizan en la programación polimórfica para recorrer un arreglo o una lista enlazada de apuntadores a objetos de varios niveles de una jerarquía. Los apuntadores en una lista de este tipo son todos apuntadores de la clase base. (En el capítulo 22, Biblioteca de plantillas estándar (STL), se presenta un tratamiento detallado sobre los iteradores). Una lista de apuntadores a objetos de la clase base `FiguraBidimensional` podría contener apuntadores a objetos de las clases `Cuadrado`, `Círculo`, `Triángulo`, etcétera. Al usar el polimorfismo para enviar un mensaje `dibujar`, desde un apuntador `FiguraBidimensional *` a cada objeto en la lista, se dibujaría cada objeto correctamente en la pantalla.

13.6 Ejemplo práctico: sistema de nómina mediante el uso de polimorfismo

En esta sección volvemos a examinar la jerarquía `EmpleadoPorComision-EmployeeBaseMasComision` que exploramos a lo largo de la sección 12.4. En este ejemplo utilizamos una clase abstracta y polimorfismo para realizar cálculos de nómina con base en el tipo de empleado. Creamos una jerarquía de empleados mejorada para resolver el siguiente problema:

Una empresa paga a sus empleados por semana. Los empleados son de cuatro tipos: los empleados asalariados reciben un salario semanal fijo, sin importar el número de horas trabajadas; los empleados por hora reciben un salario por hora y un pago extra por las horas trabajadas que excedan a las 40 horas; los empleados por comisión reciben un porcentaje de sus ventas y los empleados por comisión con salario base reciben un salario base, más un porcentaje de sus ventas. Para el periodo de pago actual, la empresa ha decidido recompensar a los empleados con salario base más comisión, agregando un 10 porciento a sus salarios base. La empresa desea implementar un programa en C++ que realice sus cálculos de nómina por medio del polimorfismo.

Utilizamos la clase abstracta `Empleado` para representar el concepto general de un empleado. Las clases que se derivan directamente de `Empleado` son: `EmpleadoAsalariado`, `EmpleadoPorComision` y `EmpleadoPorHoras`. La clase `EmpleadoBaseMasComision` (que se deriva de `EmpleadoPorComision`) representa el último tipo de empleado. El diagrama de clases de UML de la figura 13.11 muestra la jerarquía de herencia para nuestra aplicación de nómina de empleados polimórfica. Observe que el nombre de la clase abstracta `Empleado` está en cursiva, según la convención del UML.

La clase base abstracta `Empleado` declara la “interfaz” para la jerarquía; es decir, el conjunto de funciones miembro que un programa puede invocar en todos los objetos `Empleado`. Cada empleado, sin importar la forma en que se calculen sus ingresos, tiene un primer nombre, un apellido paterno y un número de seguro social, por lo que aparecen los datos miembro `private` `primerNombre`, `apellidoPaterno` y `numeroSeguroSocial` en la clase base abstracta `Empleado`.

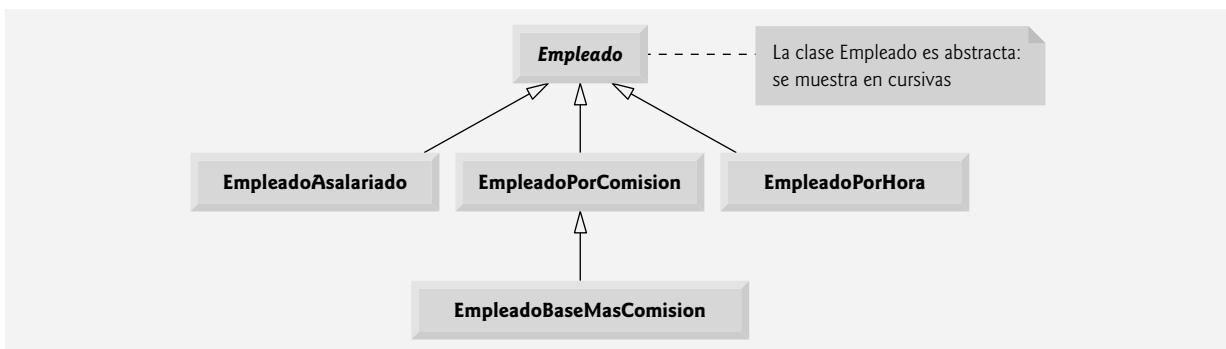


Figura 13.11 | Diagrama de clases de UML de la jerarquía `Empleado`.



Observación de Ingeniería de Software 13.10

Una clase derivada puede heredar la interfaz o implementación de una clase base. Las jerarquías diseñadas para la herencia de implementación tienden a tener su funcionalidad en un nivel alto en la jerarquía; cada nueva clase derivada hereda una o más funciones miembro que se definieron en una clase base, y la clase derivada utiliza las definiciones de la clase base. Las jerarquías diseñadas para la herencia de interfaz tienden a tener su funcionalidad en un nivel bajo en la jerarquía; una clase base especifica una o más funciones que deben definirse para cada clase en la jerarquía (es decir, tienen el mismo prototipo), pero las clases derivadas individuales proporcionan sus propias implementaciones de la(s) función(es).

En las siguientes secciones se implementa la jerarquía de la clase `Empleado`. Las primeras cinco secciones implementan cada una de las clases abstractas o concretas. La última sección implementa un programa de prueba que crea objetos de todas estas clases y procesa los objetos mediante el polimorfismo.

13.6.1 Creación de la clase base abstracta `Empleado`

La clase `Empleado` (figuras 13.13 y 13.14, que veremos con detalle en breve) proporciona las funciones `ingresos` e `imprimir`, además de varias funciones `get` y `set` que manipulan los datos miembro de `Empleado`. Una función `ingresos` se aplica evidentemente en forma genérica a todos los empleados, pero cada cálculo de los ingresos depende de la clase del empleado. Por lo tanto, declaramos `ingresos` como `virtual` pura en la clase base `Empleado` debido a que una implementación predeterminada no tiene sentido para esa función; no hay suficiente información para determinar qué cantidad debe devolver `ingresos`. Cada clase derivada sobrescribe a `ingresos` con una implementación apropiada. Para calcular los ingresos de un empleado, el programa asigna la dirección de un objeto empleado a un apuntador de la clase base `Empleado`, y después invoca a la función `ingresos` en ese objeto. Mantenemos un vector de apuntadores `Empleado`, cada uno de los cuales apunta a un objeto `Empleado` (desde luego, no puede haber objetos `Empleado` debido a que es una clase base abstracta; sin embargo, debido a la herencia todos los objetos de todas las clases derivadas de `Empleado` pueden considerarse como objetos `Empleado`). El programa itera a través del vector y llama a la función `ingresos` para cada objeto `Empleado`. C++ procesa estas llamadas a funciones en forma polimórfica. Al incluir a `ingresos` como una función `virtual` pura en `Empleado`, se obliga a todas las clases derivadas directas de `Empleado` que deseen ser una clase concreta a sobreescibir `ingresos`. Esto permite al diseñador de la jerarquía de clases exigir que cada clase derivada proporcione un cálculo apropiado del pago, si esa clase derivada va a ser sin duda concreta.

La función `imprimir` en la clase `Empleado` muestra el primer nombre, apellido paterno y número de seguro social del empleado. Como veremos, cada clase derivada de `Empleado` sobrescribe a la función `imprimir` para mostrar el tipo del empleado (por ejemplo, "empleado asalariado:"), seguido del resto de la información del empleado.

El diagrama de la figura 13.12 muestra a cada una de las cinco clases en la jerarquía hacia abajo en el lado izquierdo, y a las funciones `ingresos` e `imprimir` a lo largo de la parte superior. Para cada clase, el diagrama muestra los resultados deseados de cada función. Observe que la clase `Empleado` especifica "`=0`" para la función `ingresos`, para indicar que es una función `virtual` pura. Cada clase derivada sobrescribe a esta función para proporcionar una implementación apropiada. No listamos las funciones `get` y `set` de la clase base `Empleado` debido a que no se sobreesciben en ninguna de las clases derivadas; cada una de estas funciones es heredada y utilizada "así como está" por cada una de las clases derivadas.

Vamos a considerar el archivo de encabezado de la clase `Empleado` (figura 13.13). Las funciones miembro `public` incluyen un constructor que recibe el primer nombre, apellido paterno y número de seguro social como argumentos (línea 12); funciones `set` que establecen el primer nombre, apellido paterno y número de seguro social (líneas 14, 17 y 20,

	ingresos	imprimir
Empleado	= 0	primerNombre apellidoPaterno número de seguro social: NSS
Empleado-Asalariado	salarioSemanal	empleado asalariado: primerNombre apellidoPaterno número de seguro social: NSS salario semanal: salariosemanal
Empleado-PorHoras	$\begin{aligned} & \text{Si } horas \leq 40 \\ & \quad sueldo * horas \\ & \text{Si } horas > 40 \\ & \quad (40 * sueldo) + \\ & \quad ((horas - 40) \\ & \quad * sueldo * 1.5) \end{aligned}$	empleado por horas: primerNombre apellidoPaterno número de seguro social: NSS sueldo semanal: sueldo; horas trabajadas: horas
Empleado-porComision	tarifaComision * ventasBrutas	empleado por comisión: primerNombre apellidoPaterno número de seguro social: NSS ventas brutas: ventasBrutas; tarifa de comisión: tarifaComision
Empleado-BaseMasComision	salarioBase + (tarifaComision * ventasBrutass)	empleado por comisión con salario base: primerNombre apellidoPaterno número de seguro social: NSS ventas brutas: ventasBrutas; tarifa de comisión: tarifaComision; salario base: Salariobase

Figura 13.12 | Interfaz polimórfica para las clases de la jerarquía Empleado.

respectivamente); funciones *get* que devuelven el primer nombre, apellido paterno y número de seguro social (líneas 15, 18 y 21, respectivamente); la función *virtual* pura *ingresos* (línea 24) y la función *virtual* *imprimir* (línea 25).

Recuerde que declaramos a *ingresos* como una función *virtual* pura, debido a que primero debemos conocer el tipo de *Empleado* específico para determinar los cálculos apropiados de *ingresos*. Al declarar esta función como *virtual* pura se indica que cada clase derivada concreta *debe* proporcionar una implementación apropiada para *ingresos*, y que un programa puede usar apuntadores de la clase base *Empleado* para invocar a la función *ingresos* de manera polimórfica para cualquier tipo de *Empleado*.

La figura 13.14 contiene las implementaciones de las funciones miembro para la clase *Empleado*. No se proporciona una implementación para la función *virtual* *ingresos*. Observe que el constructor de *Empleado* (líneas 10 a 15) no valida el número de seguro social. Por lo general, se debe proporcionar dicha validación. Un ejercicio en el capítulo 12 pide al lector que valide un número de seguro social para asegurar que se encuentre en la forma **###-##-####**, en donde cada # representa a un dígito.

```

1 // Fig. 13.13: Empleado.h
2 // Clase base abstracta Empleado.
3 #ifndef EMPLEADO_H
4 #define EMPLEADO_H
5
6 #include <string> // clase string estándar de C++
7 using std::string;
8
9 class Empleado
10 {
11 public:
12     Empleado( const string &, const string &, const string & );
13 }
```

Figura 13.13 | Archivo de encabezado de la clase *Empleado*. (Parte I de 2).

```

14 void setPrimerNombre( const string & ); // establece el primer nombre
15 string getPrimerNombre() const; // devuelve el primer nombre
16
17 void setApellidoPaterno( const string & ); // establece el apellido paterno
18 string getApellidoPaterno() const; // devuelve el apellido paterno
19
20 void setNumeroSeguroSocial( const string & ); // establece el NSS
21 string getNumeroSeguroSocial() const; // devuelve el NSS
22
23 // la función virtual pura hace de Empleado una clase base abstracta
24 virtual double ingresos() const = 0; // virtual pura
25 virtual void imprimir() const; // virtual
26 private:
27     string primerNombre;
28     string apellidoPaterno;
29     string numeroSeguroSocial;
30 }; // fin de la clase Empleado
31
32 #endif // EMPLEADO_H

```

Figura 13.13 | Archivo de encabezado de la clase Empleado. (Parte 2 de 2).

```

1 // Fig. 13.14: Empleado.cpp
2 // Definiciones de las funciones miembro de la clase base abstracta Empleado.
3 // Nota: no se proporcionan definiciones para las funciones virtuales puras.
4 #include <iostream>
5 using std::cout;
6
7 #include "Empleado.h" // definición de la clase Empleado
8
9 // constructor
10 Empleado::Empleado( const string &nombre, const string &apellido,
11                     const string &nss )
12     : primerNombre( nombre ), apellidoPaterno( apellido ), numeroSeguroSocial( nss )
13 {
14     // cuerpo vacío
15 } // fin del constructor de Empleado
16
17 // establece el primer nombre
18 void Empleado::setPrimerNombre( const string &nombre )
19 {
20     primerNombre = nombre;
21 } // fin de la función setPrimerNombre
22
23 // devuelve el primer nombre
24 string Empleado::getPrimerNombre() const
25 {
26     return primerNombre;
27 } // fin de la función getPrimerNombre
28
29 // establece el apellido paterno
30 void Empleado::setApellidoPaterno( const string &apellido )
31 {
32     apellidoPaterno = apellido;
33 } // fin de la función setApellidoPaterno
34
35 // devuelve el apellido paterno
36 string Empleado::getApellidoPaterno() const
37 {
38     return apellidoPaterno;

```

Figura 13.14 | Archivo de implementación de la clase Empleado. (Parte I de 2).

```

39 } // fin de la función getApellidoPaterno
40
41 // establece el número de seguro social
42 void Empleado::setNumeroSeguroSocial( const string &nss )
43 {
44     numeroSeguroSocial = nss; // debe validar
45 } // fin de la función setNumeroSeguroSocial
46
47 // devuelve el número de seguro social
48 string Empleado::getNumeroSeguroSocial() const
49 {
50     return numeroSeguroSocial;
51 } // fin de la función getNumeroSeguroSocial
52
53 // imprime la información del Empleado (virtual, pero no virtual pura)
54 void Empleado::imprimir() const
55 {
56     cout << getPrimerNombre() << ' ' << getApellidoPaterno()
57         << "\nnumero de seguro social: " << getNumeroSeguroSocial();
58 } // fin de la función imprimir

```

Figura 13.14 | Archivo de implementación de la clase `Empleado`. (Parte 2 de 2).

Observe que la función virtual `imprimir` (figura 13.14, líneas 54 a 58) proporciona una implementación que se sobrescribirá en cada una de las clases derivadas. Cada una de estas funciones utilizará, sin embargo, la versión de `imprimir` de la clase abstracta para imprimir información común para todas las clases en la jerarquía de `Empleado`.

13.6.2 Creación de la clase derivada concreta `EmpleadoAsalariado`

La clase `EmpleadoAsalariado` (figuras 13.15 y 13.16) se deriva de la clase `Empleado` (línea 8 de la figura 13.15). Las funciones miembro `public` incluyen a un constructor que recibe un primer nombre, un apellido paterno, un número de seguro social y un salario semanal como argumentos (líneas 11 y 12); una función `set` para asignar un nuevo valor no negativo al miembro de datos `salarioSemanal` (línea 14); una función `get` para devolver el valor de `salario-`

```

1 // Fig. 13.15: EmpleadoAsalariado.h
2 // Clase EmpleadoAsalariado derivada de Empleado.
3 #ifndef ASALARIADO_H
4 #define ASALARIADO_H
5
6 #include "Empleado.h" // definición de la clase Empleado
7
8 class EmpleadoAsalariado : public Empleado
9 {
10 public:
11     EmpleadoAsalariado( const string &, const string &,
12                         const string &, double = 0.0 );
13
14     void setSalarioSemanal( double ); // establece el salario semanal
15     double getSalarioSemanal() const; // devuelve el salario semanal
16
17     // la palabra clave virtual indica el intento de sobrescribir
18     virtual double ingresos() const; // calcula los ingresos
19     virtual void imprimir() const; // imprime el objeto EmpleadoAsalariado
20 private:
21     double salarioSemanal; // salario por semana
22 }; // fin de la clase EmpleadoAsalariado
23
24 #endif // ASALARIADO_H

```

Figura 13.15 | Archivo de encabezado de la clase `EmpleadoAsalariado`.

Semanal (línea 15); una función **virtual** llamada **ingresos** que calcula los ingresos de un **EmpleadoAsalariado** (línea 18) y una función **virtual** llamada **imprimir** (línea 19) que imprime el tipo del empleado, a saber, "empleado asalariado:" seguido de la información específica del empleado producida por la función **imprimir** de la clase base **Empleado** y la función **getSalarioSemanal** de **EmpleadoAsalariado**.

La figura 13.16 contiene las implementaciones de las funciones miembro para **EmpleadoAsalariado**. El constructor de la clase pasa el primer nombre, apellido paterno y número de seguro social al constructor de **Empleado** (línea 11) para inicializar los datos miembro **private** que se heredan de la clase base, pero que no están accesibles en la clase derivada. La función **ingresos** (líneas 30 a 33) sobrescribe la función **virtual** pura **ingresos** en **Empleado** para proporcionar una implementación concreta que devuelva el salario semanal del **EmpleadoAsalariado**. Si no implementáramos **ingresos**, la clase **EmpleadoAsalariado** sería una clase abstracta y cualquier intento de instanciar un objeto de la clase produciría un error de compilación (y, desde luego, queremos que **EmpleadoAsalariado** aquí sea una clase concreta). Observe que en el archivo de encabezado de la clase **EmpleadoAsalariado** declaramos las funciones miembro **ingresos** e **imprimir** como **virtual** (líneas 18 y 19 de la figura 13.15); en realidad, colocar la palabra clave **virtual** antes de estas funciones miembro es redundante. Las definimos como **virtual** en la clase base **Empleado**, por lo que siguen siendo funciones **virtual** a través de la jerarquía de clases. En la *Buena práctica de programación 13.1* vimos que declarar dichas funciones explícitamente como **virtual** en cada nivel de la jerarquía puede promover la claridad del programa.

```

1 // Fig. 13.16: EmpleadoAsalariado.cpp
2 // Definiciones de las funciones miembro de la clase EmpleadoAsalariado.
3 #include <iostream>
4 using std::cout;
5
6 #include "EmpleadoAsalariado.h" // definición de la clase EmpleadoAsalariado
7
8 // constructor
9 EmpleadoAsalariado::EmpleadoAsalariado( const string &nombre,
10   const string &apellido, const string &nss, double salario )
11 : Empleado( nombre, apellido, nss )
12 {
13   setSalarioSemanal( salario );
14 } // fin del constructor de EmpleadoAsalariado
15
16 // establece el salario
17 void EmpleadoAsalariado::setSalarioSemanal( double salario )
18 {
19   salarioSemanal = ( salario < 0.0 ) ? 0.0 : salario;
20 } // fin de la función setSalarioSemanal
21
22 // devuelve el salario
23 double EmpleadoAsalariado::getSalarioSemanal() const
24 {
25   return salarioSemanal;
26 } // fin de la función getSalarioSemanal
27
28 // calcula los ingresos;
29 // sobrescribe a la función virtual pura ingresos en Empleado
30 double EmpleadoAsalariado::ingresos() const
31 {
32   return getSalarioSemanal();
33 } // fin de la función ingresos
34
35 // imprime la información del EmpleadoAsalariado
36 void EmpleadoAsalariado::imprimir() const
37 {
38   cout << "empleado asalariado: ";
39   Empleado::imprimir(); // reutiliza la función imprimir de la clase base abstracta
40   cout << "\nsalario semanal: " << getSalarioSemanal();
41 } // fin de la función imprimir

```

Figura 13.16 | Archivo de implementación de la clase **EmpleadoAsalariado**.

La función `imprimir` de la clase `EmpleadoAsalariado` (líneas 36 a 41 de la figura 13.16) sobrescribe a la función `imprimir` de `Empleado`. Si la clase `EmpleadoAsalariado` no sobrescribiera a `imprimir`, `EmpleadoAsalariado` heredaría la versión de `imprimir` correspondiente a `Empleado`. En ese caso, la función `imprimir` de `EmpleadoAsalariado` simplemente devolvería el nombre completo del empleado y el número de seguro social, lo cual no representa en forma adecuada a un `EmpleadoAsalariado`. Para imprimir la información completa de un `EmpleadoAsalariado`, la función `imprimir` de la clase derivada imprime el texto "`empleado asalariado`" seguido de la información específica de la clase base `Empleado` (es decir, primer nombre, apellido paterno y número de seguro social) que se imprime al invocar la función `imprimir` de la clase base, usando el operador de resolución de ámbito binario (línea 39); éste es un buen ejemplo de reutilización de código. Los resultados producidos por la función `imprimir` de `EmpleadoAsalariado` contienen el salario semanal del empleado que se obtiene al invocar la función `getSalarioSemanal` de la clase.

13.6.3 Creación de la clase derivada concreta `EmpleadoPorHoras`

La clase `EmpleadoPorHora` (figuras 13.17 y 13.18) también se deriva de la clase `Empleado` (línea 8 de la figura 13.17). Las funciones miembro `public` incluyen un constructor (líneas 11 y 12) que recibe como argumentos un primer nombre, un apellido paterno, un número de seguro social, un sueldo por hora y el número de horas trabajadas; funciones `set` que asignan nuevos valores a los datos miembro `sueldoHora` y `horasTrabajadas`, respectivamente (líneas 14 y 17); funciones `get` para devolver los valores de `sueldoHora` y `horasTrabajadas`, respectivamente (líneas 15 y 18); una función `virtual` llamada `ingresos` que calcula los ingresos de un `EmpleadoPorHoras` (línea 21) y una función `virtual` llamada `imprimir` que imprime el tipo del empleado, a saber, "`empleado por horas:`", junto con información específica del empleado (línea 22).

```

1 // Fig. 13.17: EmpleadoPorHoras.h
2 // Definición de la clase EmpleadoPorHoras.
3 #ifndef PORHORAS_H
4 #define PORHORAS_H
5
6 #include "Empleado.h" // definición de la clase Empleado
7
8 class EmpleadoPorHoras : public Empleado
9 {
10 public:
11     EmpleadoPorHoras( const string &, const string &,
12                       const string &, double = 0.0, double = 0.0 );
13
14     void setSueldo( double ); // establece el sueldo por hora
15     double getSueldo() const; // devuelve el sueldo por hora
16
17     void setHoras( double ); // establece las horas trabajadas
18     double getHoras() const; // devuelve las horas trabajadas
19
20     // la palabra clave virtual indica el intento de sobrescribir
21     virtual double ingresos() const; // calcula los ingresos
22     virtual void imprimir() const; // imprime el objeto EmpleadoPorHoras
23 private:
24     double sueldo; // sueldo por hora
25     double horas; // horas trabajadas por semana
26 }; // fin de la clase EmpleadoPorHoras
27
28 #endif // PORHORAS_H

```

Figura 13.17 | Archivo de encabezado de la clase `EmpleadoPorHoras`.

```

1 // Fig. 13.18: EmpleadoPorHoras.cpp
2 // Definiciones de las funciones miembro de EmpleadoPorHoras.
3 #include <iostream>
4 using std::cout;
5

```

Figura 13.18 | Archivo de implementación de la clase `EmpleadoPorHoras`. (Parte I de 2).

```

6 #include "EmpleadoPorHoras.h" // definición de la clase EmpleadoPorHoras
7
8 // constructor
9 EmpleadoPorHoras::EmpleadoPorHoras( const string &nombre, const string &apellido,
10 const string &nss, double sueldoHora, double horasTrabajadas )
11 : Empleado( nombre, apellido, nss )
12 {
13     setSueldo( sueldoHora ); // valida el sueldo por hora
14     setHoras( horasTrabajadas ); // valida las horas trabajadas
15 } // fin del constructor de EmpleadoPorHoras
16
17 // establece el sueldo
18 void EmpleadoPorHoras::setSueldo( double sueldoHora )
19 {
20     sueldo = ( sueldoHora < 0.0 ? 0.0 : sueldoHora );
21 } // fin de la función setSueldo
22
23 // devuelve el sueldo
24 double EmpleadoPorHoras::getSueldo() const
25 {
26     return sueldo;
27 } // fin de la función getSueldo
28
29 // establece las horas trabajadas
30 void EmpleadoPorHoras::setHoras( double horasTrabajadas )
31 {
32     horas = ( ( horasTrabajadas >= 0.0 ) && ( horasTrabajadas <= 168.0 ) ) ?
33         horasTrabajadas : 0.0 ;
34 } // fin de la función setHoras
35
36 // devuelve las horas trabajadas
37 double EmpleadoPorHoras::getHoras() const
38 {
39     return horas;
40 } // fin de la función getHoras
41
42 // calcula los ingresos;
43 // sobrescribe la función virtual pura ingresos en Empleado
44 double EmpleadoPorHoras::ingresos() const
45 {
46     if ( getHoras() <= 40 ) // no hay tiempo extra
47         return getSueldo() * getHoras();
48     else
49         return 40 * getSueldo() + ( ( getHoras() - 40 ) * getSueldo() * 1.5 );
50 } // fin de la función ingresos
51
52 // imprime la información del EmpleadoPorHoras
53 void EmpleadoPorHoras::imprimir() const
54 {
55     cout << "empleado por horas: ";
56     Empleado::imprimir(); // code reuse
57     cout << "\nsueldo por hora: " << getSueldo() <<
58     "; horas trabajadas: " << getHoras();
59 } // fin de la función imprimir

```

Figura 13.18 | Archivo de implementación de la clase EmpleadoPorHoras. (Parte 2 de 2).

La figura 13.18 contiene las implementaciones de las funciones miembro para la clase `EmpleadoPorHoras`. En las líneas 18 a 21 y 30 a 34 se definen funciones *set* que asignan nuevos valores a los datos miembro `sueldoHora` y `horasTrabajadas`, respectivamente. La función `setSueldo` (líneas 18 a 21) asegura que `sueldoHora` sea no negativo, y la función `setHoras` (líneas 30 a 34) asegura que el miembro de datos `horasTrabajadas` se encuentre entre 0 y 168 (el número total de horas en una semana). Las funciones *get* de la clase `EmpleadoPorHoras` se implementan en las líneas 24

a 27 y 37 a 40. No declaramos estas funciones `virtual`, por lo que las clases derivadas de la clase `EmpleadoPorHoras` no las pueden sobrescribir (aunque las clases derivadas ciertamente pueden redefinirlas). Observe que el constructor de `EmpleadoPorHoras`, al igual que el constructor de `EmpleadoAsalariado`, pasa el primer nombre, el apellido paterno y el número de seguro social al constructor de la clase base `Empleado` (línea 11) para inicializar los datos miembro `private` heredados que se declaran en la clase base. Además, la función `imprimir` de `EmpleadoPorHoras` llama a la función `imprimir` de la clase base (línea 56) para imprimir la información específica de `Empleado` (es decir, primer nombre, apellido paterno y número de seguro social); éste es otro buen ejemplo de reutilización de código.

13.6.4 Creación de la clase derivada concreta `EmpleadoPorComision`

La clase `EmpleadoPorComision` (figuras 13.19 y 13.20) se deriva de la clase `Empleado` (línea 8 de la figura 13.19). Las implementaciones de las funciones miembro (figura 13.20) incluyen a un constructor (líneas 9 a 15) que recibe un primer nombre, un apellido paterno, un número de seguro social, un monto de ventas y una tarifa de comisión; funciones `set` (líneas 18 a 21 y 30 a 33) para asignar nuevos valores a los datos miembro `tarifaComision` y `ventasBrutas`, respectivamente; funciones `get` (líneas 24 a 27 y 36 a 39) que obtienen los valores de estos datos miembro; la función `ingresos` (líneas 43 a 46) para calcular los ingresos de un `EmpleadoPorComision` y la función `imprimir` (líneas 49 a 55), que imprime el tipo del empleado, a saber, "empleado por comision:", y la información específica del empleado. El constructor de `EmpleadoPorComision` también pasa el primer nombre, apellido paterno y número de seguro social al constructor de `Empleado` (línea 11) para inicializar los datos miembro `private` de `Empleado`. La función `imprimir` llama a la función `imprimir` de la clase base (línea 52) para mostrar la información específica de `Empleado` (es decir, primer nombre, apellido paterno y número de seguro social).

```

1 // Fig. 13.19: EmpleadoPorComision.h
2 // Clase EmpleadoPorComision derivada de Empleado.
3 #ifndef COMISION_H
4 #define COMISION_H
5
6 #include "Empleado.h" // definición de la clase Empleado
7
8 class EmpleadoPorComision : public Empleado
9 {
10 public:
11     EmpleadoPorComision( const string &, const string &,
12                          const string &, double = 0.0, double = 0.0 );
13
14     void setTarifaComision( double ); // establece la tarifa de comisión
15     double getTarifaComision() const; // devuelve la tarifa de comisión
16
17     void setVentasBrutas( double ); // establece el monto de ventas brutas
18     double getVentasBrutas() const; // devuelve el monto de ventas brutas
19
20     // la palabra clave virtual indica la intención de sobrescribir
21     virtual double ingresos() const; // calcula los ingresos
22     virtual void imprimir() const; // imprime el objeto EmpleadoPorComision
23 private:
24     double ventasBrutas; // ventas brutas semanales
25     double tarifaComision; // porcentaje de comisión
26 }; // fin de la clase EmpleadoPorComision
27
28 #endif // COMISION_H

```

Figura 13.19 | Archivo de encabezado de la clase `EmpleadoPorComision`.

```

1 // Fig. 13.20: EmpleadoPorComision.cpp
2 // Definiciones de las funciones miembro de la clase EmpleadoPorComision.
3 #include <iostream>
4 using std::cout;

```

Figura 13.20 | Archivo de implementación de la clase `EmpleadoPorComision`. (Parte 1 de 2).

```

5 #include "EmpleadoPorComision.h" // definición de la clase EmpleadoPorComision
6
7 // constructor
8 EmpleadoPorComision::EmpleadoPorComision( const string &nombre,
9     const string &apellido, const string &nss, double ventas, double tarifa )
10    : Empleado( nombre, apellido, nss )
11 {
12     setVentasBrutas( ventas );
13     setTarifaComision( tarifa );
14 }
15 // fin del constructor de EmpleadoPorComision
16
17 // establece la tarifa de comisión
18 void EmpleadoPorComision::setTarifaComision( double tarifa )
19 {
20     tarifaComision = ( ( tarifa > 0.0 && tarifa < 1.0 ) ? tarifa : 0.0 );
21 } // fin de la función setTarifaComision
22
23 // devuelve la tarifa de comisión
24 double EmpleadoPorComision::getTarifaComision() const
25 {
26     return tarifaComision;
27 } // fin de la función getTarifaComision
28
29 // establece el monto de ventas brutas
30 void EmpleadoPorComision::setVentasBrutas( double ventas )
31 {
32     ventasBrutas = ( ( ventas < 0.0 ) ? 0.0 : ventas );
33 } // fin de la función setVentasBrutas
34
35 // devuelve el monto de ventas brutas
36 double EmpleadoPorComision::getVentasBrutas() const
37 {
38     return ventasBrutas;
39 } // fin de la función getVentasBrutas
40
41 // calcula los ingresos;
42 // sobrescribe la función virtual pura ingresos en Empleado
43 double EmpleadoPorComision::ingresos() const
44 {
45     return getTarifaComision() * getVentasBrutas();
46 } // fin de la función ingresos
47
48 // imprime la información del EmpleadoPorComision
49 void EmpleadoPorComision::imprimir() const
50 {
51     cout << "empleado por comision: ";
52     Empleado::imprimir(); // reutilización de código
53     cout << "\nventas brutas: " << getVentasBrutas()
54         << "; tarifa de comision: " << getTarifaComision();
55 } // fin de la función imprimir

```

Figura 13.20 | Archivo de implementación de la clase `EmpleadoPorComision`. (Parte 2 de 2).

13.6.5 Creación de la clase derivada concreta indirecta `EmpleadoBaseMasComision`

La clase `EmpleadoBaseMasComision` (figuras 13.21 y 13.22) hereda directamente de la clase `EmpleadoPorComision` (línea 8 de la figura 13.21), y por lo tanto es una clase derivada *indirecta* de la clase `Empleado`. Las implementaciones de las funciones miembro de la clase `EmpleadoBaseMasComision` incluyen a un constructor (líneas 10 a 16 de la figura 13.22) que recibe como argumentos un primer nombre, un apellido paterno, un número de seguro social, un monto de ventas, una tarifa de comisión y un salario base. Después pasa el primer nombre, apellido paterno, número de seguro

social, monto de ventas y tarifa de comisión al constructor de `EmpleadoPorComision` (línea 13) para inicializar los miembros heredados. `EmpleadoBaseMasComision` también contiene una función `set` (líneas 19 a 22) para asignar un nuevo valor al miembro de datos `salarioBase` y una función `get` (líneas 25 a 28) para devolver el valor de `salarioBase`. La función `ingresos` (líneas 32 a 35) calcula los ingresos de un `EmpleadoBaseMasComision`. Observe que en la línea 34 de la función `ingresos` se llama a la función `ingresos` de la clase base `EmpleadoPorComision` para calcular la porción basada en comisiones de los ingresos del empleado. Éste es un buen ejemplo de reutilización de código. La función `imprimir` de `EmpleadoBaseMasComision` (líneas 38 a 43) imprime el texto "con salario base", seguido de los resultados de la función `imprimir` de la clase `EmpleadoPorComision` (otro ejemplo de reutilización de código), y después el salario base. La salida resultante empieza con el texto "con salario base empleado por comision: " seguido del resto de la información de `EmpleadoBaseMasComision`. Recuerde que la función `imprimir` de `EmpleadoPorComision` muestra el primer nombre, apellido paterno y número de seguro social del empleado al invocar la función `imprimir` de su clase base (es decir, `Empleado`), otro ejemplo más de reutilización de código. Observe que la función `imprimir` de `EmpleadoBaseMasComision` inicia una cadena de funciones que abarca los tres niveles de la jerarquía de `Empleado`.

```

1 // Fig. 13.21: EmpleadoBaseMasComision.h
2 // Clase EmpleadoBaseMasComision derivada de Empleado.
3 #ifndef BASEMAS_H
4 #define BASEMAS_H
5
6 #include "EmpleadoPorComision.h" // definición de la clase EmpleadoPorComision
7
8 class EmpleadoBaseMasComision : public EmpleadoPorComision
9 {
10 public:
11     EmpleadoBaseMasComision( const string &, const string &,
12                             const string &, double = 0.0, double = 0.0, double = 0.0 );
13
14     void setSalarioBase( double ); // establece el salario base
15     double getSalarioBase() const; // devuelve el salario base
16
17     // la palabra clave virtual indica el intento de sobrescribir
18     virtual double ingresos() const; // calcula los ingresos
19     virtual void imprimir() const; // imprime el objeto EmpleadoBaseMasComision
20 private:
21     double salarioBase; // salario base por semana
22 }; // fin de la clase EmpleadoBaseMasComision
23
24 #endif // BASEMAS_H

```

Figura 13.21 | Archivo de encabezado de `EmpleadoBaseMasComision`.

```

1 // Fig. 13.22: EmpleadoBaseMasComision.cpp
2 // Definiciones de las funciones miembro de EmpleadoBaseMasComision.
3 #include <iostream>
4 using std::cout;
5
6 // definición de la clase EmpleadoBaseMasComision
7 #include "EmpleadoBaseMasComision.h"
8
9 // constructor
10 EmpleadoBaseMasComision::EmpleadoBaseMasComision(
11     const string &nomb, const string &apellido, const string &nss,
12     double ventas, double tarifa, double salario )
13     : EmpleadoPorComision( nomb, apellido, nss, ventas, tarifa )
14 {
15     setSalarioBase( salario ); // valida y almacena el salario base

```

Figura 13.22 | Archivo de implementación de la clase `EmpleadoBaseMasComision`. (Parte I de 2).

```

16 } // fin del constructor de EmpleadoBaseMasComision
17
18 // establece el salario base
19 void EmpleadoBaseMasComision::setSalarioBase( double salario )
20 {
21     salarioBase = ( ( salario < 0.0 ) ? 0.0 : salario );
22 } // fin de la función setSalarioBase
23
24 // devuelve el salario base
25 double EmpleadoBaseMasComision::getSalarioBase() const
26 {
27     return salarioBase;
28 } // fin de la función getSalarioBase
29
30 // calcula los ingresos;
31 // sobrescribe la función virtual pura ingresos en Empleado
32 double EmpleadoBaseMasComision::ingresos() const
33 {
34     return getSalarioBase() + EmpleadoPorComision::ingresos();
35 } // fin de la función ingresos
36
37 // imprime la información del EmpleadoBaseMasComision
38 void EmpleadoBaseMasComision::imprimir() const
39 {
40     cout << "con salario base ";
41     EmpleadoPorComision::imprimir(); // reutilización de código
42     cout << "; salario base: " << getSalarioBase();
43 } // fin de la función imprimir

```

Figura 13.22 | Archivo de implementación de la clase `EmpleadoBaseMasComision`. (Parte 2 de 2).

13.6.6 Demostración del procesamiento polimórfico

Para evaluar nuestra jerarquía de `Empleado`, el programa de la figura 13.23 crea un objeto de cada una de las cuatro clases concretas `EmpleadoAsalariado`, `EmpleadoPorHoras`, `EmpleadoPorComision` y `EmpleadoBaseMasComision`. El programa manipula estos objetos, primero con la vinculación estática y después en forma polimórfica, usando un vector de apuntadores `Empleado`. En las líneas 31 a 38 se crean objetos de cada una de las cuatro clases derivadas concretas de `Empleado`. En las líneas 43 a 51 se imprime la información y los ingresos de cada `Empleado`. La invocación a cada función miembro en las líneas 43 a 51 es un ejemplo de vinculación estática; en tiempo de compilación, como estamos usando manejadores de nombres (y no apuntadores o referencias que podrían establecerse en tiempo de ejecución), el compilador puede identificar el tipo de cada objeto para determinar cuáles funciones de `imprimir` e `ingresos` se van a llamar.

```

1 // Fig. 13.23: fig13_23.cpp
2 // Procesamiento de objetos de clases derivadas de Empleado en forma
3 // individual y polimórfica, mediante el uso de la vinculación dinámica.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7 using std::fixed;
8
9 #include <iomanip>
10 using std::setprecision;
11
12 #include <vector>
13 using std::vector;

```

Figura 13.23 | Programa controlador de la jerarquía de clases de `Empleado`. (Parte 1 de 4).

```
14 // incluye las definiciones de las clases en la jerarquía de Empleado
15 #include "Empleado.h"
16 #include "EmpleadoAsalariado.h"
17 #include "EmpleadoPorHoras.h"
18 #include "EmpleadoPorComision.h"
19 #include "EmpleadoBaseMasComision.h"
20
21 void virtualViaApuntador( const Empleado * const ); // prototipo
22 void virtualViaReferencia( const Empleado & ); // prototipo
23
24 int main()
25 {
26     // establece el formato de salida de punto flotante
27     cout << fixed << setprecision( 2 );
28
29     // crea objetos de las clases derivadas
30     EmpleadoAsalariado empleadoAsalariado(
31         "John", "Smith", "111-11-1111", 800 );
32     EmpleadoPorHoras empleadoPorHoras(
33         "Karen", "Price", "222-22-2222", 16.75, 40 );
34     EmpleadoPorComision empleadoPorComision(
35         "Sue", "Jones", "333-33-3333", 10000, .06 );
36     EmpleadoBaseMasComision empleadoBaseMasComision(
37         "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
38
39     cout << "Empleados procesados en forma individual, usando vinculacion estatica:\n\n";
40
41     // imprime la información de cada empleado y sus ingresos, usando vinculación estática
42     empleadoAsalariado.imprimir();
43     cout << "\nobtuvo $" << empleadoAsalariado.ingresos() << "\n\n";
44     empleadoPorHoras.imprimir();
45     cout << "\nobtuvo $" << empleadoPorHoras.ingresos() << "\n\n";
46     empleadoPorComision.imprimir();
47     cout << "\nobtuvo $" << empleadoPorComision.ingresos() << "\n\n";
48     empleadoBaseMasComision.imprimir();
49     cout << "\nobtuvo $" << empleadoBaseMasComision.ingresos()
50         << "\n\n";
51
52     // crea un vector de cuatro apunadores de la clase base
53     vector < Empleado * > empleados( 4 );
54
55     // inicializa el vector con objetos Empleado
56     empleados[ 0 ] = &empleadoAsalariado;
57     empleados[ 1 ] = &empleadoPorHoras;
58     empleados[ 2 ] = &empleadoPorComision;
59     empleados[ 3 ] = &empleadoBaseMasComision;
60
61     cout << "Empleados procesados en forma polimorifica mediante vinculacion dinamica:\n\n";
62
63     // llama a virtualViaApuntador para imprimir la información de cada Empleado
64     // y a ingresos mediante el uso de la vinculación dinámica
65     cout << "Llamadas a funciones virtuales realizadas desde apunadores de la clase base:\n\n";
66
67     for ( size_t i = 0; i < empleados.size(); i++ )
68         virtualViaApuntador( empleados[ i ] );
69
70     // llama a virtualViaReferencia para imprimir la información de cada Empleado
71     // y a ingresos mediante el uso de vinculación dinámica
72     cout << "Llamadas a funciones virtuales realizadas desde referencias de la clase base:\n\n";
```

Figura 13.23 | Programa controlador de la jerarquía de clases de Empleado. (Parte 2 de 4).

```

74
75     for ( size_t i = 0; i < empleados.size(); i++ )
76         virtualViaReferencia( *empleados[ i ] ); // observe la desreferencia
77
78     return 0;
79 } // fin de main
80
81 // llama a las funciones virtuales imprimir e ingresos de Empleado desde un
82 // apuntador de la clase base mediante la vinculación dinámica
83 void virtualViaApuntador( const Empleado * const claseBasePtr )
84 {
85     claseBasePtr->imprimir();
86     cout << "\nobtuvo $" << claseBasePtr->ingresos() << "\n\n";
87 } // fin de la función virtualViaApuntador
88
89 // llama a las funciones virtuales imprimir e ingresos de Empleado desde una
90 // referencia de la clase base mediante la vinculación dinámica
91 void virtualViaReferencia( const Empleado &claseBaseRef )
92 {
93     claseBaseRef.imprimir();
94     cout << "\nobtuvo $" << claseBaseRef.ingresos() << "\n\n";
95 } // fin de la función virtualViaReferencia

```

Empleados procesados en forma individual, usando vinculacion estatica:

empleado asalariado: John Smith
 numero de seguro social: 111-11-1111
 salario semanal: 800.00
 obtuvo \$800.00

empleado por horas: Karen Price
 numero de seguro social: 222-22-2222
 sueldo por hora: 16.75; horas trabajadas: 40.00
 obtuvo \$670.00

empleado por comision: Sue Jones
 numero de seguro social: 333-33-3333
 ventas brutas: 10000.00; tarifa de comision: 0.06
 obtuvo \$600.00

con salario base empleado por comision: Bob Lewis
 numero de seguro social: 444-44-4444
 ventas brutas: 5000.00; tarifa de comision: 0.04; salario base: 300.00
 obtuvo \$500.00

Empleados procesados en forma polimorfica mediante vinculacion dinamica:

Llamadas a funciones virtuales realizadas desde apuntadores de la clase base:

empleado asalariado: John Smith
 numero de seguro social: 111-11-1111
 salario semanal: 800.00
 obtuvo \$800.00

empleado por horas: Karen Price
 numero de seguro social: 222-22-2222
 sueldo por hora: 16.75; horas trabajadas: 40.00
 obtuvo \$670.00

empleado por comision: Sue Jones
 numero de seguro social: 333-33-3333
 ventas brutas: 10000.00; tarifa de comision: 0.06
 obtuvo \$600.00

(continúa...)

Figura 13.23 | Programa controlador de la jerarquía de clases de Empleado. (Parte 3 de 4).

```

con salario base empleado por comision: Bob Lewis
numero de seguro social: 444-44-4444
ventas brutas: 5000.00; tarifa de comision: 0.04; salario base: 300.00
obtuvo $500.00

Llamadas a funciones virtuales realizadas desde referencias de la clase base:

empleado asalariado: John Smith
numero de seguro social: 111-11-1111
salario semanal: 800.00
obtuvo $800.00

empleado por horas: Karen Price
numero de seguro social: 222-22-2222
sueldo por hora: 16.75; horas trabajadas: 40.00
obtuvo $670.00

empleado por comision: Sue Jones
numero de seguro social: 333-33-3333
ventas brutas: 10000.00; tarifa de comision: 0.06
obtuvo $600.00

con salario base empleado por comision: Bob Lewis
numero de seguro social: 444-44-4444
ventas brutas: 5000.00; tarifa de comision: 0.04; salario base: 300.00
obtuvo $500.00

```

Figura 13.23 | Programa controlador de la jerarquía de clases de Empleado. (Parte 4 de 4).

En la línea 54 se asigna el vector empleados, que contiene cuatro apunadores Empleado. En la línea 57 se orienta empleados[0] al objeto empleadoAsalariado. En la línea 58 se orienta empleados[1] al objeto empleadoPorHoras. En la línea 59 se orienta empleados[2] al objeto empleadoPorComision. En la línea 60 se orienta empleados[3] al objeto empleadoBaseMasComision. El compilador permite estas asignaciones, ya que un EmpleadoAsalariado es un Empleado, un EmpleadoPorHoras es un Empleado, un EmpleadoPorComision es un Empleado y un EmpleadoBaseMasComision es un Empleado. Por lo tanto, podemos asignar las direcciones de los objetos EmpleadoAsalariado, EmpleadoPorHoras, EmpleadoPorComision y EmpleadoBaseMasComision a los apunadores de la clase base Empleado (aun y cuando Empleado sea una clase abstracta).

El ciclo en las líneas 68 y 69 recorre el vector empleados e invoca a la función virtualViaApuntador (líneas 83 a 87) para cada elemento en empleados. La función virtualViaApuntador recibe en el parámetro claseBasePtr (de tipo const Empleado * const) la dirección almacenada en un elemento de empleados. Cada llamada a virtualViaApuntador utiliza a claseBasePtr para invocar a las funciones virtual imprimir (línea 85) e ingresos (línea 86). Observe que la función virtualViaApuntador no contiene ninguna información de los tipos EmpleadoAsalariado, EmpleadoPorHoras, EmpleadoPorComision o EmpleadoBaseMasComision. La función sólo sabe acerca del tipo de su clase base Empleado. Por lo tanto, en tiempo de compilación, el compilador no puede saber cuáles funciones de la clase concreta debe llamar a través de claseBasePtr. Aun en tiempo de ejecución, cada invocación a una función virtual llama a la función en el objeto al que apunta claseBasePtr en ese momento. La salida ilustra que las funciones apropiadas para cada clase se invocan sin duda, y que se muestra la información apropiada de cada objeto. Por ejemplo, el salario semanal se muestra para el EmpleadoAsalariado, y las ventas brutas se muestran para EmpleadoPorComision y EmpleadoBaseMasComision. Observe además que al obtener los ingresos de cada Empleado en forma polimórfica en la línea 86 se producen los mismos resultados que obtener los ingresos de esos empleados mediante la vinculación estática en las líneas 44, 46, 48 y 50. Todas las llamadas a las funciones virtual imprimir e ingresos se resuelven en tiempo de compilación con la vinculación dinámica.

Por último, otra instrucción for (líneas 75 y 76) recorre a empleados e invoca la función virtualViaReferencia (líneas 91 a 95) para cada elemento en el vector. La función virtualViaReferencia recibe en su parámetro claseBaseRef (de tipo const Empleado &) una referencia formada por la acción de desreferenciar el apuntador almacenado en cada elemento de empleados (línea 76). Cada llamada a virtualViaReferencia invoca a las funciones virtual imprimir (línea 93) e ingresos (línea 94) a través de la referencia claseBaseRef para demostrar que el procesamiento polimórfico ocurre también con las referencias de la clase base. Cada invocación a una función virtual llama a la función en el objeto al que claseBaseRef hace referencia en tiempo de ejecución. Éste es otro ejemplo de vinculación

dinámica. Los resultados producidos usando referencias de la clase base son idénticos a los resultados que se producen usando apuntadores de la clase base.

13.7 (Opcional) Polimorfismo, funciones virtuales y vinculación dinámica “detrás de las cámaras”

C++ facilita la programación del polimorfismo. Evidentemente, es posible programar para el polimorfismo en los lenguajes no orientados a objetos como C, pero para ello se requieren manipulaciones de apuntadores complejas y potencialmente peligrosas. Esta sección habla acerca de cómo C++ puede implementar el polimorfismo, las funciones `virtual` y la vinculación dinámica de manera interna. Esto proporcionará al lector una sólida comprensión de la forma en que realmente funcionan estas herramientas. Lo que es más importante, le ayudará a apreciar la sobrecarga del polimorfismo; en términos de consumo de memoria y tiempo de procesador adicionales. Esto le ayudará a determinar cuándo usar el polimorfismo y cuándo evitarlo. Como veremos en el capítulo 22, los componentes de la STL se implementaron sin polimorfismo ni funciones `virtual`; esto se hizo para evitar la sobrecarga asociada en tiempo de ejecución y lograr un rendimiento óptimo para cumplir con los requerimientos únicos de la STL.

Primero explicaremos las estructuras de datos que el compilador de C++ genera en tiempo de compilación para dar soporte al polimorfismo en tiempo de ejecución. Veremos que el polimorfismo se lleva a cabo a través de tres niveles de apuntadores (es decir, “triple indirección”). Despues mostraremos cómo un programa en ejecución utiliza estas estructuras de datos para ejecutar funciones `virtual` y lograr la vinculación dinámica asociada con el polimorfismo. Observe que nuestra discusión explica una posible implementación; esto no es un requerimiento del lenguaje.

Cuando C++ compila una clase que tiene una o más funciones `virtual`, genera una **tabla de funciones virtuales (*vtable*)** para esa clase. Un programa en ejecución utiliza la *vtable* para seleccionar la implementación de la función apropiada cada vez que se llama a una función `virtual` de esa clase. La columna de más a la izquierda de la figura 13.24 ilustra las *vtables* para las clases `Empleado`, `EmpleadoAsalariado`, `EmpleadoPorHoras`, `EmpleadoPorComision` y `EmpleadoBaseMasComision`.

En la *vtable* para la clase `Empleado`, el apuntador a la primera función se establece en 0 (es decir, el apuntador nulo). Esto se hace debido a que la función `ingresos` es una función `virtual` pura, y por lo tanto carece de una implementación. El apuntador a la segunda función apunta a la función `imprimir`, la cual muestra el nombre completo y número de seguro social del empleado. [Nota: hemos abreviado la salida de cada función `imprimir` en esta figura para conservar espacio]. Cualquier clase que tenga uno o más apuntadores nulos en su *vtable* es una clase abstracta. Las clases sin apuntadores nulos en su *vtable* (como `EmpleadoAsalariado`, `EmpleadoPorHoras`, `EmpleadoPorComision` y `EmpleadoBaseMasComision`) son clases concretas.

La clase `EmpleadoAsalariado` sobrescribe a la función `ingresos` para regresar el salario semanal del empleado, de manera que el apuntador a función apunta a la función `ingresos` de la clase `EmpleadoAsalariado`. Esta clase también sobrescribe a `imprimir`, por lo que el apuntador a función correspondiente apunta a la función miembro de `EmpleadoAsalariado` que imprime el texto `"Empleado asalariado: "` seguido del nombre del empleado, su número de seguro social y su salario semanal.

El apuntador a la función `ingresos` en la *vtable* para la clase `EmpleadoPorHoras` apunta a la función `ingresos` de `EmpleadoAsalariado` que devuelve el `sueldoHora` del empleado, multiplicado por el número de horas trabajadas. Observe que para conservar espacio, hemos omitido el hecho de que los empleados por horas reciben un sueldo de tiempo y medio extra por las horas extras trabajadas. El apuntador a la función `imprimir` apunta a la versión de la función correspondiente a `EmpleadoPorHoras`, la cual imprime el texto `"Empleado por horas: "`, el nombre, número de seguro social, sueldo por hora y las horas trabajadas del empleado. Ambas funciones sobrescriben a las funciones en la clase `Empleado`.

El apuntador a la función `ingresos` en la *vtable* para la clase `EmpleadoPorComision` apunta a la función `ingresos` de `EmpleadoPorComision` que devuelve las ventas brutas del empleado, multiplicadas por la tarifa de comisión. El apuntador a la función `imprimir` apunta a la versión de `EmpleadoPorComision` de la función, la cual imprime el tipo, nombre, número de seguro social, tarifa de comisión y ventas brutas del empleado. Como en la clase `EmpleadoPorHoras`, ambas funciones sobrescriben a las funciones en la clase `Empleado`.

El apuntador a la función `ingresos` en la *vtable* para la clase `EmpleadoBaseMasComision` apunta a la función `ingresos` de `EmpleadoBaseMasComision`, la cual devuelve el salario base del empleado más las ventas brutas, multiplicadas por la tarifa de comisión. El apuntador a la función `imprimir` apunta a la versión de la función correspondiente a `EmpleadoBaseMasComision`, la cual imprime el salario base del empleado más el tipo, nombre, número de seguro social, tarifa de comisión y ventas brutas. Ambas funciones sobrescriben a las funciones en la clase `EmpleadoPorComision`.

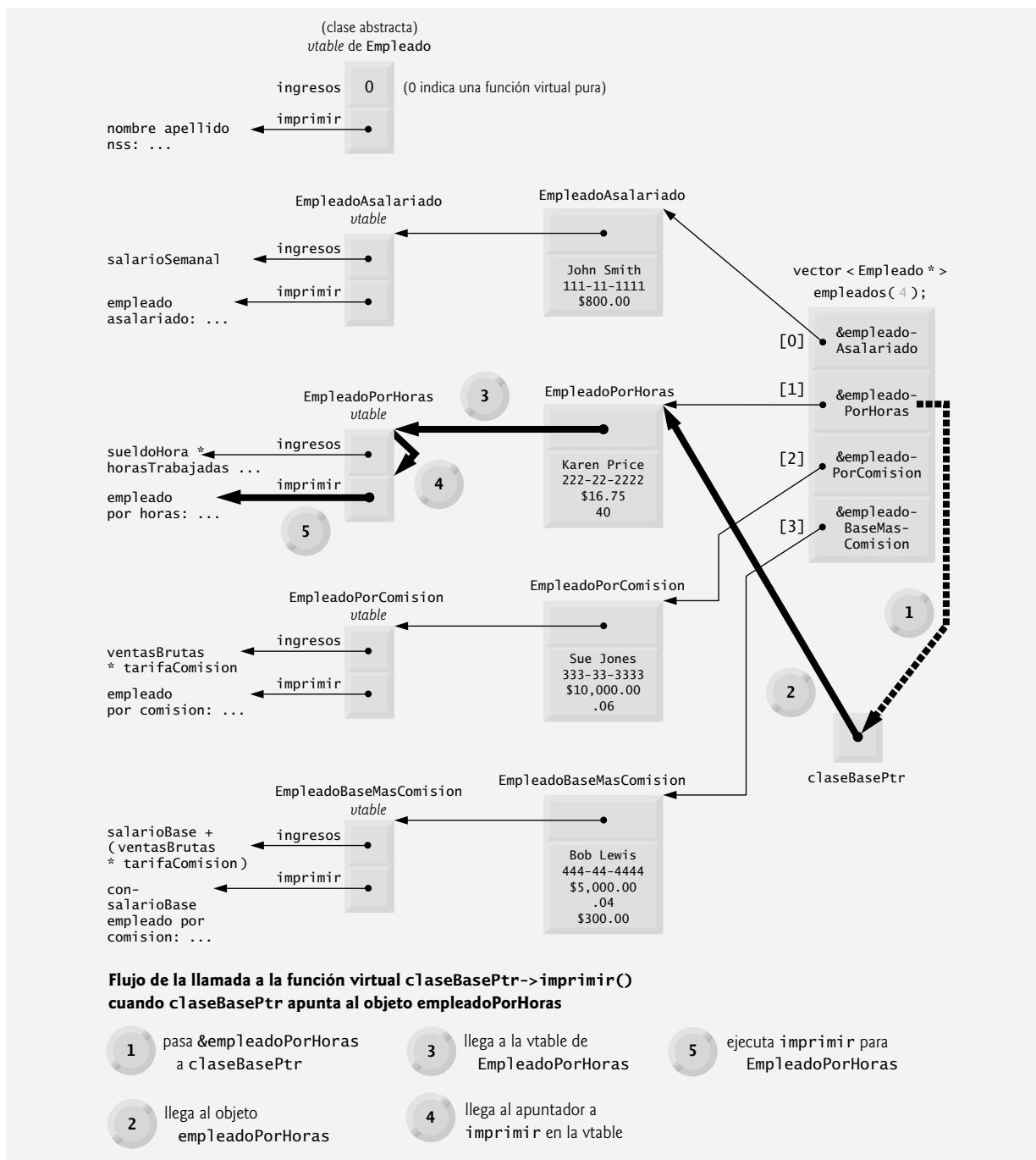


Figura 13.24 | Cómo funcionan las llamadas a funciones virtual.

Observe que en nuestro caso de estudio de `Empleado`, cada clase concreta proporciona su propia implementación para las funciones virtual `ingresos` e `imprimir`. El lector aprendió que cada clase que hereda directamente de la clase base abstracta `Empleado` debe implementar a `ingresos` para poder ser una clase concreta, ya que `ingresos` es una función virtual pura. Sin embargo, estas clases no necesitan implementar la función `imprimir` para considerarse concretas; `imprimir` no es una función virtual pura y las clases derivadas pueden heredar la implementación de `imprimir` correspondiente a la clase `Empleado`. Lo que es más, la clase `EmpleadoBaseMasComision` no necesita implementar ni la función `imprimir` ni la función `ingresos`; ambas implementaciones de las funciones se pueden heredar de la clase

EmpleadoPorComision. Si una clase en nuestra jerarquía fuera a heredar las implementaciones de las funciones de esta forma, los apuntadores de la *vtable* para estas funciones simplemente apuntarían a la implementación de la función que se vaya a heredar. Por ejemplo, si `EmpleadoBaseMasComision` no sobrescribiera a `ingresos`, el apuntador a la función `ingresos` en la *vtable* para la clase `EmpleadoBaseMasComision` apuntaría a la misma función `ingresos` que a la que apunta la *vtable* para la clase `EmpleadoPorComision`.

El polimorfismo se lleva a cabo a través de una elegante estructura de datos que implica tres niveles de apuntadores. Hemos hablado sobre un nivel: los apuntadores a las funciones en la *vtable*. Éstos apuntan a las funciones actuales que se ejecutan cuando se invoca a una función `virtual`.

Ahora consideraremos el segundo nivel de apuntadores. Cada vez que se instancia un objeto de una clase con una o más funciones `virtual`, el compilador adjunta al objeto un apuntador a la *vtable* para esa clase. Por lo general, este apuntador está en la parte frontal del objeto, pero no se requiere implementarlo de esa forma. En la figura 13.24, estos apuntadores se asocian con los objetos creados en la figura 13.23 (un objeto para cada uno de los tipos `EmpleadoAsalariado`, `EmpleadoPorHoras`, `EmpleadoPorComision` y `EmpleadoBaseMasComision`). Observe que el diagrama muestra los valores de cada uno de los datos miembro del objeto. Por ejemplo, el objeto `empleadoAsalariado` contiene un apuntador a la *vtable* de `EmpleadoAsalariado`; el objeto también contiene los valores John Smith, 111-11-1111 y \$800.00.

El tercer nivel de apuntadores simplemente contiene los manejadores para los objetos que reciben las llamadas a las funciones `virtual`. Los manejadores en este nivel también pueden ser referencias. Observe que la figura 13.24 describe el vector `empleados` que contiene apuntadores `Empleado`.

Ahora veamos cómo se ejecuta una llamada a una función `virtual` ordinaria. Considere la llamada `claseBasePtr->imprimir()` en la función `virtualViaApuntador` (línea 85 de la figura 13.23). Suponga que `claseBasePtr` contiene `empleados[1]` (es decir, la dirección del objeto `EmpleadoPorHoras` en `empleados`). Cuando el compilador compila esta instrucción, determina que la llamada sin duda se está realizando a través de un apuntador de la clase base, y que `imprimir` es una función `virtual`.

El compilador determina que `imprimir` es la *segunda* entrada en cada una de las *vtables*. Para localizar esta entrada, el compilador observa que necesitará omitir la primera entrada. Por ende, el compilador compila un **desplazamiento** de cuatro bytes (cuatro bytes para cada apuntador en las máquinas populares de 32 bits de la actualidad, y sólo hay que omitir un apuntador) en la tabla de apuntadores de código objeto de lenguaje máquina para encontrar el código que ejecutará la llamada a la función `virtual`.

El compilador genera código que realiza las siguientes operaciones [Nota: los números en la lista corresponden a los números con círculos en la figura 13.24]:

1. Seleccionar la *i*-ésima entrada de `empleados` (en este caso, la dirección del objeto `EmpleadoPorHoras`), y pasarlala como un argumento a la función `virtualViaApuntador`. Esto establece el parámetro `claseBasePtr` para que apunte a `EmpleadoPorHoras`.
2. Desreferenciar ese apuntador para llegar al objeto `EmpleadoPorHoras`; que, como recordará, empieza con un apuntador a la *vtable* de `EmpleadoPorHoras`.
3. Desreferenciar el apuntador de la *vtable* de `EmpleadoPorHoras` para llegar a la *vtable* de `EmpleadoPorHoras`.
4. Omitir el desplazamiento de cuatro bytes para seleccionar el apuntador a la función `imprimir`.
5. Desreferenciar el apuntador a la función `imprimir` para formar el “nombre” de la función actual que se va a ejecutar, y usar el operador de llamada a función () para ejecutar la función `imprimir` apropiada, que en este caso imprime el tipo, nombre, número de seguro social, sueldo por hora y horas trabajadas del empleado.

Las estructuras de datos de la figura 13.24 pueden parecer complejas, pero esta complejidad es administrada por el compilador y se oculta al programador, lo cual hace de la programación polimórfica un proceso simple. Las operaciones de desreferenciamiento de apuntadores y los accesos a memoria que ocurren en cada llamada a una función `virtual` requieren cierto tiempo de ejecución adicional. Las *vtables* y los apuntadores a una *vtable* que se agregan a los objetos requieren cierta memoria adicional. Ahora el lector tiene suficiente información para determinar si las funciones `virtual` son apropiadas para sus programas.

Tip de rendimiento 13.1



El polimorfismo, según su implementación común con funciones `virtual` y vinculación dinámica en C++, es eficiente. Los programadores pueden utilizar estas herramientas con un impacto nominal en el rendimiento.



Tip de rendimiento 13.2

Las funciones virtuales y la vinculación dinámica permiten la programación polimórfica como una alternativa a la programación con la lógica de switch. Por lo común, los compiladores optimizadores generan código polimórfico que se ejecuta con la misma eficiencia que la lógica basada en switch, codificada a mano. La sobrecarga del polimorfismo es aceptable para la mayoría de las aplicaciones. Pero en ciertas situaciones (aplicaciones de tiempo real con requerimientos estrictos en cuanto al rendimiento, por ejemplo), la sobrecarga del polimorfismo puede ser demasiado alta.



Observación de Ingeniería de Software 13.11

La vinculación dinámica permite a los distribuidores de software independientes (ISVs) distribuir el software sin revelar secretos propietarios. Las distribuciones de software pueden consistir sólo de archivos de encabezado y de archivos de código objeto; no hay necesidad de revelar el código fuente. Así, los desarrolladores de software pueden utilizar la herencia para derivar nuevas clases a partir de las que proporcionan los ISVs. Otro software que haya trabajado con las clases proporcionadas por los ISVs, seguirá trabajando con las clases derivadas y utilizará las funciones virtual sobreescritas en estas clases (mediante la vinculación dinámica).

13.8 Ejemplo práctico: sistema de nómina mediante el uso de polimorfismo e información de tipos en tiempo de ejecución con conversión descendente, dynamic_cast, typeid y type_info

En el enunciado del problema anterior al principio de la sección 13.6 vimos que, para el periodo de pago actual, nuestra empresa ficticia ha decidido recompensar a cada `EmpleadoBaseMasComision`, agregando un 10 porciento a sus salarios base. Al procesar objetos `Empleado` mediante el polimorfismo en la sección 13.6.6, no tuvimos que preocuparnos por los “detalles específicos”. Ahora, sin embargo, para ajustar los salarios base de cada `EmpleadoBaseMasComision` tenemos que determinar el tipo específico de cada objeto `Empleado` en tiempo de ejecución, y después actuar en forma apropiada. Esta sección demuestra las poderosas herramientas de la información de tipos en tiempo de ejecución (RTTI) y la conversión dinámica de tipos, las cuales permiten a un programa determinar el tipo de un objeto en tiempo de ejecución, y actuar sobre ese objeto en forma acorde.

[Nota: algunos compiladores requieren habilitar la RTTI para poder usarla en un programa. Consulte la documentación de su compilador para determinar si éste tiene requerimientos similares. En Visual C++ 2005, esta opción está habilitada de manera predeterminada].

En la figura 13.25 se utiliza la jerarquía de `Empleado` desarrollada en la sección 13.6, y se incrementa en un 10 por ciento el salario base de cada `EmpleadoBaseMasComision`. En la línea 31 se declara el vector de cuatro elementos llamado `empleados`, que almacena apunadores a objetos `Empleado`. En las líneas 34 a 41 se llena el vector con las direcciones de los objetos asignados en forma dinámica de las clases `EmpleadoAsalariado` (figuras 13.15 y 13.16), `EmpleadoPorHoras` (figuras 13.17 y 13.18), `EmpleadoPorComision` (figuras 13.19 y 13.20) y `EmpleadoBaseMasComision` (figuras 13.21 y 13.22).

```

1 // Fig. 13.25: fig13_25.cpp
2 // Demostración de la conversión descendente y la información de tipos en tiempo de ejecución.
3 // NOTA: Tal vez necesite habilitar la RTTI en su compilador
4 // para poder ejecutar esta aplicación.
5 #include <iostream>
6 using std::cout;
7 using std::endl;
8 using std::fixed;
9
10 #include <iomanip>
11 using std::setprecision;
12
13 #include <vector>
14 using std::vector;
15
16 #include <typeinfo>
17

```

Figura 13.25 | Demostración de la conversión descendente y la información de tipos en tiempo de ejecución. (Parte I de 3).

```

18 // incluye las definiciones de las clases en la jerarquía de Empleado
19 #include "Empleado.h"
20 #include "EmpleadoAsalariado.h"
21 #include "EmpleadoPorHoras.h"
22 #include "EmpleadoPorComision.h"
23 #include "EmpleadoBaseMasComision.h"
24
25 int main()
26 {
27     // establece el formato de salida de punto flotante
28     cout << fixed << setprecision( 2 );
29
30     // crea un vector de cuatro apunadores de la clase base
31     vector < Empleado * > empleados( 4 );
32
33     // inicializa el vector con varios tipos de objetos Empleado
34     empleados[ 0 ] = new EmpleadoAsalariado(
35         "John", "Smith", "111-11-1111", 800 );
36     empleados[ 1 ] = new EmpleadoPorHoras(
37         "Karen", "Price", "222-22-2222", 16.75, 40 );
38     empleados[ 2 ] = new EmpleadoPorComision(
39         "Sue", "Jones", "333-33-3333", 10000, .06 );
40     empleados[ 3 ] = new EmpleadoBaseMasComision(
41         "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
42
43     // procesa en forma polimórfica cada elemento en el vector empleados
44     for ( size_t i = 0; i < empleados.size(); i++ )
45     {
46         empleados[ i ]->imprimir(); // imprime la información del empleado
47         cout << endl;
48
49         // conversión descendente del apuntador
50         EmpleadoBaseMasComision *derivadaPtr =
51             dynamic_cast < EmpleadoBaseMasComision * >
52             ( empleados[ i ] );
53
54         // determina si el elemento apunta al empleado por
55         // comisión con salario base
56         if ( derivadaPtr != 0 ) // 0 si no es un EmpleadoBaseMasComision
57         {
58             double salarioBaseAnterior = derivadaPtr->getSalarioBase();
59             cout << "salario base anterior: $" << salarioBaseAnterior << endl;
60             derivadaPtr->setSalarioBase( 1.10 * salarioBaseAnterior );
61             cout << "el nuevo salario base con aumento del 10% es: $"
62                 << derivadaPtr->getSalarioBase() << endl;
63         } // fin de if
64
65         cout << "obtuvo $" << empleados[ i ]->ingresos() << "\n\n";
66     } // fin de for
67
68     // libera los objetos a los que apuntan los elementos del vector
69     for ( size_t j = 0; j < empleados.size(); j++ )
70     {
71         // imprime el nombre de la clase
72         cout << "eliminando objeto de "
73             << typeid( *empleados[ j ] ).name() << endl;
74
75         delete empleados[ j ];
76     } // fin de for
77
78     return 0;
79 } // fin de main

```

Figura 13.25 | Demostración de la conversión descendente y la información de tipos en tiempo de ejecución. (Parte 2 de 3).

```

empleado asalariado: John Smith
numero de seguro social: 111-11-1111
salario semanal: 800.00
obtuvo $800.00

empleado por horas: Karen Price
numero de seguro social: 222-22-2222
sueldo por hora: 16.75; horas trabajadas: 40.00
obtuvo $670.00

empleado por comision: Sue Jones
numero de seguro social: 333-33-3333
ventas brutas: 10000.00; tarifa de comision: 0.06
obtuvo $600.00

con salario base empleado por comision: Bob Lewis
numero de seguro social: 444-44-4444
ventas brutas: 5000.00; tarifa de comision: 0.04; salario base: 300.00
salario base anterior: $300.00
el nuevo salario base con aumento del 10% es: $330.00
obtuvo $530.00

eliminando objeto de class EmpleadoAsalariado
eliminando objeto de class EmpleadoPorHoras
eliminando objeto de class EmpleadoPorComision
eliminando objeto de class EmpleadoBaseMasComision

```

Figura 13.25 | Demostración de la conversión descendente y la información de tipos en tiempo de ejecución. (Parte 3 de 3).

La instrucción `for` en las líneas 44 a 66 itera a través del vector `empleados` y muestra la información de cada `Empleado`, para lo cual invoca a la función miembro `imprimir` (línea 46). Recuerde que como `imprimir` se declara `virtual` en la clase base `Empleado`, el sistema invoca a la función `imprimir` del objeto de la clase derivada apropiada.

En este ejemplo, al encontrarnos con objetos `EmpleadoBaseMasComision`, deseamos incrementar su salario base en un 10 por ciento. Como procesamos a los empleados de manera genérica (es decir, polimórfica), no podemos (con las técnicas que hemos aprendido) estar seguros de qué tipo de `Empleado` se está manipulando en cualquier momento dado. Esto crea un problema, ya que debemos identificar los empleados `EmpleadoBaseMasComision` al encontrarlos, para que puedan recibir el aumento del 10 por ciento en su salario. Para lograr esto, utilizamos el operador `dynamic_cast` (línea 51) para determinar si el tipo de cada objeto es `EmpleadoBaseMasComision`. Ésta es la operación de conversión descendente a la que hicimos referencia en la sección 13.3.3. En las líneas 50 a 52 se realiza una conversión descendente dinámica de `empleados[i]`, del tipo `Empleado *` al tipo `EmpleadoBaseMasComision *`. Si el elemento de vector apunta a un objeto que es un objeto `EmpleadoBaseMasComision`, entonces la dirección de ese objeto se asigna a `comisionPtr`; en caso contrario, se asigna 0 al apuntador de la clase derivada `derivadaPtr`.

Si el valor devuelto por el operador `dynamic_cast` en las líneas 50 a 52 no es 0, el objeto es del tipo correcto, y la instrucción `if` (líneas 56 a 63) realiza el procesamiento especial requerido para el objeto `EmpleadoBaseMasComision`. En las líneas 58, 60 y 62 se invocan las funciones `getSalarioBase` y `setSalarioBase` de `EmpleadoBaseMasComision` para obtener y actualizar el salario del empleado.

En la línea 65 se invoca a la función miembro `ingresos` en el objeto al que apunta `empleados[i]`. Recuerde que `ingresos` se declara como `virtual` en la clase base, por lo que el programa invoca a la función `ingresos` del objeto de la clase derivada; otro ejemplo de vinculación dinámica.

En las líneas 69 a 76 se muestra el tipo del objeto de cada empleado y se utiliza el operador `delete` para desasignar la memoria dinámica a la que apunta cada elemento `vector`. El operador `typeid` (línea 73) devuelve una referencia a un objeto de la clase `type_info` que contiene la información acerca del tipo de su operando, incluyendo el nombre de ese tipo. Al invocarse, la función miembro `name` de `type_info` (línea 73) devuelve una cadena basada en apuntador que contiene el nombre del tipo (por ejemplo, "class `EmpleadoBaseMasComision`") del argumento que se pasa a `typeid`. Para usar `typeid`, el programa debe incluir el archivo de encabezado `<typeinfo>` (línea 16).

Tip de portabilidad 13.1

La cadena devuelta por la función miembro `name` de `type_info` puede variar de un compilador a otro.



Observe que evitamos varios errores de compilación en este ejemplo al realizar una conversión descendente de un apuntador `Empleado` a un apuntador `EmpleadoBaseMasComision` (líneas 50 a 52). Si eliminamos el operador `dynamic_cast` de la línea 51 y tratamos de asignar el apuntador `Empleado` actual directamente al apuntador `derivadaPtr` de `EmpleadoBaseMasComision`, recibiremos un error de compilación. C++ no permite a un programa asignar un apuntador de la clase base a un apuntador de la clase derivada, debido a que la relación “*es un*” no se aplica; un `EmpleadoPorComision` no es un `EmpleadoBaseMasComision`. La relación “*es un*” se aplica sólo entre la clase derivada y sus clases base, no viceversa.

De manera similar, si en las líneas 58, 60 y 62 se utilizara el apuntador de la clase derivada actual de `empleados` (en vez de usar el apuntador `derivadaPtr` de la clase derivada) para invocar a las funciones `getSalarioBase` y `setSalarioBase` que sólo pertenecen a la clase derivada, recibiríamos un error de compilación en cada una de estas líneas. Como vimos en la sección 13.3.3, no está permitido tratar de invocar las funciones que sólo pertenecen a la clase derivada a través de un apuntador de la clase base. Aunque las líneas 58, 60 y 62 se ejecutan sólo si `derivadaPtr` no es 0 (es decir, puede realizarse la conversión), no podemos tratar de invocar las funciones `getSalarioBase` y `setSalarioBase` de la clase derivada `EmpleadoBaseMasComision` en el apuntador de la clase base `Empleado`. Recuerde que, al utilizar un apuntador de la clase base `Empleado`, sólo podemos invocar las funciones que se encuentran en la clase base `Empleado`: `ingresos`, `imprimir` y las funciones `get` y `set` de `Empleado`.

13.9 Destructores virtuales

Puede ocurrir un problema al utilizar polimorfismo para procesar los objetos de una jerarquía de clases que se asignan en forma dinámica. Hasta ahora hemos visto los **destructores no virtuales**: destructores que no se declaran con la palabra clave `virtual`. Si se destruye explícitamente un objeto de la clase derivada con un destructor no virtual mediante la aplicación del operador `delete` a un apuntador de la clase base al objeto, el estándar de C++ especifica que el comportamiento es indefinido.

La solución simple para este problema es crear un **destructor virtual** (es decir, un destructor que se declara con la palabra clave `virtual`) en la clase base. Esto hace a todos los destructores de la clase `virtual`, *aun y cuando no tienen el mismo nombre que el destructor de la clase derivada*. Ahora, si un objeto en la jerarquía se destruye explícitamente al aplicar el operador `delete` a un apuntador de la clase base, se hace una llamada al destructor para la clase apropiada con base en el objeto al que apunta el apuntador de la clase base. Recuerde, cuando se destruye un objeto de la clase derivada, la parte del objeto de la clase derivada correspondiente a la clase base también se destruye, por lo que es importante que también se ejecuten los destructores de la clase derivada y la clase base. El destructor de la clase base se ejecuta de manera automática, después del destructor de la clase derivada.



Tip para prevenir errores 13.2

Si una clase tiene funciones virtual, proporcione un destructor virtual aunque no se requiera uno para la clase. Esto asegura que se invoque a un destructor personalizado de la clase derivada (si hay uno) cuando un objeto de la clase derivada se destruya mediante un apuntador de la clase base.



Error común de programación 13.5

Los constructores no pueden ser virtual. Declarar un constructor como virtual es un error de compilación.

13.10 (Opcional) Ejemplo práctico de Ingeniería de Software: incorporación de la herencia en el sistema ATM

Ahora volveremos a analizar nuestro diseño del sistema ATM para ver cómo se podría beneficiar mediante la herencia. Para aplicar la herencia, primero buscaremos las características comunes entre las clases en el sistema. Crearemos una jerarquía de herencia para modelar clases similares (pero no idénticas) de una manera más eficiente y elegante que nos permita procesar objetos de esas clases en forma polimórfica. Después modificaremos nuestro diagrama de clases para incorporar las nuevas relaciones de herencia. Por último, demostraremos cómo se traduce nuestro diseño actualizado en archivos de encabezado de C++.

En la sección 3.11 nos topamos con el problema de representar una transacción financiera en el sistema. En vez de crear una clase para representar a todos los tipos de transacciones, decidimos crear tres clases individuales de transacciones (`SolicitudSaldo`, `Retiro` y `Deposito`) para representar las transacciones que puede realizar el sistema ATM. En la figura 13.26 se muestran los atributos y operaciones de estas clases. Observe que tienen un atributo (`númeroCuenta`)

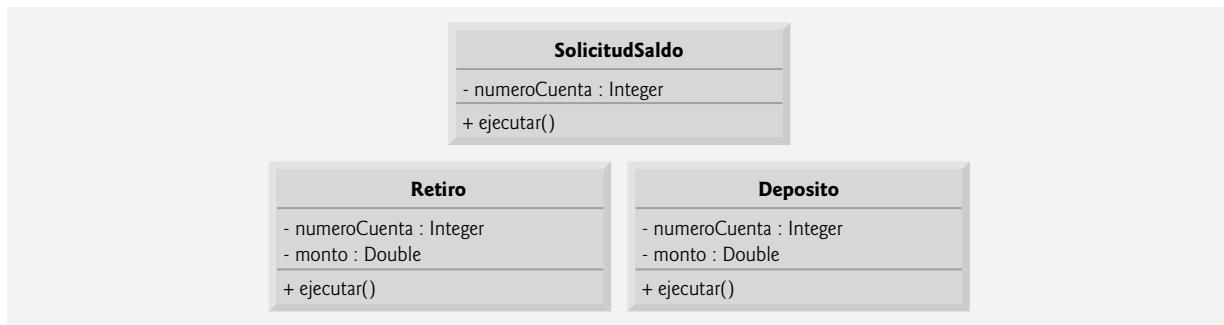


Figura 13.26 | Atributos y operaciones de las clases **SolicitudSaldo**, **Retiro** y **Deposito**.

y una operación (**ejecutar**) en común. Cada clase requiere que el atributo **numeroCuenta** especifique la cuenta a la que se aplica la transacción. Cada clase contiene la operación **ejecutar**, que el ATM invoca para realizar la transacción. Es evidente que **SolicitudSaldo**, **Retiro** y **Deposito** representan *tipos de* transacciones. La figura 13.26 revela las características comunes entre las clases de transacciones, por lo que el uso de la herencia para factorizar las características comunes parece apropiado para diseñar estas clases. Colocamos la funcionalidad común en la clase base **Transaccion** y derivamos las clases **SolicitudSaldo**, **Retiro** y **Deposito** de **Transaccion** (figura 13.27).

UML especifica una relación conocida como **generalización** para modelar la herencia. La figura 13.27 es el diagrama de clases que modela la relación de herencia entre la clase base **Transaccion** y sus tres clases derivadas. Las flechas con puntas triangulares huecas indican que las clases **SolicitudSaldo**, **Retiro** y **Deposito** se derivan de la clase **Transaccion**. Se dice que la clase **Transaccion** es una generalización de sus clases derivadas. Se dice que las clases derivadas son **especializaciones** de la clase **Transaccion**.

Las clases **SolicitudSaldo**, **Retiro** y **Deposito** comparten el atributo entero **numeroCuenta**, por lo que factorizamos este atributo común y lo colocamos en la clase base **Transaccion**. Ya no listamos a **numeroCuenta** en el segundo compartimiento de cada clase derivada, ya que las tres clases derivadas heredan este atributo de **Transaccion**. Sin embargo, recuerde que las clases derivadas no pueden acceder a los atributos **private** de una clase base. Por lo tanto, incluimos la función miembro **public getNumeroCuenta** en la clase **Transaccion**. Cada clase derivada heredará esta función miembro, con lo cual podrá acceder a su **numeroCuenta** según sea necesario para ejecutar una transacción.

De acuerdo con la figura 13.26, las clases **SolicitudSaldo**, **Retiro** y **Deposito** también comparten la operación **ejecutar**, por lo que la clase base **Transaccion** debe contener la función miembro **public ejecutar**. Sin embargo, no tiene sentido implementar a **ejecutar** en la clase **Transaccion**, ya que la funcionalidad que proporciona esta función miembro depende del tipo específico de la transacción actual. Por lo tanto, declaramos la función miembro **ejecutar** como una función **virtual pura** en la clase base **Transaccion**. Esto hace a **Transaccion** una clase base abstracta, y obliga a que cualquier clase derivada de **Transaccion** que deba ser una clase concreta (es decir, **SolicitudSaldo**, **Retiro**

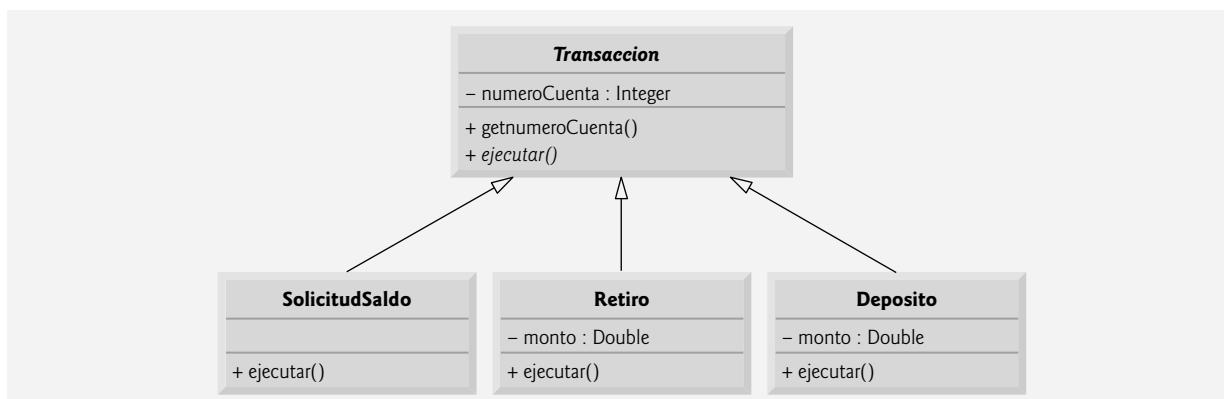


Figura 13.27 | Diagrama de clases que modela la relación de generalización entre la clase base **Transaccion** y las clases derivadas **SolicitudSaldo**, **Retiro** y **Deposito**.

y **Deposito**) implemente la función miembro **virtual pura ejecutar** para que la clase derivada sea concreta. UML requiere que coloquemos los nombres de clases abstractas (y las funciones **virtual puras; operaciones abstractas** en UML) en cursivas, por lo cual **Transaccion** y su función miembro **ejecutar** aparecen en cursivas en la figura 13.27. Observe que la operación **ejecutar** no está en cursivas en las clases derivadas **SolicitudSaldo**, **Retiro** y **Deposito**. Cada clase derivada sobrescribe la función miembro **ejecutar** de la clase base **Transaccion** con una implementación apropiada. Observe que la figura 13.27 incluye la operación **ejecutar** en el tercer compartimiento de las clases **SolicitudSaldo**, **Retiro** y **Deposito**, ya que cada clase tiene una implementación concreta distinta de la función miembro sobrescrita.

Como aprendió en este capítulo, una clase derivada puede heredar la interfaz o la implementación de una clase base. En comparación con una jerarquía diseñada para la herencia de implementación, una diseñada para la herencia de interfaz tiende a tener su funcionalidad a un nivel más bajo en la jerarquía; una clase base indica una o más funciones que cada clase debe definir en la jerarquía, pero las clases derivadas individuales proporcionan sus propias implementaciones de la(s) función(es). La jerarquía de herencia diseñada para el sistema ATM aprovecha este tipo de herencia, la cual proporciona al ATM una forma elegante de ejecutar todas las transacciones “en general”. Cada clase derivada de **Transaccion** hereda ciertos detalles de implementación (por ejemplo, el miembro de datos **numeroCuenta**), pero el beneficio primario de incorporar la herencia en nuestro sistema es que las clases derivadas comparten una interfaz común (por ejemplo, la función miembro **virtual ejecutar**). El ATM puede orientar un apuntador de **Transaccion** a cualquier transacción, y cuando el ATM invoque a **ejecutar** a través de este apuntador, se ejecutará automáticamente la versión de **ejecutar** apropiada a esa transacción (es decir, la versión implementada en el archivo .cpp de esa clase derivada). Por ejemplo, suponga que un usuario opta por realizar una solicitud de saldo. El ATM orienta un apuntador **Transaccion** a un nuevo objeto de la clase **SolicitudSaldo**; el compilador permite esto debido a que una **SolicitudSaldo** es una **Transaccion**. Cuando el ATM utiliza este apuntador para invocar a **ejecutar**, se hace una llamada a la versión de **ejecutar** correspondiente a **SolicitudSaldo**.

Este enfoque polimórfico también facilita la extensibilidad del sistema. Si deseamos crear un nuevo tipo de transacción (por ejemplo, una transferencia de fondos o el pago de un recibo), tan sólo tenemos que crear una clase derivada de **Transaccion** adicional que sobrescriba la función miembro **ejecutar** con una versión apropiada para el nuevo tipo de transacción. Sólo tendríamos que realizar pequeñas modificaciones al código del sistema para permitir que los usuarios seleccionen el nuevo tipo de transacción del menú principal y para que la clase ATM cree instancias y ejecute objetos de la nueva clase derivada. La clase ATM podría ejecutar transacciones del nuevo tipo utilizando el código actual, ya que éste ejecuta todas las transacciones en forma idéntica.

Como aprendió antes en este capítulo, una clase abstracta tal como **Transaccion** es una para la cual el programador nunca tendrá la intención de crear instancias de objetos. Una clase abstracta sólo declara los atributos y comportamientos comunes de sus clases derivadas en una jerarquía de herencia. La clase **Transaccion** define el concepto de lo que significa ser una transacción que tiene un número de cuenta y puede ejecutarse. Tal vez usted se pregunte por qué nos tomamos la molestia de incluir la función miembro **virtual pura ejecutar** en la clase **Transaccion**, si carece de una implementación concreta. En concepto, incluimos esta función miembro porque corresponde al comportamiento que define a todas las transacciones: ejecutarse. Técnicamente, debemos incluir la función miembro **ejecutar** en la clase base **Transaccion**, de manera que la clase ATM (o cualquier otra clase) pueda invocar mediante el polimorfismo a la versión sobrescrita de esta función miembro correspondiente a cada clase derivada, a través de un apuntador o de una referencia **Transaccion**.

Las clases derivadas **SolicitudSaldo**, **Retiro** y **Deposito** heredan el atributo **numeroCuenta** de la clase base **Transaccion**, pero las clases **Retiro** y **Deposito** contienen el atributo adicional **monto** que las diferencia de la clase **SolicitudSaldo**. Las clases **Retiro** y **Deposito** requieren este atributo adicional para almacenar el monto de dinero que el usuario desea retirar o depositar. La clase **SolicitudSaldo** no necesita dicho atributo, puesto que sólo requiere un número de cuenta para ejecutarse. Aun y cuando dos de las tres clases derivadas de **Transaccion** comparten este atributo, no lo colocamos en la clase base **Transaccion**; en la clase base sólo colocamos las características comunes para *todas* las clases derivadas, de manera que las clases derivadas no hereden atributos (y operaciones) innecesarios.

La figura 13.28 presenta un diagrama de clases actualizado de nuestro modelo, en el cual se incorpora la herencia y se introduce la clase **Transaccion**. Modelamos una asociación entre la clase **ATM** y la clase **Transaccion** para mostrar que la clase **ATM**, en cualquier momento dado, está ejecutando una transacción o no lo está (es decir, existen cero o un objetos de tipo **Transaccion** en el sistema, en un momento dado). Como un **Retiro** es un tipo de **Transaccion**, ya no dibujamos una línea de asociación directamente entre la clase **ATM** y la clase **Retiro**; la clase derivada **Retiro** hereda la asociación de la clase base **Transaccion** con la clase **ATM**. Las clases derivadas **SolicitudSaldo** y **Deposito** también heredan esta asociación, que reemplaza las asociaciones anteriormente omitidas entre las clases **SolicitudSaldo** y **Deposito**, y la clase **ATM**. Observe de nuevo el uso de puntas de flecha triangulares huecas para indicar las especializaciones de la clase **Transaccion**, como se indica en la figura 13.27.

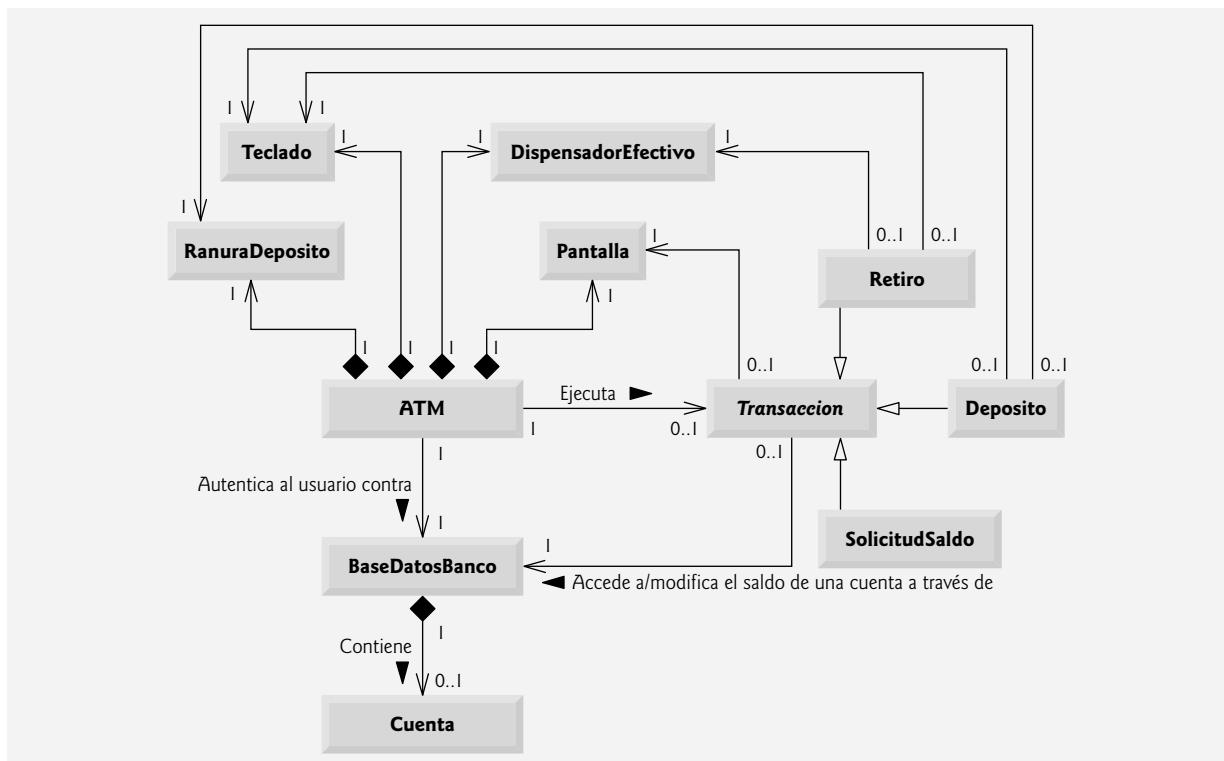


Figura 13.28 | Diagrama de clases del sistema ATM (en el que se incorpora la herencia). Observe que el nombre de la clase abstracta (*Transaccion*) aparece en cursivas.

También agregamos una asociación entre la clase *Transaccion* y la clase *BaseDatosBanco* (figura 13.28). Todos los objetos *Transaccion* requieren una referencia a *BaseDatosBanco*, de manera que puedan acceder a (y modificar) la información de las cuentas. Debido a que cada clase derivada de *Transaccion* hereda esta referencia, ya no tenemos que modelar la asociación entre la clase *Retiro* y *BaseDatosBanco*. Observe que la asociación entre las clases *Transaccion* y *BaseDatosBanco* reemplaza a las asociaciones anteriormente omitidas entre las clases *SolicitudSaldo* y *Deposito*, y la clase *BaseDatosBanco*.

Incluimos una asociación entre la clase *Transaccion* y la clase *Pantalla*, debido a que todos los objetos *Transaccion* muestran los resultados al usuario a través de la *Pantalla*. Cada clase derivada hereda esa asociación. Por ende, ya no incluimos la asociación que modelamos antes entre *Retiro* y *Pantalla*. La clase *Retiro* aún participa en las asociaciones con *DispensadorEfectivo* y *Teclado*. No movemos estas asociaciones a la clase base *Transaccion* debido a que la asociación con el *Teclado* sólo se aplica a las clases *Retiro* y *Deposito*, y la asociación con el *DispensadorEfectivo* sólo se aplica a la clase *Retiro*.

Nuestro diagrama de clases que incorpora la herencia (figura 13.28) también modela a *Deposito* y *SolicitudSaldo*. Mostramos las asociaciones entre *Deposito* y tanto *RanuraDeposito* como *Teclado*. Observe que la clase *SolicitudSaldo* no participa en asociaciones más que las heredadas de la clase *Transaccion*; un objeto *SolicitudSaldo* sólo interactúa con la *BaseDatosBanco* y con la *Pantalla*.

El diagrama de clases de la figura 9.20 mostraba los atributos y las operaciones con marcadores de visibilidad. Ahora presentamos un diagrama de clases modificado en la figura 13.29 que incluye la clase base abstracta *Transaccion*. Este diagrama abreviado no muestra las relaciones de herencia (éstas aparecen en la figura 13.28), sino los atributos y operaciones después de haber empleado la herencia en nuestro sistema. Observe que el nombre de la clase abstracta *Transaccion* y el nombre de la operación abstracta *ejecutar* en la clase *Transaccion* aparecen en cursivas. Para ahorrar espacio, como hicimos en la figura 4.24, no incluimos esos atributos que se muestran mediante las asociaciones en la figura 13.28; sin embargo, los incluimos en la implementación de C++ en el apéndice G. También omitimos todos los parámetros de operación, como hicimos en la figura 9.20; al incorporar la herencia no se ven afectados los parámetros ya modelados en las figuras 6.36 a 6.39.

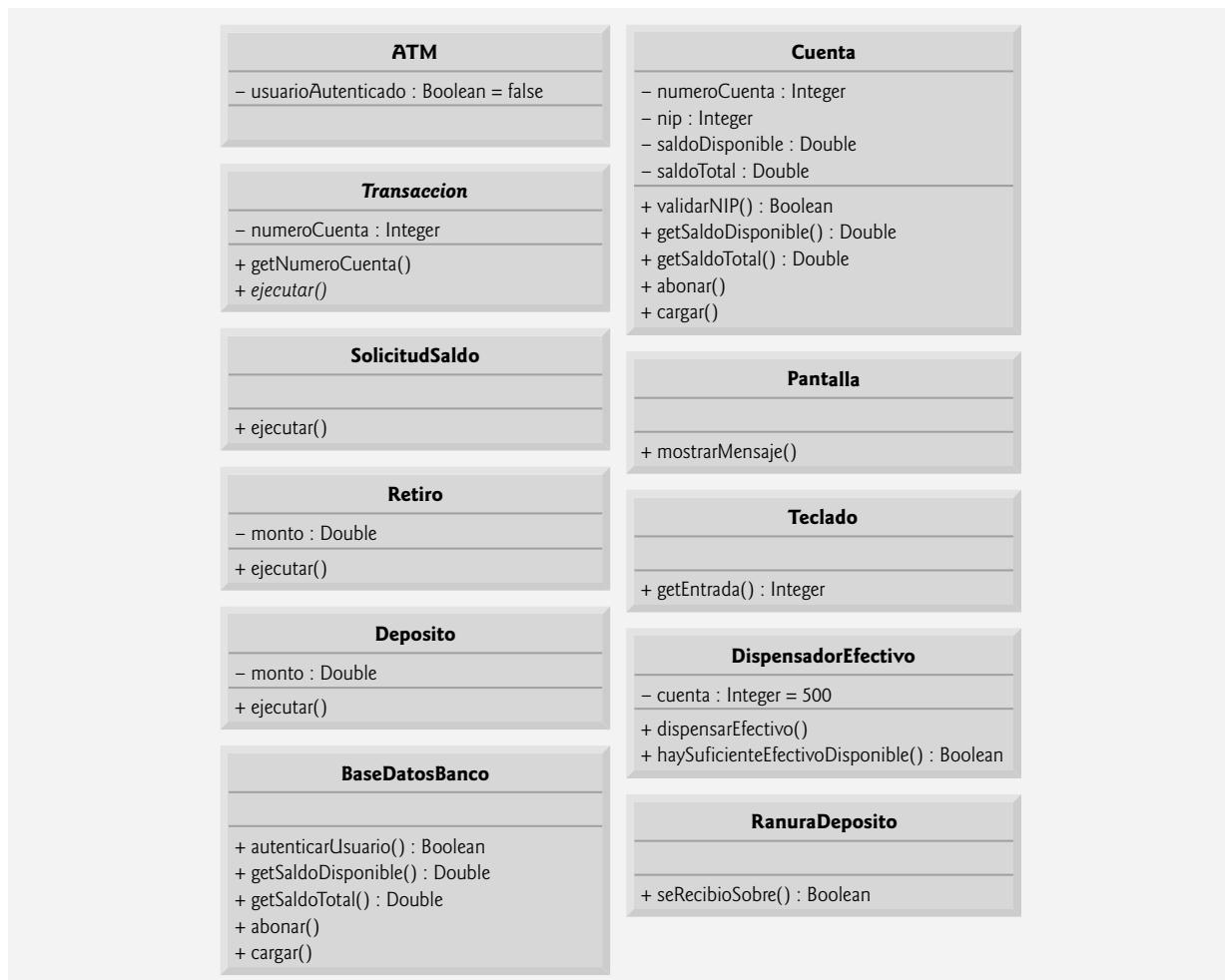


Figura 13.29 | Diagrama de clases después de incorporar la herencia al sistema.



Observación de Ingeniería de Software 13.12

Un diagrama de clases completo muestra todas las asociaciones entre clases, junto con todos los atributos y operaciones para cada clase. Cuando el número de atributos, operaciones y asociaciones de las clases es sustancial (como en las figuras 13.28 y 13.29), una buena práctica que promueve la legibilidad es dividir esta información entre dos diagramas de clases: uno que se enfoca en las asociaciones y el otro en los atributos y operaciones. Sin embargo, al examinar las clases modeladas de esta forma, es imprescindible considerar ambos diagramas de clases para obtener una visión completa de las clases. Por ejemplo, uno debe hacer referencia a la figura 13.28 para observar la relación de herencia entre **Transaccion** y sus clases derivadas, la cual se omite de la figura 13.29.

Implementación del diseño del sistema ATM en el que se incorpora la herencia

En la sección 9.11 empezamos a implementar el diseño del sistema ATM en código de C++. Ahora modificaremos nuestra implementación para incorporar la herencia, usando la clase **Retiro** como ejemplo.

- Si la clase A es una generalización de la clase B, entonces la clase B se deriva (y es una especialización) de la clase A. Por ejemplo, la clase base abstracta **Transaccion** es una generalización de la clase **Retiro**. Por ende, la clase **Retiro** se deriva (y es una especialización) de la clase **Transaccion**. La figura 13.30 contiene una parte del archivo de encabezado de la clase **Retiro**, en la cual la definición de la clase indica la relación de herencia entre **Retiro** y **Transaccion** (línea 9).
- Si la clase A es una clase abstracta y la clase B se deriva de la clase A, entonces la clase B debe implementar las funciones `virtual` puras de la clase A, si la clase B va a ser una clase concreta. Por ejemplo, la clase **Transaccion**

contiene la función `virtual pura ejecutar`, por lo que la clase `Retiro` debe implementar esta función miembro si queremos crear una instancia de un objeto `Retiro`. La figura 13.31 contiene el archivo de encabezado de C++ para la clase `Retiro` de las figuras 13.28 y 13.29. La clase `Retiro` hereda el miembro de datos `numeroCuenta` de la clase base `Transaccion`, por lo que `Retiro` no necesita declarar este miembro de datos. La clase `Retiro` también hereda referencias a las clases `Pantalla` y `BaseDatosBanco` de su clase base `Transaccion`, por lo que no incluimos estas referencias en nuestro código. La figura 13.29 especifica el atributo `monto` y la operación `ejecutar` para la clase `Retiro`. La línea 19 de la figura 13.31 declara un datos miembro para el atributo `monto`. La línea 16 contiene el prototipo de función para la operación `ejecutar`. Recuerde que, para ser una clase concreta, la clase derivada `Retiro` debe proporcionar una implementación concreta de la función `virtual pura ejecutar` en la clase base `Transaccion`. El prototipo en la línea 16 indica nuestra intención de sobrescribir la función `virtual pura` de la clase base. El programador debe proporcionar este prototipo si va a proporcionar una implementación en el archivo `.cpp`. Presentamos esta implementación en el apéndice G. Las referencias `teclado` y `dispensadorEfectivo` (líneas 20 y 21) son datos miembro derivados de las asociaciones de `Retiro` en la figura 13.28. En la implementación de esta clase en el apéndice G, un constructor inicializa estas referencias a objetos actuales. Una vez más, para poder compilar las declaraciones de las referencias en las líneas 20 y 21, incluimos las declaraciones anticipadas en las líneas 8 y 9.

```

1 // Fig. 13.30 Retiro.h
2 // Definición de la clase Retiro que representa una transacción de retiro
3 #ifndef RETIRO_H
4 #define RETIRO_H
5
6 #include "Transaccion.h" // definición de la clase Transaccion
7
8 // la clase Retiro se deriva de la clase base Transaccion
9 class Retiro : public Transaccion
10 {
11 }; // fin de la clase Retiro
12
13 #endif // RETIRO_H

```

Figura 13.30 | Definición de la clase `Retiro` que se deriva de `Transaccion`.

```

1 // Fig. 13.31: Retiro.h
2 // Definición de la clase Retiro que representa una transacción de retiro
3 #ifndef RETIRO_H
4 #define RETIRO_H
5
6 #include "Transaccion.h" // definición de la clase Transaccion
7
8 class Teclado; // declaración anticipada de la clase Teclado
9 class DispensadorEfectivo // declaración anticipada de la clase DispensadorEfectivo
10
11 // la clase Retiro se deriva de la clase base Transaccion
12 class Retiro : public Transaccion
13 {
14 public:
15     // función miembro que sobrescribe a ejecutar en la clase base Transaccion
16     virtual void ejecutar(); // realiza la transacción
17 private:
18     // atributos
19     double monto; // monto a retirar
20     Teclado &teclado; // referencia al teclado del ATM
21     CashDispenser &cashDispenser; // referencia al dispensador de efectivo del ATM
22 }; // fin de la clase Retiro
23
24 #endif // RETIRO_H

```

Figura 13.31 | Archivo de encabezado de la clase `Retiro`, con base en las figuras 13.28 y 13.29.

Conclusión del caso de estudio del ATM

Esto concluye nuestro diseño orientado a objetos del sistema ATM. En el apéndice G aparece una implementación completa del sistema ATM, en 877 líneas de código. Esta implementación funcional utiliza nociones clave de programación, incluyendo clases, objetos, encapsulamiento, visibilidad, composición, herencia y polimorfismo. El código contiene muchos comentarios y se conforma a las prácticas de codificación que usted ya ha aprendido. Dominar este código es un maravilloso logro culminante para usted, después de estudiar los capítulos 1 a 13.

Ejercicios de autoevaluación del caso de estudio de Ingeniería de Software

- 13.1** UML utiliza una flecha con una _____ para indicar una relación de generalización.
- punta con relleno sólido
 - punta triangular sin relleno
 - punta hueca en forma de diamante
 - punta lineal
- 13.2** Indique si el siguiente enunciado es *verdadero* o *falso*, y si es *falso*, explique por qué: UML requiere que subrayemos los nombres de las clases abstractas y los nombres de las operaciones.
- 13.3** Escriba un archivo de encabezado de C++ para empezar a implementar el diseño para la clase `Transaccion` que se especifica en las figuras 13.28 y 13.29. Asegúrese de incluir las referencias `private`, con base en las asociaciones de la clase `Transaccion`. Asegúrese también de incluir las funciones `set public` para cualquiera de los datos miembro `private` que deben utilizar las clases derivadas para realizar sus tareas.

Respuestas a los ejercicios de autoevaluación del caso de estudio de Ingeniería de Software

- 13.1** b.
- 13.2** Falso. UML requiere que se escriban los nombres de las clases abstractas y de las operaciones en cursiva.
- 13.3** El diseño para la clase `Transaccion` produce el archivo de encabezado de la figura 13.32. En la implementación del apéndice G, un constructor inicializa los atributos de referencia `private pantalla` y `baseDatosBanco` con objetos actuales, y las funciones miembro `getPantalla` y `getBaseDatosBanco` acceden a estos atributos. Estas funciones miembro permiten que las clases derivadas de `Transaccion` accedan a la pantalla del ATM e interactúen con la base de datos del banco.

```

1 // Fig. 13.32: Transaccion.h
2 // Definición de la clase base abstracta Transaccion.
3 #ifndef TRANSACCION_H
4 #define TRANSACCION_H
5
6 class Pantalla; // declaración anticipada de la clase Pantalla
7 class BaseDatosBanco; // declaración anticipada de la clase BaseDatosBanco
8
9 class Transaccion
10 {
11 public:
12     int getNumeroCuenta(); // devuelve el número de cuenta
13     Pantalla &getPantalla(); // devuelve la referencia a la pantalla
14     BaseDatosBanco &getBaseDatosBanco(); // devuelve la referencia a la base de datos del banco
15
16     // función virtual pura para realizar la transacción
17     virtual void ejecutar() = 0; // se sobrescribe en las clases derivadas
18 private:
19     int numeroCuenta; // indica la cuenta involucrada
20     Pantalla &pantalla; // referencia a la pantalla del ATM
21     BaseDatosBanco &baseDatosBanco; // referencia a la base de datos de información de las
22     cuentas
23 };
24 #endif // TRANSACCION_H

```

Figura 13.32 | Archivo de encabezado de la clase `Transaccion`, basado en las figuras 13.28 y 13.29.

13.11 Repaso

En este capítulo hablamos sobre el polimorfismo, que nos permite “programar en forma general” en vez de “programar en forma específica”, y mostramos cómo esto hace a los programas más extensibles. Empezamos con un ejemplo sobre cómo el polimorfismo permitiría a un administrador de pantalla mostrar varios objetos “espaciales”. Después demostramos cómo se pueden orientar los apuntadores de clases base y de clases derivadas a objetos de clases base y de clases derivadas. Dijimos que es natural orientar apuntadores de clase base a objetos de clase base, al igual que orientar apuntadores de clase derivada a objetos de clase derivada. También es natural orientar apuntadores de clase base a apuntadores de clase derivada, ya que un objeto de una clase derivada *es un* objeto de su clase base. El lector aprendió por qué es peligroso orientar apuntadores de clase derivada a objetos de clase base, y por qué el compilador no permite dichas asignaciones. Presentamos las funciones `virtual`, las cuales permiten llamar a las funciones apropiadas cuando se hace referencia a objetos en varios niveles de una jerarquía de herencia (en tiempo de ejecución) mediante apuntadores de clase base. A esto se le conoce como vinculación dinámica o postergada. Después hablamos sobre las funciones `virtual` puras (funciones `virtual` que no proporcionan una implementación) y las clases abstractas (clases con una o más funciones `virtual` puras). También aprendió que las clases abstractas no se pueden utilizar para instanciar objetos, mientras que las clases concretas sí se pueden usar. Después demostramos el uso de clases abstractas en una jerarquía de herencia. El lector aprendió cómo trabaja el polimorfismo “detrás de las cámaras” con `vtables` que el compilador crea. Hablamos sobre la conversión descendente de apuntadores de clase base a apuntadores de clase derivada, para permitir a un programa llamar a las funciones miembro que sólo pertenecen a la clase derivada. El capítulo concluyó con una discusión de los destructores `virtual` y cómo aseguran éstos que se ejecuten todos los destructores apropiados en una jerarquía de herencia en un objeto de una clase derivada, cuando ese objeto se elimina mediante un apuntador de la clase base.

En el siguiente capítulo hablaremos sobre las plantillas, una sofisticada característica de C++ que permite a los programadores definir una familia de clases o funciones relacionadas con un solo segmento de código.

Resumen

Sección 13.1 Introducción

- El polimorfismo nos permite “programar en forma general” en vez de “programar en forma específica”.
- El polimorfismo nos permite escribir programas que procesen objetos de clases que sean parte de la misma jerarquía de clases, como si todos fueran objetos de la clase base de la jerarquía.
- Con el polimorfismo, podemos diseñar e implementar sistemas que sean fácilmente extensibles; pueden agregarse nuevas clases con poca (o ninguna) modificación a las porciones generales del programa, siempre y cuando las nuevas clases sean parte de la jerarquía de herencias que el programa procesa en forma genérica. Las únicas partes de un programa que deben alterarse para dar cabida a nuevas clases son aquellas que requieren un conocimiento directo de las nuevas clases que agregamos a la jerarquía.
- La información de tipos en tiempo de ejecución (RTTI) y la conversión dinámica de tipos permiten a un programa determinar el tipo de un objeto en tiempo de ejecución, y actuar sobre ese objeto de manera acorde.

Sección 13.2 Ejemplos de polimorfismo

- Con el polimorfismo, una función puede ocasionar que ocurran distintas acciones, dependiendo del tipo del objeto en el que se invoca la función.
- Con las funciones `virtual` y el polimorfismo, es posible diseñar e implementar sistemas que puedan extenderse con mayor facilidad. Los programas pueden escribirse para procesar objetos de tipos que tal vez no existían cuando el programa estaba en desarrollo.

Sección 13.3 Relaciones entre los objetos en una jerarquía de herencia

- C++ permite el polimorfismo: la habilidad de que los objetos de distintas clases relacionadas por la herencia respondan de manera distinta a la misma llamada a una función miembro.
- El polimorfismo se implementa a través de funciones `virtual` y vinculación dinámica.
- Cuando se realiza una solicitud a través de un apuntador o referencia de la clase base para usar una función `virtual`, C++ selecciona la función sobrescrita correcta en la clase derivada apropiada, asociada con el objeto.
- Si una función `virtual` se llama mediante la referencia a un objeto específico por su nombre y mediante el uso del operador punto de selección de miembros, la referencia se resuelve en tiempo de compilación (a esto se le conoce como vinculación estática); la función `virtual` que se llama es la que está definida para la clase de ese objeto específico.
- Las clases derivadas pueden proporcionar sus propias implementaciones de una función `virtual` de la clase base si es necesario, pero si no, se utiliza la implementación de la clase base.

Sección 13.4 Tipos de campos e instrucciones switch

- La programación polimórfica con funciones virtual puede eliminar la necesidad de la lógica de switch. El programador puede usar el mecanismo de funciones virtual para realizar la lógica equivalente de manera automática, evitando con ello los tipos de errores que se asocian comúnmente con la lógica de switch.

Sección 13.5 Clases abstractas y funciones virtual puras

- En muchas situaciones es conveniente definir clases abstractas para las que nunca se tendrá la intención de crear objetos. Como estas clases se utilizan sólo como clases base, nos referimos a ellas como clases base abstractas. No se pueden instanciar objetos de una clase base abstracta.
- Las clases a partir de las cuales se instancian objetos se conocen como clases concretas.
- Una clase se hace abstracta al declarar una o más de sus funciones virtual como puras. Una función virtual pura tiene un especificador puro (=0) en su declaración.
- Si una clase se deriva de una clase con una función virtual pura y esa clase derivada no proporciona una definición para esa función virtual pura, entonces esa función virtual pura sigue siendo pura en la clase derivada. En consecuencia, la clase derivada es también una clase abstracta.
- Aunque no podemos instanciar objetos de clases base abstractas, podemos declarar apuntadores y referencias a objetos de clases base abstractas. Dichos apuntadores y referencias se pueden utilizar para permitir manipulaciones polimórficas de los objetos de clases derivadas que se instancian de clases derivadas concretas.

Sección 13.7 (Opcional) Polimorfismo, funciones virtuales y vinculación dinámica “detrás de las cámaras”

- La vinculación dinámica requiere que en tiempo de ejecución, la llamada a una función miembro virtual se dirija a la versión de la función virtual apropiada para esa clase. Una tabla de funciones virtual, conocida como vtable, se implementa como un arreglo que contiene apuntadores a funciones. Cada clase con funciones virtual tiene una vtable. Para cada función virtual en la clase, la vtable tiene una entrada que contiene un apuntador a función que apunta a la versión de la función virtual que se debe usar para un objeto de esa clase. La función virtual a utilizar para una clase específica podría ser la función definida en esa clase, o podría ser una función heredada ya sea de manera directa o indirecta de una clase base en un nivel más alto en la jerarquía.
- Cuando una clase base proporciona una función miembro virtual, las clases derivadas pueden sobrescribir a la función virtual, pero no tienen que sobrescribirla. Por ende, una clase derivada puede usar una versión de una función virtual correspondiente a la clase base.
- Cada objeto de una clase con funciones virtual contiene un apuntador a la vtable para esa clase. Cuando se hace una llamada a una función desde un apuntador de la clase base a un objeto de la clase derivada, se obtiene el apuntador a la función apropiada en la vtable y se desreferencia para completar la llamada en tiempo de ejecución. Esta búsqueda en la vtable y la desreferencia del apuntador requieren una sobrecarga nominal en tiempo de ejecución.
- Cualquier clase que tenga uno o más apuntadores a 0 en su vtable es una clase abstracta. Las clases sin apuntadores a 0 en la vtable son clases concretas.
- Se agregan nuevos tipos de clases a los sistemas con regularidad. Las nuevas clases se acomodan mediante la vinculación dinámica (también conocida como vinculación postergada). El tipo de un objeto no se necesita conocer en tiempo de compilación para poder compilar una llamada a una función virtual. En tiempo de ejecución, se llamará a la función miembro apropiada para el objeto al que apunte el apuntador.

Sección 13.8 Ejemplo práctico: sistema de nómina mediante el uso de polimorfismo e información de tipos en tiempo de ejecución con conversión descendente, dynamic_cast, typeid y type_info

- El operador dynamic_cast comprueba el tipo del objeto al que apunta el apuntador, y determina si este tipo tiene una relación “es un” con el tipo al que se está convirtiendo el apuntador. Si hay una relación “es un”, dynamic_cast devuelve la dirección del objeto. Si no, dynamic_cast devuelve 0.
- El operador typeid devuelve una referencia a un objeto de la clase type_info que contiene información acerca del tipo de su operando, incluyendo el nombre del tipo. Para usar typeid, el programa debe incluir el archivo de encabezado <type_info>.
- Al invocarse, la función miembro name de type_info devuelve una cadena basada en apuntador que contiene el nombre del tipo que representa el objeto type_info.
- Los operadores dynamic_cast y typeid son parte de la característica de información de tipos en tiempo de ejecución (RTTI) de C++, la cual permite a un programa determinar el tipo de un objeto en tiempo de ejecución.

Sección 13.9 Destructores virtuales

- Debemos declarar el destructor de la clase base como virtual si la clase contiene funciones virtual. Esto hace a todos los destructores de la clase derivada virtual, aun y cuando no tengan el mismo nombre que el destructor de la clase base. Si un objeto en la jerarquía se destruye de manera explícita al aplicar el operador delete a un apuntador de la clase base que apunte a un objeto de la clase derivada, se llama al destructor para la clase apropiada. Despues de que se ejecuta el destructor de

una clase derivada, se ejecutan los destructores para todas las clases base de esa clase hacia arriba en la jerarquía; el destructor de la clase raíz se ejecuta al último.

Terminología

apuntador a la <i>vtable</i> del objeto	herencia de implementación
apuntador a <i>vtable</i>	herencia de interfaz
apuntador de clase base a objeto de clase base	lógica de <i>switch</i>
apuntador de clase base a objeto de clase derivada	manipulación de apuntadores peligrosa
apuntador de clase derivada a objeto de clase base	<i>name</i> , función de la clase <i>type_info</i>
apuntador de clase derivada a objeto de clase derivada	polimorfismo
clase abstracta	polimorfismo como alternativa a la lógica de <i>switch</i>
clase base abstracta	programación en forma específica
clase concreta	programación en forma general
clase iteradora	programación polimórfica
conversión descendente	RTTI (información de tipos en tiempo de ejecución)
conversión dinámica de tipos	sobrescribir una función
desplazamiento en una <i>vtable</i>	tabla de funciones virtuales (<i>vtable</i>)
destructor no virtual	< <i>type_info</i> >, archivo de encabezado
destructor <i>virtual</i>	<i>type_info</i> , clase
determinar la función a ejecutar en forma dinámica	<i>typeid</i> , operador
<i>dynamic_cast</i>	vinculación dinámica
especificador puro (con funciones virtuales)	vinculación estática
flujo de control de una llamada a una función <i>virtual</i>	vinculación postergada
función <i>virtual</i>	<i>virtual</i> , palabra clave
función <i>virtual</i> pura	<i>vtable</i>

Ejercicios de autoevaluación

13.1 Complete las siguientes oraciones:

- Tratar a un objeto de la clase base como un _____ puede provocar errores lógicos.
- El polimorfismo ayuda a eliminar la lógica de _____.
- Si una clase contiene al menos una función *virtual* pura, es una clase _____.
- Las clases a partir de las cuales pueden instanciarse objetos se llaman clases _____.
- El operador _____ se puede usar para realizar conversiones descendentes con los apuntadores de la clase base en forma segura.
- El operador *typeid* devuelve una referencia a un objeto _____.
- El _____ implica el uso de un apuntador o referencia de la clase base para invocar funciones *virtual* en objetos de la clase base y la clase derivada.
- Las funciones que pueden sobrescribirse se declaran mediante la palabra clave _____.
- Al proceso de convertir un apuntador de la clase base en un apuntador de la clase derivada se le conoce como _____.

13.2 Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

- Todas las funciones *virtual* en una clase base abstracta se deben declarar como funciones *virtual* puras.
- Es peligroso tratar de hacer referencia a un objeto de la clase derivada con un manejador de la clase base.
- Para hacer a una clase abstracta, se declara como *virtual*.
- Si una clase base declara a una función *virtual* pura, una clase derivada debe implementar la función para convertirse en una clase concreta.
- La programación polimórfica puede eliminar la necesidad de la lógica de *switch*.

Respuestas a los ejercicios de autoevaluación

13.1 a) objeto de la clase derivada. b) switch. c) abstracta. d) concretas. e) *dynamic_cast*. f) *type_info*. g) Polimorfismo. h) *virtual*. i) conversión descendente.

13.2 a) Falso. Una clase base abstracta puede incluir funciones virtuales con implementaciones. b) Falso. Es peligroso hacer referencia a un objeto de la clase base con un manejador de la clase derivada. c) Falso. Las clases nunca se declaran *virtual*. En vez de ello, una clase se hace abstracta al incluir por lo menos una función *virtual* pura en ella. d) Verdadero. e) Verdadero.

Ejercicios

- 13.3** ¿Cómo es que el polimorfismo le permite programar “en forma general”, en lugar de hacerlo “en forma específica”? Hable sobre las ventajas clave de la programación “en forma general”.
- 13.4** Hable sobre los problemas de programar con la lógica de `switch`. Explique por qué el polimorfismo puede ser una alternativa efectiva al uso de la lógica de `switch`.
- 13.5** Explique la diferencia entre heredar la interfaz y heredar la implementación. ¿En qué difieren las jerarquías de herencia diseñadas para heredar la interfaz, de las jerarquías diseñadas para heredar la implementación?
- 13.6** ¿Qué son las funciones `virtual`? Describa una circunstancia en la que las funciones `virtual` serían apropiadas.
- 13.7** Explique la diferencia entre la vinculación estática y la vinculación dinámica. Explique el uso de las funciones `virtual` y la `vtable` en la vinculación dinámica.
- 13.8** Explique la diferencia entre las funciones `virtual` y las funciones `virtual` puras.
- 13.9** Sugiera uno o más niveles de clases base abstractas para la jerarquía de `Figura` que vimos en este capítulo, y que se muestra en la figura 12.3. (El primer nivel es `Figura`, y el segundo nivel consiste en las clases `FiguraBidimensional` y `FiguraTridimensional`.)
- 13.10** ¿Cómo promueve el polimorfismo la extensibilidad?
- 13.11** Se le ha pedido que desarrolle un simulador de vuelo que tenga salidas gráficas elaboradas. Explique por qué la programación polimórfica podría ser especialmente efectiva para un problema de esta naturaleza.
- 13.12** (*Modificación al sistema de nómina*) Modifique el sistema de nómina de las figuras 13.13 a 13.23 para incluir el miembro de datos `private` llamado `fechaNacimiento` en la clase `Empleado`. Use la clase `Fecha` de las figuras 11.12 y 11.13 para representar el cumpleaños de un empleado. Suponga que la nómina se procesa una vez al mes. Cree un vector de referencias `Empleado` para guardar los diversos objetos `Empleado`. En un ciclo, calcule la nómina para cada `Empleado` (mediante el polimorfismo) y agregue una bonificación de \$100.00 a la cantidad de pago de nómina de la persona, si el mes actual es el mes en el que ocurre el cumpleaños de ese `Empleado`.
- 13.13** (*Jerarquía de figuras*) Implemente la jerarquía de `Figura` diseñada en el ejercicio 12.7 (que se basa en la jerarquía de la figura 12.3). Cada `FiguraBidimensional` debe contener la función `getArea` para calcular el área de la figura bidimensional. Cada `FiguraTridimensional` debe tener las funciones miembro `getArea` y `getVolumen` para calcular el área superficial y el volumen, respectivamente, de la figura tridimensional. Cree un programa que utilice un vector de apuntadores `Figura` a objetos de cada clase concreta en la jerarquía. El programa deberá imprimir el objeto al cual apunta cada elemento del vector. Además, en el ciclo que procesa a todas las figuras en el vector, determine si cada figura es `FiguraBidimensional` o `FiguraTridimensional`. Si es `FiguraBidimensional`, muestre su área. Si es `FiguraTridimensional`, muestre su área y su volumen.
- 13.14** (*Administrador de pantallas polimórfico mediante el uso de la jerarquía de figuras*) Desarrolle un paquete de gráficos básicos. Use la jerarquía de `Figura` implementada en el ejercicio 13.13. Limítese a figuras bidimensionales tales como cuadrados, rectángulos, triángulos y círculos. Interactúe con el usuario. Deje que éste especifique la posición, tamaño, figura y caracteres de relleno a usar para dibujar cada figura. El usuario puede especificar más de un objeto de la misma figura. A medida que cree cada figura, coloque un apuntador `Figura *` a cada nuevo objeto `Figura` en un arreglo. Cada clase de `Figura` deberá tener ahora su propia función miembro `dibujar`. Escriba un administrador de pantallas polimórfico que recorra el arreglo, enviando mensajes `dibujar` a cada objeto en el arreglo para formar una imagen de la pantalla. Vuelva a dibujar la imagen de la pantalla cada vez que el usuario especifique una figura adicional.
- 13.15** (*Jerarquía de herencia Paquete*) Use la jerarquía de herencia `Paquete` creada en el ejercicio 12.9 para crear un programa que muestre la información de la dirección y que calcule los costos de envío para varios objetos `Paquete`. El programa debe contener un vector de apuntadores `Paquete` a objetos de las clases `PaqueteDosDias` y `PaqueteNocturno`. Itere a través del vector para procesar los objetos `Paquete` mediante el polimorfismo. Para cada `Paquete`, invoque a funciones `get` para obtener la información de las direcciones del emisor y del receptor, y después imprimir las dos direcciones como deben aparecer en las etiquetas de envío. Además, llame a la función miembro `calcularCosto` de cada `Paquete` e imprima el resultado. Lleve la cuenta del costo de envío total para todos los objetos `Paquete` en el vector, y muestre este total cuando termine el ciclo.
- 13.16** (*Programa bancario polimórfico mediante el uso de la jerarquía Cuenta*) Desarrolle un programa bancario polimórfico mediante el uso de la jerarquía `Cuenta` creada en el ejercicio 12.10. Cree un vector de apuntadores `Cuenta` a objetos `CuentaAhorros` y `CuentaCheques`. Para cada `Cuenta` en el vector, permita al usuario especificar un monto de dinero a retirar de la `Cuenta`, usando la función miembro `cargar`, y un monto de dinero a depositar en la `Cuenta` mediante el uso de la función miembro `abonar`. A medida que procese cada `Cuenta`, determine su tipo. Si una `Cuenta` es una `CuentaAhorros`, calcule el monto de interés que se debe a la `Cuenta` usando la función miembro `CalcularInteres`, y después agregue el interés al saldo actual mediante la función miembro `abonar`. Después de procesar una `Cuenta`, imprima el saldo de la cuenta actualizado que se obtiene al invocar a la función miembro `getSaldo` de la clase base.



*Detrás de ese patrón
externo las tenues figuras
se hacen más claras día
con día. Siempre es la
misma figura, sólo que muy
numerosa.*

—Charlotte Perkins Gilman

*Cada hombre inteligente
ve el mundo desde un
ángulo distinto al de sus
compañeros.*

—Havelock Ellis

*... nuestra individualidad
especial, a diferencia
de nuestra humanidad
genérica.*

—Oliver Wendell Holmes, Sr.

Plantillas

OBJETIVOS

En este capítulo aprenderá a:

- Usar las plantillas de funciones para crear de manera conveniente un grupo de funciones relacionadas (sobrecargadas).
- Conocer la diferencia entre las plantillas de funciones y las especializaciones de plantillas de funciones.
- Usar las plantillas de clase para crear un grupo de tipos relacionados.
- Conocer la diferencia entre las plantillas de clases y las especializaciones de plantillas de clases.
- Sobrecargar las plantillas de funciones.
- Comprender las relaciones entre las plantillas, las funciones friend, la herencia y los miembros estáticos.

- 14.1 Introducción
- 14.2 Plantillas de funciones
- 14.3 Sobrecarga de plantillas de funciones
- 14.4 Plantillas de clases
- 14.5 Parámetros sin tipo y tipos predeterminados para las plantillas de clases
- 14.6 Notas acerca de las plantillas y la herencia
- 14.7 Notas acerca de las plantillas y funciones friend
- 14.8 Notas acerca de las plantillas y miembros static
- 14.9 Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)

14.1 Introducción

En este capítulo hablaremos sobre una de las características de reutilización de software más poderosas de C++, a saber las plantillas. Las **plantillas de funciones** y las **plantillas de clases** permiten a los programadores especificar, con un solo segmento de código, un rango completo de funciones relacionadas (sobrecargadas) (llamadas **especializaciones de plantillas de funciones**) o un rango completo de clases relacionadas (llamadas **especializaciones de plantillas de clases**). A esta técnica se le conoce como **programación genérica**.

Podríamos escribir una sola plantilla de función para una función de ordenamiento de arreglos, y después hacer que C++ genere especializaciones de plantillas de funciones separadas que ordenen arreglos `int`, arreglos `float`, arreglos `string`, y así sucesivamente. En el capítulo 6 presentamos las plantillas de funciones. En este capítulo presentaremos una discusión y un ejemplo adicionales.

Podríamos escribir una sola plantilla de clase para una clase tipo pila, y después hacer que C++ genere especializaciones de plantillas de clases separadas, como una clase “pila de `int`”, una clase “pila de `float`”, una clase “pila de `string`”, y así sucesivamente.

Observe la diferencia entre las plantillas y las especializaciones de plantillas: las plantillas de funciones y las plantillas de clases son como las plantillas que usamos para trazar figuras; las especializaciones de plantillas de funciones y las especializaciones de plantillas de clases son como los trazos separados que tienen todos la misma figura, pero podrían, por ejemplo, dibujarse en distintos colores.

En este capítulo presentaremos una plantilla de función y una plantilla de clase. También consideraremos las relaciones entre las plantillas y otras características de C++, como la sobrecarga, la herencia, las funciones friend y los miembros `static`. El diseño y los detalles de los mecanismos de las plantillas que veremos aquí se basan en el trabajo de Bjarne Stroustrup que presentó en su artículo “*Parameterized Types for C++*”, que se publicó en la conferencia “*Proceedings of the USENIX C++ Conference*” llevada a cabo en Denver, Colorado, en octubre de 1988.

Este capítulo es sólo una introducción a las plantillas. El capítulo 22, Biblioteca de plantillas estándar (STL), presenta un tratamiento detallado de las clases contenedoras de plantillas, los iteradores y algoritmos de la STL. El capítulo 22 contiene docenas de ejemplos basados en plantillas de código activo, que ilustran técnicas de programación de plantillas más sofisticadas que las que utilizamos aquí.



Observación de Ingeniería de Software 14.1

La mayoría de los compiladores de C++ requieren que la definición completa de una plantilla aparezca en el archivo de código fuente cliente que utiliza la plantilla. Por esta razón y por motivos de reutilización, las plantillas se definen a menudo en archivos de encabezado, que después se incluyen (mediante #include) en los archivos de código fuente cliente apropiados. Para las plantillas de clases, esto significa que las funciones miembro también están definidas en el archivo de encabezado.

14.2 Plantillas de funciones

Por lo general, las funciones sobrecargadas realizan operaciones *similares* o *idénticas* en distintos tipos de datos. Si las operaciones son *idénticas* para cada tipo, pueden expresarse en forma más compacta y conveniente mediante el uso de las plantillas de funciones. Al principio, se escribe una sola definición de plantilla de función. Con base en los tipos de los argumentos que se proporcionan de manera explícita, o que se infieren de las llamadas a esta función, el compilador genera funciones de código fuente separadas (es decir, especializaciones de plantillas de funciones) para

manejar cada llamada a una función en forma apropiada. En C, esta tarea puede realizarse mediante el uso de **macros** creadas con la directiva del preprocesador `#define` (vea el apéndice F, Preprocesador). Sin embargo, las macros pueden tener graves efectos secundarios y no permitir que el compilador realice la comprobación de tipos. Las plantillas de funciones proporcionan una solución completa, al igual que las macros, pero permiten la comprobación de tipos completa.



Tip para prevenir errores 14.1

Al igual que las macros, las plantillas de funciones permiten la reutilización de software. A diferencia de las macros, las plantillas de funciones ayudan a eliminar muchos tipos de errores a través del escrutinio de la comprobación de tipos completa en C++.

Todas las **definiciones de plantillas de funciones** empiezan con la palabra clave `template` seguida de una lista de **parámetros de plantilla** para la plantilla de función encerrada entre los signos (`<` y `>`); a cada parámetro de plantilla que representa un tipo se le debe anteponer cualquiera de las palabras clave intercambiables `class` o `typename`, como en

```
template< typename T >
o
template< class TipoElemento >
o
template< typename TipoBorde, typename TipoRelleno >
```

Los parámetros de plantilla de los tipos de la definición de una plantilla de función se utilizan para especificar los tipos de los argumentos para la función, para especificar el tipo de valor de retorno de la función y para declarar variables dentro de la misma. La definición de la función va después, y tiene la misma apariencia que cualquier otra definición de función. Observe que las palabras clase `typename` y `class` que se utilizan para especificar los parámetros de plantilla de función en realidad significan “cualquier tipo integrado, o tipo definido por el usuario”.



Error común de programación 14.1

Si no se coloca la palabra clave `class` o la palabra clave `typename` antes de cada parámetro de plantilla de tipo de una plantilla de función, se produce un error de sintaxis.

Ejemplo: la plantilla de función `imprimirArreglo`

Vamos a examinar la plantilla de función `imprimirArreglo` en la figura 14.1, líneas 8 a 15. La plantilla de función `imprimirArreglo` declara (línea 8) un solo parámetro de plantilla `T` (`T` puede ser cualquier identificador válido) para el tipo del arreglo que va a imprimir la función `imprimirArreglo`; `T` se conoce como un **parámetro de plantilla de tipo**, o **parámetro de tipo**. En la sección 14.5 veremos los parámetros de plantilla sin tipo.

Cuando el compilador detecta una invocación a la función `imprimirArreglo` en el programa cliente (por ejemplo, las líneas 30, 35 y 40), el compilador utiliza sus herramientas de resolución de sobrecarga para encontrar una definición de la función `imprimirArreglo` que coincida mejor con la llamada a la función. En este caso, la única función `imprimirArreglo` con el número apropiado de parámetros es la plantilla de función `imprimirArreglo` (líneas 8 a 15). Consideré la llamada a la función en la línea 30. El compilador compara el tipo del primer argumento de `imprimirArreglo` (`int *` en la línea 30) con el primer parámetro de la plantilla de función `imprimirArreglo` (`const T * const` en la línea 9), y deduce que el argumento sería consistente con el parámetro si se reemplazara el parámetro de tipo `T` con `int`. Después, el compilador sustituye `int` para `T` a lo largo de la definición de plantilla y compila una especialización de `imprimirArreglo` que pueda mostrar un arreglo de valores `int`. En la figura 14.1, el compilador crea tres especializaciones de `imprimirArreglo`: una que espera un arreglo `int`, una que espera un arreglo `double` y una que espera un arreglo `char`. Por ejemplo, la especialización de plantilla de función para el tipo `int` es

```
void imprimirArreglo( const int * const arreglo, int cuenta )
{
    for ( int i = 0; i < cuenta; i++ )
        cout << arreglo[ i ] << " ";
    cout << endl;
} // fin de la función imprimirArreglo
```

El nombre de un parámetro de plantilla se puede declarar sólo una vez en la lista de parámetros de plantilla de un encabezado de plantilla, pero se puede utilizar varias veces en el encabezado y cuerpo de la función. Los nombres de los parámetros de plantilla entre las plantillas de funciones no necesitan ser únicos.

La figura 14.1 demuestra la plantilla de función `imprimirArreglo` (líneas 8 a 15). El programa empieza declarando el arreglo `int` de cinco elementos llamado `a`, el arreglo `double` de siete elementos llamado `b` y el arreglo `char` de seis elementos llamado `c` (líneas 23 a 25, respectivamente). Después, el programa imprime cada arreglo llamando a `imprimirArreglo`; una vez con un primer argumento `a` de tipo `int *` (línea 30), una vez con un primer argumento `b` de tipo `double *` (línea 35) y una vez con un primer argumento `c` de tipo `char *` (línea 40). Por ejemplo, la llamada en la línea 30 causa que el compilador infiera que `T` es `int` y que instancie una especialización de la plantilla de función `imprimir-`

```

1 // Fig 14.1: fig14_01.cpp
2 // Uso de funciones de plantilla.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // definición de la plantilla de función imprimirArreglo
8 template< typename T >
9 void imprimirArreglo( const T * const arreglo, int cuenta )
10 {
11     for ( int i = 0; i < cuenta; i++ )
12         cout << arreglo[ i ] << " ";
13
14     cout << endl;
15 } // fin de la plantilla de función imprimirArreglo
16
17 int main()
18 {
19     const int aCuenta = 5; // tamaño del arreglo a
20     const int bCuenta = 7; // tamaño del arreglo b
21     const int cCuenta = 6; // tamaño del arreglo c
22
23     int a[ aCuenta ] = { 1, 2, 3, 4, 5 };
24     double b[ bCuenta ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
25     char c[ cCuenta ] = "HOLA"; // 5a posición para el carácter nulo
26
27     cout << "El arreglo a contiene:" << endl;
28
29     // Llama a la especialización de plantilla de función entera
30     imprimirArreglo( a, aCuenta );
31
32     cout << "El arreglo b contiene:" << endl;
33
34     // Llama a la especialización de plantilla de función double
35     imprimirArreglo( b, bCuenta );
36
37     cout << "El arreglo c contiene:" << endl;
38
39     // Llama a la especialización de plantilla de función carácter
40     imprimirArreglo( c, cCuenta );
41     return 0;
42 } // fin de main

```

```

El arreglo a contiene:
1 2 3 4 5
El arreglo b contiene:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
El arreglo c contiene:
H O L A

```

Figura 14.1 | Especializaciones de plantillas de funciones de la plantilla de función `imprimirArreglo`.

`Arreglo`, para la cual el parámetro de tipo `T` es `int`. La llamada en la línea 35 causa que el compilador infiera que `T` es `double` y que instancie una segunda especialización de la plantilla de función `imprimirArreglo`, para la cual el parámetro `T` es `double`. La llamada en la línea 40 causa que el compilador infiera que `T` es `char` y que instancie una tercera especialización de la plantilla de función `imprimirArreglo`, para la cual el parámetro de tipo `T` es `char`. Es importante observar que si `T` (línea 8) representa un tipo definido por el usuario (que no lo hace en la figura 14.1), debe haber un operador de inserción de flujo sobre cargado para ese tipo; en caso contrario, el primer operador de inserción de flujo en la línea 12 no se compilará.



Error común de programación 14.2

Si se invoca a una plantilla con un tipo definido por el usuario, y si esa plantilla utiliza funciones u operadores (por ejemplo, ==, +, <=) con objetos de ese tipo de clase, entonces esas funciones y operadores deben sobre cargarse para el tipo definido por el usuario. Olvidar sobre cargar dichos operadores produce errores de compilación.

En este ejemplo, el mecanismo de las plantillas evita que tengamos que escribir tres funciones sobre cargadas separadas con los prototipos

```
void imprimirArreglo( const int *, int );
void imprimirArreglo( const double *, int );
void imprimirArreglo( const char *, int );
```

todos los cuales utilizan el mismo código, excepto por el tipo `T` (según su uso en la línea 9).



Tip de rendimiento 14.1

Aunque las plantillas ofrecen beneficios de reutilización de software, recuerde que las especializaciones múltiples de plantillas de funciones y las especializaciones múltiples de plantillas de clases se instancian en un programa (en tiempo de compilación), a pesar del hecho de que las plantillas sólo se escriben una vez. Estas copias pueden consumir una cantidad considerable de memoria. Por lo general esto no es un problema, debido a que el código generado por la plantilla es del mismo tamaño que el código que se hubiera escrito para producir las funciones sobre cargadas separadas.

14.3 Sobre carga de plantillas de funciones

Las plantillas de funciones y la sobre carga están estrechamente relacionadas. Las especializaciones de plantillas de funciones que se generan de una plantilla de función tienen todas el mismo nombre, por lo que el compilador utiliza la resolución de sobre carga para invocar a la función apropiada.

Una plantilla de función se puede sobre cargar de varias formas. Podemos proporcionar otras plantillas de funciones que especifiquen el mismo nombre de función pero distintos parámetros. Por ejemplo, la plantilla de función `imprimirArreglo` de la figura 14.1 podría sobre cargarse con otra plantilla de función `imprimirArreglo` con los parámetros adicionales `subindiceInferior` y `subindiceSuperior` para especificar la porción del arreglo a imprimir (vea el ejercicio 14.4).

Una plantilla de función también puede sobre cargarse al proporcionar a las funciones que no son de plantilla el mismo nombre, pero distintos argumentos. Por ejemplo, la plantilla de función `imprimirArreglo` de la figura 14.1 podría sobre cargarse con una versión que no es de plantilla, y que imprime específicamente un arreglo de cadenas de caracteres en un formato tabular ordenado (vea el ejercicio 14.5).

El compilador realiza un proceso de concordancia para determinar qué función debe llamar cuando se invoca a una función. En primer lugar, el compilador busca todas las plantillas de función que coincidan con la función nombrada en la llamada a la función, y crea especializaciones con base en los argumentos de la llamada a la función. Despues, el compilador busca todas las funciones ordinarias que coincidan con la función nombrada en la llamada a la función. Si una de las funciones ordinarias, o de las especializaciones de plantillas de función, es la mejor coincidencia para la llamada a la función, se utiliza esa función ordinaria o esa especialización. Si una función ordinaria y una especialización son igualmente buenas coincidencias para la llamada a la función, entonces se utiliza la función ordinaria. En caso contrario, si hay varias coincidencias para la llamada a la función, el compilador considera que la llamada es ambigua y genera un mensaje de error.



Error común de programación 14.3

Un error de compilación se produce si no se puede encontrar una definición de función que coincida con una llamada a una función específica, o si hay varias coincidencias que el compilador considere ambiguas.

14.4 Plantillas de clases

Es posible comprender el concepto de una “pila” (una estructura de datos en la que insertamos elementos en la parte superior y los recuperamos en el orden “último en entrar, primero en salir”) de manera independiente al tipo de los elementos que se van a colocar en la pila. Sin embargo, para instanciar una pila, se debe especificar un tipo de datos. Esto crea una maravillosa oportunidad para la reutilización de software. Necesitamos los medios para describir la noción de una pila en forma genérica, e instanciar clases que sean versiones de tipo específico de esta clase de pila genérica. C++ proporciona esta herramienta a través de las plantillas de clases.



Observación de Ingeniería de Software 14.2

Las plantillas de clases fomentan la reutilización de software, al permitir que se creen instancias de versiones de tipos específicos de las clases genéricas.

Las plantillas de clases son **tipos parametrizados**, debido a que requieren uno o más parámetros de tipo para especificar cómo personalizar una “clase genérica” para formar una especialización de plantilla de clase.

Para producir una variedad de especializaciones de plantilla de clase, se escribe sólo una definición de plantilla de clase. Cada vez que se necesita una especialización de plantilla de clase adicional, se utiliza una notación simple y concisa, y el compilador escribe el código fuente para la especialización que requerimos. Por ejemplo, nuestra plantilla de clase **Pila** podría ser la base para crear muchas clases **Pila** (como “**Pila de double**”, “**Pila de int**”, “**Pila de char**”, “**Pila de Empleado**”, etc.) utilizadas en un programa.

Creación de la plantilla de clase **Pila< T >**

Observe la definición de la plantilla de clase **Pila** en la figura 14.2. Parece una definición de clase convencional, excepto que se le antepone el siguiente encabezado (línea 6):

```
template< typename T >
```

para especificar una definición de plantilla de clase con el parámetro de tipo **T** que actúa como un receptáculo para el tipo de la clase **Pila** que se vaya a crear. No necesitamos utilizar específicamente el identificador **T**; se puede utilizar cualquier identificador válido. El tipo de elemento a almacenar en esta **Pila** se menciona de manera genérica como **T** a lo largo del encabezado de la clase **Pila** y de las definiciones de las funciones miembro. En un momento le mostraremos cómo se asocia **T** con un tipo específico, como **double** o **int**. Debido a la forma en que está diseñada esta plantilla de clase, hay dos restricciones para los tipos de datos no fundamentales que se utilicen con esta **Pila**: deben tener un constructor predeterminado (para usarlo en la línea 44 y crear el arreglo que almacena los elementos de la pila), y deben soportar el operador de asignación (líneas 56 y 70).

Las definiciones de las funciones miembro de una plantilla de clase son plantillas de función. Las definiciones de las funciones miembro que aparecen fuera de la definición de la plantilla de clase empiezan con el siguiente encabezado:

```
template< typename T >
```

(líneas 40, 51 y 65). Así, cada definición se asemeja a la definición de una función convencional, excepto que el tipo de elemento de **Pila** siempre se lista en forma genérica como el parámetro de tipo **T**. El operador de resolución de ámbito binario se utiliza con el nombre de la plantilla de clase **Stack< T >** (líneas 41, 52 y 66) para enlazar cada una de las definiciones de las funciones miembro con el alcance de la plantilla de clase. En este caso, el nombre de la clase genérica es **Pila< T >**. Cuando se instancia **pilaDouble** como el tipo **Pila< double >**, la especialización de plantilla de función del constructor de **Pila** utiliza **new** para crear un arreglo de elementos de tipo **double** para representar la pila (línea 44). La instrucción

```
pilaPtr = new T[ tamanio ];
```

en la definición de la plantilla de clase **Pila** se genera mediante el compilador en la especialización de plantilla de clase **Pila< double >** como

```
pilaPtr = new double[ tamanio ];
```

```

1 // Fig. 14.2: Pila.h
2 // Plantilla de clase Pila.
3 #ifndef PILA_H
4 #define PILA_H
```

Figura 14.2 | Plantilla de clase **Pila**. (Parte I de 3).

```
5  template< typename T >
6  class Pila
7  {
8  public:
9    Pila( int = 10 ); // constructor predeterminado (Pila con tamaño de 10)
10   // destructor
11   ~Pila()
12   {
13     delete [] pilaPtr; // desasigna el espacio interno para Pila
14   } // fin del destructor ~Pila
15
16   bool push( const T & ); // mete un elemento en la Pila
17   bool pop( T & ); // saca un elemento de la Pila
18
19   // determina si la Pila está vacía
20   bool estaVacia() const
21   {
22     return cima == -1;
23   } // fin de la función estaVacia
24
25   // determina si la Pila está llena
26   bool estaLlena() const
27   {
28     return cima == tamanio - 1;
29   } // fin de la función estaLlena
30
31 private:
32   int tamanio; // # de elementos en la pila
33   int cima; // ubicación del elemento superior (-1 significa vacío)
34   T *pilaPtr; // apuntador a la representación interna de la Pila
35 }; // fin de la plantilla de clase Pila
36
37 // constructor
38 template< typename T >
39 Pila< T >::Pila( int s )
40   : tamanio( s > 0 ? s : 10 ), // valida el tamaño
41     cima( -1 ), // al principio la Pila está vacía
42     pilaPtr( new T[ tamanio ] ) // asigna memoria para los elementos
43 {
44   // cuerpo vacío
45 } // fin de la plantilla del constructor de Pila
46
47 // mete elemento en la Pila;
48 // si tiene éxito devuelve true; en caso contrario devuelve false
49 template< typename T >
50 bool Pila< T >::push( const T &valorMeter )
51 {
52   if ( !estaLlena() )
53   {
54     pilaPtr[ ++cima ] = valorMeter; // coloca el elemento en la Pila
55     return true; // se pudo meter
56   } // fin de if
57
58   return false; // no se pudo meter
59 } // fin de la plantilla de función push
60
61 // saca un elemento de la Pila;
62 // si tiene éxito devuelve true; en caso contrario devuelve false
63 template< typename T >
64 bool Pila< T >::pop( T &valorSacar )
```

Figura 14.2 | Plantilla de clase Pila. (Parte 2 de 3).

```

67  {
68      if ( !estaVacia() )
69      {
70          valorSacar = pilaPtr[ cima-- ]; // elimina el elemento de la Pila
71          return true; // se pudo sacar
72      } // fin de if
73
74      return false; // no se pudo sacar
75  } // fin de la plantilla de función pop
76
77 #endif

```

Figura 14.2 | Plantilla de clase *Pila*. (Parte 3 de 3).

*Creación de un controlador para probar la plantilla de clase *Pila*< T >*

Ahora vamos a considerar el controlador (figura 14.3) que ejecuta la plantilla de clase *Pila*. El controlador empieza por instanciar el objeto *pilaDouble* de tamaño 5 (línea 11). Este objeto se declara como objeto de la clase *Pila< double >* (se pronuncia como “Pila de *double*”). El compilador asocia el tipo *double* con el parámetro de tipo *T* en la plantilla de clase para producir el código fuente para una clase *Pila* de tipo *double*. Aunque las plantillas ofrecen beneficios de reutilización de software, recuerde que las especializaciones múltiples de plantillas de clase se instancian en un programa (en tiempo de compilación), aun y cuando la plantilla sólo se escribe una vez.

En las líneas 17 a 21 se invoca a *push* para colocar los valores *double* 1.1, 2.2, 3.3, 4.4 y 5.5 en *pilaDouble*. El ciclo *while* termina cuando el controlador trata de meter (*push*) un sexto valor en *pilaDouble* (que está llena, ya que contiene un máximo de cinco elementos). Observe que la función *push* devuelve *false* cuando no puede meter un valor en la pila.¹

En las líneas 27 y 28 se invoca *pop* en un ciclo *while* para eliminar los cinco valores de la pila (observe en la salida de la figura 14.3 que los valores se sacan en orden “último en entrar, primero en salir”). Cuando el controlador trata de sacar un sexto valor, la *pilaDouble* está vacía, por lo que el ciclo de *pop* termina.

```

1 // Fig. 14.3: fig14_03.cpp
2 // Programa de prueba de la plantilla de clase Pila.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Pila.h" // definición de la plantilla de clase Pila
8
9 int main()
10 {
11     Pila< double > pilaDouble( 5 ); // tamaño 5
12     double valorDouble = 1.1;
13
14     cout << "Metiendo elementos a pilaDouble\n";
15
16     // mete 5 valores double a pilaDouble
17     while ( pilaDouble.push( valorDouble ) )
18     {
19         cout << valorDouble << ' ';
20         valorDouble += 1.1;
21     } // fin de while

```

Figura 14.3 | Programa de prueba de la plantilla de clase *Pila*. (Parte 1 de 2).

- La clase *Pila* (figura 14.2) proporciona la función *estaVacia*, que el programador puede usar para determinar si la pila está vacía antes de intentar una operación *push*. Esto evitaría el error potencial de meter datos en una pila llena. En el capítulo 16, Manejo de excepciones, si la operación no se puede completar, la función *push* “lanzaría una excepción”. Puede escribir código para “atrapar” esa excepción, y luego decidir cómo manejarla en forma apropiada para la aplicación. La misma técnica se puede utilizar con la función *pop* cuando se hace un intento por sacar (*pop*) un elemento de una pila vacía.

```

22
23     cout << "\nLa pila esta llena. No se puede meter " << valorDouble
24     << "\n\nSacando elementos de pilaDouble\n";
25
26     // saca elementos de pilaDouble
27     while ( pilaDouble.pop( valorDouble ) )
28         cout << valorDouble << ' ';
29
30     cout << "\nLa pila esta vacia. No se puede sacar\n";
31
32     Pila< int > pilaInt; // tamaño predeterminado 10
33     int valorInt = 1;
34     cout << "\nMetiendo elementos a pilaInt\n";
35
36     // mete 10 enteros a pilaInt
37     while ( pilaInt.push( valorInt ) )
38     {
39         cout << valorInt++ << ' ';
40     } // fin de while
41
42     cout << "\nLa pila esta llena. No se puede meter " << valorInt
43     << "\n\nSacando elementos de pilaInt\n";
44
45     // saca elementos de pilaInt
46     while ( pilaInt.pop( valorInt ) )
47         cout << valorInt << ' ';
48
49     cout << "\nLa pila esta vacia. No se puede sacar" << endl;
50     return 0;
51 } // fin de main

```

```

Metiendo elementos a pilaDouble
1.1 2.2 3.3 4.4 5.5
La pila esta llena. No se puede meter 6.6

Sacando elementos de pilaDouble
5.5 4.4 3.3 2.2 1.1
La pila esta vacia. No se puede sacar

Metiendo elementos a pilaInt
1 2 3 4 5 6 7 8 9 10
La pila esta llena. No se puede meter 11

Sacando elementos de pilaInt
10 9 8 7 6 5 4 3 2 1
La pila esta vacia. No se puede sacar

```

Figura 14.3 | Programa de prueba de la plantilla de clase Pila. (Parte 2 de 2).

En la línea 32 se instancia la pila de enteros `pilaInt` con la declaración

```
Pila< int > pilaInt;
```

(se pronuncia “`pilaInt` es una Pila de `int`”). Como no se especifica un tamaño, el valor predeterminado es 10 como se especifica en el constructor predeterminado (figura 14.2, línea 10). En las líneas 37 a 40 se itera y se invoca a `push` para colocar valores en `pilaInt` hasta que esté llena, y después en las líneas 46 y 47 se itera y se invoca a `pop` para eliminar valores de `pilaInt` hasta que esté vacía. Una vez más, observe en los resultados que los valores se sacan en orden “último en entrar, primero en salir”.

Creación de plantillas de función para probar la plantilla de clase `Pila< T >`

Observe que el código en la función `main` de la figura 14.3 es casi idéntica para las manipulaciones de `pilaDouble` en las líneas 11 a 30 y las manipulaciones de `pilaInt` en las líneas 32 a 50. Esto presenta otra oportunidad para usar una plantilla de función. En la figura 14.4 se define la plantilla de función `pruebaPila` (líneas 14 a 38) para realizar las mismas tareas

```

1 // Fig. 14.4: fig14_04.cpp
2 // Programa de prueba de la plantilla de clase Pila. La función main usa una
3 // plantilla de función para manipular objetos de tipo Pila< T >.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <string>
9 using std::string;
10
11 #include "Pila.h" // definición de la plantilla de clase Pila
12
13 // plantilla de función para manipular Pila< T >
14 template< typename T >
15 void pruebaPila(
16     Pila< T > &laPila, // referencia a Pila< T >
17     T valor, // valor inicial a meter
18     T incremento, // incremento para los valores subsiguientes
19     const string nombrePila ) // nombre del objeto Pila< T >
20 {
21     cout << "\nMetiendo elementos a " << nombrePila << '\n';
22
23     // mete el elemento en la Pila
24     while ( laPila.push( valor ) )
25     {
26         cout << valor << ' ';
27         valor += incremento;
28     } // fin de while
29
30     cout << "\nLa pila esta llena. No se puede meter " << valor
31     << "\n\nSacando elementos de " << nombrePila << '\n';
32
33     // saca elementos de la Pila
34     while ( laPila.pop( valor ) )
35         cout << valor << ' ';
36
37     cout << "\nLa pila esta vacia. No se puede sacar" << endl;
38 } // fin de la plantilla de función pruebaPila
39
40 int main()
41 {
42     Pila< double > pilaDouble( 5 ); // tamaño 5
43     Pila< int > pilaInt; // tamaño predeterminado: 10
44
45     pruebaPila( pilaDouble, 1.1, 1.1, "pilaDouble" );
46     pruebaPila( pilaInt, 1, 1, "pilaInt" );
47
48     return 0;
49 } // fin de main

```

```

Metiendo elementos a pilaDouble
1.1 2.2 3.3 4.4 5.5
La pila esta llena. No se puede meter 6.6
Sacando elementos de pilaDouble
5.5 4.4 3.3 2.2 1.1
La pila esta vacia. No se puede sacar
Metiendo elementos a pilaInt
1 2 3 4 5 6 7 8 9 10
La pila esta llena. No se puede meter 11
Sacando elementos de pilaInt
10 9 8 7 6 5 4 3 2 1
La pila esta vacia. No se puede sacar

```

Figura 14.4 | Cómo pasar un objeto de la plantilla Pila a una plantilla de función.

que `main` en la figura 14.3: meter (`push`) una serie de valores en una `Pila< T >` y sacar (`pop`) los valores de una `Pila< T >`. La plantilla de función `pruebaPila` utiliza el parámetro de plantilla `T` (que se especifica en la línea 14) para representar el tipo de datos almacenado en la `Pila< T >`. La plantilla de función recibe cuatro argumentos (líneas 16 a 19): una referencia a un objeto de tipo `Pila< T >`, un valor de tipo `T` que será el primer valor que se meta en la `Pila< T >`, un valor de tipo `T` que se utiliza para incrementar los valores que se meten en la `Pila< T >` y un objeto `string` que representa el nombre del objeto `Pila< T >` para fines de imprimir los resultados. La función `main` (líneas 40 a 49) instancia un objeto de tipo `Pila< double >` que se llama `pilaDouble` (línea 42) y un objeto de tipo `Pila< int >` que se llama `pilaInt` (línea 43), y utiliza estos objetos en las líneas 45 y 46. El compilador infiere el tipo de `T` para `pruebaPila` del tipo utilizado para instanciar el primer argumento de la función (es decir, el tipo utilizado para instanciar `pilaDouble` o `pilaInt`). Los resultados de la figura 14.4 coinciden precisamente con los resultados de la figura 14.3.

14.5 Parámetros sin tipo y tipos predeterminados para las plantillas de clases

La plantilla de clase `Pila` de la sección 14.4 sólo utiliza un parámetro de tipo en el encabezado de la plantilla (figura 14.2, línea 6). También es posible usar **parámetros de plantilla sin tipo**, o **parámetros sin tipo**, que pueden tener argumentos predeterminados y se tratan como valores `const`. Por ejemplo, el encabezado de la plantilla podría modificarse para recibir un parámetro `int` llamado `elementos` de la siguiente manera:

```
template< typename T, int elementos > // parámetro sin tipo llamado elementos
```

Después, podría usarse una declaración tal como

```
Pila< double, 100 > cifrasDeVentasMasRecientes;
```

podría utilizarse para instanciar (en tiempo de compilación) una especialización de plantilla de la clase `Pila` de 100 elementos de valores `double`, llamada `cifrasDeVentasMasRecientes`; esta especialización de la plantilla de clase sería de tipo `Pila< double, 100 >`. Así, el encabezado de la clase podría contener un miembro de datos `private` con una declaración de un arreglo tal como

```
T contenedorPila[ elementos ]; // arreglo para alojar el contenido de Pila
```

Además, un parámetro de tipos puede especificar un tipo predeterminado. Por ejemplo,

```
template< typename T = string > // el tipo predeterminado es string
```

podría especificar que una `Pila` contiene objetos `string` de manera predeterminada. Después, una declaración tal como

```
Pila<> descripcionesTrabajos;
```

podría utilizarse para instanciar una especialización de la plantilla de clase `Pila` de objetos `string`, llamada `descripcionesTrabajos`; esta especialización de la plantilla de clase sería de tipo `Pila< string >`. Los parámetros de tipo predeterminado deben ser los parámetros de más a la derecha en la lista de parámetros de tipo de la plantilla. Cuando se instancia una clase con dos o más tipos predeterminados, si un tipo omitido no es el parámetro de tipo de más a la derecha en la lista de parámetros de tipo, entonces también deben omitirse todos los parámetros de tipo a la derecha de ese tipo.

Tip de rendimiento 14.2

 Cuando sea apropiado, especifique el tamaño de una clase contenedora (como una clase tipo arreglo o tipo pila) en tiempo de compilación (posiblemente a través de un parámetro de plantilla sin tipo). Esto elimina la sobrecarga en tiempo de ejecución por utilizar `new` para crear el espacio en forma dinámica.

Observación de Ingeniería de Software 14.3

 Al especificar el tamaño de un contenedor en tiempo de compilación se evita el error en tiempo de ejecución potencialmente fatal, si `new` no puede obtener la memoria necesaria.

En los ejercicios, se le pedirá que utilice un parámetro sin tipos para crear una plantilla para nuestra clase `Array` desarrollada en el capítulo 11. Esta plantilla permitirá instanciar objetos `Array` con un número especificado de elementos de un tipo especificado en tiempo de compilación, en vez de crear espacio para los objetos `Array` en tiempo de ejecución.

En algunos casos, tal vez no sea posible utilizar un tipo específico con una plantilla de clase. Por ejemplo, la plantilla `Pila` de la figura 14.2 requiere que los tipos definidos por el usuario que se van a almacenar en una `Pila` proporcionen un constructor predeterminado y un operador de asignación. Si un tipo específico definido por el usuario no va a funcionar con nuestra plantilla `Pila` o requiere un procesamiento personalizado, puede definir una **especialización explícita** de la plantilla de clase para un tipo específico. Vamos a suponer que deseamos crear una especialización explícita llamada `Pila` para objetos `Empleado`. Para ello, hay que formar una nueva clase con el nombre `Pila< Empleado >` de la siguiente manera:

```
template<>
class Pila< Empleado >
{
    // cuerpo de la definición de la clase
};
```

Observe que la especialización explícita `Pila< Empleado >` es un reemplazo completo para la plantilla de clase `Pila` que es específica para el tipo `Empleado`; no utiliza nada de la plantilla de clase original, e incluso puede tener miembros diferentes.

14.6 Notas acerca de las plantillas y la herencia

Las plantillas y la herencia se relacionan de varias formas:

- Una plantilla de clase se puede derivar de una especialización de plantilla de clase.
- Una plantilla de clase se puede derivar de una clase que no sea de plantilla.
- Una especialización de plantilla de clase se puede derivar de una especialización de plantilla de clase.
- Una clase que no sea de plantilla se puede derivar de una especialización de plantilla de clase.

14.7 Notas acerca de las plantillas y funciones friend

Hemos visto que las funciones y clases completas se pueden declarar como `friend` de las clases que no son de plantillas. Con las plantillas de clases, se puede establecer la amistad entre una plantilla de clase y una función global, una función miembro de otra clase (posiblemente una especialización de plantilla de clase), o incluso con una clase completa (posiblemente una especialización de plantilla de clase).

A lo largo de esta sección, vamos a suponer que hemos definido una plantilla de clase para una clase llamada `X` con un solo parámetro de tipo `T`, como en:

```
template< typename T > class X
```

Bajo esta suposición, es posible hacer que una función `f1` sea amiga (`friend`) de cualquier especialización de plantilla de clase que se instancie de la plantilla de clase para la clase `X`. Para ello, podemos usar una declaración de amistad de la forma

```
friend void f1();
```

Por ejemplo, la función `f1` es amiga de `X< double >`, `X< string >` y `X< Empleado >`, etc.

También es posible hacer que una función `f2` sea amiga sólo de una especialización de plantilla de clase con el mismo argumento de tipo. Para ello, se utiliza una declaración de la forma

```
friend void f2( X< T > & );
```

Por ejemplo, si `T` es `float`, la función `f2(X< float > &)` es una amiga de la especialización de plantilla de clase `X< float >`, pero no es amiga de la especialización de plantilla de clase `X< string >`.

Podemos declarar que una función miembro de otra clase es amiga de cualquier especialización de plantilla de clase generada a partir de la plantilla de clase. Para ello, la declaración `friend` debe calificar el nombre de la función miembro de otra clase mediante el uso del nombre de la clase y el operador de resolución de ámbito binario, como en:

```
friend void A::f3();
```

La declaración hace a la función miembro `f3` de la clase `A` una amiga de cualquier especialización de plantilla de clase que se instancie de la plantilla de clase anterior. Por ejemplo, la función `f3` de la clase `A` es amiga de `X< double >`, `X< string >` y `X< Empleado >`, etcétera.

Al igual que con una función global, la función miembro de otra clase puede ser amiga de sólo una especialización de plantilla de clase con el mismo argumento de tipo. Una declaración de amistad de la forma

```
friend void C< T >::f4( X< T > & );
```

para un tipo particular `T` como `float`, hace que la función miembro

```
C< float >::f4( X< float > & )
```

sea una función amiga de *sólo* la especialización de plantilla de clase `X< float >`.

En algunos casos es conveniente hacer que todo un conjunto de funciones miembro de una clase sean amigas de una plantilla de clase. En este caso, una declaración `friend` de la forma

```
friend class Y;
```

hace que cada función miembro de la clase `Y` sea amiga de cada especialización de plantilla de clase producida a partir de la plantilla de clase `X`.

Finalmente, es posible hacer que todas las funciones miembro de una especialización de plantilla de clase sean amigas de otra especialización de plantilla de clase con el mismo argumento de tipo. Por ejemplo, una declaración `friend` de la forma:

```
friend class Z< T >;
```

indica que cuando se instancia una especialización de plantilla de clase con un tipo específico para `T` (como `float`), todos los miembros de `class Z< float >` se convierten en amigos de la especialización de plantilla de clase `X< float >`. En varios ejemplos del capítulo 20, Estructuras de datos, utilizaremos esta relación específica.

14.8 Notas acerca de las plantillas y miembros static

¿Qué hay sobre los datos miembro `static`? Recuerde que, con una clase que no es de plantilla, se comparte una copia de cada miembro de datos `static` entre todos los objetos de la clase, y el miembro de datos `static` se debe inicializar en alcance de archivo.

Cada especialización de plantilla de clase que se instancie de una plantilla de clase tiene su propia copia de cada miembro de datos `static` de la plantilla de clase; todos los objetos de esa especialización comparten ese único miembro de datos `static`. Además, al igual que con los datos miembro `static` de las clases que no son de plantilla, deben definirse datos miembro `static` de las especializaciones de plantilla de clase y, si es necesario, deben inicializarse en alcance de archivo. Cada especialización de plantilla de clase obtiene su propia copia de las funciones miembro `static` de la plantilla de clase.

14.9 Repaso

En este capítulo presentamos una de las características más poderosas de C++: las plantillas. El lector aprendió a utilizar plantillas de funciones para permitir al compilador producir un conjunto de especializaciones de plantillas de funciones que representan a un grupo de funciones sobrecargadas relacionadas. También vimos cómo sobrecargar una plantilla de función para crear una versión especializada de una función que maneje el procesamiento de un tipo de datos específico, de una manera que difiera de las otras especializaciones de la plantilla de función. Después, el lector aprendió acerca de las plantillas de clases y las especializaciones de plantillas de clases. Vimos ejemplos de cómo usar una plantilla de clase para crear un grupo de tipos relacionados, en donde cada uno de ellos realiza un procesamiento idéntico sobre tipos de datos diferentes. Por último, aprendió acerca de algunas de las relaciones entre las plantillas, funciones `friend`, herencia y miembros `static`.

En el siguiente capítulo hablaremos sobre muchas de las herramientas de E/S de C++, y demostraremos varios manipuladores de flujos que realizan diversas tareas de formato.

Resumen

Sección 14.1 Introducción

- Las plantillas nos permiten especificar un rango de funciones (sobrecargadas) (conocidas como especializaciones de plantillas de funciones) o un rango de clases relacionadas (conocidas como especializaciones de plantillas de clases).

Sección 14.2 Plantillas de funciones

- Para utilizar especializaciones de plantillas de funciones, debemos escribir una sola definición de plantilla de función. Con base en los tipos de argumentos proporcionados en las llamadas a esta función, C++ genera especializaciones separadas para manejar cada tipo de llamada en forma apropiada. Éstas se compilan junto con el resto del código fuente de un programa.

- Todas las definiciones de plantillas de funciones empiezan con la palabra clave `template`, seguida de parámetros de plantilla para la plantilla de función, encerrados entre los signos `<` y `>`; a cada parámetro de plantilla que representa a un tipo se le debe anteponer la palabra `class` o `typename`. Las palabras claves `typename` y `class` que se utilizan para especificar los parámetros de plantillas de funciones indican “cualquier tipo integrado o tipo definido por el usuario”.
- Los parámetros de plantilla de las definiciones de plantillas se utilizan para especificar los tipos de argumentos para la función, el tipo de valor de retorno de la función y para declarar variables en la función.
- El nombre de un parámetro de plantilla se puede declarar sólo una vez en la lista de parámetros de tipo de un encabezado de plantilla. Los nombres de los parámetros de tipos formales entre las plantillas de función no necesitan ser únicos.

Sección 14.3 Sobre carga de plantillas de funciones

- Una plantilla de función se puede sobre cargar de varias formas. Podemos proporcionar otras plantillas de función que especifiquen el mismo nombre de función, pero distintos parámetros. Una plantilla de función puede sobre cargarse también si se proporciona a otras funciones que no sean de plantilla el mismo nombre de función, pero distintos parámetros.

Sección 14.4 Plantillas de clases

- Las plantillas de clases proporcionan los medios para describir una clase de forma genérica, y para instanciar clases que son versiones de tipos específicos de esta clase genérica.
- Las plantillas de clases se llaman tipos parametrizados; requieren parámetros de tipo para especificar cómo personalizar una plantilla de clase genérica para formar una especialización de plantilla de clase específica.
- Para usar especializaciones de una plantilla de clase, hay que escribir una plantilla de clase. Cuando se necesita una nueva clase de un tipo específico, se utiliza una notación concisa y el compilador escribe el código fuente para la especialización de la plantilla de clase.
- Una definición de plantilla de clase tiene una apariencia similar a la definición de una clase convencional, excepto que se le anteponen las palabras `template< typename T >` (o `template< class T >`) para indicar que es una definición de plantilla de clase con el parámetro de tipo `T`, que actúa como receptáculo para el tipo de la clase que se va a crear. El tipo `T` se menciona a lo largo del encabezado de la clase y de las definiciones de las funciones miembro, como un nombre de tipo genérico.
- Las definiciones de las funciones miembro que están fuera de una plantilla de clase empiezan con `template< typename T >` (o `template< class T >`). Después, cada definición de función se asemeja a la definición de una función convencional, excepto que los datos genéricos en la clase siempre se listan de manera genérica como el parámetro de tipo `T`. El operador de resolución de ámbito binario se utiliza con el nombre de la plantilla de clase para enlazar la definición de cada función miembro con el alcance de la plantilla de clase.

Sección 14.5 Parámetros sin tipo y tipos predeterminados para las plantillas de clases

- Es posible utilizar parámetros sin tipo en el encabezado de una plantilla de clase o de función.
- Se puede proporcionar una especialización explícita de una plantilla de clase para sobre escribir una plantilla de clase para un tipo específico.

Sección 14.6 Notas acerca de las plantillas y la herencia

- Una plantilla de clase se puede derivar de una especialización de plantilla de clase. Una plantilla de clase se puede derivar de una clase que no sea de plantilla. Una especialización de plantilla de clase se puede derivar de una especialización de plantilla de clase. Una clase que no sea de plantilla se puede derivar de una especialización de plantilla de clase.

Sección 14.7 Notas acerca de las plantillas y las funciones friend

- Las funciones y las clases completas pueden ser declaradas como amigas de las clases que no son de plantilla. Con las plantillas de clase, se pueden declarar arreglos de amistad. La amistad se puede establecer entre una plantilla de clase y una función global, una función miembro de otra clase (posiblemente una especialización de plantilla de clase), o incluso con una clase completa (probablemente una especialización de plantilla de clase).

Sección 14.8 Notas acerca de las plantillas y los miembros static

- Cada especialización de plantilla de clase que se instancia de una plantilla de clase tiene su propia copia de cada miembro de datos `static` de la plantilla de clase; todos los objetos de esa especialización comparten ese dato miembro `static`. Y al igual que con los datos miembro `static` de las clases que no son de plantilla, los datos miembro `static` de las especializaciones de plantilla de clase se deben definir y, si es necesario, inicializar en alcance de archivo.
- Cada especialización de una plantilla de clase obtiene una copia de las funciones miembro `static` de la plantilla de clase.

Terminología

`class`, palabra clave en un parámetro de tipo de plantilla
definición de plantilla de clase

definición de plantilla de función
especialización de plantilla de clase

especialización de plantilla de función	parámetro de tipo
especialización explícita	parámetro sin tipo
friend de una plantilla	plantilla de clase
función miembro de una especialización de plantilla de clase	plantilla de función
función miembro static de una especialización de plantilla de clase	programación genérica
función miembro static de una plantilla de clase	signos < y >
macro	sobrecarga de una plantilla de función
miembro de datos static de una especialización de plantilla de clase	template, palabra clave
miembro de datos static de una plantilla de clase	template< class T >
parámetro de plantilla	template< typename T >
parámetro de plantilla de tipo	tipo parametrizado
parámetro de plantilla sin tipo	typename
	typename, palabra clave

Ejercicios de autoevaluación

- 14.1** Indique si cada uno de los siguientes enunciados es *verdadero* o *falso*. En caso de ser *falso*, explique por qué.
- Los parámetros de plantilla de la definición de una plantilla de función se utilizan para especificar los tipos de argumentos para la función, para especificar el tipo de valor de retorno de la función y para declarar variables dentro de la función.
 - Las palabras clave `typename` y `class` que se utilizan con un parámetro de tipo de plantilla indican específicamente “cualquier tipo de clase definida por el usuario”.
 - Una plantilla de función se puede sobrecargar mediante otra plantilla de función con el mismo nombre de función.
 - Los nombres de los parámetros de plantilla entre las definiciones de plantilla deben ser únicos.
 - Cada definición de función miembro fuera de una plantilla de clase debe empezar con un encabezado de plantilla.
 - Una función `friend` de una plantilla de clase debe ser una especialización de plantilla de función.
 - Si se generan varias especializaciones de una plantilla de clase a partir de una sola plantilla de clase con un solo miembro de datos `static`, cada una de las especializaciones de la plantilla de clase comparte una sola copia del miembro de datos `static` de esa plantilla de clase.
- 14.2** Complete los siguientes enunciados:
- Las plantillas nos permiten especificar, con un solo segmento de código, un rango completo de funciones relacionadas, conocidas como _____, o un rango completo de clases relacionadas, conocidas como _____.
 - Todas las definiciones de plantillas de funciones empiezan con la palabra clave _____, seguida de una lista de parámetros de plantilla para la función, encerrados entre _____.
 - Las funciones relacionadas que se generan a partir de una plantilla de función tienen todos el mismo nombre, por lo que el compilador utiliza la resolución _____ para invocar a la función apropiada.
 - A las plantillas de clases también se les llama tipos _____.
 - El operador _____ se utiliza con el nombre de una plantilla de clase para enlazar cada definición de función miembro con el alcance de la plantilla de clase.
 - Al igual que con los datos miembro `static` de las clases que no son de plantilla, los datos miembro `static` de las especializaciones de la plantilla de clase también se deben definir y, de ser necesario, inicializar en alcance de _____.

Respuestas a los ejercicios de autoevaluación

- 14.1** a) Verdadero. b) Falso. Las palabras clave `typename` y `class` en este contexto también permiten un parámetro de tipo que sea de un tipo fundamental. c) Verdadero. d) Falso. Los nombres de los parámetros de plantilla entre las plantillas de funciones no necesitan ser únicos. e) Verdadero. f) Falso. Podría ser una función que no sea de plantilla. g) Falso. Cada especialización de la plantilla de clase tendrá su propia copia del miembro de datos `static`.

- 14.2** a) especializaciones de plantilla de función, especializaciones de plantilla de clase. b) `template`, los signos < y >. c) sobrecarga. d) parametrizados. e) resolución de ámbito binario. f) archivo.

Ejercicios

- 14.3** Escriba una plantilla de función llamada `ordenSeleccion`, con base en el programa de la figura 8.15. Escriba un programa controlador que reciba como entrada, ordene e imprima los valores de un arreglo `int` y de un arreglo `float`.
- 14.4** Sobrecrega la plantilla de función `imprimirArreglo` de la figura 14.1, de manera que reciba dos argumentos enteros adicionales, a saber, `int subindiceInferior` e `int subindiceSuperior`. Una llamada a esta función imprimirá sólo la parte designada del arreglo. Valide `subindiceInferior` y `subindiceSuperior`; si cualquiera de éstos está fuera de rango, o si `subindiceSuperior` es menor o igual a `subindiceInferior`, la función `imprimirArreglo` sobrecregada deberá devolver 0; en caso contrario, `imprimirArreglo` deberá devolver el número de elementos impresos. Después modifique `main` para ejecutar ambas versiones de `imprimirArreglo` en los arreglos `a`, `b` y `c` (líneas 23 a 25 de la figura 14.1). Asegúrese de probar todas las capacidades de ambas versiones de `imprimirArreglo`.
- 14.5** Sobrecrega la plantilla de función `imprimirArreglo` de la figura 14.1 con una versión que no sea de plantilla y que imprima de manera específica un arreglo de caracteres en formato tabular y ordenado por columnas.
- 14.6** Escriba una plantilla de función simple para la función `esIgualA` que compara sus dos argumentos del mismo tipo con el operador de igualdad (`==`), y devuelve `true` si son iguales, y `false` si no son iguales. Use esta plantilla de función en un programa que llame a `esIgualA` sólo con una variedad de tipos integrados. Ahora escriba una versión separada del programa que llame a `esIgualA` con un tipo de clase definida por el usuario, pero que no sobrecrega el operador de igualdad. ¿Qué ocurre si tratamos de ejecutar este programa? Ahora sobrecrega el operador de igualdad (con la función operador) `operator==`. ¿Qué ocurre ahora si tratamos de ejecutar este programa?
- 14.7** Use un parámetro sin tipo de plantilla `int` llamado `numeroDeElementos`, y un parámetro de tipo `tipoElemento` para ayudar a crear una plantilla para la clase `Array` (figuras 11.6 y 11.7) que desarrollamos en el capítulo 11. Esta plantilla permitirá instanciar objetos `Array` con un número especificado de elementos de un tipo especificado en tiempo de compilación.
- 14.8** Escriba un programa con la plantilla de clase `Array`. La plantilla puede instanciar un objeto `Array` de cualquier tipo de elementos. Sobrecreba la plantilla con una definición específica para un objeto `Array` de elementos `float` (`class Array < float >`). El controlador debe demostrar la creación de una instancia de un objeto `Array` de `int` a través de la plantilla, y debe mostrar que un intento por instanciar un objeto `Array` de `float` utiliza la definición que se proporciona en `class Array < float >`.
- 14.9** Explique la diferencia entre los términos “plantilla de función” y “especialización de plantilla de función”.
- 14.10** ¿Qué es más parecido a una plantilla común, utilizada para dibujar en papel: una plantilla de clase o una especialización de plantilla de clase? Explique su respuesta.
- 14.11** ¿Cuál es la relación entre las plantillas de función y la sobrecregación?
- 14.12** ¿Por qué podríamos elegir usar una plantilla de función en vez de una macro?
- 14.13** ¿Qué problema de rendimiento se puede producir al usar plantillas de función y plantillas de clase?
- 14.14** El compilador realiza un proceso de concordancia para determinar cuál especialización de plantilla de función debe llamar al invocar una función. ¿Bajo qué circunstancias podría un intento por realizar una concordancia producir un error de compilación?
- 14.15** ¿Por qué es apropiado hacer referencia a una plantilla de clase como un tipo parametrizado?
- 14.16** Explique por qué un programa en C++ utilizaría la instrucción
- ```
Array< Empleado > listaTrabajadores(100);
```
- 14.17** Revise su respuesta al ejercicio 14.16. ¿Por qué podría un programa en C++ usar la instrucción
- ```
Array< Empleado > listaTrabajadores;
```
- 14.18** Explique el uso de la siguiente notación en un programa en C++:
- ```
template< typename T > Array< T >::Array(int s)
```
- 14.19** ¿Por qué se podría usar un parámetro sin tipo con una plantilla de clase para un contenedor tal como un arreglo o una pila?
- 14.20** Suponga que una plantilla de clase tiene el encabezado
- ```
template< typename T > class Ct1
```

Describa las relaciones de amistad establecidas al colocar cada una de las siguientes declaraciones `friend` dentro de esta plantilla de clase. Los identificadores que empiezan con “f” son funciones, los identificadores que empiezan con “C” son clases, los identificadores que empiezan con “Ct” son plantillas de clases y T es un parámetro de tipo de plantilla (es decir, T puede representar cualquier tipo fundamental o de clase).

- a) `friend void f1();`
- b) `friend void f2(Ct1< T > &);`
- c) `friend void C2::f3();`
- d) `friend void Ct3< T >::f4(Ct1< T > &);`
- e) `friend class C4;`
- f) `friend class Ct5< T >;`

14.21 Suponga que la plantilla de clase `Empleado` tiene un miembro de datos `static` llamado `cuenta`. Suponga que se instancian tres especializaciones de plantilla de clase a partir de esa plantilla de clase. ¿Cuántas copias del miembro de datos `static` existirán? ¿Cómo se restringirá el uso de cada una (si acaso se restringe)?



La conciencia... no aparece por sí misma cortada en pequeños pedazos... Un "río" o un "flujo" son las metáforas por las cuales se describe con más naturalidad.

—William James

Todas las noticias que se pueden imprimir.

—Adolph S. Ochs

No elimine el sitio histórico sobre el límite de los campos.

—Amenehope

Entrada y salida de flujos

OBJETIVOS

En este capítulo aprenderá a:

- Usar la entrada/salida de flujos orientados a objetos de C++.
- Dar formato a la entrada y la salida.
- Conocer la jerarquía de la clase de E/S de flujos.
- Usar manipuladores de flujos.
- Controlar la justificación y el relleno de caracteres.
- Determinar el éxito o la falla de las operaciones de entrada/salida.
- Enlazar los flujos de salida a los flujos de entrada.

- 15.1** Introducción
- 15.2** Flujos
 - 15.2.1** Comparación entre flujos clásicos y flujos estándar
 - 15.2.2** Archivos de encabezado de la biblioteca `iostream`
 - 15.2.3** Clases y objetos de entrada/salida de flujos
- 15.3** Salida de flujos
 - 15.3.1** Salida de variables `char *`
 - 15.3.2** Salida de caracteres mediante la función miembro `put`
- 15.4** Entrada de flujos
 - 15.4.1** Funciones miembro `get` y `getline`
 - 15.4.2** Funciones miembro `peek`, `putback` e `ignore` de `istream`
 - 15.4.3** E/S con seguridad de tipos
- 15.5** E/S sin formato mediante el uso de `read`, `write` y `gcount`
- 15.6** Introducción a los manipuladores de flujos
 - 15.6.1** Base de flujos integrales: `dec`, `oct`, `hex` y `setbase`
 - 15.6.2** Precisión de punto flotante (`precision`, `setprecision`)
 - 15.6.3** Anchura de campos (`width`, `setw`)
 - 15.6.4** Manipuladores de flujos de salida definidos por el usuario
- 15.7** Estados de formato de flujos y manipuladores de flujos
 - 15.7.1** Ceros a la derecha y puntos decimales (`showpoint`)
 - 15.7.2** Justificación (`left`, `right` e `internal`)
 - 15.7.3** Relleno de caracteres (`fill`, `setfill`)
 - 15.7.4** Base de flujos integrales (`dec`, `oct`, `hex`, `showbase`)
 - 15.7.5** Números de punto flotante; notación científica y fija (`scientific`, `fixed`)
 - 15.7.6** Control de mayúsculas/minúsculas (`uppercase`)
 - 15.7.7** Especificación de formato booleano (`boolalpha`)
 - 15.7.8** Establecer y restablecer el estado de formato mediante la función miembro `flags`
- 15.8** Estados de error de los flujos
- 15.9** Enlazar un flujo de salida a un flujo de entrada
- 15.10** Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)

15.1 Introducción

Las bibliotecas estándar de C++ proporcionan un extenso conjunto de herramientas de entrada/salida. En este capítulo hablaremos sobre un rango de herramientas suficientes para realizar la mayoría de las operaciones de E/S comunes, y presentaremos las generalidades de las herramientas restantes. Anteriormente en el libro hablamos sobre algunas de estas características; ahora proporcionaremos un tratamiento más completo. Muchas de las características de E/S que veremos están orientadas a objetos. Este estilo de E/S hace uso de otras características de C++, como las referencias, la sobrecarga de funciones y la sobrecarga de operadores.

C++ utiliza la **E/S con seguridad de tipos**. Cada operación de E/S se ejecuta de una manera sensible al tipo de datos. Si se ha definido una función miembro de E/S para manejar un tipo de datos específico, entonces se hace una llamada a esa función miembro para manejar ese tipo de datos. Si no hay coincidencia entre el tipo de los datos actuales

y una función para manejar ese tipo de datos, el compilador genera un error. Por ende, los datos inapropiados no pueden “infiltrarse” por el sistema (como puede ocurrir en C, con lo cual se permiten ciertos errores sutiles y raros).

Los usuarios pueden especificar cómo realizar operaciones de E/S para objetos de tipos definidos por el usuario, para lo cual sobrecargan el operador de inserción de flujo (`<<`) y el operador de extracción de flujo (`>>`). Esta extensibilidad es una de las características más valiosas de C++.



Observación de Ingeniería de Software 15.1

Use el estilo de E/S de C++ exclusivamente en programas de C++, aun y cuando el estilo E/S de C++ esté disponible para los programadores de C++.



Tip para prevenir errores 15.1

La E/S en C++ es segura para los tipos.



Observación de Ingeniería de Software 15.2

C++ permite un tratamiento común de la E/S para los tipos predefinidos y los tipos definidos por el usuario. Estas características comunes facilitan el desarrollo y la reutilización del software.

15.2 Flujos

La E/S en C++ ocurre en forma de **flujos**, que son secuencias de bytes. En las operaciones de entrada, los bytes fluyen de un dispositivo (teclado, unidad de disco, conexión de red, etc.) a la memoria principal. En las operaciones de salida, los bytes fluyen de la memoria principal a un dispositivo (pantalla, impresora, unidad de disco, conexión de red, etcétera.)

Una aplicación asocia un significado a los bytes. Éstos podrían representar caracteres, datos crudos, imágenes de gráficos, voz digital, video digital o cualquier otra información que pueda requerir una aplicación.

Los mecanismos de E/S del sistema deben transferir bytes de los dispositivos a la memoria (y viceversa) en forma consistente y confiable. A menudo, dichas transferencias implican cierto movimiento mecánico, como la rotación de un disco o de una cinta, o la pulsación de teclas en un teclado. El tiempo que toman estas transferencias es por lo general mucho mayor que el tiempo que requiere el procesador para manipular los datos en forma interna. Por ende, las operaciones de E/S requieren un proceso cuidadoso de planeación y optimización para asegurar un rendimiento óptimo.

C++ proporciona herramientas de E/S de “bajo nivel” y de “alto nivel”. Las herramientas de E/S de bajo nivel (**E/S sin formato**) especifican que se debe transferir cierto número de bytes de un dispositivo a la memoria, o de la memoria a un dispositivo. En dichas transferencias, el byte individual es el tema de interés. Dichas herramientas de bajo nivel proporcionan transferencias de alta velocidad y alto volumen, pero no son especialmente convenientes para los programadores.

Por lo general, los programadores prefieren una visión de la E/S a un nivel más alto (**E/S con formato**), en donde los bytes se agrupan en unidades significativas tales como enteros, números de punto flotante, caracteres, cadenas y tipos definidos por el usuario. Estas herramientas orientadas a los tipos son satisfactorias para la mayoría de las operaciones de E/S, exceptuando el procesamiento de archivos de alto volumen.



Tip de rendimiento 15.1

Use la E/S sin formato para el mejor rendimiento en el procesamiento de archivos de gran volumen.



Tip de portabilidad 15.1

El uso de la E/S sin formato puede producir problemas de portabilidad, ya que los datos sin formato no son portables entre todas las plataformas.

15.2.1 Comparación entre flujos clásicos y flujos estándar

En el pasado, las bibliotecas de flujos clásicos de C++ permitían la entrada y salida de objetos `char`. Como por lo general un `char` ocupa un byte, sólo puede representar un conjunto limitado de caracteres (como los del conjunto de caracteres ASCII). Sin embargo, muchos lenguajes utilizan alfabetos que contienen más caracteres de los que puede representar un solo byte `char`. El conjunto de caracteres ASCII no proporciona estos caracteres; el **conjunto de caracteres Unicode®** sí. Unicode es un conjunto de caracteres internacional extenso, que representa la mayor parte de los lenguajes, “comercialmente viables” del mundo, símbolos matemáticos y mucho más. Para obtener más información sobre Unicode, visite www.unicode.org.

C++ incluye las **bibliotecas de flujos estándar**, que permiten a los desarrolladores crear sistemas capaces de realizar operaciones de E/S con caracteres Unicode. Para este propósito, C++ incluye un tipo de carácter adicional llamado `wchar_t`, el cual puede almacenar caracteres Unicode de 2 bytes. El estándar de C++ también rediseñó las clases de flujos clásicos de C++, que sólo proporcionan objetos `char`, como plantillas de clase con especializaciones separadas para procesar caracteres de los tipos `char` y `wchar_t`, respectivamente. En este libro utilizamos el tipo `char` de las plantillas de clases.

15.2.2 Archivos de encabezado de la biblioteca `iostream`

La biblioteca `iostream` de C++ proporciona cientos de herramientas de E/S. Varios archivos de encabezado contienen porciones de la interfaz de la biblioteca.

La mayoría de los programas de C++ incluyen el archivo de encabezado `<iostream>`, el cual declara los servicios básicos requeridos para todas las operaciones de E/S de flujos. El archivo de encabezado `<iostream>` define los objetos `cin`, `cout`, `cerr` y `clog`, que corresponden al flujo de entrada estándar, el flujo de salida estándar, el flujo de error estándar sin búfer y el flujo de error estándar con búfer, respectivamente. (En la sección 15.2.3 hablaremos sobre `cerr` y `clog`). Se proporcionan servicios de E/S sin formato y con formato.

El encabezado `<iomanip>` declara servicios útiles para realizar E/S con formato, con los denominados **manipuladores de flujos parametrizados**, tales como `setw` y `setprecision`.

El encabezado `<fstream>` declara servicios para el procesamiento de archivos controlado por el usuario. Utilizamos este encabezado en los programas de procesamiento de archivos del capítulo 17, Procesamiento de archivos.

Por lo general, las implementaciones de C++ contienen otras bibliotecas relacionadas con las operaciones de E/S que proporcionan herramientas específicas del sistema, como el control de los dispositivos de propósito especial para la E/S de audio y video.

15.2.3 Clases y objetos de entrada/salida de flujos

La biblioteca `iostream` proporciona muchas plantillas para el manejo de las operaciones de E/S comunes. Por ejemplo, la plantilla de clase `basic_istream` soporta las operaciones de entrada de flujos, la plantilla de clase `basic_ostream` soporta las operaciones de salida de flujos, y la plantilla de clase `basic_iostream` soporta tanto operaciones de entrada de flujos como de salida de flujos. Cada plantilla tiene una especialización de plantilla predefinida que permite la E/S con objetos `char`. Además, la biblioteca `iostream` proporciona un conjunto de especificadores `typedef` que proporcionan alias para estas especializaciones de plantilla. El especificador `typedef` declara sinónimos (alias) para los tipos de datos previamente definidos. Algunas veces los programadores utilizan `typedef` para crear nombres de tipos más cortos o más legibles. Por ejemplo, la instrucción

```
typedef Carta *CartaPtr;
```

define un nombre de tipo adicional, `CartaPtr`, como un sinónimo para el tipo `Carta *`. Observe que al crear un nombre mediante el uso de `typedef` no se crea un tipo de datos; `typedef` sólo crea un nombre de tipo que puede usarse en el programa. En la sección 21.5 hablaremos sobre `typedef` con detalle. La definición `typedef istream` representa una especialización de `basic_stream` que permite la entrada de objetos `char`. De manera similar, la definición `typedef ostream` representa una especialización de `basic_ostream` que permite la salida de objetos `char`. Además, la definición `typedef iostream` representa una especialización de `basic_iostream` que permite la entrada y salida de objetos `char`. A lo largo de este capítulo utilizaremos estas definiciones `typedef`.

Jerarquía de plantillas de E/S de flujos y sobrecarga de operadores

Las plantillas `basic_istream` y `basic_ostream` se derivan a través de la herencia simple de la plantilla base `basic_ios`.¹ La plantilla `basic_iostream` se deriva a través de la herencia múltiple² de las plantillas `basic_istream` y `basic_ostream`. El diagrama de clases de UML de la figura 15.1 sintetiza estas relaciones de herencia.

La sobrecarga de operadores proporciona una notación conveniente para realizar operaciones de entrada/salida. El operador de desplazamiento a la izquierda (`<<`) se sobrecarga para designar la salida de flujos, y se conoce como el operador de inserción de flujo. El operador de desplazamiento a la derecha (`>>`) se sobrecarga para designar la entrada de flujos, y se conoce como el operador de extracción de flujo. Estos operadores se utilizan con los objetos de flujo estándar `cin`, `cout`, `cerr` y `clog`, y comúnmente con los objetos de flujo definidos por el usuario.

1. En este capítulo hablaremos sobre las plantillas sólo en el contexto de las especializaciones de plantillas que permiten la E/S de objetos `char`. Estas especializaciones son clases y, por ende, pueden heredar unas de otras.
 2. La herencia múltiple se discute en el capítulo 25, Otros temas.

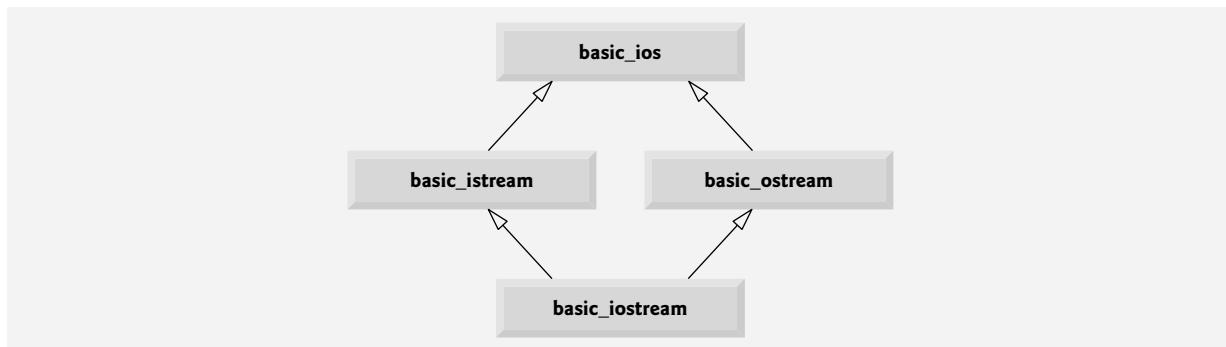


Figura 15.1 | Porción de la jerarquía de plantillas de E/S de flujos.

Los objetos de flujo estándar `cin`, `cout`, `cerr` y `clog`

El objeto predefinido `cin` es una instancia de `istream`, y se dice está “conectado a” (o unido a) el dispositivo de entrada estándar, que por lo general es el teclado. El operador de extracción de flujo (`>>`) que se utiliza en la siguiente instrucción hace que se introduzca un valor para la variable entera `calificacion` (suponiendo que `calificacion` se haya declarado como variable `int`) de `cin` a la memoria:

```
cin >> calificacion; // los datos "fluyen" en la dirección de las flechas
```

Observe que el compilador determina el tipo de datos de `calificacion` y selecciona el operador de extracción de flujo sobrecargado apropiado. Suponiendo que `calificaciones` se haya declarado en forma apropiada, el operador de extracción de flujo no requiere información adicional sobre el tipo (como es el caso, por ejemplo, en la E/S estilo C). El operador `>>` se sobrecarga para introducir elementos de datos de los tipos integrados, cadenas y valores de apuntadores.

El objeto predefinido `cout` es una instancia de `ostream` y se dice que está “conectado a” el dispositivo de salida estándar, que por lo general es la pantalla. El operador de inserción de flujo (`<<`) que se utiliza en la siguiente instrucción hace que el valor de la variable `calificacion` se envíe de la memoria al dispositivo de salida estándar:

```
cout << calificacion; // los datos "fluyen" en la dirección de las flechas
```

Observe que el compilador también determina el tipo de datos de `calificacion` (suponiendo que `calificacion` se haya declarado en forma apropiada) y selecciona el operador de inserción de flujo apropiado, de manera que el operador de inserción de flujo no requiere información adicional sobre el tipo. El operador `<<` se sobrecarga para imprimir elementos de datos de los tipos integrados, cadenas y valores de apuntadores.

El objeto predefinido `cerr` es una instancia de `ostream` y se dice que “está conectado a” el dispositivo de error estándar. Las operaciones de salida hacia el objeto `cerr` son **sin búfer**, lo cual implica que cada inserción de flujo en `cerr` hace que su salida aparezca de inmediato; esto es apropiado para notificar a un usuario oportunamente acerca de los errores.

El objeto predefinido `clog` es una instancia de la clase `ostream` y se dice que “está conectado a” el dispositivo de error estándar. Las salidas hacia `clog` son **con búfer**. Esto significa que cada inserción en `clog` podría hacer que su salida se contenga en un búfer hasta que éste se llene, o hasta que se vacíe. El uso de búfer es una técnica para mejorar el rendimiento de las operaciones de E/S que se describe en los cursos de sistemas operativos.

Plantillas de procesamiento de archivos

El procesamiento de archivos en C++ utiliza las plantillas de clases `basic_ifstream` (para la entrada de archivos), `basic_ofstream` (para la salida de archivos) y `basic_fstream` (para la entrada y salida de archivos). Cada plantilla de clase tiene una especialización de plantilla predefinida que permite la E/S de objetos `char`. C++ proporciona un conjunto de definiciones `typedef` que proporcionan alias para estas especializaciones de plantilla. Por ejemplo, la definición `typedef ifstream` representa una especialización de `basic_ifstream` que permite la entrada de objetos `char` desde un archivo. De manera similar, `typedef ofstream` representa una especialización de `basic_ofstream` que permite la salida de objetos `char` hacia un archivo. Además, `typedef fstream` representa una especialización de `basic_fstream` que permite la entrada y salida de objetos `char` desde, y hacia, un archivo. La plantilla `basic_ifstream` hereda de `basic_istream`, `basic_ofstream` hereda de `basic_ostream` y `basic_fstream` hereda de `basic_iostream`. El diagrama de clases de UML de la figura 15.2 sintetiza las diversas relaciones de herencia de las clases relacionadas con la E/S. La jerarquía de clases completa de E/S de flujos proporciona la mayoría de las herramientas que requieren los programadores. Consulte la referencia de las bibliotecas de clases de su sistema de C++ para obtener información adicional sobre el procesamiento de archivos.

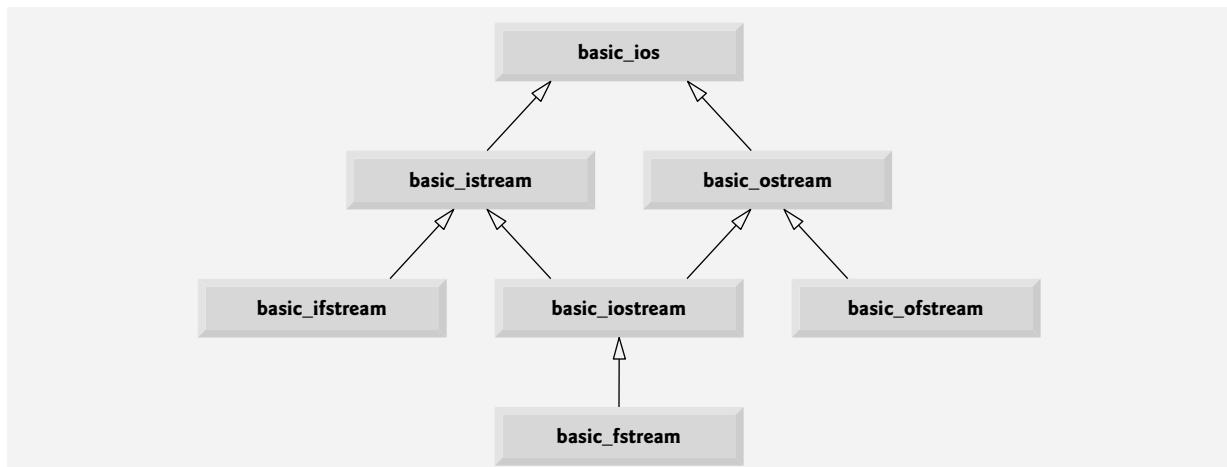


Figura 15.2 | Parte de la jerarquía de plantillas de E/S de flujos, que muestra las plantillas principales de procesamiento de archivos.

15.3 Salida de flujos

La clase `ostream` proporciona las herramientas de salida con formato y sin formato. Las herramientas para la salida incluyen la salida de tipos de datos estándar con el operador de inserción de flujo (`<<`); la salida de caracteres mediante la función miembro `put`; la salida sin formato mediante la función miembro `write` (sección 15.5); la salida de enteros en los formatos decimal, octal y hexadecimal con puntos decimales forzados (sección 15.6.1), la salida de valores de punto flotante con precisión variada (sección 15.6.2), con puntos decimales obligados (sección 15.7.1), en notación científica y en notación fija (sección 15.7.5); la salida de datos justificados en campos con anchuras designadas (sección 15.7.2); la salida de datos en campos llenos con caracteres especificados (sección 15.7.3); y la salida de letras mayúsculas en notación científica y notación hexadecimal (sección 15.7.6).

15.3.1 Salida de variables `char` *

C++ determina los tipos de datos de manera automática, una mejora sobre C. Esta característica algunas veces se “interpone en el camino”. Por ejemplo, suponga que deseamos imprimir el valor de una variable `char *` en una cadena de caracteres (es decir, la dirección de memoria del primer carácter de esa cadena). Sin embargo, el operador `<<` se ha sobre cargado para imprimir datos de tipo `char *` como una cadena con terminación nula. La solución es convertir el valor `char *` en `void *` (de hecho, esto debería hacerse con cualquier variable apuntador que el programador desee imprimir como una dirección). La figura 15.3 demuestra cómo imprimir una variable `char *` en los formatos de cadena y de dirección. Observe que la dirección se imprime como un número hexadecimal (base 16). [Nota: para aprender más acerca de los números hexadecimales, lea el apéndice D, Sistemas numéricos]. En las secciones 15.6.1, 15.7.4, 15.7.5 y 15.7.7 veremos más acerca de cómo controlar las bases de los números. [Nota: la dirección de memoria que se muestra en la salida del programa de la figura 15.3 puede diferir de un compilador a otro].

15.3.2 Salida de caracteres mediante la función miembro `put`

Podemos usar la función miembro `put` para imprimir caracteres. Por ejemplo, la instrucción

```
cout.put( 'A' );
```

muestra un solo carácter A. Las llamadas a `put` se pueden poner en cascada, como en la instrucción

```
cout.put( 'A' ).put( '\n' );
```

que imprime la letra A seguida de un carácter de nueva línea. Al igual que con `<<`, la instrucción anterior se ejecuta de esta forma, debido a que el operador punto (.) asocia de izquierda a derecha, y la función miembro `put` devuelve una referencia al objeto `ostream` (`cout`) que recibió la llamada a `put`. La función `put` también se puede llamar con una expresión numérica que represente un valor ASCII, como en la siguiente instrucción:

```
cout.put( 65 );
```

que también imprime A.

```

1 // Fig. 15.3: Fig15_03.cpp
2 // Impresión de la dirección almacenada en una variable char *.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     char *palabra = "nuevamente";
10
11    // muestra el valor de char *, y después muestra el valor de char *
12    // static_cast a void *
13    cout << "El valor de la palabra es: " << palabra << endl
14    << "El valor de static_cast< void * >( palabra ) es: "
15    << static_cast< void * >( palabra ) << endl;
16
17    return 0;
18 } // fin de main

```

```

El valor de la palabra es: nuevamente
El valor de static_cast< void * >( palabra ) es: 0041675C

```

Figura 15.3 | Impresión de la dirección almacenada en una variable char *.

15.4 Entrada de flujos

Ahora vamos a considerar la entrada de flujos. La clase `istream` proporciona las herramientas de entrada con formato y sin formato. Por lo general, el operador de extracción de flujo (es decir, el operador `>>` sobrecargado) omite los **caracteres de espacio en blanco** (como espacios, tabuladores y caracteres de nueva línea) en el flujo de entrada; más adelante veremos cómo modificar este comportamiento. Después de cada entrada, el operador de extracción de flujo devuelve una referencia al objeto flujo que recibió el mensaje de extracción (por ejemplo, `cin` en la expresión `cin >> calificacion`). Si se utiliza esa referencia como una condición (por ejemplo, en la condición de continuación de ciclo de una instrucción `while`), la función del operador de conversión `void *` sobrecargado del flujo se invoca de manera implícita para convertir la referencia en un valor de apuntador no nulo, o en el apuntador nulo con base en el éxito o fracaso de la última operación de entrada. Un apuntador no nulo se convierte en el valor `bool true` para indicar éxito, y el apuntador nulo se convierte en el valor `bool false` para indicar fracaso. Cuando se hace un intento de leer más allá del final de un flujo, el operador de conversión sobrecargado `void *` del flujo devuelve el apuntador nulo para indicar el fin del archivo.

Cada objeto flujo contiene un conjunto de **bites de estado** que se utilizan para controlar el estado del flujo (es decir, aplicar formato, establecer estados de error, etc.). El operador de conversión sobrecargado `void *` utiliza estos bites para determinar si debe devolver un apuntador no nulo o el apuntador nulo. La extracción de flujo hace que se establezca el bit `failbit` del flujo si se introducen datos del tipo incorrecto, y hace que se establezca el bit `badbit` del flujo si falla la operación. En las secciones 15.7 y 15.8 hablaremos sobre los bites de estado de un flujo con detalle, y después le mostraremos cómo evaluar estos bites después de una operación de E/S.

15.4.1 Funciones miembro `get` y `getline`

La función miembro `get` sin argumentos recibe como entrada un carácter del flujo designado (incluyendo caracteres de espacio en blanco y otros caracteres no gráficos, como la secuencia de teclas que representa el fin de archivo) y lo devuelve como el valor de la llamada a la función. Esta versión de `get` devuelve `EOF` cuando se encuentra el fin del archivo.

Uso de las funciones miembro `eof`, `get` y `put`

La figura 15.4 demuestra el uso de las funciones miembro `eof` y `get` en el flujo de entrada `cin`, y la función miembro `put` en el flujo de salida `cout`. El programa primero imprime el valor de `cin.eof()` [es decir, `false` (0 en la salida)] para mostrar que no ha ocurrido el fin de archivo en `cin`. El usuario introduce una línea de texto y oprime *Intro* seguido del fin de archivo (`<Ctrl>-z` en sistemas Microsoft Windows, `<Ctrl>-d` en sistemas UNIX y Macintosh). En la línea 17 se lee cada carácter, que en la línea 18 se envía como salida a `cout` mediante la función miembro `put`. Al encontrar el fin de archivo la instrucción `while` termina, y en la línea 22 se muestra el valor de `cin.eof()`, que ahora es `true` (1 en la salida), para mostrar que se ha establecido el fin de archivo en `cin`. Observe que este programa utiliza la versión de la función miembro `get` de `istream` que no recibe argumentos y devuelve el carácter que se está introduciendo (línea 17). La función `eof` devuelve `true` sólo después de que el programa trata de leer más allá del último carácter en el flujo.

```

1 // Fig. 15.4: Fig15_04.cpp
2 // Uso de las funciones miembro get, put y eof.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10     int caracter; // usa int, ya que char no puede representar EOF
11
12     // pide al usuario que introduzca una linea de texto
13     cout << "Antes de la entrada, cin.eof() es " << cin.eof() << endl
14     << "Escriba un enunciado seguido del fin de archivo:" << endl;
15
16     // usa get para leer cada carácter; usa put para mostrarlo
17     while ( ( caracter = cin.get() ) != EOF )
18         cout.put( caracter );
19
20     // muestra el carácter de fin de archivo
21     cout << "\nEOF en este sistema es: " << caracter << endl;
22     cout << "Despues de introducir EOF, cin.eof() es " << cin.eof() << endl;
23     return 0;
24 } // fin de main

```

```

Antes de la entrada, cin.eof() es 0
Escriba un enunciado seguido del fin de archivo:
Prueba de las funciones miembro get y put
Prueba de las funciones miembro get y put
^Z

EOF en este sistema es: -1
Despues de introducir EOF, cin.eof() es 1

```

Figura 15.4 | Funciones miembro get, put y eof.

La función miembro `get` con un argumento de referencia de carácter introduce el siguiente carácter del flujo de entrada (aun si es un carácter de espacio en blanco) y lo almacena en el argumento tipo carácter. Esta versión de `get` devuelve una referencia al objeto `istream` para el que se está invocando la función miembro `get`.

Una tercera versión de `get` recibe tres argumentos: un arreglo de caracteres, un límite de tamaño y un delimitador (con el valor predeterminado '\n'). Esta versión lee caracteres del flujo de entrada. Lee un carácter menos que el número máximo especificado de caracteres y termina, o se termina tan pronto como se lea el delimitador. Se inserta un carácter nulo para terminar la cadena de entrada en el arreglo de caracteres que el programa utiliza como búfer. El delimitador no se coloca en el arreglo de caracteres, sino que permanece en el flujo de entrada (el delimitador será el siguiente carácter que se lea). Así, el resultado de una segunda función `get` consecutiva es una línea vacía, a menos que el carácter delimitador se elimine del flujo de entrada (posiblemente con `cin.ignore()`).

Comparación entre `cin` y `cin.get`

La figura 15.5 compara la entrada mediante el uso de la extracción de flujo con `cin` (que lee caracteres hasta que se encuentra un carácter de espacio en blanco) y la entrada mediante el uso de `cin.get`. Observe que la llamada a `cin.get` (línea 24) no especifica un delimitador, por lo que se utiliza el carácter predeterminado '\n'.

```

1 // Fig. 15.5: Fig15_05.cpp
2 // Contraste entre la entrada de una cadena mediante cin y cin.get.
3 #include <iostream>
4 using std::cin;

```

Figura 15.5 | Entrada de una cadena mediante el uso de `cin` con la extracción de flujo, en comparación con la entrada mediante el uso de `cin.get`. (Parte 1 de 2).

```

5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10     // crea dos arreglos char, cada una con 80 elementos
11     const int TAMANIO = 80;
12     char bufer1[ TAMANIO ];
13     char bufer2[ TAMANIO ];
14
15     // usa cin para introducir caracteres en bufer1
16     cout << "Escriba un enunciado:" << endl;
17     cin >> bufer1;
18
19     // muestra el contenido de bufer1
20     cout << "\nLa cadena leida con cin fue:" << endl
21         << bufer1 << endl << endl;
22
23     // usa cin.get para introducir caracteres en bufer2
24     cin.get( bufer2, TAMANIO );
25
26     // muestra el contenido de bufer2
27     cout << "La cadena leida con cin.get fue:" << endl
28         << bufer2 << endl;
29     return 0;
30 } // fin de main

```

Escriba un enunciado:

Contraste entre la entrada de una cadena mediante cin y cin.get

La cadena leída con cin fue:

Contraste

La cadena leída con cin.get fue:

entre la entrada de una cadena mediante cin y cin.get

Figura 15.5 | Entrada de una cadena mediante el uso de `cin` con la extracción de flujo, en comparación con la entrada mediante el uso de `cin.get`. (Parte 2 de 2).

Uso de la función miembro `getline`

La función miembro `getline` opera de manera similar a la tercera versión de la función miembro `get` e inserta un carácter nulo después de la línea en el arreglo de caracteres. La función `getline` elimina el delimitador del flujo (es decir, lee el carácter y lo descarta), pero no lo almacena en el arreglo de caracteres. El programa de la figura 15.6 demuestra el uso de la función miembro `getline` para introducir una línea de texto (línea 15).

```

1  // Fig. 15.6: Fig15_06.cpp
2  // Introducción de caracteres mediante la función miembro getline de cin.
3  #include <iostream>
4  using std::cin;
5  using std::cout;
6  using std::endl;
7
8  int main()
9  {
10     const int TAMANIO = 80;
11     char bufer[ TAMANIO ]; // crea un arreglo de 80 caracteres
12
13     // introduce caracteres en bufer mediante la función getline de cin
14     cout << "Escriba un enunciado:" << endl;

```

Figura 15.6 | Introducción de datos tipo carácter con la función miembro `getline` de `cin`. (Parte 1 de 2).

```

15     cin.getline( bufer, TAMANIO );
16
17     // muestra el contenido de bufer
18     cout << "\nEl enunciado introducido es:" << endl << bufer << endl;
19     return 0;
20 } // fin de main

```

Escriba un enunciado:

Uso de la función miembro getline

El enunciado introducido es:

Uso de la función miembro getline

Figura 15.6 | Introducción de datos tipo carácter con la función miembro `getline` de `cin`. (Parte 2 de 2).

15.4.2 Funciones miembro peek, putback e ignore de istream

La función miembro `ignore` de `istream` lee y descarta un número designado de caracteres (el valor predeterminado es un carácter) o termina al momento de encontrar un delimitador designado (el delimitador predeterminado es EOF, que hace que `ignore` salte hasta el fin del archivo cuando lee datos del mismo).

La función miembro `putback` coloca el carácter anterior, obtenido por una operación `get` de un flujo de entrada, de vuelta a ese flujo. Esta función es útil para las aplicaciones que exploran un flujo de entrada en busca de un campo que empiece con un carácter específico. Cuando se introduce ese carácter, la aplicación devuelve el carácter al flujo, por lo que éste se puede incluir en los datos de entrada.

La función miembro `peek` devuelve el siguiente carácter de un flujo de entrada, pero no lo elimina del flujo.

15.4.3 E/S con seguridad de tipos

C++ ofrece la E/S con seguridad de tipos. Los operadores `<<` y `>>` se sobrecargan para aceptar elementos de datos de tipos específicos. Si se procesan datos inesperados, se establecen varios bits de error, que el usuario puede evaluar para determinar si una operación de E/S tuvo éxito o fracasó. Si el operador `<<` no se ha sobrecargado para un tipo definido por el usuario y el programador intenta realizar operaciones de entrada o salida con el contenido de un objeto de ese tipo definido por el usuario, el compilador reporta un error. Esto permite al programa “permanecer con el control”. En la sección 15.8 hablaremos acerca de estos estados de error.

15.5 E/S sin formato mediante el uso de read, write y gcount

La entrada/salida sin formato se lleva a cabo mediante las funciones miembro `read` y `write` de `istream` y `ostream`, respectivamente. La función miembro `read` introduce cierto número de bytes en un arreglo de caracteres en la memoria; la función miembro `write` envía bytes de salida desde un arreglo de caracteres. Estos bytes no tienen ningún tipo de formato. Se reciben como entrada o se envían de salida como bytes puros. Por ejemplo, la llamada

```

char bufer[ ] = "FELIZ CUMPLEAÑOS";
cout.write( bufer, 10 );

```

imprime los primeros 10 bytes de `bufer` (incluyendo caracteres nulos, si los hay, que hagan que termine la salida con `cout` y `<<`). La llamada

```
cout.write( "ABCDEFGHIJKLMNPQRSTUVWXYZ", 10 );
```

muestra los primeros 10 caracteres del alfabeto.

La función miembro `read` introduce un número designado de caracteres en un arreglo de caracteres. Si se leen menos caracteres que el número designado, se establece el bit `failbit`. En la sección 15.8 le mostraremos cómo determinar si se ha establecido `failbit`. La función miembro `gcount` reporta el número de caracteres leídos por la última operación de entrada.

La figura 15.7 demuestra las funciones miembro `read` y `gcount` de `istream`, y la función miembro `write` de `ostream`. El programa introduce 20 caracteres (de una secuencia de entrada más larga) en el arreglo `bufer` con `read` (línea 15), determina el número de caracteres introducidos con `gcount` (línea 19) y envía como salida los caracteres en `bufer` con `write` (línea 19).

```

1 // Fig. 15.7: Fig15_07.cpp
2 // E/S sin formato mediante el uso de read, gcount y write.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10     const int TAMANIO = 80;
11     char bufer[ TAMANIO ]; // crea un arreglo de 80 caracteres
12
13     // usa la función read para introducir caracteres en el búfer
14     cout << "Escriba un enunciado:" << endl;
15     cin.read( bufer, 20 );
16
17     // usa las funciones write y gcount para mostrar los caracteres del búfer
18     cout << endl << "El enunciado que escribio fue:" << endl;
19     cout.write( bufer, cin.gcount() );
20     cout << endl;
21     return 0;
22 } // fin de main

```

Escriba un enunciado:

Uso de las funciones miembro read, write y gcount

El enunciado que escribio fue:

Uso de read, write

Figura 15.7 | E/S con formato mediante el uso de las funciones miembro `read`, `gcount` y `write`.

15.6 Introducción a los manipuladores de flujos

C++ proporciona varios **manipuladores de flujos** que realizan tareas de formato. Los manipuladores de flujos proporcionan herramientas para establecer las anchuras de los campos, establecer la precisión, establecer y quitar el formato de estado, establecer el carácter de relleno en los campos, vaciar flujos, insertar una nueva línea en el flujo de salida (y vaciar el flujo), insertar un carácter nulo en el flujo de salida y omitir el espacio en blanco en el flujo de entrada. Estas características se describen en las siguientes secciones.

15.6.1 Base de flujos integrales: `dec`, `oct`, `hex` y `setbase`

Los enteros se interpretan generalmente como valores decimales (base 10). Para cambiar la base en la que se interpretan los enteros en un flujo, inserte el manipulador `hex` para establecer la base en hexadecimal (base 16) o inserte el manipulador `oct` para establecer la base en octal (base 8). Inserte el manipulador `dec` para restablecer la base del flujo en decimal. Todos éstos son manipuladores pegajosos.

La base de un flujo también se puede cambiar mediante el manipulador de flujos `setbase`, el cual recibe un argumento entero de 10, 8 o 16 para establecer la base en decimal, octal o hexadecimal, respectivamente. Debido a que `setbase` recibe un argumento, se conoce como manipulador de flujo parametrizado. El uso de `setbase` (o de cualquier otro manipulador parametrizado) requiere la inclusión del archivo de encabezado `<iomanip>`. El valor de la base del flujo permanece igual hasta que se cambia de manera explícita; las opciones de `setbase` son “pegajosas”. La figura 15.8 demuestra los manipuladores de flujos `hex`, `oct`, `dec` y `setbase`.

```

1 // Fig. 15.8: Fig15_08.cpp
2 // Uso de los manipuladores de flujos hex, oct, dec y setbase.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::dec;
7 using std::endl;

```

Figura 15.8 | Los manipuladores de flujos `hex`, `oct`, `dec` y `setbase`. (Parte I de 2).

```

8  using std::hex;
9  using std::oct;
10
11 #include <iomanip>
12 using std::setbase;
13
14 int main()
15 {
16     int numero;
17
18     cout << "Escriba un número decimal: ";
19     cin >> numero; // recibe el número de entrada
20
21     // usa el manipulador de flujo hex para mostrar un número hexadecimal
22     cout << numero << " en hexadecimal es: " << hex
23     << numero << endl;
24
25     // usa el manipulador de flujo oct para mostrar un número octal
26     cout << dec << numero << " en octal es: "
27     << oct << numero << endl;
28
29     // usa el manipulador de flujo setbase para mostrar un número decimal
30     cout << setbase( 10 ) << numero << " en decimal es: "
31     << numero << endl;
32
33     return 0;
34 } // fin de main

```

```

Escriba un numero decimal: 20
20 en hexadecimal es: 14
20 en octal es: 24
20 en decimal es: 20

```

Figura 15.8 | Los manipuladores de flujos hex, oct, dec y setbase. (Parte 2 de 2).

15.6.2 Precisión de punto flotante (precision, setprecision)

Para controlar la **precisión** de los números de punto flotante (es decir, el número de dígitos a la derecha del punto decimal), podemos usar el manipulador de flujo **setprecision** o la función miembro **precision** de **ios_base**. Una llamada a uno de estos miembros establece la precisión para todas las operaciones de salida subsecuentes, hasta la siguiente llamada para establecer la precisión. Una llamada a la función miembro **precision** sin argumento devuelve la opción de precisión actual (esto es lo que necesitamos usar para poder restaurar la precisión original en un momento dado, una vez que ya no sea necesaria una opción “pegajosa”). El programa de la figura 15.9 utiliza tanto la función miembro **precision** (línea 28) como el manipulador **setprecision** (línea 37) para imprimir una tabla que muestra la raíz cuadrada de 2, en donde la precisión varía de 0 a 9.

```

1 // Fig. 15.9: Fig15_09.cpp
2 // Control de la precisión de los valores de punto flotante.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 #include <cmath>
12 using std::sqrt; // prototipo de sqrt
13

```

Figura 15.9 | Precisión de los valores de punto flotante. (Parte 1 de 2).

```

14 int main()
15 {
16     double raiz2 = sqrt( 2.0 ); // calcula la raíz cuadrada de 2
17     int posiciones; // precisión, varía de 0 a 9
18
19     cout << "Raiz cuadrada de 2 con precisiones de 0 a 9." << endl
20         << "Precision establecida mediante la función miembro precision "
21         << "de ios_base:" << endl;
22
23     cout << fixed; // usa el formato de punto fijo
24
25     // muestra la raíz cuadrada usando la función precision de ios_base
26     for ( posiciones = 0; posiciones <= 9; posiciones++ )
27     {
28         cout.precision( posiciones );
29         cout << raiz2 << endl;
30     } // fin de for
31
32     cout << "\nPrecision establecida por el manipulador de flujo "
33         << "setprecision:" << endl;
34
35     // establece la precisión para cada dígito, y después muestra la raíz cuadrada
36     for ( posiciones = 0; posiciones <= 9; posiciones++ )
37         cout << setprecision( posiciones ) << raiz2 << endl;
38
39     return 0;
40 } // fin de main

```

Raiz cuadrada de 2 con precisiones de 0 a 9.
 Precision establecida mediante la función miembro precision de ios_base:

```

1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562

```

Precision establecida por el manipulador de flujo setprecision:

```

1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562

```

Figura 15.9 | Precisión de los valores de punto flotante. (Parte 2 de 2).

15.6.3 Anchura de campos (width, setw)

La función miembro **width** (de la clase base **ios_base**) establece la anchura de campo (es decir, el número de posiciones de caracteres en los que debe imprimirse un valor, o el número máximo de caracteres que deben introducirse) y devuelve la anchura anterior. Si los valores que se imprimen son menos que la anchura de campo, se insertan **caracteres de relleno** como **relleno** (padding). Un valor más ancho que la anchura designada no se truncará; se imprimirá el número completo. La función **width** sin argumento devuelve la configuración actual.



Error común de programación 15.1

La opción de anchura se aplica sólo para la siguiente inserción o extracción (es decir, la opción de anchura no es “pegajosa”); después de esto, la anchura se establece de manera implícita en 0 (es decir, la entrada y la salida se llevarán a cabo con las opciones predeterminadas). Suponer que la opción de anchura se aplica a todas las operaciones de salida subsiguientes es un error lógico.



Error común de programación 15.2

Cuando un campo no es lo bastante ancho como para manejar las salidas, éstas se imprimen con la anchura necesaria, lo cual puede producir resultados confusos.

La figura 15.10 demuestra el uso de la función miembro `width` en operaciones de entrada y de salida. Observe que, al introducir datos en un arreglo `char`, se leerá un máximo de caracteres igual a uno menos la anchura, ya que se toma en cuenta el carácter nulo que se va a colocar en la cadena de entrada. Recuerde que la extracción de flujo termina al encontrar espacio en blanco a la derecha. El manipulador de flujo `setw` también se puede usar para establecer la anchura de los campos.

[Nota: cuando se pide al usuario la entrada en la figura 15.10, éste debe introducir una línea de texto y oprimir la tecla `Intro`, seguida del fin de archivo (`<Ctrl>-z` en sistemas Microsoft Windows, `<Ctrl>-d` en sistemas UNIX y Macintosh)].

```

1 // Fig. 15.10: Fig15_10.cpp
2 // Demostración de la función miembro width.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10     int valorAnchura = 4;
11     char enunciado[ 10 ];
12
13     cout << "Escriba un enunciado:" << endl;
14     cin.width( 5 ); // introduce sólo 5 caracteres de enunciado
15
16     // establece la anchura de campo y después muestra los caracteres con base en esa anchura
17     while ( cin >> enunciado )
18     {
19         cout.width( valorAnchura++ );
20         cout << enunciado << endl;
21         cin.width( 5 ); // introduce 5 caracteres más de enunciado
22     } // fin de while
23
24     return 0;
25 } // fin de main

```

Escriba un enunciado:

Esta es una prueba de la función miembro width

Esta

es
una
prue
ba
de
la
func
ion
miem
bro
widt
h

Figura 15.10 | La función miembro `width` de la clase `ios_base`.

15.6.4 Manipuladores de flujos de salida definidos por el usuario

El programador puede crear sus propios manipuladores de flujos.³ La figura 15.11 muestra la creación y uso de los nuevos manipuladores de flujos no parametrizados `alarma` (líneas 10 a 13), `retornoCarro` (líneas 16 a 19), `tab` (líneas 22 a 25) y `finLinea` (líneas 29 a 32). Para los manipuladores de flujos de salida, el tipo de valor de retorno y el parámetro

```

1 // Fig. 15.11: Fig15_11.cpp
2 // Creación y prueba de manipuladores de flujos
3 // no parametrizados, definidos por el usuario.
4 #include <iostream>
5 using std::ostream;
6 using std::cout;
7 using std::flush;
8
9 // manipulador alarma (usa la secuencia de escape \a)
10 ostream& alarma( ostream& salida )
11 {
12     return salida << '\a'; // emite el sonido del sistema
13 } // fin del manipulador alarma
14
15 // manipulador retornoCarro (usa la secuencia de escape \r)
16 ostream& retornoCarro( ostream& salida )
17 {
18     return salida << '\r'; // emite el retorno de carro
19 } // fin del manipulador retornoCarro
20
21 // manipulador tab (usa la secuencia de escape \t)
22 ostream& tab( ostream& salida )
23 {
24     return salida << '\t'; // emite el tabulador
25 } // fin del manipulador tab
26
27 // manipulador finLinea (usa la secuencia de escape \n y la
28 // función miembro flush)
29 ostream& finLinea( ostream& salida )
30 {
31     return salida << '\n' << flush; // emite fin de línea parecido a endl
32 } // fin del manipulador finLinea
33
34 int main()35 {
35     // usa los manipuladores tab y finLinea
36     cout << "Prueba del manipulador tab:" << finLinea
37         << 'a' << tab << 'b' << tab << 'c' << finLinea;
38
39     cout << "Prueba de los manipuladores retornoCarro y alarma:"
40         << finLinea << ".....";
41
42     cout << alarma; // usa el manipulador alarma
43
44     // usa los manipuladores ret y finLinea
45     cout << retornoCarro << "----" << finLinea;
46     return 0;
47 } // fin de main

```

Prueba del manipulador tab:

a b c

Prueba de los manipuladores retornoCarro y alarma:

Figura 15.11 | Manipuladores de flujos no parametrizados, definidos por el usuario.

3. El programador también puede crear sus propios manipuladores de flujos parametrizados. Este concepto está más allá del alcance de este libro.

deben ser de tipo `ostream &`. Cuando en la línea 37 se inserta el manipulador `finLinea` en el flujo de salida, se hace una llamada a la función `finLinea` y en la línea 31 se imprime la secuencia de escape `\n` junto con el manipulador `flush` al flujo de salida estándar `cout`. De manera similar, cuando en las líneas 37 a 46 se insertan los manipuladores `tab`, `alarma` y `retornoCarro` en el flujo de salida, se hacen llamadas a sus funciones correspondientes: `tab` (línea 22), `alarma` (línea 10) y `retornoCarro` (línea 16), que a su vez imprimen varias secuencias de escape.

15.7 Estados de formato de flujos y manipuladores de flujos

Se pueden utilizar varios manipuladores de flujos para especificar los tipos de formato a realizar durante las operaciones de E/S de flujos. Los manipuladores de flujos controlan la configuración del formato de la salida. La figura 15.12 ilustra cada manipulador de flujo que controla un estado de formato de un flujo dado. Todos estos manipuladores pertenecen a la clase `ios_base`. En las siguientes secciones mostraremos ejemplos de la mayoría de estos manipuladores de flujos.

Manipulador de flujo	Descripción
<code>skipws</code>	Omite los caracteres de espacio en blanco en un flujo de entrada. Esta opción se restablece con el manipulador de flujo <code>noskipws</code> .
<code>left</code>	Justifica la salida a la izquierda en un campo. Si es necesario, aparecen caracteres de relleno a la derecha.
<code>right</code>	Justifica la salida a la derecha en un campo. Si es necesario, aparecen caracteres de relleno a la izquierda.
<code>internal</code>	Indica que el signo de un número debe justificarse a la izquierda en un campo, y que la magnitud del número se debe justificar a la derecha en ese mismo campo (es decir, deben aparecer caracteres de relleno entre el signo y el número).
<code>dec</code>	Especifica que los enteros deben tratarse como valores decimales (base 10).
<code>oct</code>	Especifica que los enteros se deben tratar como valores octales (base 8).
<code>hex</code>	Especifica que los enteros se deben tratar como valores hexadecimales (base 16).
<code>showbase</code>	Especifica que la base de un número se debe imprimir adelante del mismo (un 0 a la izquierda para los valores octales; 0x o 0X a la izquierda para los valores hexadecimales). Esta opción se restablece con el manipulador de flujo <code>noshowbase</code> .
<code>showpoint</code>	Especifica que los números de punto flotante se deben imprimir con un punto decimal. Esto se usa generalmente con <code>fixed</code> para garantizar cierto número de dígitos a la derecha del punto decimal, aun y cuando sean ceros. Esta opción se restablece con el manipulador de flujo <code>noshowpoint</code> .
<code>uppercase</code>	Especifica que deben usarse letras mayúsculas (es decir, X y de la A a la F) en un entero hexadecimal, y que se debe usar la letra E al representar un valor de punto flotante en notación científica. Esta opción se restablece con el manipulador de flujo <code>nouppercase</code> .
<code>showpos</code>	Especifica que a los números positivos se les debe anteponer un signo positivo (+). Esta opción se restablece con el manipulador de flujo <code>noshowpos</code> .
<code>scientific</code>	Especifica la salida de un valor de punto flotante en notación científica.
<code>fixed</code>	Especifica la salida de un valor de punto flotante en notación de punto fijo, con un número específico de dígitos a la derecha del punto decimal.

Figura 15.12 | Manipuladores de flujo de formato de estado de `<iostream>`.

15.7.1 Ceros a la derecha y puntos decimales (`showpoint`)

El manipulador de flujo `showpoint` obliga a que un número de punto flotante se imprima con su punto decimal y ceros a la derecha. Por ejemplo, el valor de punto flotante `79.0` se imprime como `79` sin usar `showpoint`, y se imprime como `79.000000` (o con todos los ceros que se especifiquen mediante la precisión actual) usando `showpoint`. Para restablecer la opción de `showpoint`, hay que imprimir el manipulador de flujo `noshowpoint`. El programa de la figura 15.13 muestra

```

1 // Fig. 15.13: Fig15_13.cpp
2 // Uso de showpoint para controlar la impresión de
3 // ceros a la derecha y puntos decimales para valores double.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7 using std::showpoint;
8
9 int main()
10 {
11     // muestra los valores double con formato de flujo predeterminado
12     cout << "Antes de usar showpoint" << endl
13         << "9.9900 se imprime como: " << 9.9900 << endl
14         << "9.9000 se imprime como: " << 9.9000 << endl
15         << "9.0000 se imprime como: " << 9.0000 << endl << endl;
16
17     // muestra el valor double después de showpoint
18     cout << showpoint
19         << "Despues de usar showpoint" << endl
20         << "9.9900 se imprime como: " << 9.9900 << endl
21         << "9.9000 se imprime como: " << 9.9000 << endl
22         << "9.0000 se imprime como: " << 9.0000 << endl;
23
24     return 0;
25 } // fin de main

```

```

Antes de usar showpoint
9.9900 se imprime como: 9.99
9.9000 se imprime como: 9.9
9.0000 se imprime como: 9

Despues de usar showpoint
9.9900 se imprime como: 9.99000
9.9000 se imprime como: 9.90000
9.0000 se imprime como: 9.00000

```

Figura 15.13 | Control de la impresión de ceros a la derecha y puntos decimales en los valores de punto flotante.

cómo usar el manipulador de flujo `showpoint` para controlar la impresión de ceros a la derecha y puntos decimales para los valores de punto flotante. Recuerde que la precisión predeterminada de un número de punto flotante es 6. Cuando no se utilizan los manipuladores de flujo `fixed` o `scientific`, la precisión representa el número de dígitos significativos a mostrar (es decir, el número total de dígitos a mostrar), no el número de dígitos a mostrar después del punto decimal.

15.7.2 Justificación (`left`, `right` e `internal`)

Los manipuladores de flujos `left` y `right` permiten justificar los campos a la izquierda con caracteres de relleno a la derecha, o justificarlos a la derecha con caracteres de relleno a la izquierda, respectivamente. El carácter de relleno se especifica mediante la función miembro `fill` o el manipulador de flujo parametrizado `setfill` (que veremos en la sección 15.7.3). La figura 15.14 utiliza los manipuladores `setw`, `left` y `right` para justificar a la izquierda y a la derecha los datos enteros en un campo.

```

1 // Fig. 15.14: Fig15_14.cpp
2 // Demostración de la justificación a la izquierda y a la derecha.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::left;
7 using std::right;
8
9 #include <iomanip>
10 using std::setw;

```

Figura 15.14 | Justificación a la izquierda y a la derecha con los manipuladores de flujos `left` y `right`. (Parte I de 2).

```

11
12 int main()
13 {
14     int x = 12345;
15
16     // muestra el valor de x justificado a la derecha (predeterminado)
17     cout << "La opcion predeterminada es justificado a la derecha:" << endl
18     << setw( 10 ) << x;
19
20     // usa el manipulador left para mostrar el valor de x justificado a la izquierda
21     cout << "\n\nUso de std::left para justificar x a la izquierda:\n"
22     << left << setw( 10 ) << x;
23
24     // usa el manipulador right para mostrar el valor de x justificado a la derecha
25     cout << "\n\nUso de std::right para justificar x a la derecha:\n"
26     << right << setw( 10 ) << x << endl;
27     return 0;
28 } // fin de main

```

La opcion predeterminada es justificado a la derecha:
12345

Uso de std::left para justificar x a la izquierda:
12345

Uso de std::right para justificar x a la derecha:
12345

Figura 15.14 | Justificación a la izquierda y a la derecha con los manipuladores de flujos `left` y `right`. (Parte 2 de 2).

El manipulador de flujo `internal` indica que el signo de un número (o la base, cuando se utiliza el manipulador de flujo `showbase`) debe justificarse a la izquierda dentro de un campo, que la magnitud del número se debe justificar a la derecha y que los espacios intermedios deben rellenarse con el carácter de relleno. La figura 15.15 muestra el manipulador de flujo `internal` que especifica un espaciamiento interno (línea 15). Observe que `showpos` obliga a que se imprima el signo positivo (línea 15). Para restablecer la opción de `showpos`, hay que imprimir el manipulador de flujo `noshowpos`.

15.7.3 Relleno de caracteres (`fill`, `setfill`)

La función miembro `fill` especifica el carácter de relleno que se debe utilizar con los campos justificados; si no se especifica un valor, se utilizan espacios para llenar. La función `fill` devuelve el carácter de relleno anterior. El manipulador

```

1 // Fig. 15.15: Fig15_15.cpp
2 // Impresión de un entero con espaciamiento interno y un signo positivo.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::internal;
7 using std::showpos;
8
9 #include <iomanip>
10 using std::setw;
11
12 int main()
13 {
14     // muestra el valor con espaciamiento interno y signo positivo
15     cout << internal << showpos << setw( 10 ) << 123 << endl;
16     return 0;
17 } // fin de main

```

+ 123

Figura 15.15 | Impresión de un entero con espaciamiento interno y el signo positivo.

setfill también establece el carácter de relleno. La figura 15.16 demuestra el uso de la función miembro **fill** (línea 40) y el manipulador de flujo **setfill** (líneas 44 y 47) para establecer el carácter de relleno.

```

1 // Fig. 15.16: Fig15_16.cpp
2 // Uso de la función miembro fill y el manipulador de flujo setfill para cambiar
3 // el carácter de relleno para campos más grandes que el valor impreso.
4 #include <iostream>
5 using std::cout;
6 using std::dec;
7 using std::endl;
8 using std::hex;
9 using std::internal;
10 using std::left;
11 using std::right;
12 using std::showbase;
13
14 #include <iomanip>
15 using std::setfill;
16 using std::setw;
17
18 int main()
19 {
20     int x = 10000;
21
22     // muestra x
23     cout << x << " impreso como int justificado a la derecha y a la izquierda\n"
24         << "y como hex con justificación interna.\n"
25         << "Uso del carácter de relleno predeterminado (espacio):" << endl;
26
27     // muestra x con la base
28     cout << showbase << setw( 10 ) << x << endl;
29
30     // muestra x con justificación a la izquierda
31     cout << left << setw( 10 ) << x << endl;
32
33     // muestra x como hex con justificación interna
34     cout << internal << setw( 10 ) << hex << x << endl << endl;
35
36     cout << "Uso de varios caracteres de relleno:" << endl;
37
38     // muestra x usando caracteres de relleno (justificación a la derecha)
39     cout << right;
40     cout.fill( '*' );
41     cout << setw( 10 ) << dec << x << endl;
42
43     // muestra x usando caracteres de relleno (justificación a la izquierda)
44     cout << left << setw( 10 ) << setfill( '%' ) << x << endl;
45
46     // muestra x usando caracteres de relleno (justificación interna)
47     cout << internal << setw( 10 ) << setfill( '^' ) << hex
48         << x << endl;
49
50 } // fin de main

```

```

1000 impreso como int justificado a la derecha y a la izquierda
y como hex con justificación interna
Usd del carácter de relleno predeterminado (espacio):
    1000
1000

```

Figura 15.16 | Uso de la función miembro **fill** y el manipulador de flujo **setfill** para modificar el carácter de relleno, cuando los campos son más grandes que los valores que se van a imprimir. (Parte 1 de 2).

```

0x    2710
Uso de varios caracteres de relleno
*****1000
1000%%%%%
0x^^^^2710

```

Figura 15.16 | Uso de la función miembro `fill` y el manipulador de flujo `setfill` para modificar el carácter de relleno, cuando los campos son más grandes que los valores que se van a imprimir. (Parte 2 de 2).

15.7.4 Base de flujos integrales (dec, oct, hex, showbase)

C++ proporciona los manipuladores de flujos `dec`, `hex` y `oct` para especificar que se van a mostrar enteros como valores decimales, hexadecimales y octales, respectivamente. Las inserciones de flujo usan la opción predeterminada decimal si no se utiliza uno de estos manipuladores. Con la extracción de flujo, los enteros con prefijo de 0 (cero) se tratan como valores octales, los enteros con el prefijo `0x` o `0X` se tratan como valores hexadecimales, y todos los demás enteros se tratan como valores decimales. Una vez que se especifica una base para un flujo, todos los enteros en ese flujo se procesan con esa base, hasta que se especifique una base distinta o cuando el programa termina.

El manipulador de flujo `showbase` obliga a que se imprima la base de un valor integral. Los números decimales se imprimen de manera predeterminada, los números octales se imprimen con un 0 a la izquierda, y los números decimales se imprimen con `0x` o `0X` a la izquierda (como veremos en la sección 15.7.6, el manipulador de flujo `uppercase` determina qué opción se elige). La figura 15.17 demuestra el uso del manipulador de flujo `showbase` para obligar a un entero a imprimirse en los formatos decimal, octal y hexadecimal. Para restablecer la opción de `showbase`, hay que imprimir el manipulador de flujo `noshowbase`.

```

1 // Fig. 15.17: Fig15_17.cpp
2 // Uso del manipulador de flujo showbase.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::hex;
7 using std::oct;
8 using std::showbase;
9
10 int main()
11 {
12     int x = 100;
13
14     // usa showbase para mostrar la base del número
15     cout << "Impresion de enteros, y a la derecha su base:" << endl
16     << showbase;
17
18     cout << x << endl; // imprime valor decimal
19     cout << oct << x << endl; // imprime valor octal
20     cout << hex << x << endl; // imprime valor hexadecimal
21
22 } // fin de main

```

```

Impresion de enteros, y a la derecha su base:
100
0144
0x64

```

Figura 15.17 | El manipulador de flujo `showbase`.

15.7.5 Números de punto flotante: notación científica y fija (scientific, fixed)

Los manipuladores de flujos `scientific` y `fixed` controlan el formato de salida de los números de punto flotante. El manipulador de flujo `scientific` obliga a que la salida de un número de punto flotante se muestre en formato científico. El manipulador de flujo `fixed` obliga a que un número de punto flotante muestre un número específico de dígitos

```

1 // Fig. 15.18: Fig15_18.cpp
2 // Cómo mostrar los valores de punto flotante en los formatos
3 // predeterminado del sistema, científico y fijo.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7 using std::fixed;
8 using std::scientific;
9
10 int main()
11 {
12     double x = 0.001234567;
13     double y = 1.946e9;
14
15     // muestra x e y en el formato predeterminado
16     cout << "Mostrados en el formato predeterminado:" << endl
17         << x << '\t' << y << endl;
18
19     // muestra x e y en el formato científico
20     cout << "\nMostrados en el formato científico:" << endl
21         << scientific << x << '\t' << y << endl;
22
23     // muestra x e y en formato fijo
24     cout << "\nMostrados en formato fijo:" << endl
25         << fixed << x << '\t' << y << endl;
26
27     return 0;
28 } // fin de main

```

Mostrados en el formato predeterminado:

0.00123457 1.946e+009

Mostrados en el formato científico:

1.234567e-003 1.946000e+009

Mostrados en formato fijo:

0.001235 194600000.000000

Figura 15.18 | Valores de punto flotante mostrados en los formatos predeterminado, científico y fijo.

(según lo especificado por la función miembro `precisión` o el manipulador de flujo `setprecision`) a la derecha del punto decimal. Sin usar otro manipulador, el valor del número de punto flotante determina el formato de salida.

La figura 15.18 demuestra cómo mostrar números de punto flotante en los formatos científico y fijo, usando los manipuladores de flujos `scientific` (línea 21) y `fixed` (línea 25). El formato exponencial en notación científica podría diferir de un compilador a otro.

15.7.6 Control de mayúsculas/minúsculas (uppercase)

El manipulador de flujo `uppercase` imprime una X o E mayúscula con valores hexadecimales enteros o con valores de punto flotante en notación científica, respectivamente (figura 15.19). El uso del manipulador de flujo `uppercase` también hace que todas las letras en un valor hexadecimal sean mayúsculas. De manera predeterminada, las letras para los valores hexadecimales y los exponentes en los valores de punto flotante en notación científica aparecen en minúscula. Para restablecer la opción de `uppercase`, hay que imprimir el manipulador de flujo `nouppercase`.

```

1 // Fig. 15.19: Fig15_19.cpp
2 // El manipulador de flujo uppercase.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::hex;

```

Figura 15.19 | El manipulador de flujo `uppercase`. (Parte 1 de 2).

```

7  using std::showbase;
8  using std::uppercase;
9
10 int main()
11 {
12     cout << "Impresion de letras mayusculas en exponentes de" << endl
13     << "notacion cientifica y valores hexadecimales:" << endl;
14
15     // usa std::uppercase para mostrar letras mayúsculas; usa std::hex y
16     // std::showbase para mostrar un valor hexadecimal y su base
17     cout << uppercase << 4.345e10 << endl
18     << hex << showbase << 123456789 << endl;
19     return 0;
20 } // fin de main

```

Impresion de letras mayusculas en exponentes de
notacion científica y valores hexadecimales:
4.345E+010
0X75BCD15

Figura 15.19 | El manipulador de flujo uppercase. (Parte 2 de 2).

15.7.7 Especificación de formato booleano (boolalpha)

C++ proporciona el tipo de datos `bool`, cuyos valores pueden ser `false` o `true`, como una alternativa preferida al antiguo estilo de usar 0 para indicar `false` y un valor distinto de cero para indicar `true`. Una variable `bool` se imprime como 0 o 1 de manera predeterminada. Sin embargo, podemos usar el manipulador de flujo `boolalpha` para establecer el flujo de salida de manera que muestre los valores `bool` como las cadenas "`true`" y "`false`". Use el manipulador de flujo `noboolalpha` para establecer el flujo de salida, de manera que muestre los valores `bool` como enteros (es decir, la opción predeterminada). El programa de la figura 15.20 demuestra estos manipuladores de flujos. En la línea 14 se muestra el valor `bool`, que en la línea 11 se establece en `true`, como un entero. En la línea 18 se utiliza el manipulador `boolalpha` para mostrar el valor `bool` como una cadena. Luego, en las líneas 21 y 22 se modifica el valor de `bool` y se usa el manipulador `noboolalpha`, por lo que en la línea 25 se puede mostrar el valor `bool` como un entero. En la línea 29 se utiliza el manipulador `boolalpha` para mostrar el valor `bool` como una cadena. Tanto `boolalpha` como `noboolalpha` son opciones "pegajosas".



Buena práctica de programación 15.1

Al mostrar los valores `bool` como `true` o `false`, en vez de mostrarlos como un valor distinto de cero o un 0, respectivamente, los resultados de los programas se hacen más legibles.

```

1 // Fig. 15.20: Fig15_20.cpp
2 // Demostración de los manipuladores de flujos boolalpha y noboolalpha.
3 #include <iostream>
4 using std::boolalpha;
5 using std::cout;
6 using std::endl;
7 using std::noboolalpha;
8
9 int main()
10 {
11     bool valorBooleano = true;
12
13     // muestra el valorBooleano verdadero predeterminado
14     cout << "valorBooleano es " << valorBooleano << endl;
15
16     // muestra el valorBooleano después de usar boolalpha
17     cout << "valorBooleano (después de usar boolalpha) es "
18         << boolalpha << valorBooleano << endl << endl;

```

Figura 15.20 | Los manipuladores de flujos `boolalpha` y `noboolalpha`. (Parte 1 de 2).

```

19
20     cout << "cambio de valorBooleano y uso de noboolalpha" << endl;
21     valorBooleano = false; // cambia valorBooleano
22     cout << noboolalpha << endl; // usa noboolalpha
23
24     // muestra el valorBooleano falso predeterminado después de usar noboolalpha
25     cout << "valorBooleano es " << valorBooleano << endl;
26
27     // muestra el valorBooleano después de usar boolalpha otra vez
28     cout << "valorBooleano (después de usar boolalpha) es "
29         << boolalpha << valorBooleano << endl;
30
31 } // fin de main

```

```

valorBooleano es 1
valorBooleano (después de usar boolalpha) es true
cambio de valorBooleano y uso de noboolalpha

valorBooleano es 0
valorBooleano (después de usar boolalpha) es false

```

Figura 15.20 | Los manipuladores de flujos `boolalpha` y `noboolalpha`. (Parte 2 de 2).

15.7.8 Establecer y restablecer el estado de formato mediante la función miembro `flags`

En la sección 15.7 hemos estado usando manipuladores de flujos para modificar las características de formato de la salida. Ahora veremos cómo devolver el formato de un flujo de salida a su estado predeterminado después de haber aplicado varias manipulaciones. La función miembro `flags` sin un argumento devuelve las opciones de formato actuales como un tipo de datos `fmtflags` (de la clase `ios_base`), el cual representa el **estado del formato**. La función miembro `flags` con un argumento `fmtflags` establece el estado del formato según lo especificado por el argumento y devuelve las opciones de estado anteriores. Las opciones iniciales del valor que devuelve `flags` podrían diferir de un sistema a otro. El programa de la figura 15.21 utiliza la función miembro `flags` para guardar el estado del formato original del flujo (línea 22), y después restaura las opciones de formato originales (línea 30).

```

1 // Fig. 15.21: Fig15_21.cpp
2 // Demostración de la función miembro flags.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::ios_base;
7 using std::oct;
8 using std::scientific;
9 using std::showbase;
10
11 int main()
12 {
13     int valorEntero = 1000;
14     double valorDouble = 0.0947628;
15
16     // muestra el valor de flags, los valores int y double (formato original)
17     cout << "El valor de la variable flags es: " << cout.flags()
18         << "\nImpresión de int y double en formato original:\n"
19         << valorEntero << '\t' << valorDouble << endl << endl;
20
21     // usa la función flags de cout para guardar el formato original
22     ios_base::fmtflags formatoOriginal = cout.flags();
23     cout << showbase << oct << scientific; // cambia el formato
24

```

Figura 15.21 | La función miembro `flags`. (Parte I de 2).

```

25 // muestra el valor de flags, los valores int y double (nuevo formato)
26 cout << "El valor de la variable flags es: " << cout.flags()
27     << "\nImpresion de int y double en un nuevo formato:\n"
28     << valorEntero << '\t' << valorDouble << endl << endl;
29
30 cout.flags( formatoOriginal ); // restaura el formato
31
32 // muestra el valor de flags, los valores int y double (formato original)
33 cout << "El valor restaurado de la variable flags es: "
34     << cout.flags()
35     << "\nImpresion de los valores en su formato original otra vez:\n"
36     << valorEntero << '\t' << valorDouble << endl;
37 return 0;
38 } // fin de main

```

```

El valor de la variable flags es: 513
Impresion de int y double en formato original:
1000 0.0947628

El valor de la variable flags es: 012011
Impresion de int y double en un nuevo formato:
01750 9.476280e-002

El valor restaurado de la variable flags es: 513
Impresion de los valores en su formato original otra vez:
1000 0.0947628

```

Figura 15.21 | La función miembro `flags`. (Parte 2 de 2).

15.8 Estados de error de los flujos

El estado de un flujo puede probarse a través de los bits en la clase `ios_base`. En un momento le mostraremos cómo probar estos bits, en el ejemplo de la figura 15.22.

El bit `eofbit` se establece para un flujo de entrada, al encontrar el fin de archivo. Un programa puede usar la función miembro `eof` para determinar si se ha encontrado el fin de archivo en un flujo después de un intento de extraer datos más allá del fin del flujo. La llamada

```
cin.eof()
```

devuelve `true` si se ha encontrado el fin de archivo en `cin`, y `false` en caso contrario.

El bit `failbit` se establece para un flujo cuando ocurre un error de formato en el mismo, como cuando el programa está introduciendo enteros y se encuentra un carácter que no sea dígito en el flujo de entrada. Cuando ocurre dicho error, los caracteres no se pierden. La función miembro `fail` reporta si ha fallado una operación con un flujo. Por lo general, es posible recuperarse de dichos errores.

El bit `badbit` se establece para un flujo cuando ocurre un error que produce la pérdida de datos. La función miembro `bad` reporta si falló una operación con un flujo. Por lo general, no es posible recuperarse de dichas fallas.

```

1 // Fig. 15.22: Fig15_22.cpp
2 // Prueba de los estados de error.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10     int valorEntero;
11
12     // muestra los resultados de las funciones de cin
13     cout << "Antes de una operacion de entrada incorrecta:"
14     << "\ncin.rdstate(): " << cin.rdstate()

```

Figura 15.22 | Prueba de los estados de error. (Parte 1 de 2).

```

15      << "\n    cin.eof(): " << cin.eof()
16      << "\n    cin.fail(): " << cin.fail()
17      << "\n    cin.bad(): " << cin.bad()
18      << "\n    cin.good(): " << cin.good()
19      << "\n\nEspera un entero, pero se introduce un carácter: ";
20
21  cin >> valorEntero; // escribe el valor tipo carácter
22  cout << endl;
23
24 // muestra los resultados de las funciones de cin después de una entrada incorrecta
25 cout << "Después de una operación de entrada incorrecta:"
26     << "\ncin.rdstate(): " << cin.rdstate()
27     << "\n    cin.eof(): " << cin.eof()
28     << "\n    cin.fail(): " << cin.fail()
29     << "\n    cin.bad(): " << cin.bad()
30     << "\n    cin.good(): " << cin.good() << endl << endl;
31
32  cin.clear(); // borra el flujo
33
34 // muestra los resultados de las funciones de cin después de borrar cin
35 cout << "Después de cin.clear()" << "\ncin.fail(): " << cin.fail()
36     << "\ncin.good(): " << cin.good() << endl;
37
38 } // fin de main

```

Antes de una operación de entrada incorrecta:

```

cin.rdstate(): 0
  cin.eof(): 0
  cin.fail(): 0
  cin.bad(): 0
  cin.good(): 1

```

Espera un entero, pero se introduce un carácter: A

Después de una operación de entrada incorrecta:

```

cin.rdstate(): 2
  cin.eof(): 0
  cin.fail(): 1
  cin.bad(): 0
  cin.good(): 0

```

Después de cin.clear()

```

cin.fail(): 0
cin.good(): 1

```

Figura 15.22 | Prueba de los estados de error. (Parte 2 de 2).

El bit **goodbit** se establece para un flujo si no se establece ninguno de los bits **eofbit**, **failbit** o **badbit** para el flujo.

La función miembro **good** devuelve **true** si todas las funciones **bad**, **fail** y **eof** devuelven **false**. Las operaciones de E/S deben realizarse sólo en flujos “buenos”.

La función miembro **rdstate** devuelve el estado de error del flujo. Por ejemplo, una llamada a **cout.rdstate** devolvería el estado del flujo, que entonces se podría evaluar mediante una instrucción **switch** que examine los bits **eofbit**, **badbit**, **failbit** y **goodbit**. La forma preferida de evaluar el estado de un flujo es mediante el uso de las funciones miembro **eof**, **bad**, **fail** y **good**; el uso de estas funciones no requiere que el programador esté familiarizado con los bits de estado específicos.

La función miembro **clear** se utiliza para restaurar el estado de un flujo a “bueno”, de manera que la E/S pueda proceder en ese flujo. El argumento predeterminado para **clear** es **goodbit**, por lo que la instrucción

```
cin.clear();
```

limpia **cin** y establece el bit **goodbit** para el flujo. La instrucción

```
cin.clear( ios::failbit )
```

establece el bit `failbit`. Tal vez el programador desee hacer esto al realizar operaciones de entrada en `cin` con un tipo definido por el usuario y toparse con un problema. El nombre `clear` podría parecer inapropiado en este contexto, pero es correcto.

El programa de la figura 15.22 demuestra las funciones miembro `rdstate`, `eof`, `fail`, `bad`, `good` y `clear`. [Nota: los valores actuales que se impriman podrían ser distintos de un compilador a otro].

La función miembro `operator!` de `basic_ios` devuelve `true` si se establece el bit `badbit`, si se establece el bit `failbit` o si se establecen ambos. La función miembro `operator void *` devuelve `false` (0) si se establece el bit `badbit`, si se establece el bit `failbit` o si se establecen ambos. Estas funciones son útiles en el procesamiento de archivos cuando se está evaluando una condición `true/false` bajo el control de una instrucción de selección o de repetición.

15.9 Enlazar un flujo de salida a un flujo de entrada

Por lo general, las aplicaciones interactivas implican un objeto `istream` para entrada y un objeto `ostream` para salida. Cuando aparece un mensaje de petición en la pantalla, el usuario responde introduciendo los datos apropiados. Obviamente, la petición necesita aparecer antes de que proceda la operación de entrada. Con el uso de búfer en la salida, las salidas aparecen sólo cuando se llena el búfer, cuando las salidas se vacían de manera explícita por el programa, o de manera automática al final del programa. C++ proporciona la función miembro `tie` para sincronizar (es decir, “enlazar entre sí”) la operación de un objeto `istream` y un objeto `ostream` para asegurar que los resultados aparezcan antes de sus entradas subsiguientes. La llamada

```
cin.tie( &cout );
```

enlaza a `cout` (un objeto `ostream`) con `cin` (un objeto `istream`). En realidad, esta llamada específica es redundante, debido a que C++ realiza esta operación de manera automática para crear un entorno de entrada/salida estándar para el usuario. Sin embargo, el usuario podría enlazar otros pares `istream/ostream` de manera explícita. Para desenlazar un flujo de entrada (`flujoEntrada`) de un flujo de salida, utilice la llamada

```
flujoEntrada.tie( 0 );
```

15.10 Repaso

En este capítulo sintetizamos la forma en que C++ realiza las operaciones de entrada/salida mediante el uso de flujos. El lector aprendió acerca de las clases y objetos de E/S de flujos, así como la jerarquía de clases de plantilla de E/S de flujos. Hablamos sobre las herramientas de salida con formato y sin formato de `ostream` que realizan las funciones `put` y `write`. Vimos ejemplos acerca del uso de las herramientas de entrada con formato y sin formato de `istream` realizadas por las funciones `eof`, `get`, `getline`, `peek`, `putback`, `ignore` y `read`. Después hablamos sobre los manipuladores de flujos y las funciones miembro que realizan tareas de formato: `dec`, `oct`, `hex` y `setbase` para mostrar enteros; `precision` y `setprecision` para controlar la precisión de punto flotante; y `width` y `setw` para establecer la anchura de campo. También aprendió acerca de los manipuladores `iostream` para formato adicional y acerca de las funciones miembro: `showpoint` para mostrar el punto decimal y ceros a la derecha; `left`, `right` e `internal` para la justificación; `fill` y `setfill` para llenar con caracteres; `scientific` y `fixed` para mostrar números de punto flotante en notación científica y fija; `uppercase` para el control de mayúsculas/minúsculas; `boolalpha` para especificar el formato booleano; y `flags` junto con `fmtflags` para restablecer el estado del formato.

En el siguiente capítulo presentaremos el manejo de excepciones, que permite a los programadores tratar con ciertos problemas que puedan ocurrir durante la ejecución de un programa. Demostraremos las técnicas básicas de manejo de excepciones que a menudo permiten a un programa continuar su ejecución, como si no se hubiera encontrado un problema. También presentaremos varias clases que proporciona la Biblioteca estándar de C++ para manejar excepciones.

Resumen

Sección 15.1 Introducción

- Las operaciones de E/S se realizan de una manera sensible al tipo de los datos.

Sección 15.2 Flujos

- En C++, las operaciones de E/S se realizan en flujos. Un flujo es una secuencia de bytes.
- Los mecanismos de E/S del sistema desplazan bytes de los dispositivos a la memoria y viceversa, de manera eficiente y confiable.

- C++ proporciona herramientas de E/S de “bajo nivel” y de “alto nivel”. Las herramientas de E/S de bajo nivel especifican que debe transferirse cierto número de bytes de dispositivo a memoria, o de memoria a dispositivo. La E/S de alto nivel se realiza con los bytes agrupados en unidades significativas tales como enteros, números de punto flotante, caracteres, cadenas y tipos definidos por el usuario.
- C++ proporciona operaciones de E/S con formato y sin formato. Las transferencias de E/S sin formato son rápidas, pero procesan datos puros que son difíciles de usar para las personas. La E/S con formato procesa los datos en unidades significativas, pero requiere un tiempo de procesamiento adicional que puede degradar el rendimiento de las transferencias de datos de alto volumen.
- El archivo de encabezado `<iostream>` declara todas las operaciones de E/S con flujos.
- El encabezado `<iomanip>` declara los manipuladores de flujos parametrizados.
- El encabezado `<fstream>` declara las operaciones de procesamiento de archivos.
- La plantilla `basic_istream` soporta las operaciones de entrada con flujos.
- La plantilla `basic_ostream` soporta las operaciones de salida con flujos.
- La plantilla `basic_iostream` soporta las operaciones de entrada y de salida con flujos.
- Las plantillas `basic_istream` y `basic_ostream` se derivan a través de la herencia simple de la plantilla `basic_ios`.
- La plantilla `basic_iostream` se deriva a través de la herencia múltiple, de la plantilla `basic_istream` y de la plantilla `basic_ostream`.
- El operador de desplazamiento a la izquierda (`<<`) se sobrecarga para designar las operaciones de salida con flujos, y se conoce como el operador de inserción de flujo.
- El operador de desplazamiento a la derecha (`>>`) se sobrecarga para designar las operaciones de entrada con flujos, y se conoce como el operador de extracción de flujo.
- El objeto `cin` de `istream` está enlazado al dispositivo de entrada estándar, que por lo general es el teclado.
- El objeto `cout` de `ostream` está enlazado al dispositivo de salida estándar, que por lo general es la pantalla.
- El objeto `cerr` de `ostream` está enlazado al dispositivo de error estándar. Las operaciones de salida con `cerr` no usan búfer; cada inserción en `cerr` aparece de inmediato.
- El compilador de C++ determina los tipos de datos de manera automática para las operaciones de entrada y de salida.

Sección 15.3 Salida de flujos

- Las direcciones se despliegan en formato hexadecimal de manera predeterminada.
- Para imprimir una dirección en una variable apuntador, convierte el apuntador a `void *`.
- Las funciones miembro `put` escriben solamente un carácter. Las llamadas a `put` se pueden hacer en cascada.

Sección 15.4 Entrada de flujos

- Los flujos de entrada se realizan con el operador de extracción de flujo `>>`. Este operador omite de manera automática los caracteres de espacio en blanco en el flujo de entrada.
- El operador `>>` devuelve `false` después de encontrar el fin de archivo en un flujo.
- La extracción de flujo hace que se establezca el bit `failbit` para los datos de entrada incorrectos, y que se establezca el bit `badbit` si la operación falla.
- Se puede introducir una serie de valores mediante el uso del operador de extracción de flujo en el encabezado de un ciclo `while`. La extracción devuelve 0 al encontrarse con el fin de archivo.
- La función miembro `get` sin argumentos introduce un carácter y devuelve ese carácter; se devuelve `EOF` al encontrar el fin de archivo en el flujo.
- La función miembro `get` con un argumento de referencia de carácter introduce el siguiente carácter del flujo de entrada y lo almacena en el argumento tipo carácter. Esta versión de `get` devuelve una referencia al objeto `istream` para el que se está invocando la función miembro `get`.
- La función miembro `get` con tres argumentos [un arreglo de caracteres, un límite de tamaño y un delimitador (con el valor predeterminado de nueva línea)] lee caracteres del flujo de entrada hasta un máximo de caracteres equivalente al límite – 1, o hasta que se lee el delimitador. La cadena de entrada se termina con un carácter nulo. El delimitador no se coloca en el arreglo de caracteres, pero permanece en el flujo de entrada.
- La función miembro `getline` opera como la función miembro `get` de tres argumentos. La función `getline` elimina el delimitador del flujo de entrada, pero no lo almacena en la cadena.
- La función miembro `ignore` omite el número especificado de caracteres (el valor predeterminado es 1) en el flujo de entrada; termina al encontrar el delimitador especificado (el delimitador predeterminado es `EOF`).
- La función miembro `putback` coloca el carácter anterior obtenido mediante `get` en un flujo, de vuelta a ese flujo.
- La función miembro `peek` devuelve el siguiente carácter de un flujo de entrada, pero no lo extrae (elimina) del flujo.
- C++ ofrece la E/S con seguridad de tipos. Si los operadores `<<` y `>>` procesan datos inesperados se establecen varios bits de error, que el usuario puede usar para determinar si una operación de E/S tuvo éxito o falló. Si el operador `<<` no se ha sobre cargado para un tipo definido por el usuario, se reporta un error de compilación.

Sección 15.5 E/S sin formato mediante el uso de `read`, `write` y `gcount`

- La E/S sin formato se lleva a cabo con las funciones miembro `read` y `write`. Éstas envían o reciben cierto número de bytes hacia/ desde la memoria, empezando en una dirección de memoria designada. Se reciben o envían como bytes puros sin formato.
- La función miembro `gcount` devuelve el número de caracteres introducidos por la operación `read` anterior en ese flujo.
- La función miembro `read` introduce un número especificado de caracteres en un arreglo de caracteres. El bit `failbit` se establece si se leen menos caracteres que el número especificado.

Sección 15.6 Introducción a los manipuladores de flujos

- Para cambiar la base en la que se imprimen los enteros, se utiliza el manipulador `hex` para establecer la base en hexadecimal (base 16), o el manipulador `oct` para establecer la base en octal (base 8). El manipulador `dec` se utiliza para restablecer la base a decimal. La base permanece igual hasta que se cambia de manera explícita.
- El manipulador de flujo parametrizado `setbase` también establece la base para las operaciones de salida con enteros. El manipulador `setbase` recibe un argumento entero de 10, 8 o 16 para establecer la base.
- La precisión de punto flotante se puede controlar mediante el manipulador de flujo `setprecision`, o mediante la función miembro `precision`. Ambos establecen la precisión para todas las operaciones subsiguientes de salida, hasta la siguiente llamada para establecer la precisión. La función miembro `precision` sin argumento devuelve el valor de precisión actual.
- Los manipuladores parametrizados requieren la inclusión del archivo de encabezado `<iomanip>`.
- La función miembro `width` establece la anchura de campo y devuelve la anchura anterior. Los valores de menor anchura que el campo se llenan con caracteres de relleno. La opción de anchura de campo se aplica sólo para la siguiente inserción o extracción; la anchura de campo se establece en 0 de manera implícita (los valores subsecuentes se imprimirán con la anchura necesaria). Los valores más anchos que un campo se imprimen en su totalidad. La función `width` sin argumento devuelve la opción de anchura actual. El manipulador `setw` también establece la anchura.
- Para las operaciones de entrada, el manipulador de flujo `setw` establece un tamaño máximo de cadena; si se introduce una cadena más grande, la línea más grande se divide en piezas que no sean mayores que el tamaño designado.
- Los programadores pueden crear sus propios manipuladores de flujos.

Sección 15.7 Estados de formato de flujos y manipuladores de flujos

- El manipulador de flujo `showpoint` obliga a que un número de punto flotante se imprima con un punto decimal, y con el número de dígitos significativos especificado por la precisión.
- Los manipuladores de flujos `left` y `right` hacen que los campos se justifiquen a la izquierda con caracteres de relleno a la derecha, o que se justifiquen a la derecha con caracteres de relleno a la izquierda.
- El manipulador de flujo `internal` indica que el signo de un número (o la base, cuando se utiliza el manipulador de flujo `showbase`) debe justificarse a la izquierda dentro de un campo, su magnitud debe justificarse a la derecha y los espacios intermedios deben rellenarse con el carácter de relleno.
- La función miembro `fill` especifica el carácter de relleno a usar con los manipuladores de flujos `left`, `right` e `internal` (el espacio es el valor predeterminado); se devuelve el carácter de relleno anterior. El manipulador de flujo `setfill` también establece el carácter de relleno.
- Los manipuladores de flujos `oct`, `hex` y `dec` especifican que los enteros se van a tratar como valores octales, hexadecimales o decimales, respectivamente. Los valores predeterminados de las operaciones de salida con enteros están en decimal si no se establece ninguno de estos bits; las extracciones de flujo procesan los datos en la forma en que éstos se suministran.
- El manipulador de flujo `showbase` obliga a que se imprima la base de un valor integral.
- El manipulador de flujo `scientific` se utiliza para imprimir un número de punto flotante en formato científico. El manipulador de flujo `fixed` se utiliza para imprimir un número de punto flotante con la precisión especificada mediante la función miembro `precision`.
- El manipulador de flujo `uppercase` imprime una X o E mayúscula para los enteros hexadecimales y los valores de punto flotante en notación científica, respectivamente. Los valores hexadecimales aparecen sólo en mayúsculas.
- La función miembro `flags` sin argumento devuelve el valor `long` de las opciones actuales del estado del formato. La función `flags` con un argumento `long` establece el estado del formato especificado mediante el argumento.

Sección 15.8 Estados de error de los flujos

- El estado de un flujo se puede evaluar mediante los bits en la clase `ios_base`.
- El bit `eofbit` se establece para un flujo de entrada, una vez que se encuentra el fin de archivo durante una operación de entrada. La función miembro `eof` reporta si se ha establecido el bit `eofbit`.
- El bit `failbit` de un flujo se establece cuando ocurre un error de formato. La función miembro `fail` reporta si ha fallado una operación de flujo; por lo general es posible recuperarse de dichos errores.
- El bit `badbit` de un flujo se establece cuando ocurre un error que provoca la pérdida de datos. La función miembro `bad` reporta si dicha operación de flujo falló. Dichas fallas graves por lo general son irrecuperables.
- La función miembro `good` devuelve verdadero si todas las funciones `bad`, `fail` y `eof` devuelven `false`. Las operaciones de E/S deben realizarse sólo en los flujos “buenos”.

- La función miembro `rdstate` devuelve el estado de error del flujo.
- La función miembro `clear` restaura el estado de un flujo a “bueno”, para que puedan continuar las operaciones de E/S.

Sección 15.9 Enlazar un flujo de salida a un flujo de entrada

- C++ proporciona la función miembro `tie` para sincronizar las operaciones con `istream` y `ostream` para asegurar que los resultados aparezcan antes de las entradas subsiguientes.

Terminología

0 a la izquierda (octal)	<code>iostream</code>
0x o 0X a la izquierda (hexadecimal)	<code>istream</code>
anchura de campo	<code>left</code> , manipulador de flujo
<code>bad</code> , función miembro de <code>basic_ios</code>	manipulador de flujo
<code>badbit</code>	manipulador de flujo parametrizado
<code>basic_fstream</code> , plantilla de clase	<code>noboolalpha</code> , manipulador de flujo
<code>basic_ifstream</code> , plantilla de clase	<code>noshowbase</code> , manipulador de flujo
<code>basic_ios</code> , plantilla de clase	<code>noshowpoint</code> , manipulador de flujo
<code>basic_iostream</code> , plantilla de clase	<code>noshowpos</code> , manipulador de flujo
<code>basic_ofstream</code> , plantilla de clase	<code>noskipws</code> , manipulador de flujo
<code>basic_ostream</code> , plantilla de clase	<code>nouppercase</code> , manipulador de flujo
<code>boolalpha</code> , manipulador de flujo	<code>oct</code> , manipulador de flujo
búfer de salida	<code>ofstream</code>
carácter de relleno	operador de extracción de flujo (<code>>></code>)
carácter de relleno predeterminado (espacio)	operador de inserción de flujo (<code><<</code>)
<code>clear</code> , función miembro de <code>basic_ios</code>	<code>operator void*</code> , función miembro de <code>basic_ios</code>
<code>dec</code> , manipulador de flujo	<code>operator!=</code> , función miembro de <code>basic_ios</code>
E/S con formato	<code>ostream</code>
E/S con seguridad de tipos	<code>peek</code> , función miembro de <code>basic_istream</code>
E/S sin formato	precisión predeterminada
entrada de flujos	<code>precision</code> , función miembro de <code>basic_istream</code>
<code>eof</code> , función miembro de <code>basic_ios</code>	<code>precision</code> , función miembro de <code>ios_base</code>
<code>eofbit</code>	<code>put</code> , función miembro de <code>basic_ostream</code>
estados de formato	<code>putback</code> , función miembro de <code>basic_istream</code>
<code>fail</code> , función miembro de <code>basic_ios</code>	<code>rdstate</code> , función miembro de <code>basic_ios</code>
<code>failbit</code>	<code>read</code> , función miembro de <code>basic_istream</code>
<code>fill</code> , función miembro de <code>basic_ios</code>	relleno de caracteres (padding)
fin de archivo	<code>right</code> , manipulador de flujo
<code>fixed</code> , manipulador de flujo	salida de flujos
<code>flags</code> , función miembro de <code>ios_base</code>	salida sin búfer
flujos predefinidos	<code>scientific</code> , manipulador de flujo
<code>fmtflags</code>	<code>setbase</code> , manipulador de flujo
<code>fstream</code>	<code>setfill</code> , manipulador de flujo
<code>gcount</code> , función miembro de <code>basic_istream</code>	<code>setprecision</code> , manipulador de flujo
<code>get</code> , función miembro de <code>basic_istream</code>	<code>setw</code> , manipulador de flujo
<code>getline</code> , función miembro de <code>basic_istream</code>	<code>showbase</code> , manipulador de flujo
<code>good</code> , función miembro de <code>basic_ios</code>	<code>showpoint</code> , manipulador de flujo
<code>hex</code> , manipulador de flujo	<code>showpos</code> , manipulador de flujo
<code>ifstream</code>	<code>skipws</code> , manipulador de flujo
<code>ignore</code> , función miembro de <code>basic_istream</code>	<code>tie</code> , función miembro de <code>basic_ios</code>
<code>internal</code> , manipulador de flujo	<code>typedef</code>
<code><iomanip></code> , archivo de encabezado	<code>uppercase</code> , manipulador de flujo
<code>ios_base</code> , clase	<code>width</code> , manipulador de flujo
	<code>write</code> , función miembro de <code>basic_ostream</code>

Ejercicios de autoevaluación

15.1 Complete los siguientes enunciados:

- a) La entrada/salida en C++ ocurre en forma de _____ de bytes.
- b) Los manipuladores de flujos que dan formato a la justificación son _____, _____ y _____.

- c) La función miembro _____ se puede utilizar para establecer y restablecer el estado del formato.
- d) La mayoría de los programas de C++ que realizan operaciones de E/S deben incluir el archivo de encabezado _____ que contiene las declaraciones requeridas para todas las operaciones de E/S de flujos.
- e) Al utilizar manipuladores parametrizados, se debe incluir el archivo de encabezado _____.
- f) El archivo de encabezado _____ contiene las declaraciones requeridas para el procesamiento de archivos.
- g) La función miembro _____ de `ostream` se utiliza para realizar operaciones de salida sin formato.
- h) Las operaciones de entrada están soportadas por la clase _____.
- i) Las operaciones de salida del flujo de error estándar se dirigen a los objetos flujo _____ o _____.
- j) Las operaciones de salida están soportadas por la clase _____.
- k) El símbolo para el operador de inserción de flujo es _____.
- l) Los cuatro objetos que corresponden a los dispositivos estándar en el sistema son _____, _____, _____ y _____.
- m) El símbolo para el operador de extracción de flujo es _____.
- n) Los manipuladores de flujos _____, _____ y _____ especifican que los enteros se deben mostrar en los formatos octal, hexadecimal y decimal, respectivamente.
- o) El manipulador de flujo _____ hace que los números positivos se muestren con un signo positivo.

15.2

Indique si cada uno de los siguientes enunciados es *verdadero* o *falso*. En caso de ser *falso*, explique por qué.

- a) La función miembro `flags` de un flujo con un argumento `long` establece la variable de estado `flags` con su argumento y devuelve su valor anterior.
- b) El operador de inserción de flujo `<<` y el operador de extracción de flujo `>>` se sobrecargan para manejar todos los tipos de datos estándar [incluyendo cadenas y direcciones de memoria (sólo el de inserción de flujo)] y todos los tipos de datos definidos por el usuario.
- c) La función miembro `flags` de un flujo sin argumentos restablece el estado de formato del flujo.
- d) El operador de extracción de flujo `>>` se puede sobrecargar con una función operador que reciba como argumentos una referencia `istream` y una referencia a un tipo definido por el usuario, y devuelva una referencia `istream`.
- e) El operador de inserción de flujo `<<` se puede sobrecargar con una función operador que reciba como argumentos una referencia `istream` y una referencia a un tipo definido por el usuario, y devuelva una referencia `istream`.
- f) La entrada con el operador de extracción de flujo `>>` siempre omite los caracteres de espacio en blanco a la izquierda en el flujo de entrada, de manera predeterminada.
- g) La función miembro `rdstate` de un flujo devuelve el estado actual del flujo.
- h) Por lo general, el flujo `cout` está conectado a la pantalla.
- i) La función miembro `good` de un flujo devuelve `true` si todas las funciones miembro `bad`, `fail` y `eof` devuelven `false`.
- j) Por lo general, el flujo `cin` está conectado a la pantalla.
- k) Si ocurre un error irrecuperable durante una operación de un flujo, la función miembro `bad` devolverá `true`.
- l) La salida a `cerr` no usa búfer y la salida a `clog` tiene búfer.
- m) El manipulador de flujo `showpoint` obliga a que los valores de punto flotante se impriman con los seis dígitos pre-determinados de precisión, a menos que se haya modificado el valor de precisión, en cuyo caso los valores de punto flotante se imprimen con la precisión especificada.
- n) La función miembro `put` de `ostream` imprime el número especificado de caracteres.
- o) Los manipuladores de flujos `dec`, `oct` y `hex` sólo afectan a la siguiente operación de entrada con enteros.
- p) De manera predeterminada, las direcciones de memoria se muestran como enteros `long`.

15.3

Para cada uno de los siguientes enunciados, escriba una sola instrucción que realice la tarea indicada.

- a) Imprimir la cadena "Escriba su nombre: ".
- b) Usar un manipulador de flujo que haga que el exponente en la notación científica y las letras en los valores hexadecimales se impriman en mayúsculas.
- c) Imprimir la dirección de la variable `miString` de tipo `char *`.
- d) Usar un manipulador de flujo para asegurar que los valores de punto flotante se impriman en notación científica.
- e) Imprimir la dirección en la variable `enteroPtr` de tipo `int *`.
- f) Usar un manipulador de flujo de tal forma que, cuando se impriman valores enteros, se muestre la base entera para los valores octales y hexadecimales.
- g) Imprimir el valor al que apunta `floatPtr` de tipo `float *`.
- h) Usar una función miembro de flujo para establecer el carácter de relleno en '*' e imprimir en anchuras de campo mayores que los valores que se van a imprimir. Repita esta instrucción con un manipulador de flujo.
- i) Imprimir los caracteres '0' y 'K' en una instrucción con la función `put` de `ostream`.

- j) Obtener el valor del siguiente carácter a introducir, sin extraerlo del flujo.
- k) Introducir un solo carácter en la variable `valorChar` de tipo `char`, usando la función miembro `get` de `istream` en dos maneras distintas.
- l) Introducir y descartar los siguientes seis caracteres en el flujo de entrada.
- m) Usar la función miembro `read` de `istream` para introducir 50 caracteres en el arreglo `línea` tipo `char`.
- n) Leer 10 caracteres y colocarlos en el arreglo de caracteres `nombre`. Dejar de leer caracteres al encontrar el delimitador `'. '`. No elimine el delimitador del flujo de entrada. Escriba otra instrucción que realice esta tarea y elimine el delimitador de la entrada.
- o) Usar la función miembro `gcount` de `istream` para determinar el número de caracteres introducidos en el arreglo de caracteres `línea` mediante la última llamada a la función miembro `read` de `istream`, e imprimir ese número de caracteres, usando la función miembro `write` de `ostream`.
- p) Imprimir 124, 18.376, 'Z', 1000000 y "Cadena" separados por espacios.
- q) Imprimir la opción de precisión actual, usando una función miembro del objeto `cout`.
- r) Introducir un valor entero en la variable `int` llamada `meses`, y un valor de punto flotante en la variable `float` llamada `tasaPorcentaje`.
- s) Imprimir 1.92, 1.925 y 1.9258 separados por tabuladores y con 3 dígitos de precisión, usando un manipulador de flujo.
- t) Imprimir el entero 100 en octal, hexadecimal y decimal, usando manipuladores de flujos y separado por tabuladores.
- u) Imprimir el entero 100 en decimal, octal y hexadecimal separado por tabuladores, usando un manipulador de flujo para cambiar la base.
- v) Imprimir 1234 justificado a la derecha en un campo de 10 dígitos.
- w) Leer los caracteres en el arreglo de caracteres `línea` hasta encontrar el carácter 'z', hasta un límite de 20 caracteres (incluyendo un carácter nulo de terminación). No extraiga el carácter delimitador del flujo.
- x) Usar las variables enteras `x` y `y` para especificar la anchura de campo y precisión utilizadas para mostrar el valor `double` 87.4573, y mostrar el valor.

15.4 Identifique el error en cada una de las siguientes instrucciones y explique cómo corregirlo.

- a) `cout << "El valor de x <= y es: " << x <= y;`
 - b) La siguiente instrucción debe imprimir el valor entero de 'c'.
- ```
cout << 'c';
```
- c) `cout << ""Una cadena entre comillas"";`

**15.5** Para cada uno de los siguientes incisos, muestre los resultados.

- a) `cout << "12345" << endl;`  
`cout.width( 5 );`  
`cout.fill( '*' );`  
`cout << 123 << endl << 123;`
- b) `cout << setw( 10 ) << setfill( '$' ) << 10000;`
- c) `cout << setw( 8 ) << setprecision( 3 ) << 1024.987654;`
- d) `cout << showbase << oct << 99 << endl << hex << 99;`
- e) `cout << 100000 << endl << showpos << 100000;`
- f) `cout << setw( 10 ) << setprecision( 2 ) << scientific << 444.93738;`

## Respuestas a los ejercicios de autoevaluación

**15.1** a) flujos. b) `left`, `right` e `internal`. c) `flags`. d) `<iostream>`. e) `<iomanip>`. f) `<fstream>`. g) `write`. h) `istream`. i) `cerr` o `clog`. j) `ostream`. k) `<<`. l) `cin`, `cout`, `cerr` y `clog`. m) `>>`. n) `oct`, `hex` y `dec`. o) `showpos`.

**15.2** a) Falso. La función miembro `flags` de un flujo con un argumento `fmtflags` establece la variable de estado `flags` con su argumento y devuelve las opciones de estado anteriores. b) Falso. Los operadores de inserción de flujo y de extracción de flujo no se sobrecargan para todos los tipos definidos por el usuario. El programador de una clase debe proporcionar de manera específica las funciones operador sobrecargadas, para sobrecargar los operadores de flujo y usarlos con cada tipo definido por el usuario. c) Falso. La función miembro `flags` de un flujo sin argumentos devuelve las opciones de formato actuales como un tipo de datos `fmtflags`, el cual representa el estado del formato. d) Verdadero. e) Falso. Para sobrecargar el operador de inserción de flujo `<<`, la función operador sobrecargada debe recibir como argumentos una referencia `ostream` y una referencia a un tipo definido por el usuario, y devolver una referencia `ostream`. f) Verdadero. g) Verdadero. h) Verdadero. i) Verdadero.

j) Falso. El flujo `cin` está conectado a la entrada estándar de la computadora, que por lo general es el teclado. k) Verdadero.  
l) Verdadero. m) Verdadero. n) Falso. La función miembro `put` de `ostream` imprime su único argumento tipo carácter.  
o) Falso. Los manipuladores de flujos `dec`, `oct` y `hex` establecen el estado del formato de salida para los enteros con la base especificada, hasta que se cambia la base de nuevo o el programa termina. p) Falso. Las direcciones de memoria se muestran en formato hexadecimal de manera predeterminada. Para mostrar las direcciones como enteros `long`, la dirección se debe convertir en un valor `long`.

- 15.3**
- a) `cout << "Escriba su nombre: ";`
  - b) `cout << uppercase;`
  - c) `cout << static_cast< void * >( miString );`
  - d) `cout << scientific;`
  - e) `cout << enteroPtr;`
  - f) `cout << showbase;`
  - g) `cout << *floatPtr;`
  - h) `cout.fill( '*' );`  
`cout << setfill( '*' );`
  - i) `cout.put( '0' ). put( 'K' );`
  - j) `cin.peek();`
  - k) `valorChar = cin.get();`  
`cin.get ( valorChar );`
  - l) `cin.ignore( 6 );`
  - m) `cin.read( linea, 50 );`
  - n) `cin.get( nombre, 10, '.' );`  
`cin.getline( nombre, 10, '.' );`
  - o) `cout.write( linea, cin.gcount() );`
  - p) `cout << 124 << ' ' << 18.376 << ' ' << "Z" << 1000000 << "Cadena";`
  - q) `cout << cout.precision();`
  - r) `cin >> meses >> tasaPorcentaje;`
  - s) `cout << setprecision( 3 ) << 1.92 << '\t' << 1.925 << '\t' << 1.9258;`
  - t) `cout << oct << 100 << '\t' << hex << 100 << '\t' << dec << 100;`
  - u) `cout << 100 << '\t' << setbase( 8 ) << 100 << '\t' << setbase( 16 ) = << 100;`
  - v) `cout << setw( 10 ) << 1234;`
  - w) `cin.get( linea, 20, 'z' );`
  - x) `cout << setw( x ) << setprecision( y ) << 87.4573;`

- 15.4**
- a) *Error:* la precedencia del operador `<<` es mayor que la de `=`, lo cual hace que la instrucción se evalúe en forma incorrecta y también produce un error de compilación.

*Corrección:* coloque paréntesis alrededor de la expresión `x <= y`.

- b) *Error:* en C++ los caracteres no se tratan como enteros pequeños, como en C.

*Corrección:* imprimir el valor numérico para un carácter en el conjunto de caracteres de la computadora, el carácter debe convertirse en un valor entero, como en la siguiente instrucción:

```
cout << static_cast< int >('c');
```

- c) *Error:* los caracteres de comillas no se pueden imprimir en una cadena, a menos que se utilice una secuencia de escape.

*Corrección:* imprima la cadena en una de las siguientes formas:

```
cout << "\"Una cadena entre comillas\"";
```

- 15.5**
- a) `12345`  
`**123`  
`123`
  - b) `$$$$$10000`
  - c) `1024.988`
  - d) `0143`  
`0x63`
  - e) `100000`  
`+100000`
  - f) `4.45e+002`

## Ejercicios

**15.6** Escriba una instrucción para cada uno de los siguientes incisos:

- Imprimir el entero 40000 justificado a la izquierda en un campo de 15 dígitos.
- Leer una cadena y colocarla en la variable tipo arreglo de caracteres llamada `estado`.
- Imprimir 200 con y sin un signo.
- Imprimir el valor decimal 100 en formato hexadecimal con el prefijo `0x`.
- Leer caracteres en el arreglo `arregloChar` hasta encontrar el carácter '`'p'`', hasta un límite de 10 caracteres (incluyendo el carácter nulo de terminación). Extraiga el delimitador del flujo de entrada, y descártelo.
- Imprimir 1.234 en un campo de 9 dígitos con ceros a la izquierda.

**15.7** Escriba un programa para evaluar la introducción de valores enteros en los formatos decimal, octal y hexadecimal. Imprima cada entero leído por el programa en los tres formatos. Evalúe el programa con los siguientes datos de entrada: 10, 010, `0x10`.

**15.8** Escriba un programa que imprima valores de apuntadores, usando conversiones de tipos a todos los tipos de datos enteros. ¿Cuáles imprimen valores extraños? ¿Cuáles producen errores?

**15.9** Escriba un programa para evaluar los resultados de imprimir el valor entero 12345 y el valor de punto flotante 1.2345 en campos de diversos tamaños. ¿Qué ocurre cuando se imprimen los valores en campos que contengan menos dígitos que los valores?

**15.10** Escriba un programa que imprima el valor 100.453627 redondeado a la unidad, décima, centésima, milésima o diezmilésima más cercanas.

**15.11** Escriba un programa que reciba una cadena del teclado y determine la longitud de la cadena. Imprima la cadena en una anchura de campo que sea el doble de la longitud de la cadena.

**15.12** Escriba un programa que convierta temperaturas Fahrenheit enteras, de 0 a 212 grados, a temperaturas en grados Centígrados de punto flotante, con 3 dígitos de precisión. Utilice la siguiente fórmula:

$$\text{centigrados} = 5.0 / 9.0 * (\text{fahrenheit} - 32);$$

para realizar el cálculo. Los resultados deben imprimirse en dos columnas justificadas a la izquierda, y las temperaturas en grados Centígrados se les debe anteponer un signo tanto para los valores positivos como negativos.

**15.13** En ciertos lenguajes de programación, las cadenas se introducen entre comillas sencillas o dobles. Escriba un programa que lea las tres cadenas `"suzy"`, `'suzy'` y `'suzy'`. ¿Se ignoran las comillas simples y dobles, o se leen como parte de la cadena?

**15.4** En la figura 11.5 se sobrecargaron los operadores de extracción de flujo y de inserción de flujo para las operaciones de entrada y salida con objetos de la clase `NumeroTelefonico`. Vuelva a escribir el operador de extracción de flujo para realizar la siguiente comprobación de errores en la entrada. Se tendrá que volver a implementar la función `operator>>`.

- Introduzca el número telefónico completo en un arreglo. Pruebe que se haya introducido el número apropiado de caracteres. Debe haber un total de 14 caracteres leído para un número telefónico de la forma (800) 555-1212. Use la función miembro `clear` de `ios_base` para establecer el bit `failbit` para la entrada inapropiada.
- El código de área y el intercambio no empiezan con 0 o 1. Pruebe el primer dígito de las porciones del código de área y del intercambio del número telefónico para asegurar que ninguna empiece con 0 o 1. Use la función miembro `clear` de `ios_base` para establecer el bit `failbit` para una entrada incorrecta.
- El dígito intermedio de un código de área solía limitarse a 0 o 1 (aunque esto ha cambiado recientemente). Pruebe el dígito intermedio para un valor de 0 o 1. Use la función miembro `clear` de `ios_base` para establecer el bit `failbit` para una entrada incorrecta. Si ninguna de las operaciones anteriores provocan que se establezca el bit `failbit` para una entrada incorrecta, copie las tres partes del número telefónico en los miembros `codigoArea`, `intercambio` y `línea` del objeto `NumeroTelefonico`. Si se estableció `failbit` en la entrada, haga que el programa imprima un mensaje de error y termine, en vez de imprimir el número telefónico.

**15.15** Escriba un programa que realice cada una de las siguientes acciones:

- Crear una clase `Punto` definida por el usuario que contenga los datos miembro privados enteros `coordenadaX` y `coordenadaY`, y declarar las funciones de los operadores sobrecargados de inserción de flujo y extracción de flujo como funciones `friend` de la clase.
- Definir las funciones de los operadores de inserción de flujo y extracción de flujo. La función del operador de extracción de flujo debe determinar si los datos introducidos son válidos y, en caso contrario, debe establecer el bit `failbit` para indicar una entrada incorrecta. El operador de inserción de flujo no debe mostrar el punto después de encontrar un error en la entrada.

- c) Escriba una función `main` que pruebe la entrada y salida de la clase `Punto` definida por el usuario, usando los operadores sobrecargados de extracción de flujo y de inserción de flujo.
- 15.16** Escriba un programa que realice cada una de las siguientes acciones:
- Crear una clase `Complejo` definida por el usuario que contenga los datos miembro enteros `real` e `imaginario`, y que declare las funciones de los operadores sobrecargados de inserción de flujo y extracción de flujo como funciones `friend` de la clase.
  - Definir las funciones de los operadores de inserción de flujo y de extracción de flujo. La función del operador de extracción de flujo debe determinar si los datos introducidos son válidos y, en caso contrario, debe establecer el bit `failbit` para indicar una entrada incorrecta. La entrada deberá establecerse de manera que sea de la siguiente forma:

`3 + 8i`

- Los valores pueden ser negativos o positivos, y es posible que no se proporcione uno de los dos valores, en cuyo caso los datos miembro apropiados se deberán establecer en 0. El operador de inserción de flujo no deberá mostrar el punto si ocurrió un error de entrada. Para los valores imaginarios negativos, debe imprimirse un signo negativo en vez de un signo positivo.
  - Escriba una función `main` que pruebe la entrada y salida de la clase `Complejo` definida por el usuario, usando los operadores sobrecargados de extracción de flujo y de inserción de flujo.
- 15.17** Escriba un programa que utilice una instrucción `for` para imprimir una tabla de valores ASCII para los caracteres en el conjunto de caracteres ASCII de 33 a 126. El programa debe imprimir el valor decimal, valor octal, valor hexadecimal y valor de carácter para cada carácter. Use los manipuladores de flujos `dec`, `oct` y `hex` para imprimir los valores enteros.
- 15.18** Escriba un programa para mostrar que la función miembro `getline` y la función miembro `get` con tres argumentos de `istream` terminan la cadena de entrada con un carácter nulo de terminación de cadenas. Además, muestre que `get` deja el carácter delimitador en el flujo de entrada, mientras que `getline` extrae el carácter delimitador y lo descarta. ¿Qué ocurre con los caracteres en el flujo que no se leen?



# Manejo de excepciones

*Es cuestión de sentido común tomar un método y probarlo. Si falla, admítalo francamente y pruebe otro. Pero sobre todo, inténtelo.*

—Franklin Delano Roosevelt

*¡Oh! Arroja la peor parte de ello, y vive en forma más pura con la otra mitad.*

—William Shakespeare

*Si están corriendo y no saben hacia dónde se dirigen tengo que salir de alguna parte y atraparlos.*

—Jerome David Salinger

*¡Oh, infinita virtud! ¿Cómo sonríes desde la trampa más grande del mundo sin estar atrapada?*

—William Shakespeare

*Nunca olvido un rostro, pero en su caso haré una excepción.*

—Groucho Marx

## OBJETIVOS

En este capítulo aprenderá a :

- Distinguir las excepciones y cuándo utilizarlas.
- Usar `try`, `catch` y `throw` para detectar, manejar e indicar excepciones, respectivamente.
- Procesar excepciones no atrapadas e inesperadas.
- Declarar nuevas clases de excepciones.
- Comprender cómo la limpieza de la pila permite que las excepciones que no se atrapan en un alcance, se atrapen en otro alcance.
- Manejar las fallas de `new`.
- Usar `auto_ptr` para evitar las fugas de memoria.
- Comprender la jerarquía de excepciones estándar.

**Plan general**

- 16.1** Introducción
- 16.2** Generalidades acerca del manejo de excepciones
- 16.3** Ejemplo: manejo de un intento de dividir entre cero
- 16.4** Cuándo utilizar el manejo de excepciones
- 16.5** Volver a lanzar una excepción
- 16.6** Especificaciones de excepciones
- 16.7** Procesamiento de excepciones inesperadas
- 16.8** Limpieza de la pila
- 16.9** Constructores, destructores y manejo de excepciones
- 16.10** Excepciones y herencia
- 16.11** Procesamiento de las fallas de new
- 16.12** La clase `auto_ptr` y la asignación dinámica de memoria
- 16.13** Jerarquía de excepciones de la Biblioteca estándar
- 16.14** Otras técnicas para manejar errores
- 16.15** Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)

## 16.1 Introducción

En este capítulo presentaremos el **manejo de excepciones**. Una **excepción** es la indicación de un problema que ocurre durante la ejecución de un programa. El nombre “excepción” implica que el problema ocurre con poca frecuencia; si la “regla” es que una instrucción generalmente se ejecuta en forma correcta, entonces la “excepción a la regla” es cuando ocurre un problema. El manejo de excepciones permite a los programadores crear aplicaciones que puedan resolver (o manejar) las excepciones. En muchos casos, el manejo de una excepción permite que un programa continúe su ejecución como si no se hubiera encontrado un problema. Un problema más grave podría evitar que un programa continuara su ejecución normal, en vez de requerir al programa que notifique al usuario sobre el problema antes de terminar de una manera controlada. Las características que presentamos en este capítulo permiten a los programadores escribir **programas tolerantes a fallas y robustos**, que puedan tratar con los problemas que puedan surgir sin dejar de ejecutarse, o que terminen de una manera no dañina. El estilo y los detalles sobre el manejo de excepciones en C++ se basan, en parte, en el trabajo que Andrew Koenig y Bjarne Stroustrup presentaron en su artículo “Exception Handling for C++ (versión revisada).”<sup>1</sup>



### Tip para prevenir errores 16.1

*El manejo de excepciones ayuda a mejorar la tolerancia a fallas de un programa.*



### Observación de Ingeniería de Software 16.1

*El manejo de excepciones proporciona un mecanismo estándar para procesar los errores. Esto es especialmente importante cuando se trabaja en un proyecto con un equipo extenso de programadores.*

El capítulo empieza con una descripción general de los conceptos relacionados con el manejo de excepciones, y posteriormente se demuestran las técnicas básicas para el manejo de excepciones. Mostraremos estas técnicas mediante un ejemplo que señala cómo manejar una excepción que ocurre cuando una función intenta realizar una división entre cero. Después hablaremos sobre ciertas cuestiones adicionales sobre el manejo de excepciones, como la forma en que se deben manejar las excepciones que ocurren en un constructor o destructor, y cómo manejar las excepciones que ocurren si el operador `new` falla al asignar memoria para un objeto. Concluiremos este capítulo presentando varias clases que proporciona la Biblioteca estándar de C++ para manejar excepciones.

1. Koenig, A. y B. Stroustrup. “Exception Handling for C++ (versión revisada)”, *Proceedings of the Usenix C++ Conference*, pp. 149-176, San Francisco, abril de 1990.

## 16.2 Generalidades acerca del manejo de excepciones

Con frecuencia, los programas evalúan condiciones que determinan cómo debe proceder la ejecución del programa. Consideré el siguiente seudocódigo:

```

Realizar una tarea
Si la tarea anterior no se ejecutó correctamente
 Realizar el procesamiento de los errores
Realizar la siguiente tarea
Si la tarea anterior no se ejecutó correctamente
 Realizar el procesamiento de los errores
...

```

En este seudocódigo, empezaremos por realizar una tarea; después evaluaremos si se ejecutó en forma correcta. Si no lo hizo, realizamos el procesamiento de los errores. De otra manera, continuamos con la siguiente tarea. Aunque esta forma de manejo de excepciones funciona, al entremezclar la lógica del programa con la lógica del manejo de errores el programa podría ser difícil de leer, modificar, mantener y depurar, especialmente, en aplicaciones extensas.



### Tip de rendimiento 16.1

*Si los problemas potenciales ocurren con poca frecuencia, al entremezclar la lógica del programa y la lógica del manejo de errores se puede degradar el rendimiento del programa, ya que éste debe realizar pruebas (tal vez con frecuencia) para determinar si la tarea se ejecutó en forma correcta, y si se puede llevar a cabo la siguiente tarea.*

El manejo de excepciones permite al programador remover el código para manejo de errores de la “línea principal” de ejecución del programa, lo cual mejora la claridad y capacidad de modificación del mismo. Los programadores pueden optar por manejar cualquier excepción que deseen; todas las excepciones, todas las excepciones de cierto tipo o todas las excepciones de un grupo de tipos relacionados (por ejemplo, los tipos de excepciones que pertenecen a una jerarquía de herencia). Esta flexibilidad reduce la probabilidad de que los errores se pasen por alto y, por consecuencia, hace que los programas sean más robustos.

Con lenguajes de programación que no soportan el manejo de excepciones, los programadores a menudo retrasan la escritura de código de procesamiento de errores, o algunas veces olvidan incluirlo. Esto hace que los productos de software sean menos robustos. C++ permite al programador tratar con el manejo de excepciones fácilmente, desde el comienzo de un proyecto.

## 16.3 Ejemplo: manejo de un intento de dividir entre cero

Vamos a considerar un ejemplo simple de manejo de excepciones (figuras 16.1 y 16.2). El propósito de este ejemplo es mostrar cómo evitar un problema aritmético común: la división entre cero. En C++, la división entre cero mediante el uso de aritmética de enteros por lo general hace que un programa termine en forma prematura. En la aritmética de punto flotante, ciertas implementaciones de C++ permiten la división entre cero, en cuyo caso el resultado de infinito positivo o negativo se muestra como `INF` o `-INF`, respectivamente.

En este ejemplo definimos una función llamada `cociente`, la cual recibe dos enteros introducidos por el usuario, y divide su primer parámetro `int` entre su segundo parámetro `int`. Antes de realizar la división, la función convierte el valor del primer parámetro entero al tipo `double`. Después, el valor del segundo parámetro `int` se promueve al tipo `double` para el cálculo. Así, la función `cociente` en realidad realiza la división utilizando dos valores `double` y devuelve un resultado `double`.

Aunque la división entre cero está permitida en la aritmética de punto flotante, para el propósito de este ejemplo trataremos un intento de división entre cero como un error. Así, la función `cociente` evalúa su segundo parámetro para asegurar que no sea cero antes de permitir que continúe la división. Si el segundo parámetro es cero, la función utiliza una excepción para indicar a la función que hizo la llamada que ocurrió un problema. Así, la función que hizo la llamada (`main` en este ejemplo) puede procesar la excepción y permitir que el usuario escriba dos nuevos valores, antes de llamar a la función `cociente` de nuevo. De esta forma, el programa puede seguir ejecutándose aun después de que se introduzca un valor inapropiado, con lo cual el programa se vuelve más robusto.

El ejemplo consiste en dos archivos. `ExcepcionDivisionEntreCero.h` (figura 16.1) define una clase de excepción que representa el tipo del problema que podría ocurrir en el ejemplo, y `fig16_02.cpp` (figura 16.2) define la función `cociente` y la función `main` que llama a la clase de excepción. La función `main` contiene el código que demuestra el manejo de excepciones.

### *Definición de una clase de excepción para representar el tipo del problema que podría ocurrir*

En la figura 16.1 se define la clase ExcepcionDivisionEntreCero como una clase derivada de la clase `runtime_error` (definida en el archivo de encabezado `<stdexcept>`). La clase `runtime_error`, una clase derivada de la clase `exception` de la Biblioteca estándar (definida en el archivo de encabezado `<exception>`), es la clase base estándar de C++ para representar errores en tiempo de ejecución. La clase `exception` es la clase base estándar de C++ para todas las excepciones. (En la sección 16.13 hablaremos sobre la clase `exception` y sus clases derivadas con detalle). Una clase de excepción típica que se deriva de la clase `runtime_error` define sólo un constructor (líneas 12 y 13) que pasa una cadena de mensaje de error al constructor de la clase base `runtime_error`. Cada clase de excepción que se deriva en forma directa o indirecta de `exception` contiene la función virtual `what`, la cual devuelve el mensaje de error de un objeto excepción. No es obligatorio derivar una clase de excepción personalizada (como `ExcepcionDivisionEntreCero`) de las clases de excepciones estándar que proporciona C++. Sin embargo, esto permite a los programadores utilizar la función virtual `what` para obtener un mensaje de error apropiado. En la figura 16.2 utilizamos un objeto de esta clase `ExcepcionDivisionEntreCero` para indicar cuando se hace un intento de dividir entre cero.

```

1 // Fig. 16.1: ExcepcionDivisionEntreCero.h
2 // Definición de la clase ExcepcionDivisionEntreCero.
3 #include <stdexcept> // el archivo de encabezado stdexcept contiene runtime_error
4 using std::runtime_error; // clase runtime_error de la biblioteca estándar de C++
5
6 // los objetos ExcepcionDivisionEntreCero deben lanzarse por las funciones
7 // al detectar las excepciones de división entre cero
8 class ExcepcionDivisionEntreCero : public runtime_error
9 {
10 public:
11 // el constructor especifica el mensaje de error predeterminado
12 ExcepcionDivisionEntreCero()
13 : runtime_error("intento de dividir entre cero") {}
14 } // fin de la clase ExcepcionDivisionEntreCero

```

Figura 16.1 | Definición de la clase ExcepcionDivisionEntreCero.

```

1 // Fig. 16.2: Fig16_02.cpp
2 // Un ejemplo simple para manejar excepciones, que comprueba
3 // las excepciones de división entre cero.
4 #include <iostream>
5 using std::cin;
6 using std::cout;
7 using std::endl;
8
9 #include "ExcepcionDivisionEntreCero.h" // clase ExcepcionDivisionEntreCero
10
11 // realiza la división y lanza un objeto ExcepcionDivisionEntreCero si
12 // ocurre una excepción de división entre cero
13 double cociente(int numerador, int denominador)
14 {
15 // lanza ExcepcionDivisionEntreCero si intenta dividir entre cero
16 if (denominador == 0)
17 throw ExcepcionDivisionEntreCero(); // termina la función
18
19 // devuelve el resultado de la división
20 return static_cast< double >(numerador) / denominador;
21 } // fin de la función cociente
22
23 int main()
24 {
25 int numero1; // numerador especificado por el usuario
26 int numero2; // denominador especificado por el usuario
27 double resultado; // resultado de la división

```

Figura 16.2 | Ejemplo de manejo de excepciones que lanza excepciones al tratar de dividir entre cero. (Parte I de 2).

```

28 cout << "Escriba dos enteros (fin de archivo para terminar): ";
29
30 // permite al usuario introducir dos enteros para la división
31 while (cin >> numero1 >> numero2)
32 {
33 // el bloque try contiene código que podría lanzar una excepción
34 // y código que no se debe ejecutar si ocurre una excepción
35 try
36 {
37 resultado = cociente(numero1, numero2);
38 cout << "El cociente es: " << resultado << endl;
39 } // fin de try
40
41 // el manejador de excepciones maneja una excepción que se divide por cero
42 catch (ExcepcionDivisionEntreCero ÷ByZeroException)
43 {
44 cout << "Ocurrio una excepcion: "
45 << divideByZeroException.what() << endl;
46 } // fin de catch
47
48 cout << "\nEscriba dos enteros (fin de archivo para terminar): ";
49 } // fin de while
50
51 cout << endl;
52 return 0; // termina en forma normal
53 } // fin de main

```

Escriba dos enteros (fin de archivo para terminar): 100 7  
 El cociente es: 14.2857

Escriba dos enteros (fin de archivo para terminar): 100 0  
 Ocurrio una excepcion: intento de dividir entre cero

Escriba dos enteros (fin de archivo para terminar): ^Z

**Figura 16.2** | Ejemplo de manejo de excepciones que lanza excepciones al tratar de dividir entre cero. (Parte 2 de 2).

### Demostración del manejo de excepciones

El programa de la figura 16.2 utiliza el manejo de excepciones para envolver código que podría lanzar una excepción de “división entre cero” y para manejar esa excepción, en caso de que ocurra una. La aplicación permite al usuario introducir dos enteros, que se pasan como argumentos a la función `cociente` (líneas 13 a 21). Esta función divide su primer parámetro (`numerador`) entre su segundo parámetro (`denominador`). Suponiendo que el usuario no especifica 0 como el denominador para la división, la función `cociente` devuelve el resultado de la división. Sin embargo, si el usuario introduce un 0 para el denominador, la función `cociente` lanza una excepción. En los resultados de ejemplo, las primeras dos líneas muestran un cálculo exitoso y las siguientes dos líneas muestran un cálculo fallido, debido a un intento de dividir entre cero. Cuando ocurre la excepción, el programa informa al usuario del error y le pide que introduzca dos nuevos enteros. Una vez que hablamos sobre el código, consideraremos las entradas del usuario y el flujo de control del programa que producen estos resultados.

### Encerrar código en un bloque `try`

El programa empieza pidiendo al usuario que introduzca dos enteros. Estos enteros se introducen en la condición del ciclo `while` (línea 32). Después de que el usuario introduce valores que representan al numerador y denominador, el control del programa continúa hacia el cuerpo del ciclo (líneas 33 a 50). En la línea 38 se pasan estos valores a la función `cociente` (líneas 13 a 21), la cual divide los enteros y devuelve un resultado, o **lanza una excepción** (es decir, indica que ocurrió un error) en un intento de dividir entre cero. El manejo de excepciones está orientado a situaciones en las que la función que detecta un error no puede manejarlo.

C++ proporciona **bloques `try`** para permitir el manejo de excepciones. Un bloque `try` consiste en la palabra clave `try`, seguida de llaves (`{}`) que definen un bloque de código en el que podrían ocurrir errores. El bloque `try` encierra instrucciones que podrían ocasionar excepciones, e instrucciones que se deberían omitir si ocurre una excepción.

Observe que un bloque `try` (líneas 36 a 40) encierra la invocación de la función `cociente` y la instrucción que muestra el resultado de la división. En este ejemplo, debido a que la invocación de la función `cociente` (línea 38) puede lanzar una excepción, encerramos la invocación a esta función en un bloque `try`. Al encerrar la instrucción de salida (línea 39) en el bloque `try`, aseguramos que la salida sólo ocurrirá si la función `cociente` devuelve un resultado.



### Observación de Ingeniería de Software 16.2

*Las excepciones pueden surgir a través de código mencionado en forma explícita en un bloque try, a través de las llamadas a otras funciones y a través de llamadas a funciones con muchos niveles de anidamiento, iniciadas por el código en un bloque try.*

#### Definición de un manejador `catch` para procesar una ExcepcionDivisionEntreCero

Las excepciones se procesan mediante los manejadores `catch` (también conocidos como **manejadores de excepciones**), que atrapan y manejan las excepciones. Por lo menos debe haber un manejador `catch` (líneas 43 a 47) inmediatamente después de cada bloque `try`. Cada manejador `catch` empieza con la palabra clave `catch` y especifica entre paréntesis un **parámetro de excepción** que representa el tipo de excepción que puede procesar el manejador `catch` (en este caso, `ExcepcionDivisionEntreCero`). Cuando ocurre una excepción en un bloque `try`, el manejador `catch` que se ejecuta es aquél cuyo tipo coincide con el tipo de la excepción que ocurrió (es decir, el tipo en el bloque `catch` coincide exactamente con el tipo de excepción lanzada, o es una clase base de la misma). Si un parámetro de excepción incluye un nombre de parámetro opcional, el manejador `catch` puede usar ese nombre de parámetro para interactuar con la excepción atrapada en el cuerpo del manejador `catch`, que está delimitado por llaves (`{` y `}`). Por lo general, un manejador `catch` reporta el error al usuario, lo registra en un archivo, termina el programa sin que haya pérdida de datos o intenta una estrategia alterna para realizar la tarea fallida. En este ejemplo, el manejador `catch` simplemente reporta que el usuario trató de realizar una división entre cero. Después, el programa pide al usuario que introduzca dos nuevos valores enteros.



### Error común de programación 16.1

*Es un error de sintaxis colocar código entre un bloque try y sus correspondientes manejadores catch, o entre sus manejadores catch.*



### Error común de programación 16.2

*Cada manejador catch puede tener un solo parámetro; es un error de sintaxis especificar una lista separada por comas de parámetros de excepciones.*



### Error común de programación 16.3

*Es un error lógico atrapar el mismo tipo en dos manejadores catch distintos, que vayan después de un solo bloque try.*

#### Modelo de terminación del manejo de excepciones

Si ocurre una excepción como resultado de una instrucción en un bloque `try`, este bloque expira (es decir, termina de inmediato). A continuación, el programa busca el primer manejador `catch` que pueda procesar el tipo de excepción que ocurrió. El programa localiza el `catch` que coincide, comparando el tipo de la excepción lanzada con el tipo del parámetro de excepción de cada `catch`, hasta que el programa encuentra una coincidencia. Ocurre una coincidencia si los tipos son idénticos, o si el tipo de la excepción lanzada es una clase derivada del tipo del parámetro de excepción. Cuando ocurre una coincidencia, se ejecuta el código contenido en el manejador `catch` que coincide. Cuando un manejador `catch` termina su procesamiento al llegar a su llave derecha de cierre (`}`), se considera que la excepción se manejó y las variables locales definidas dentro del manejador `catch` (incluyendo el parámetro de `catch`) quedan fuera de alcance. El control del programa no regresa al punto en el que ocurrió la excepción (conocido como el **punto de lanzamiento**), debido a que el bloque `try` ha expirado. En vez de ello, el control continúa con la primera instrucción (línea 49) después del último manejador `catch` que sigue del bloque `try`. A esto se le conoce como **modelo de terminación del manejo de excepciones**. [Nota: algunos lenguajes usan el **modelo de reanudación del manejo de excepciones**, en el que después de manejar una excepción, el control se reanuda justo después del punto de lanzamiento]. Al igual que con cualquier otro bloque de código, cuando termina un bloque `try`, las variables definidas en el bloque quedan fuera de alcance.



### Error común de programación 16.4

*Pueden ocurrir errores lógicos si asumimos que, después de manejar una excepción, el control regresará a la primera instrucción que sigue después del punto de lanzamiento.*



### Tip para prevenir errores 16.2

*Con el manejo de excepciones, un programa puede seguir ejecutándose (en vez de terminar) después de lidiar con un problema. Esto ayuda a asegurar el tipo de aplicaciones robustas que contribuyen a lo que se conoce como computación de misión crítica, o computación crítica para los negocios.*

Si el bloque `try` completa su ejecución con éxito (es decir, si no ocurren excepciones en el bloque `try`), entonces el programa ignora los manejadores `catch` y el control del programa continúa con la primera instrucción después del último bloque `catch` que sigue de ese bloque `try`. Si no ocurren excepciones en un bloque `try`, el programa ignora el (los) manejador(es) `catch` para ese bloque.

Si una excepción que ocurre en un bloque `try` no tiene un manejador `catch` que coincida, o si una excepción ocurre en una instrucción que no se encuentre dentro de un bloque `try`, la función que contiene la instrucción termina de inmediato y el programa intenta localizar un bloque `try` circundante en la función que hizo la llamada. A este proceso se le conoce como **limpieza de la pila** y se describe en la sección 16.8.

#### *Flujo del control del programa cuando el usuario introduce un denominador distinto de cero*

Considere el flujo de control cuando el usuario introduce el numerador 100 y el denominador 7 (es decir, las primeras dos líneas de salida en la figura 16.2). En la línea 16, la función `cociente` determina que el denominador no es igual a cero, por lo que en la línea 20 se realiza la división y se devuelve el resultado (14.2857) a la línea 38 como un valor `double` (el texto `static_cast< double >` en la línea 20 asegura el tipo de valor de retorno apropiado). Después, el control del programa continúa secuencialmente desde la línea 38, por lo que en la línea 39 se muestra el resultado de la división; en la línea 40 se termina el bloque `try`. Como el bloque `try` se completó con éxito y no lanzó una excepción, el programa no ejecuta las instrucciones contenidas en el manejador `catch` (líneas 43 a 47), y el control continúa a la línea 49 (la primera línea de código después del manejador `catch`), en donde se pide al usuario que introduzca dos enteros más.

#### *Flujo del control del programa cuando el usuario escribe un denominador de cero*

Ahora vamos a considerar un caso más interesante, en el que el usuario introduce el numerador 100 y el denominador 0 (es decir, las líneas tercera y cuarta de la salida en la figura 16.2). En la línea 16, `cociente` determina que el denominador es igual a cero, lo cual indica un intento de división entre cero. En la línea 17 se lanza una excepción, que representamos como un objeto de la clase `ExcepcionDivisionEntreCero` (figura 16.1).

Para lanzar una excepción, en la línea 17 se utiliza la palabra clave `throw` seguida de un operando que representa el tipo de excepción a lanzar. Por lo general, una instrucción `throw` especifica un operando. (En la sección 16.5 veremos cómo usar una instrucción `throw` sin operandos). El operando de una instrucción `throw` puede ser de cualquier tipo. Si el operando es un objeto, lo llamamos **objeto excepción**; en este ejemplo, el objeto excepción es un objeto de tipo `ExcepcionDivisionEntreCero`. Sin embargo, un operando de `throw` también puede asumir otros valores, como el valor de una expresión que no produce un objeto (por ejemplo, `throw x > 5`) o el valor de un `int` (por ejemplo, `throw 5`). Los ejemplos en este capítulo se enfocan exclusivamente en cómo lanzar objetos excepción.



### Error común de programación 16.5

*Tenga cuidado al lanzar el resultado de una expresión condicional (? :), las reglas de promoción podrían hacer que el valor sea de un tipo distinto del esperado. Por ejemplo, al lanzar un `int` o un `double` de la misma expresión condicional, el `int` se promueve a `double`. Por lo tanto, un manejador `catch` que atrape un `int` nunca se ejecutaría, con base en dicha expresión condicional.*

Como parte de lanzar una excepción, se crea el operando `throw` y se utiliza para inicializar el parámetro en el manejador `catch`, el cual veremos en breve. En este ejemplo, la instrucción `throw` en la línea 17 crea un objeto de la clase `ExcepcionDivisionEntreCero`. Cuando la línea 17 lanza la excepción, la función `cociente` sale de inmediato. Por lo tanto, en la línea 17 se lanza la excepción antes de que la función `cociente` pueda realizar la división en la línea 20. Esta es una característica central del manejo de excepciones: una función debe lanzar una excepción *antes* de que el error tenga la oportunidad de manifestarse.

Ya que decidimos encerrar la invocación de la función `cociente` (línea 38) en un bloque `try`, el control del programa entra al manejador `catch` (líneas 43 a 47) que está justo después del bloque `try`. Este manejador `catch` sirve como el manejador de excepciones para la excepción de división entre cero. En general, cuando se lanza una excepción dentro de un bloque `try`, la excepción se atrapa mediante un manejador `catch` que especifica el tipo que coincide con la excepción lanzada. En este programa, el manejador `catch` especifica que atrapa objetos `ExcepcionDivisionEntreCero`; este

tipo coincide con el tipo del objeto lanzado en la función `cociente`. En realidad, el manejador `catch` atrapa una referencia al objeto `ErrorDivisionEntreCero` creado por la instrucción `throw` de la función `cociente` (línea 17). El objeto excepción se mantiene mediante el mecanismo de manejo de excepciones.



### Tip de rendimiento 16.2

*Al atrapar un objeto excepción por referencia, se elimina la sobrecarga de copiar el objeto que representa la excepción lanzada.*



### Buena práctica de programación 16.1

*Al asociar cada tipo de error en tiempo de ejecución con un objeto excepción con nombre apropiado, se mejora la claridad del programa.*

El cuerpo del manejador `catch` (líneas 45 y 46) imprime el mensaje de error asociado devuelto por la función que llaman a `catch` de la clase base `runtime_error`. Esta función devuelve la cadena que el constructor de `ErrorDivisionEntreCero` (líneas 12 y 13 de la figura 16.1) pasó al constructor de la clase base `runtime_error`.

## 16.4 Cuándo utilizar el manejo de excepciones

El manejo de excepciones está diseñado para procesar **errores síncronos**, que ocurren cuando se ejecuta una instrucción. Ejemplos comunes de estos errores son los subíndices de arreglos fuera de rango, el desbordamiento aritmético (es decir, un valor fuera del rango de valores representables), la división entre cero, los parámetros de función inválidos y la asignación fallida de memoria (debido a la falta de memoria). El manejo de excepciones no está diseñado para procesar los errores asociados con los eventos **síncronos** (por ejemplo, completar la E/S de disco, la llegada de mensajes de red, los clics del ratón y las pulsaciones de tecla), los cuales ocurren en paralelo con, y de manera independiente a, el flujo de control del programa.



### Observación de Ingeniería de Software 16.3

*Incorpore su estrategia de manejo de excepciones en su sistema, a partir del comienzo del proceso de diseño. Puede ser difícil incluir un manejo efectivo de las excepciones, después de haber implementado un sistema.*



### Observación de Ingeniería de Software 16.4

*El manejo de excepciones proporciona una sola técnica uniforme para procesar problemas. Esto ayuda a los programadores, que trabajan en proyectos extensos, a comprender el código de procesamiento de errores de los demás programadores.*



### Observación de Ingeniería de Software 16.5

*Evite usar el manejo de excepciones como una forma alternativa de flujo de control. Estas excepciones “adicionales” pueden “interponerse” en las excepciones de tipos de errores genuinos.*



### Observación de Ingeniería de Software 16.6

*El manejo de excepciones simplifica la combinación de los componentes de software, y les permite trabajar en conjunto de manera efectiva, al permitir que los componentes predefinidos comuniquen los problemas a los componentes específicos de una aplicación, que a su vez pueden procesar los problemas en forma específica para la aplicación.*

El mecanismo de manejo de errores también es útil para procesar problemas que ocurren cuando un programa interactúa con los elementos de software, como las funciones miembro, los constructores, los destructores y las clases. En vez de manejar los problemas de manera interna, dichos elementos de software utilizan comúnmente las excepciones para notificar a los programas cuando ocurren problemas. Esto permite a los programadores implementar el manejo de errores personalizado para cada aplicación.



### Tip de rendimiento 16.3

*Cuando no ocurren excepciones, el código de manejo de excepciones afecta muy poco (o nada) al rendimiento. Por ende, los programas que implementan el manejo de excepciones operan con mayor eficiencia que los programas que entremezclan el código de manejo de errores con la lógica del programa.*



## Observación de Ingeniería de Software 16.7

*Las funciones con condiciones de errores comunes deben devolver 0 o NULL (u otros valores apropiados), en vez de lanzar excepciones. Un programa que llame a dicha función puede comprobar el valor de retorno para determinar si la llamada a la función tuvo éxito o fracasó.*

Por lo general, las aplicaciones complejas consisten en componentes predefinidos de software y componentes específicos de cada aplicación que utilizan los componentes predefinidos. Cuando un componente predefinido encuentra un problema, ese componente necesita un mecanismo para comunicar el problema al componente específico de la aplicación; el componente predefinido no puede saber de antemano cómo procesa cada aplicación un problema que se presenta.

## 16.5 Volver a lanzar una excepción

Es posible que un manejador de excepciones, al momento de recibir una excepción, decida que no puede procesar esa excepción o que puede procesar la excepción sólo de manera parcial. En tales casos, el manejador de excepciones puede diferir el manejo de excepciones (o tal vez una porción de éste) a otro manejador de excepciones. En cualquier caso, para lograr esto hay que **volver a lanzar la excepción** mediante la siguiente instrucción:

```
throw;
```

Sin importar el que un manejador pueda o no procesar (incluso de manera parcial) una excepción, el manejador puede volver a lanzar la excepción para seguirla procesando fuera de éste. El siguiente bloque `try` circundante detecta la excepción que se volvió a lanzar, y un manejador `catch` listado después de ese bloque `try` circundante trata de manejarla.



## Error común de programación 16.6

*Al ejecutar una instrucción `throw` vacía que se sitúa fuera de un manejador `catch` se produce una llamada a la función `terminate`, la cual abandona el procesamiento de la excepción y termina el programa de inmediato.*

El programa de la figura 16.3 demuestra cómo volver a lanzar una excepción. En el bloque `try` de `main` (líneas 32 a 37), la línea 35 llama a la función `lanzarExcepcion` (líneas 11 a 27). Esta función también contiene un bloque `try` (líneas 14 a 18), desde el que la instrucción `throw` en la línea 17 lanza una instancia de la clase `exception` de la biblioteca estándar. El manejador `catch` de la función `lanzarExcepcion` (líneas 19 a 24) atrapa esta excepción, imprime un mensaje de error (línea 21 y 22) y vuelve a lanzar la excepción (línea 23). Esto termina la función `lanzarExcepcion` y devuelve el control a la línea 35 en el bloque `try...catch` en `main`. El bloque `try` termina (por lo que la línea 36 no se ejecuta), y el manejador `catch` en `main` (líneas 38 a 41) atrapa esta excepción e imprime un mensaje de error (línea 40). [Nota: como no utilizamos los parámetros de excepción en los manejadores `catch` de este ejemplo, omitimos los nombres de los parámetros de excepción y especificamos sólo el tipo de excepción que se debe atrapar (líneas 19 y 38)].

```

1 // Fig. 16.3: Fig16_03.cpp
2 // Demostración de cómo volver a lanzar excepciones.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <exception>
8 using std::exception;
9
10 // lanza, atrapa y vuelve a lanzar la excepción
11 void lanzarExcepcion()
12 {
13 // lanza la excepción y la atrapa de inmediato
14 try
15 {
16 cout << " la función lanzarExcepcion lanza una excepción\n";
17 throw excepcion(); // genera la excepción
18 } // fin de try
19 catch (exception &) // maneja la excepción
20 }
```

Figura 16.3 | Volver a lanzar una excepción. (Parte I de 2).

```

21 cout << " La excepcion se manejo en la funcion lanzarExcepcion"
22 << "\n La funcion lanzarExcepcion vuelve a lanzar la excepcion";
23 throw; // vuelve a lanzar la excepcion para seguir procesandola
24 } // fin de catch
25
26 cout << "Esto tampoco se debe imprimir\n";
27 } // fin de la función lanzarExcepcion
28
29 int main()
30 {
31 // lanza la excepción
32 try
33 {
34 cout << "\nmain invoca a la funcion lanzarExcepcion\n";
35 lanzarExcepcion();
36 cout << "Esto no se debe imprimir\n";
37 } // fin de try
38 catch (excepcion &) // maneja la excepción
39 {
40 cout << "\n\nLa excepcion se manejo en main\n";
41 } // fin de catch
42
43 cout << "El control del programa continua despues de catch en main\n";
44 return 0;
45 } // fin de main

```

```

main invoca a la funcion lanzarExcepcion
La funcion lanzarExcepcion lanza una excepcion
La excepcion se manejo en la funcion lanzarExcepcion
La funcion lanzarExcepcion vuelve a lanzar la excepcion

La excepcion se manejo en main
El control del programa continua despues de catch en main

```

**Figura 16.3** | Volver a lanzar una excepción. (Parte 2 de 2).

## 16.6 Especificaciones de excepciones

Una especificación de excepciones opcional (también conocida como **lista throw**) enumera una lista de excepciones que puede lanzar una función. Por ejemplo, considere la siguiente declaración de una función:

```

int unaFuncion(double valor)
 throw (ExcepcionA, ExcepcionB, ExcepcionC)
{
 // cuerpo de la función
}

```

En esta definición, la especificación de la función, que empieza con la palabra clave **throw** justo después del paréntesis de cierre de la lista de parámetros de la función, indica que la función **unaFuncion** puede lanzar excepciones de los tipos **ExcepcionA**, **ExcepcionB** y **ExcepcionC**. Una función sólo puede lanzar excepciones de los tipos indicados por la especificación, o excepciones de cualquier tipo derivado de estos tipos. Si la función lanza (**throw**) una excepción que no pertenezca a un tipo especificado, el mecanismo de manejo de excepciones llama a la función **unexpected**, la cual termina el programa.

Una función que no proporciona una especificación de excepciones puede lanzar cualquier excepción. Al colocar **throw()** (una especificación de excepciones vacía) después de la lista de parámetros de una función, se establece que esa función no lanza excepciones. Si la función intenta lanzar una excepción, se invoca la función **unexpected**. En la sección 16.7 mostraremos cómo se puede personalizar la función **unexpected** mediante una llamada a la función **set\_unexpected**.

### Error común de programación 16.7



Al lanzar una excepción que no se haya declarado en la especificación de excepciones de una función, se produce una llamada a la función **unexpected**.



### Tip para prevenir errores 16.3

*El compilador no generará un error de compilación si una función contiene una expresión `throw` para una excepción no listada en la especificación de excepciones de la función. Sólo se produce un error cuando esa función intenta lanzar esa excepción en tiempo de ejecución. Para evitar sorpresas en tiempo de ejecución, el programador debe comprobar cuidadosamente su código para asegurar que las funciones no lancen excepciones que no estén listadas en sus especificaciones de excepciones.*

## 16.7 Procesamiento de excepciones inesperadas

La función `unexpected` llama a la función registrada con la función `set_unexpected` (definida en el archivo de encabezado `<exception>`). Si no se ha registrado una función de esta forma, se hace una llamada a la función `terminate` de manera predeterminada. Los casos en los que se hace una llamada a la función `terminate` son:

1. el mecanismo de excepción no puede encontrar un `catch` que coincida para una excepción que se haya lanzado.
2. un destructor intenta lanzar una excepción durante la limpieza de la pila.
3. hay un intento de volver a lanzar una excepción cuando no se esté manejando una excepción en un momento dado.
4. una llamada a la función `unexpected` realiza una llamada a la función `terminate` de manera predeterminada.

(En la sección 15.5.1 del Documento del Estándar de C++ se describen varios casos adicionales). La función `set_terminate` puede especificar la función a invocar cuando se haga la llamada a `terminate`. En caso contrario, `terminate` llama a `abort`, con lo cual se termina el programa sin llamar a los destructores de cualquier objeto restante de una clase de almacenamiento estático o automático. Esto podría conllevar a fugas de recursos cuando un programa termina de forma prematura.

Las funciones `set_terminate` y `set_unexpected` devuelven un apuntador a la última función llamada por `terminate` y `unexpected`, respectivamente (0, la primera vez que se llama a cada una). Esto permite al programador guardar el apuntador a la función, para poder restaurarlo más adelante. Las funciones `set_terminate` y `set_unexpected` reciben como argumentos apuntadores a funciones con tipos de valores de retorno `void` y sin argumentos.

Si la última acción de una función de terminación definida por el programador no es salir de un programa, se hará una llamada a la función `abort` para terminar la ejecución del programa una vez que se ejecuten las demás instrucciones de la función de terminación definida por el programador.

## 16.8 Limpieza de la pila

Cuando se lanza una excepción pero no se atrapa en un alcance específico, la pila de llamadas a funciones se “limpia” y se hace un intento de atrapar (`catch`) la excepción en el siguiente bloque `try...catch` exterior. Limpiar la pila de llamadas a funciones significa que la función en la que no se atrapó la excepción termina, todas las variables locales en la función se destruyen y el control regresa a la instrucción que invocó originalmente a esa función. Si un bloque `try` encierra esa instrucción, se hace un intento de atrapar la excepción. Si un bloque `try` no encierra esa instrucción, se vuelve a realizar la limpieza de la pila. Si no hay un manejador `catch` que atrape a esta excepción, se hace una llamada a la función `terminate` para terminar el programa. El programa de la figura 16.4 demuestra la limpieza de la pila:

```

1 // Fig. 16.4: Fig16_04.cpp
2 // Demostración de la limpieza de la pila.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <stdexcept>
8 using std::runtime_error;
9
10 // funcion3 lanza un error en tiempo de ejecución
11 void funcion3() throw (runtime_error)
12 {

```

Figura 16.4 | Limpieza de la pila. (Parte I de 2).

```

13 cout << "En la funcion 3" << endl;
14
15 // no hay bloque try, se realiza la limpieza de la pila, devuelve el control a funcion2
16 throw runtime_error("runtime_error en funcion3"); // no imprime
17 } // fin de funcion3
18
19 // funcion2 invoca a funcion3
20 void funcion2() throw (runtime_error)
21 {
22 cout << "funcion3 se llama dentro de funcion2" << endl;
23 funcion3(); // se realiza la limpieza de la pila, devuelve el control a funcion1
24 } // fin de funcion2
25
26 // funcion1 invoca a funcion2
27 void funcion1() throw (runtime_error)
28 {
29 cout << "funcion2 se llama dentro de funcion1" << endl;
30 funcion2(); // se realiza la limpieza de la pila, devuelve el control a main
31 } // fin de funcion1
32
33 // demuestra la limpieza de la pila
34 int main()
35 {
36 // invoca a funcion1
37 try
38 {
39 cout << "funcion1 se llama dentro de main" << endl;
40 funcion1(); // llama a funcion1, que lanza runtime_error
41 } // fin de try
42 catch (runtime_error &error) // maneja el error en tiempo de ejecución
43 {
44 cout << "Ocurrio una excepcion: " << error.what() << endl;
45 cout << "La excepcion se manejo en main" << endl;
46 } // fin de catch
47
48 return 0;
49 } // fin de main

```

```

funcion1 se llama dentro de main
funcion2 se llama dentro de funcion1
funcion3 se llama dentro de funcion2
En la funcion 3
Ocurrio una excepcion: runtime_error en funcion3
La excepcion se manejo en main

```

**Figura 16.4** | Limpieza de la pila. (Parte 2 de 2).

En `main`, el bloque `try` (líneas 37 a 41) llama a `funcion1` (líneas 27 a 31). A continuación, `funcion1` llama a `funcion2` (líneas 20 a 24), que a su vez llama a `funcion3` (líneas 11 a 17). En la línea 16 de `funcion3` se lanza un objeto `runtime_error`. Sin embargo, debido a que ningún bloque `try` encierra la instrucción `throw` en la línea 16, se realiza la limpieza de la pila; `funcion3` termina en la línea 16, y después devuelve el control a la instrucción en `funcion2` que invocó a `funcion3` (línea 23). Como no hay un bloque `try` que encierre la línea 23, se realiza la limpieza de la pila otra vez; `funcion2` termina en la línea 23 y devuelve el control a la instrucción en `funcion1` que invocó a `funcion2` (línea 30). Como no hay bloque `try` que encierre la línea 30, se realiza la limpieza de la pila una vez más; `funcion1` termina en la línea 30 y devuelve el control a la instrucción en `main` que invocó a `funcion1` (línea 40). El bloque `try` de las líneas 37 a 41 encierra esta instrucción, por lo que el primer manejador `catch` que coincide y que está ubicado después de este bloque `try` (líneas 42 a 46) atrapa y procesa la excepción. En la línea 44 se usa la función `what` para mostrar el mensaje de la excepción. Recuerde que la función `what` es una función `virtual` de la clase `exception` que una clase derivada puede sobrescribir para que devuelva un mensaje de error apropiado.

## 16.9 Constructores, destructores y manejo de excepciones

Primero vamos a hablar sobre una cuestión que hemos mencionado, pero todavía no hemos resuelto de manera satisfactoria: ¿qué ocurre cuando se detecta un error en un constructor? Por ejemplo, ¿cómo debe responder el constructor de un objeto cuando `new` falla debido a que no pudo asignar la memoria requerida para almacenar la representación interna de ese objeto? Como el constructor no puede devolver un valor para indicar un error, debemos elegir un medio alternativo de indicar que el objeto no se ha construido en forma apropiada. Un esquema es devolver el objeto mal construido y esperar que alguien que lo use realice las pruebas apropiadas para determinar que se encuentra en un estado inconsistente. Otro esquema es establecer una variable fuera del constructor. La alternativa preferida es requerir que el constructor lance una excepción que contenga la información del error, con lo cual se ofrece una oportunidad para que el programa maneje la falla.

Antes de que un constructor lance una excepción, se hacen llamadas a los destructores para cualquier objeto miembro que se construya como parte del objeto que se va a construir. Se hacen llamadas a los destructores para todos los objetos automáticos construidos en un bloque `try` antes de lanzar una excepción. Se garantiza que la limpieza de la pila se habrá completado en el momento en el que un manejador de excepciones se empiece a ejecutar. Si un destructor invocado como resultado de la limpieza de la pila lanza una excepción, se hace una llamada a `terminate`.

Si un objeto tiene objetos miembro, y si se lanza una excepción antes de que el objeto exterior se construya por completo, entonces se ejecutarán los destructores para los objetos miembro que se hayan construido antes de la ocurrencia de la excepción. Si se ha construido parcialmente un arreglo de objetos cuando ocurre una excepción, sólo se llamará a los destructores para los objetos construidos en el arreglo.

Una excepción podría impedir la operación de código que por lo general liberaría un recurso, con lo cual se produciría una fuga de recursos. Una técnica para resolver este problema es inicializar un objeto local para adquirir el recurso. Cuando ocurra una excepción, se invocará al destructor para ese objeto y se podrá liberar el recurso.



### Tip para prevenir errores 16.4

*Cuando se lanza una excepción desde el constructor para un objeto que se cree en una nueva expresión, se libera la memoria asignada en forma dinámica para ese objeto.*

## 16.10 Excepciones y herencia

Se pueden衍生 varias clases de excepciones de una clase base común, como vimos en la sección 16.3, cuando creamos la clase `ExcepcionDivisionEntreCero` como una clase derivada de la clase `exception`. Si un manejador `catch` maneja un apuntador o referencia a un objeto excepción del tipo de una clase base, también puede atrapar un apuntador o referencia a todos los objetos de las clases que se deriven públicamente de esa clase base; esto permite el procesamiento polimórfico de los errores relacionados.



### Tip para prevenir errores 16.5

*El uso de la herencia con excepciones permite a un manejador de excepciones atrapar los errores relacionados con una notación concisa. Una metodología es atrapar cada tipo de apuntador o referencia a un objeto excepción de clase derivada en forma individual, pero una metodología más concisa es atrapar mejor apuntadores o referencias a objetos excepción de la clase base. Además, al atrapar apuntadores o referencias a objetos excepción de una clase derivada se pueden cometer errores, en especial si el programador olvida evaluar de manera explícita uno o más de los tipos apuntador o referencia de la clase derivada.*

## 16.11 Procesamiento de las fallas de new

El estándar de C++ especifica que, cuando falla el operador `new`, lanza una excepción `bad_alloc` (definida en el archivo de encabezado `<new>`). Sin embargo, algunos compiladores no están en conformidad con el estándar de C++, y por lo tanto utilizan la versión de `new` que devuelve 0 al fallar. Por ejemplo, Microsoft Visual C++ 2005 lanza una excepción `bad_alloc` cuando falla `new`, mientras que Microsoft Visual C++ 6.0 devuelve 0 cuando falla `new`.

Los compiladores varían en cuanto a su soporte para el manejo de fallas de `new`. Muchos compiladores de C++ anteriores devuelven 0 de manera predeterminada cuando falla `new`. Algunos compiladores soportan que `new` lance una excepción si se incluye el archivo de encabezado `<new>` (o `<new.h>`). Otros compiladores lanzan `bad_alloc` de manera predeterminada, no importa si está o no incluido el archivo de encabezado `<new>`. Consulte con la documentación de su compilador para determinar el soporte que ofrece para el manejo de fallas de `new`.

En esta sección presentaremos tres ejemplos de fallas de `new`. El primer ejemplo devuelve 0 cuando falla `new`. El segundo ejemplo utiliza la versión de `new` que lanza una excepción `bad_alloc` cuando falla `new`. El tercer ejemplo utiliza la función `set_new_handler` para manejar las fallas de `new`. [Nota: los ejemplos de las figuras 16.5 y 16.7 asignan cantidades extensas de memoria dinámica, lo cual podría hacer que su computadora se vuelva lenta].

### Caso en el que new devuelve 0 al fallar

La figura 16.5 demuestra cómo new devuelve 0 al fallar en asignar la cantidad requerida de memoria. La instrucción `for` en las líneas 13 a 24 debería iterar 50 veces y, en cada pasada, asignar un arreglo de 50,000,000 valores `double` (es decir, 400,000,000 bytes, ya que un `double` es por lo general de 8 bytes). La instrucción `if` en la línea 17 evalúa el resultado de cada operación de `new` para determinar si `new` asignó la memoria con éxito. Si `new` falla y devuelve 0, en la línea 19 se imprime un mensaje de error y el ciclo termina. [Nota: usamos Microsoft Visual C++ 6.0 para ejecutar este ejemplo, ya que Microsoft Visual C++ 2005 lanza una excepción `bad_alloc` cuando falla `new`, en vez de devolver 0].

Los resultados muestran que el programa sólo realizó tres iteraciones antes de que `new` fallara, y el ciclo terminó. Los resultados del lector podrían diferir con base en la memoria física, el espacio en disco disponible para la memoria virtual en su sistema y el compilador que esté utilizando.

### Caso en el que new lanza `bad_alloc` al fallar

La figura 16.6 demuestra cómo new lanza `bad_alloc` al fallar en asignar la memoria solicitada. La instrucción `for` (líneas 20 a 24) dentro del bloque `try` debe iterar 50 veces y en cada pasada asigne un arreglo de 50,000,000 valores `double`. Si `new` falla y lanza una excepción `bad_alloc`, el ciclo termina y el programa continúa en la línea 28, en donde el manejador `catch` atrapa y procesa la excepción. En las líneas 30 y 31 se imprime el mensaje "Ocurrió una excepción:" seguido del mensaje devuelto de la versión de la función `what` correspondiente a la clase base `exception` (es decir, un mensaje específico de la excepción, definido por la implementación, como "Allocation Failure" en Microsoft Visual

```

1 // Fig. 16.5: Fig16_05.cpp
2 // Demostración de la función new previa al estándar que devuelve 0
3 // cuando falla la asignación de memoria.
4 #include <iostream>
5 using std::cerr;
6 using std::cout;
7
8 int main()
9 {
10 double *ptr[50];
11
12 // orienta cada ptr[i] a un bloque extenso de memoria
13 for (int i = 0; i < 50; i++)
14 {
15 ptr[i] = new double[50000000]; // asigna un bloque extenso
16
17 if (ptr[i] == 0) // ¿falló new al asignar la memoria?
18 {
19 cerr << "La asignacion de memoria fallo para ptr[" << i << "]\n";
20 break;
21 } // fin de if
22 else // asignación de memoria exitosa
23 cout << "ptr[" << i << "] apunta a 50,000,000 valores double\n";
24 } // fin de for
25
26 return 0;
27 } // fin de main

```

ptr[0] apunta a 50,000,000 valores double  
 ptr[1] apunta a 50,000,000 valores double  
 ptr[2] apunta a 50,000,000 valores double  
 La asignacion de memoria fallo para ptr[3]

**Figura 16.5** | Demostración de cómo la función `new` previa al estándar devuelve 0 cuando falla la asignación de memoria.

```

1 // Fig. 16.6: Fig16_06.cpp
2 // Demostración de new estándar que lanza una excepción
3 // bad_alloc cuando no se puede asignar memoria.

```

**Figura 16.6** | new lanza `bad_alloc` al fallar. (Parte 1 de 2).

```

4 #include <iostream>
5 using std::cerr;
6 using std::cout;
7 using std::endl;
8
9 #include <new> // operador new estándar
10 using std::bad_alloc;
11
12 int main()
13 {
14 double *ptr[50];
15
16 // orienta cada ptr[i] a un bloque extenso de memoria
17 try
18 {
19 // asigna memoria para ptr[i]; new lanza bad_alloc al fallar
20 for (int i = 0; i < 50; i++)
21 {
22 ptr[i] = new double[50000000]; // puede lanzar una excepción
23 cout << "ptr[" << i << "] apunta a 50,000,000 nuevos valores double\n";
24 } // fin de for
25 } // fin de try
26
27 // maneja la excepción bad_alloc
28 catch (bad_alloc &excepcionAsignacionMemoria)
29 {
30 cerr << "Ocurrio una excepcion: "
31 << excepcionAsignacionMemoria.what() << endl;
32 } // fin de catch
33
34 return 0;
35 } // fin de main

```

```

ptr[0] apunta a 50,000,000 nuevos valores double
ptr[1] apunta a 50,000,000 nuevos valores double
ptr[2] apunta a 50,000,000 nuevos valores double
ptr[3] apunta a 50,000,000 nuevos valores double
Ocurrio una excepcion: bad allocation

```

Figura 16.6 | new lanza bad\_alloc al fallar. (Parte 2 de 2).

C++ 2005). Los resultados muestran que el programa sólo realizó tres iteraciones del ciclo antes de que fallara new y se lanzara la excepción bad\_alloc. Los resultados del lector podrían diferir con base en la memoria física, espacio en disco disponible para la memoria virtual en su sistema, y el compilador que esté utilizando.

El estándar de C++ especifica que los compiladores en conformidad con el estándar pueden seguir utilizando una versión de new que devuelva 0 al fallar. Para este propósito, el archivo de encabezado <new> define el objeto **nothrow** (de tipo **nothrow\_t**), que se utiliza de la siguiente manera:

```
double *ptr = new(noexcept) double[50000000];
```

La instrucción anterior utiliza la versión de new que no lanza excepciones bad\_alloc (es decir, noexcept) para asignar un arreglo de 50,000,000 valores double.



### Observación de Ingeniería de Software 16.8

*Para hacer los programas más robustos, utilice la versión de new que lanza excepciones bad\_alloc al fallar.*

#### Manejo de las fallas de new mediante la función set\_new\_handler

Una característica adicional para manejar las fallas de new es la función **set\_new\_handler** (cuyo prototipo se encuentra en el archivo de encabezado estándar <new>). Esta función recibe como argumento un apuntador a una función que no recibe argumentos y devuelve void. Este apuntador apunta a la función que se llamará si new falla. Esto proporciona al programador una metodología uniforme para manejar todas las fallas de new, sin importar que ocurra una falla en

el programa. Una vez que `set_new_handler` registra un **manejador de new** en el programa, el operador `new` no lanza `bad_alloc` en el futuro; en vez de ello, difiere el manejo de errores a la función manejadora de `new`.

Si `new` asigna memoria con éxito, devuelve un apuntador a esa memoria. Si `new` falla en asignar memoria y `set_new_handler` no registró una función manejadora de `new`, `new` lanza una excepción `bad_alloc`. Si `new` falla al asignar memoria y se ha registrado una función manejadora de `new`, se hace una llamada a esta función. El estándar de C++ especifica que la función manejadora de `new` debe realizar una de las siguientes tareas:

1. Hacer más memoria disponible al eliminar otra parte de la memoria asignada en forma dinámica (o indicando al usuario que cierre otras aplicaciones) y regresar al operador `new` para tratar de asignar memoria otra vez.
2. Lanzar una excepción de tipo `bad_alloc`.
3. Llamar a la función `abort` o `exit` (ambas se encuentran en el archivo de encabezado `<cstdlib>`) para terminar el programa.

La figura 16.7 demuestra el uso de `set_new_handler`. La función `nuevoManejadorPersonalizado` (líneas 14 a 18) imprime un mensaje de error (línea 16) y después termina el programa mediante una llamada a `abort` (línea 17). Los resultados muestran que el programa sólo realizó tres iteraciones del ciclo, debido a que `new` falló e invocó a la función `nuevoManejadorPersonalizado`. Los resultados del lector podrían diferir con base en la memoria física, el espacio disponible para la memoria virtual en su sistema y el compilador que utilice para compilar el programa.

```
1 // Fig. 16.7: Fig16_07.cpp
2 // Demostración de set_new_handler.
3 #include <iostream>
4 using std::cerr;
5 using std::cout;
6
7 #include <new> // operador new estándar y set_new_handler
8 using std::set_new_handler;
9
10 #include <cstdlib> // prototipo de la función abort
11 using std::abort;
12
13 // maneja la característica de asignación de memoria
14 void nuevoManejadorPersonalizado()
15 {
16 cerr << "Se llamo a nuevoManejadorPersonalizado";
17 abort();
18 } // fin de la función nuevoManejadorPersonalizado
19
20 // uso de set_new_handler para manejar la asignación de memoria fallida
21 int main()
22 {
23 double *ptr[50];
24
25 // especifica que se debe llamar a nuevoManejadorPersonalizado
26 // al fallar la asignación de memoria
27 set_new_handler(nuevoManejadorPersonalizado);
28
29 // orienta cada ptr[i] a un bloque extenso de memoria; se llamará a
30 // nuevoManejadorPersonalizado al fallar la asignación de memoria
31 for (int i = 0; i < 50; i++)
32 {
33 ptr[i] = new double[50000000]; // puede lanzar una excepción
34 cout << "ptr[" << i << "] apunta a 50,000,000 nuevos valores double\n";
35 } // fin de for
36
37 return 0;
38 } // fin de main
```

Figura 16.7 | `set_new_handler` especifica la función a llamar cuando falla `new`. (Parte I de 2).

```

ptr[0] apunta a 50,000,000 nuevos valores double
ptr[1] apunta a 50,000,000 nuevos valores double
ptr[2] apunta a 50,000,000 nuevos valores double
se llama a nuevoManejadorPersonalizado

```

Figura 16.7 | `set_new_handler` especifica la función a llamar cuando falla `new`. (Parte 2 de 2).

## 16.12 La clase `auto_ptr` y la asignación dinámica de memoria

Una práctica común de programación es asignar la memoria dinámica, asignar la dirección de la memoria a un apuntador, usar el apuntador para manipular la memoria y desasignar la memoria con `delete` cuando ésta ya no sea necesaria. Si ocurre una excepción después de una asignación exitosa de memoria, pero antes de que se ejecute la instrucción `delete`, podría ocurrir una fuga de memoria. El estándar de C++ proporciona la plantilla de clase `auto_ptr` en el archivo de encabezado `<memory>` para lidiar con esta situación.

Un objeto de la clase `auto_ptr` mantiene un apuntador a la memoria que se asigna en forma dinámica. Cuando se hace una llamada al destructor de un objeto `auto_ptr` (por ejemplo, cuando un objeto `auto_ptr` queda fuera de alcance), realiza una operación `delete` con su miembro de datos apuntador. La plantilla de clase `auto_ptr` proporciona los operadores sobrecargados `*` y `->`, de manera que un objeto `auto_ptr` se puede utilizar de la misma forma que una variable apuntador común. En la figura 16.10 se demuestra un objeto `auto_ptr` que apunta a un objeto de la clase `Entero` asignado en forma dinámica (figuras 16.8-16.9).

```

1 // Fig. 16.8: Entero.h
2 // Definición de la clase Entero.
3
4 class Entero
5 {
6 public:
7 Entero(int i = 0); // constructor predeterminado de Entero
8 ~Entero(); // destructor de Entero
9 void setEntero(int i); // establece el valor de un Entero
10 int getEntero() const; // devuelve el valor de un Entero
11 private:
12 int valor;
13 } // fin de la clase Entero

```

Figura 16.8 | Definición de la clase `Entero`.

```

1 // Fig. 16.9: Entero.cpp
2 // Definición de las funciones miembro de Entero.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Entero.h"
8
9 // constructor predeterminado de Entero
10 Entero::Entero(int i)
11 : valor(i)
12 {
13 cout << "Constructor para Entero " << valor << endl;
14 } // fin del constructor de Entero
15
16 // destructor de Entero
17 Entero::~Entero()
18 {
19 cout << "Destructor para Entero " << valor << endl;

```

Figura 16.9 | Definiciones de las funciones miembro de la clase `Entero`. (Parte 1 de 2).

```

20 } // fin del destructor de Entero
21
22 // establece el valor de Entero
23 void Entero::setEntero(int i)
24 {
25 valor = i;
26 } // fin de la función setEntero
27
28 // devuelve el valor de Entero
29 int Entero::getEntero() const
30 {
31 return valor;
32 } // fin de la función getEntero

```

**Figura 16.9** | Definiciones de las funciones miembro de la clase Entero. (Parte 2 de 2).

En la línea 18 de la figura 16.10 se crea el objeto `ptrAEntero` de `auto_ptr`, y se inicializa con un apuntador a un objeto `Entero` asignado en forma dinámica, que contiene el valor 7. En la línea 21 se utiliza el operador `->` sobrecargado de `auto_ptr` para invocar a la función `setEntero` en el objeto `Entero` que maneja `ptrAEntero`. En la línea 24 se utiliza el operador `*` sobrecargado de `auto_ptr` para desreferenciar a `ptrAEntero`, y después se utiliza el operador punto `(.)` para invocar a la función `getEntero` en el objeto `Entero`. Al igual que un apuntador regular, los operadores sobrecargados `->` y `*` de un objeto `auto_ptr` se pueden utilizar para acceder al objeto al que apunta el objeto `auto_ptr`.

Como `ptrAEntero` es una variable automática local en `main`, se destruye cuando `main` termina. El destructor de `auto_ptr` obliga a que se lleve a cabo una operación `delete` del objeto `Entero` al que apunta `ptrAEntero`, que a su vez

```

1 // Fig. 16.10: Fig16_10.cpp
2 // Demostración de auto_ptr.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <memory>
8 using std::auto_ptr; // definición de la clase auto_ptr
9
10 #include "Entero.h"
11
12 // usa auto_ptr para manipular un objeto Entero object
13 int main()
14 {
15 cout << "Creacion de un objeto auto_ptr que apunta a un objeto Entero\n";
16
17 // "orienta" auto_ptr al objeto Entero
18 auto_ptr< Entero > ptrAEntero(new Entero(7));
19
20 cout << "\nUso de auto_ptr para manipular el objeto Entero\n";
21 ptrAEntero->setEntero(99); // usa auto_ptr para set el valor de Entero
22
23 // usa auto_ptr para obtener el valor de Entero
24 cout << "Entero despues de setEntero: " << (*ptrAEntero).getEntero()
25 return 0;
26 } // fin de main

```

Creacion de un objeto auto\_ptr que apunta a un objeto Entero  
Constructor para Entero 7

Uso de auto\_ptr para manipular el objeto Entero  
Entero despues de setEntero: 99

Destructor para Entero 99

**Figura 16.10** | Un objeto `auto_ptr` maneja la memoria asignada en forma dinámica.

llama al destructor de la clase `Entero`. La memoria que ocupa `Entero` se libera, sin importar cómo sale el control del bloque (por ejemplo, mediante una instrucción `return` o mediante una excepción). Lo que es más importante, al utilizar esta técnica se pueden evitar fugas de memoria. Por ejemplo, suponga que una función devuelve un apuntador orientado a cierto objeto. Por desgracia, la función llamadora que recibe este apuntador tal vez no elimine el objeto mediante `delete`, con lo cual se produce una fuga de memoria. No obstante, si la función devuelve un objeto `auto_ptr` que apunte al objeto, éste se eliminará de manera automática cuando se haga la llamada al destructor del objeto `auto_ptr`.

Sólo un objeto `auto_ptr` puede poseer un objeto asignado en forma dinámica en un momento dado, y el objeto no puede ser un arreglo. Al usar su operador de asignación sobrecargado o su constructor de copia, un objeto `auto_ptr` puede transferir la propiedad de la memoria dinámica que maneja. El último objeto `auto_ptr` que mantiene el apuntador a la memoria dinámica eliminará la memoria. Esto hace de `auto_ptr` un mecanismo ideal para devolver memoria asignada en forma dinámica al código cliente. Cuando el objeto `auto_ptr` queda fuera de alcance en el código cliente, el destructor de `auto_ptr` elimina la memoria dinámica.

## 16.13 Jerarquía de excepciones de la Biblioteca estándar

La experiencia ha demostrado que las excepciones se pueden clasificar en varias categorías. La Biblioteca estándar de C++ incluye una jerarquía de clases de excepciones (figura 16.11). Como vimos por primera vez en la sección 16.3, esta jerarquía está encabezada por la clase base `exception` (definida en el archivo de encabezado `<exception>`), la cual contiene la función `virtual what`, que las clases derivadas pueden sobreescribir para generar los mensajes de error apropiados.

Las clases derivadas inmediatas de la clase base `exception` incluyen a `runtime_error` y `logic_error` (ambas definidas en el encabezado `<stdexcept>`), cada una de las cuales tiene varias clases derivadas. De `exception` también se derivan las excepciones lanzadas por los operadores de C++; por ejemplo, `bad_alloc` es lanzada por `new` (sección 16.11), `bad_cast` es lanzada por `dynamic_cast` (capítulo 13) y `bad_typeid` es lanzada por `typeid` (capítulo 13). Incluir un `bad_exception` en la lista `throw` de una función significa que, si ocurre una excepción inesperada, la función `unexpected` puede lanzar a `bad_exception` en vez de terminar la ejecución del programa (de manera predeterminada) o llamar a otra función especificada por `set_unexpected`.

### Error común de programación 16.8



*Colocar un manejador `catch` que atrape un objeto de la clase base antes de un `catch` que atrape un objeto de una clase derivada de esa clase base es un error lógico. El `catch` de la clase base atrapa todos los objetos de las clases derivadas de esa clase base, por lo que el `catch` de la clase derivada nunca se ejecutará.*

La clase `logic_error` es la clase base de varias clases de excepciones estándar que indican errores en la lógica del programa. Por ejemplo, la clase `invalid_argument` indica que se pasó un argumento inválido a una función. (Sin duda, la codificación apropiada puede evitar que lleguen argumentos inválidos a una función). La clase `length_error` indica que para ese objeto se utilizó una longitud mayor que el tamaño máximo permitido para el objeto que se está manipulando. La clase `out_of_range` indica que un valor, como un subíndice en un arreglo, excedió su rango permitido de valores.

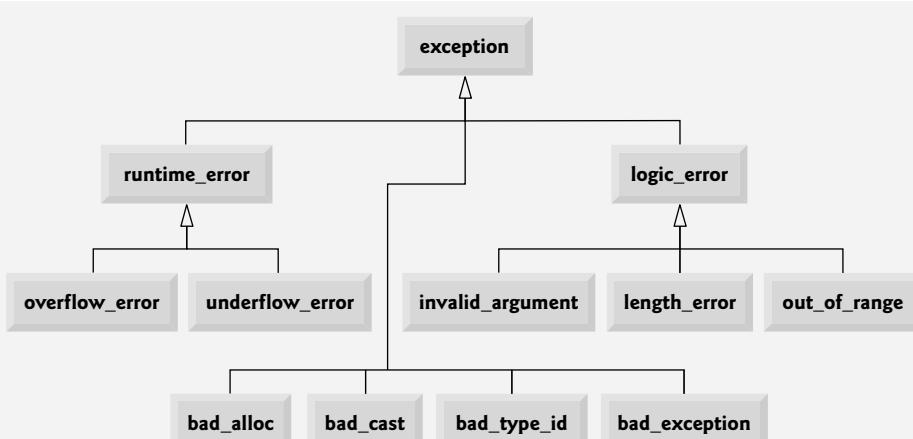


Figura 16.11 | Clases de excepciones de la Biblioteca estándar.

La clase `runtime_error`, que utilizamos brevemente en la sección 16.8, es la clase base de varias otras clases de excepciones estándar que indican errores en tiempo de ejecución. Por ejemplo, la clase `overflow_error` describe un **error de desbordamiento aritmético** (es decir, el resultado de una operación aritmética es mayor que el número más grande que se puede almacenar en la computadora) y la clase `underflow_error` describe un **error de subdesbordamiento** (es decir, el resultado de una operación aritmética es menor que el número más pequeño que se puede almacenar en la computadora).



### Error común de programación 16.9

*Las clases de excepciones definidas por el programador no necesitan derivarse de la clase exception. Por ende, al escribir catch( exception cualquierExcepcion ) no se garantiza que se atraparán todas las excepciones que un programa pudiera encontrar.*



### Tip para prevenir errores 16.6

*Para atrapar todas las excepciones que se puedan lanzar en un bloque try, use catch(...). Una debilidad al atrapar las excepciones de esta forma es que el tipo de las excepciones atrapadas se desconoce en tiempo de compilación. Otra debilidad es que, sin un parámetro con nombre, no hay forma de hacer referencia al objeto excepción dentro del manejador de la excepción.*



### Observación de Ingeniería de Software 16.9

*La jerarquía de exception estándar es un buen punto de inicio para crear excepciones. Los programadores pueden crear programas que puedan lanzar excepciones estándar, excepciones derivadas de las excepciones estándar o sus propias excepciones que no se deriven de las excepciones estándar.*



### Observación de Ingeniería de Software 16.10

*Use catch(...) para realizar una recuperación que no dependa del tipo de la excepción (por ejemplo, al liberar recursos comunes). La excepción se puede volver a lanzar para alertar a los manejadores catch circundantes más específicos.*



## 16.14 Otras técnicas para manejar errores

En capítulos anteriores hemos visto varias formas de tratar con situaciones excepcionales. A continuación se sintetizan éstas y otras técnicas para manejar errores:

- Ignorar la excepción. Si ocurre una excepción, el programa podría fallar como resultado de la excepción que no se atrapó. Esto es devastador para los productos de software comerciales y el software de misión crítica y propósito especial, pero para el software desarrollado para nuestros propios fines, es común ignorar muchos tipos de errores.



### Error común de programación 16.10

*Abortar un componente del programa debido a una excepción que no se atrapó podría dejar un recurso (como un flujo de archivo o un dispositivo de E/S) en un estado en el que otros programas no puedan adquirirlo. A esto se le conoce como fuga de recursos.*

- Abortar el programa. Sin duda, esto evita que un programa se ejecute hasta completarse y que produzca resultados incorrectos. Para muchos tipos de errores esta técnica es apropiada, en especial para los errores no fatales que permiten a un programa ejecutarse hasta completarse (con lo cual el programador podría pensar que el programa funciona correctamente). Esta estrategia es inapropiada para las aplicaciones de misión crítica. Las cuestiones sobre los recursos también son de importancia aquí; si un programa obtiene un recurso, debe liberarlo antes de terminar.
- Establecer indicadores de errores. El problema con esta metodología es que los programas tal vez no comprueben estos indicadores de errores en todos los puntos en los que los errores podrían resultar problemáticos. Otro problema es que el programa, después de procesar el problema, tal vez no borre los indicadores de errores.
- Probar la condición de error, generar un mensaje de error y llamar a `exit` (en `<cstdlib>`) para pasar un código de error apropiado al entorno del programa.

- Usar las funciones `setjump` y `longjump`. Estas funciones de la biblioteca <csetjmp> permiten al programador especificar un salto inmediato, desde una llamada a una función con muchos niveles de anidamiento hasta un manejador de errores. Sin usar `setjump` o `longjump`, un programa debe ejecutar varios retornos para salir de las llamadas a la función con muchos niveles de anidamiento. Las funciones `setjump` y `longjump` son peligrosas, ya que limpian la pila sin llamar a los destructores para los objetos automáticos. Esto puede provocar problemas graves.
- Ciertos tipos de errores tienen herramientas dedicadas para manejarlos. Por ejemplo, cuando el operador `new` no puede asignar memoria, se puede llamar a una función `new_handler` para manejar el error. Esta función se puede personalizar, para lo cual se suministra un nombre de función como argumento para `set_new_handler`, como vimos en la sección 16.11.

## 16.15 Repaso

En este capítulo el lector aprendió a usar el manejo de excepciones para lidiar con los errores en un programa. Aprendió que el manejo de excepciones permite a los programadores eliminar el código de manejo de errores de la “línea principal” de ejecución del programa. Demostramos el manejo de excepciones en el contexto de un ejemplo de división entre cero. También mostramos cómo usar bloques `try` para encerrar código que puede lanzar una excepción, y cómo usar manejadores `catch` para lidiar con las excepciones que puedan surgir. Aprendió a lanzar y volver a lanzar excepciones, y cómo manejar las excepciones que ocurren en los constructores. El capítulo continuó con discusiones acerca del procesamiento de las fallas de `new`, la asignación dinámica de memoria con la clase `auto_ptr` y la jerarquía de excepciones de la biblioteca estándar. En el siguiente capítulo aprenderá acerca del procesamiento de archivos, incluyendo la forma en que se almacenan los datos persistentes, y cómo se manipulan.

## Resumen

### Sección 16.1 Introducción

- Una excepción es una indicación de un problema que ocurre durante la ejecución de un programa.
- El manejo de excepciones nos permite crear programas que pueden resolver los problemas que ocurren en tiempo de ejecución; por lo general esto permite a los programas continuar su ejecución, como si no se hubiera encontrado ningún problema. Los problemas más graves pueden requerir que un programa notifique al usuario acerca del problema, antes de terminar de una manera controlada.

### Sección 16.2 Generalidades acerca del manejo de excepciones

- El manejo de excepciones permite al programador eliminar el código de manejo de errores de la “línea principal” de ejecución del programa, lo cual mejora la claridad del programa y su capacidad de modificación.

### Sección 16.3 Ejemplo: manejo de un intento de dividir entre cero

- La clase `exception` es la clase base estándar de C++ para las excepciones. La clase `exception` proporciona la función virtual `what`, que devuelve un mensaje de error apropiado y puede sobrescribirse en las clases derivadas.
- La clase `runtime_error` (definida en el encabezado <stdexcept>) es la clase base estándar de C++ para representar los errores en tiempo de ejecución.
- C++ utiliza el modelo de terminación del manejo de excepciones.
- Un bloque `try` consiste en la palabra clave `try`, seguida de llaves `{}` que definen un bloque de código en el que podrían ocurrir excepciones. El bloque `try` encierra instrucciones que podrían producir excepciones, e instrucciones que no deben ejecutarse si se producen excepciones.
- Por lo menos debe haber un manejador `catch` justo después de un bloque `try`. Cada manejador `catch` especifica un parámetro de excepción que representa el tipo de excepción que el manejador `catch` puede procesar.
- Si un parámetro de excepción incluye un nombre de parámetro opcional, el manejador `catch` puede usar ese nombre de parámetro para interactuar con un objeto excepción atrapado.
- El punto en el programa en el que ocurre una excepción se conoce como el punto de lanzamiento.
- Si ocurre una excepción en un bloque `try`, este bloque expira y el control del programa se transfiere al primer `catch` en el que coincide el tipo del parámetro de excepción con el de la excepción lanzada.
- Cuanto un bloque `try` termina, las variables locales definidas en el bloque quedan fuera de alcance.
- Cuando un bloque `try` termina debido a una excepción, el programa busca el primer manejador `catch` que pueda procesar el tipo de excepción que ocurrió. Para localizar el `catch` que coincide, el programa compara el tipo de la excepción lanza-

da con el tipo del parámetro de excepción de cada catch hasta que el programa encuentra una coincidencia. Ocurre una coincidencia si los tipos son idénticos, o si el tipo de la excepción lanzada es una clase derivada del tipo del parámetro de excepción. Cuando ocurre una coincidencia, se ejecuta el código contenido dentro del manejador catch que coincide.

- Cuando un manejador catch termina su procesamiento, el parámetro de catch y las variables locales definidas dentro del manejador catch quedan fuera de alcance. Cualquier manejador catch restante que corresponda al bloque try se ignora, y la ejecución se reanuda en la primera línea de código después de la secuencia try...catch.
- Si no ocurren excepciones en un bloque try, el programa ignora el (los) manejador(es) catch para ese bloque. La ejecución del programa se reanuda con la siguiente instrucción después de la secuencia try...catch.
- Si una excepción que ocurre en un bloque try no tiene un manejador catch que coincida, o si ocurre una excepción en una instrucción que no esté en un bloque try, la función que contiene la instrucción termina de inmediato y el programa trata de localizar un bloque try circundante en la función que hizo la llamada. A este proceso se le conoce como limpieza de la pila.
- Para lanzar una excepción, use la palabra clave throw seguida de un operando que representa el tipo de excepción a lanzar. El operando de una instrucción throw puede ser de cualquier tipo.

#### **Sección 16.4 Cuándo utilizar el manejo de excepciones**

- El manejo de excepciones es para errores sincrónicos, que ocurren cuando se ejecuta una instrucción.
- El manejo de excepciones no está diseñado para procesar los errores asociados con los eventos asíncronos, que ocurren en paralelo con (y de manera independiente a) el flujo de control del programa.

#### **Sección 16.5 Volver a lanzar una excepción**

- El manejador de excepciones puede diferir el manejo de una excepción (o tal vez una porción de ésta) a otro manejador de excepciones. En cualquier caso, para lograr esto el manejador vuelve a lanzar la excepción.
- Algunos ejemplos comunes de excepciones son los subíndices de arreglos fuera de rango, el desbordamiento aritmético, la división entre cero, los parámetros inválidos de funciones y las asignaciones de memoria fallidas.

#### **Sección 16.6 Especificaciones de excepciones**

- Una especificación de excepciones opcional enumera una lista de excepciones que una función puede lanzar. Una función sólo puede lanzar excepciones de los tipos indicados por la especificación de excepciones, o excepciones de cualquier tipo que se derive de estos tipos. Si una función lanza una excepción que no pertenezca a uno de los tipos especificados, se hace una llamada a la función unexpected y el programa termina.
- Una función sin especificación de excepciones puede lanzar cualquier excepción. La especificación de excepciones vacía throw() indica que una función no lanza excepciones. Si una función con una especificación de excepciones vacía intenta lanzar una excepción, se invoca la función unexpected.

#### **Sección 16.7 Procesamiento de excepciones inesperadas**

- La función unexpected llama a la función registrada con la función set\_unexpected. Si no se ha registrado ninguna función de esta forma, se hace una llamada a la función terminate de manera predeterminada.
- La función set\_terminate puede especificar la función a invocar cuando se hace la llamada a terminate. En caso contrario, terminate llama a abort, la cual termina el programa sin llamar a los destructores de los objetos declarados como static y auto.
- Las funciones set\_terminate y set\_unexpected devuelven un apuntador a la última función llamada por terminate y unexpected, respectivamente (0, la primera vez que se llama a cada una de ellas). Esto permite al programador guardar el apuntador de funciones, para poder restaurarlo más adelante.
- Las funciones set\_terminate y set\_unexpected reciben como argumentos apuntadores a funciones con tipos de valores de retorno void y sin argumentos.
- Si una función de terminación definida por el programador no sale de un programa, se hará una llamada a la función abort después de que la función de terminación definida por el programador termine de ejecutarse.

#### **Sección 16.8 Limpieza de la pila**

- Limpiar la pila de llamadas a funciones significa que la función en la que la excepción no se atrapó termina, todas las variables locales en esa función se destruyen y el control regresa a la instrucción que invocó originalmente a esa función.

#### **Sección 16.9 Constructores, destructores y manejo de excepciones**

- Las excepciones lanzadas por un constructor hacen que se llame a los destructores de todos los objetos creados como parte del objeto que se está construyendo, antes de que se lance la excepción.
- Cada objeto automático construido en un bloque try se destruye antes de lanzar una excepción.
- La limpieza de la pila se completa antes de que un manejador de excepciones empiece a ejecutarse.
- Si un destructor invocado como resultado de la limpieza de la pila lanza una excepción, se hace una llamada a terminate.

- Si un objeto tiene objetos miembros, y si se lanza una excepción antes de que el objeto exterior esté completamente construido, entonces se ejecutarán los destructores de los objetos miembro que se hayan construido antes de que ocurra la excepción.
- Si se ha construido parcialmente un arreglo de objetos cuando ocurre una excepción, sólo se llamará a los destructores para los objetos elementos del arreglo que estén construidos.
- Cuando se lanza una excepción desde el constructor para un objeto que se crea en una expresión `new`, se libera la memoria asignada en forma dinámica para ese objeto.

### Sección 16.10 Excepciones y herencia

- Si un manejador `catch` atrapa un apuntador o referencia a un objeto excepción del tipo de una clase base, también puede atrapar un apuntador o referencia a todos los objetos de las clases que se deriven públicamente de esa clase base; esto permite el procesamiento polimórfico de los errores relacionados.

### Sección 16.11 Procesamiento de las fallas de `new`

- El documento del estándar de C++ especifica que, cuando falla el operador `new`, lanza una excepción `bad_alloc` (definida en el archivo de encabezado `<new>`).
- La función `set_new_handler` recibe como argumento un apuntador a una función que no recibe argumentos y devuelve `void`. Este apuntador apunta a la función que se llamará en caso de que falle `new`.
- Una vez que `set_new_handler` registra un manejador de `new` en el programa, el operador `new` no lanza `bad_alloc` al fallar; en vez de ello difiere el manejo del error a la función manejadora de `new`.
- Si `new` asigna memoria con éxito, devuelve un apuntador a esa memoria.
- Si ocurre una excepción después de una asignación exitosa de memoria, pero antes de que se ejecute la instrucción `delete`, podría ocurrir una fuga de memoria.

### Sección 16.12 La clase `auto_ptr` y la asignación dinámica de memoria

- La Biblioteca estándar de C++ proporciona la plantilla de clase `auto_ptr` para lidiar con las fugas de memoria.
- Un objeto de la clase `auto_ptr` mantiene un apuntador a la memoria asignada en forma dinámica. El destructor de un objeto `auto_ptr` realiza una operación `delete` en el miembro de datos apuntador del objeto `auto_ptr`.
- La plantilla de clase `auto_ptr` proporciona los operadores sobrecargados `*` y `->`, para que se pueda utilizar un objeto `auto_ptr` de la misma manera que una variable apuntador ordinaria. Un objeto `auto_ptr` también transfiere la propiedad de la memoria dinámica que maneja a través de su constructor de copia y su operador de asignación sobrecargado.

### Sección 16.13 Jerarquía de excepciones de la Biblioteca estándar

- La Biblioteca estándar de C++ incluye una jerarquía de clases de excepciones. Esta jerarquía está encabezada por la clase base `exception`.
- Las clases derivadas inmediatas de la clase base `exception` son `runtime_error` y `logic_error` (ambas definidas en el encabezado `<stdexcept>`), cada una de las cuales tiene varias clases derivadas.
- Varios operadores lanzan excepciones estándar; el operador `new` lanza `bad_alloc`, el operador `dynamic_cast` lanza `bad_cast` y el operador `typeid` lanza `bad_typeid`.
- Incluir `bad_exception` en la lista `throw` de una función significa que, si ocurre una excepción inesperada, la función `unexpected` puede lanzar a `bad_exception` en vez de terminar la ejecución del programa, o llamar a otra función especificada por `set_unexpected`.

## Terminología

|                                            |                                                        |
|--------------------------------------------|--------------------------------------------------------|
| <code>abort</code> , función               | errores síncronos                                      |
| aplicación robusta                         | especificación de excepciones                          |
| atrapar todas las excepciones              | especificación de excepciones vacía                    |
| atrapar una excepción                      | evento asíncrono                                       |
| <code>auto_ptr</code> , plantilla de clase | excepción                                              |
| <code>bad_alloc</code> , excepción         | <code>&lt;exception&gt;</code> , archivo de encabezado |
| <code>bad_cast</code> , excepción          | <code>exception</code> , clase                         |
| <code>bad_exception</code> , excepción     | <code>invalid_argument</code> , excepción              |
| <code>bad_typeid</code> , excepción        | lanzar una excepción                                   |
| <code>catch(...)</code>                    | lanzar una excepción inesperada                        |
| <code>catch</code> , manejador             | <code>length_error</code> , excepción                  |
| <code>catch</code> , palabra clave         | limpieza de la pila                                    |
| error de desbordamiento aritmético         | <code>logic_error</code> , excepción                   |
| error de subdesbordamiento aritmético      | manejador de excepciones                               |

|                                                     |                                                                        |
|-----------------------------------------------------|------------------------------------------------------------------------|
| manejar una excepción                               | <code>set_new_handler</code> , función                                 |
| manejo de excepciones                               | <code>set_terminate</code> , función                                   |
| <code>&lt;memory&gt;</code> , archivo de encabezado | <code>set_unexpected</code> , función                                  |
| modelo de reanudación del manejo de excepciones     | <code>&lt;stdexcept&gt;</code> , archivo de encabezado                 |
| modelo de terminación del manejo de excepciones     | <code>terminate</code> , función                                       |
| <code>new</code> , manejador de fallas              | <code>throw</code> , lista                                             |
| <code>nothrow</code> , objeto                       | <code>throw</code> , palabra clave                                     |
| objeto excepción                                    | <code>throw</code> sin argumentos                                      |
| <code>out_of_range</code> , excepción               | <code>try</code> , bloque                                              |
| <code>overflow_error</code> , excepción             | <code>try</code> , palabra clave                                       |
| parámetro de excepción                              | <code>underflow_error</code> , excepción                               |
| programas tolerantes a errores                      | <code>unexpected</code> , función                                      |
| punto de lanzamiento                                | volver a lanzar una excepción                                          |
| <code>runtime_error</code> , excepción              | <code>what</code> , función virtual de la clase <code>exception</code> |

## Ejercicios de autoevaluación

- 16.1** Liste cinco ejemplos comunes de excepciones.
- 16.2** Dé varias razones por las que las técnicas de manejo de excepciones no se deben utilizar para el control convencional de un programa.
- 16.3** ¿Por qué son las excepciones apropiadas para lidiar con los errores producidos por las funciones de biblioteca?
- 16.4** ¿Qué es una “fuga de recursos”?
- 16.5** Si no se lanzan excepciones en un bloque `try`, ¿a dónde procede el control después de que el bloque `try` termina de ejecutarse?
- 16.6** ¿Qué ocurre si se lanza una excepción fuera de un bloque `try`?
- 16.7** Dé una ventaja clave y una desventaja clave de utilizar `catch(...)`.
- 16.8** ¿Qué ocurre si ningún manejador `catch` coincide con el tipo de un objeto lanzado?
- 16.9** ¿Qué ocurre si varios manejadores coinciden con el tipo del objeto lanzado?
- 16.10** ¿Por qué un programador debe especificar un tipo de clase base como el tipo de un manejador `catch`, y después lanzar objetos de tipos de clases derivadas?
- 16.11** Suponga que está disponible un manejador `catch` que coincide exactamente con el tipo de un objeto excepción. ¿Bajo qué circunstancias podría ejecutarse un manejador diferente para los objetos excepción de ese tipo?
- 16.12** ¿Al lanzar una excepción se debe terminar el programa?
- 16.13** ¿Qué ocurre cuando un manejador `catch` lanza una excepción?
- 16.14** ¿Qué hace la instrucción `throw`?
- 16.15** ¿Cómo se restringen los tipos de excepciones que puede lanzar una función?
- 16.16** ¿Qué ocurre si una función lanza una excepción de un tipo no permitido por la especificación de excepciones para la función?
- 16.17** ¿Qué ocurre con los objetos automáticos que se han construido en un bloque `try` cuando ese bloque lanza una excepción?

## Respuestas a los ejercicios de autoevaluación

- 16.1** Memoria insuficiente para satisfacer una petición de `new`, subíndice de arreglo fuera de límites, desbordamiento aritmético, división entre cero, parámetros de función inválidos.
- 16.2** (a) El manejo de excepciones está diseñado para manejar las situaciones que ocurren con poca frecuencia, y que a menudo provocan que el programa termine, de manera que los escritores de los compiladores no tengan que implementar el manejo de excepciones para obtener un rendimiento óptimo. (b) Por lo general, el flujo de control con las estructuras de control convencionales es más claro y eficiente que con las excepciones. (c) Pueden ocurrir problemas debido a que la pila se limpia cuando ocurre una excepción, y tal vez no se liberen los recursos asignados antes de la excepción. (d) Las excepciones “adicionales” hacen que sea más difícil para el programador manejar el número más grande de excepciones.
- 16.3** Es improbable que una función de biblioteca realice el procesamiento de errores que cumplan con las necesidades específicas de cada usuario.

**16.4** Un programa que termina en forma abrupta podría dejar un recurso en un estado en el que otros programas no puedan adquirirlo, o el programa en sí no podría readquirir un recurso “en fuga”.

**16.5** Los manejadores de excepciones (en los manejadores `catch`) para ese bloque `try` se omiten, y el programa reanuda su ejecución después del último manejador `catch`.

**16.6** Una excepción lanzada fuera de un bloque `try` provoca una llamada a `terminate`.

**16.7** La forma `catch(...)` atrapa cualquier tipo de excepción lanzada en un bloque `try`. Una ventaja es que se atraparán todas las posibles excepciones. Una desventaja es que `catch` no tiene parámetro, por lo que no puede hacer referencia a la información en el objeto lanzado y no puede conocer la causa de la excepción.

**16.8** Esto hace que la búsqueda de una coincidencia continúe en el siguiente bloque `try` circundante, si hay uno. A medida que continúa este proceso, podría determinarse en un momento dado que no hay un manejador en el programa que coincida con el tipo del objeto lanzado; en este caso se llama a `terminate`, que de manera predeterminada llama a `abort`. Se puede proporcionar una función `terminate` alternativa como argumento para `set_terminate`.

**16.9** El primer manejador de excepciones que coincide después del bloque `try` se ejecuta.

**16.10** Ésa es una excelente manera de atrapar tipos relacionados de excepciones.

**16.11** Un manejador de la clase base atraparía objetos de todos los tipos de clases derivadas.

**16.12** No, pero termina el bloque en el que se lanza la excepción.

**16.13** La excepción será procesada por un manejador `catch` (si existe) asociado con el bloque `try` (si existe) que encierra al manejador `catch` que provocó la excepción.

**16.14** Vuelve a lanzar la excepción si aparece en un manejador `catch`; en caso contrario, se hace una llamada a la función `unexpected`.

**16.15** Proporciona una especificación de excepciones que lista los tipos de excepciones que puede lanzar la función.

**16.16** Se hace una llamada a la función `unexpected`.

**16.17** El bloque `try` expira, provocando que se llame a los destructores para cada uno de estos objetos.

## Ejercicios

**16.18** Liste varias condiciones excepcionales que han ocurrido a lo largo del libro. Liste todas las condiciones excepcionales adicionales que pueda. Para cada una de estas excepciones, describa brevemente cómo manejaría generalmente un programa la excepción, usando las técnicas para manejo de excepciones descritas en este capítulo. Algunas excepciones comunes son la división entre cero, el desbordamiento aritmético, el subíndice de un arreglo fuera de límites, agotamiento del almácén de memoria libre, etcétera.

**16.19** ¿Bajo qué circunstancias no se debe proporcionar un nombre de parámetro al definir el tipo del objeto que será atrapado por un manejador?

**16.20** Un programa contiene la siguiente instrucción:

```
throw;
```

¿En dónde se esperaría encontrar comúnmente dicha instrucción? ¿Qué pasaría si esa instrucción apareciera en una parte distinta del programa?

**16.21** Compare y contraste el manejo de excepciones con los otros esquemas de procesamiento de errores descritos en el texto.

**16.22** ¿Por qué no deben usarse las excepciones como una forma alterna de control del programa?

**16.23** Describa una técnica para manejar las excepciones relacionadas.

**16.24** Hasta este capítulo, hemos encontrado que puede ser complicado lidiar con los errores detectados por los constructores. El manejo de excepciones nos proporciona un mejor medio de manejar dichos errores. Considere un constructor para la clase `String`. El constructor utiliza `new` para obtener espacio del almácén de memoria libre. Suponga que `new` falla. Muestre cómo lidiaría con esto sin el manejo de excepciones. Hable sobre las cuestiones clave. Muestre cómo lidiaría con dicho agotamiento de memoria con el manejo de excepciones. Explique por qué la metodología de manejo de excepciones es superior.

**16.25** Suponga que un programa lanza una excepción y que se empieza a ejecutar el manejador de excepciones apropiado. Ahora suponga que el manejador de excepciones en sí lanza la misma excepción. ¿Crea esto una recursividad infinita? Escriba un programa para comprobar su observación.

- 16.26** Use la herencia para crear varias clases derivadas de `runtime_error`. Después muestre que un manejador `catch` que especifique la clase base puede atrapar excepciones de las clases derivadas.
- 16.27** Escriba una expresión condicional que devuelva un valor `double` o un `int`. Proporcione un manejador `catch int` y un manejador `catch double`. Muestre que sólo se ejecuta el manejador `catch double`, sin importar que se devuelva el valor `int` o `double`.
- 16.28** Escriba un programa que genere y maneje una excepción por agotamiento de memoria. Su programa debe iterar en una petición para crear memoria dinámica a través del operador `new`.
- 16.29** Escriba un programa que ilustre que todos los destructores para los objetos construidos en un bloque se llaman antes de que se lance una excepción desde ese bloque.
- 16.30** Escriba un programa que ilustre que se llama a los destructores de los objetos miembro, sólo para aquellos objetos miembro que se construyeron antes de que ocurriera una excepción.
- 16.31** Escriba un programa que demuestre cómo se atrapan varios tipos de excepciones con el manejador de excepciones `catch(...)`.
- 16.32** Escriba un programa que ilustre que el orden de los manejadores de excepciones es importante. El primer manejador que coincida es el que se ejecuta. Trate de compilar y ejecutar su programa de dos maneras distintas, para mostrar que se ejecutan dos manejadores distintos con dos efectos diferentes.
- 16.33** Escriba un programa que muestre cómo un constructor pasa información acerca de la falla del constructor a un manejador de excepciones después de un bloque `try`.
- 16.34** Escriba un programa que ilustre cómo volver a lanzar una excepción.
- 16.35** Escriba un programa que ilustre que una función con su propio bloque `try` no tiene que atrapar todos los posibles errores generados dentro del bloque `try`. Algunas excepciones se pueden escabullir hasta (y ser manejadas en) los alcances exteriores.
- 16.36** Escriba un programa que lance una excepción desde una función con muchos niveles de anidamiento, y que de todas formas el manejador `catch` que sigue después del bloque `try` que encierra la cadena de llamadas atrape la excepción.



*Leí una parte en su totalidad.*

—Samuel Goldwyn

*Una gran memoria no hace a un filósofo; cualquier cosa que sea más que un diccionario se le puede llamar gramática.*

—John Henry, Cardenal Newman

*Sólo puedo suponer que un documento “No archivar” se archiva en un archivo “No archivar”.*

—Senador Frank Church  
Audencia del subcomité de inteligencia del Senado, 1975

# Procesamiento de archivos

En este capítulo aprenderá a:

- Crear, leer, escribir y actualizar archivos.
- Procesar archivos secuenciales.
- Procesar archivos de acceso aleatorio.
- Utilizar operaciones de E/S sin formato de alto rendimiento.
- Conocer las diferencias entre los datos con formato y el procesamiento de archivos de datos puros.
- Crear un programa para procesar transacciones, usando el procesamiento de archivos de acceso aleatorio.

**Plan general**

- 17.1** Introducción
- 17.2** Jerarquía de datos
- 17.3** Archivos y flujos
- 17.4** Creación de un archivo secuencial
- 17.5** Cómo leer datos de un archivo secuencial
- 17.6** Actualización de archivos secuenciales
- 17.7** Archivos de acceso aleatorio
- 17.8** Creación de un archivo de acceso aleatorio
- 17.9** Cómo escribir datos al azar a un archivo de acceso aleatorio
- 17.10** Cómo leer de un archivo de acceso aleatorio en forma secuencial
- 17.11** Ejemplo práctico: un programa para procesar transacciones
- 17.12** Generalidades acerca de la serialización de objetos
- 17.13** Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)

## 17.1 Introducción

El almacenamiento de datos en variables y arreglos es temporal. Los **archivos** se utilizan para la **persistencia de los datos**: datos de retención permanente. Las computadoras almacenan archivos en **dispositivos de almacenamiento secundario** como discos duros, CDs, DVDs, unidades flash y cintas magnéticas. En este capítulo explicaremos cómo construir programas en C++ para crear, actualizar y procesar archivos de datos. Consideraremos los archivos secuenciales y los archivos de acceso aleatorio. Compararemos el procesamiento de archivos de datos con formato y el procesamiento de archivos de datos puros. Examinaremos las técnicas para introducir datos de, y enviar datos a flujos `string` en vez de archivos en el capítulo 18, La clase `string` y el procesamiento de flujos de cadena.

## 17.2 Jerarquía de datos

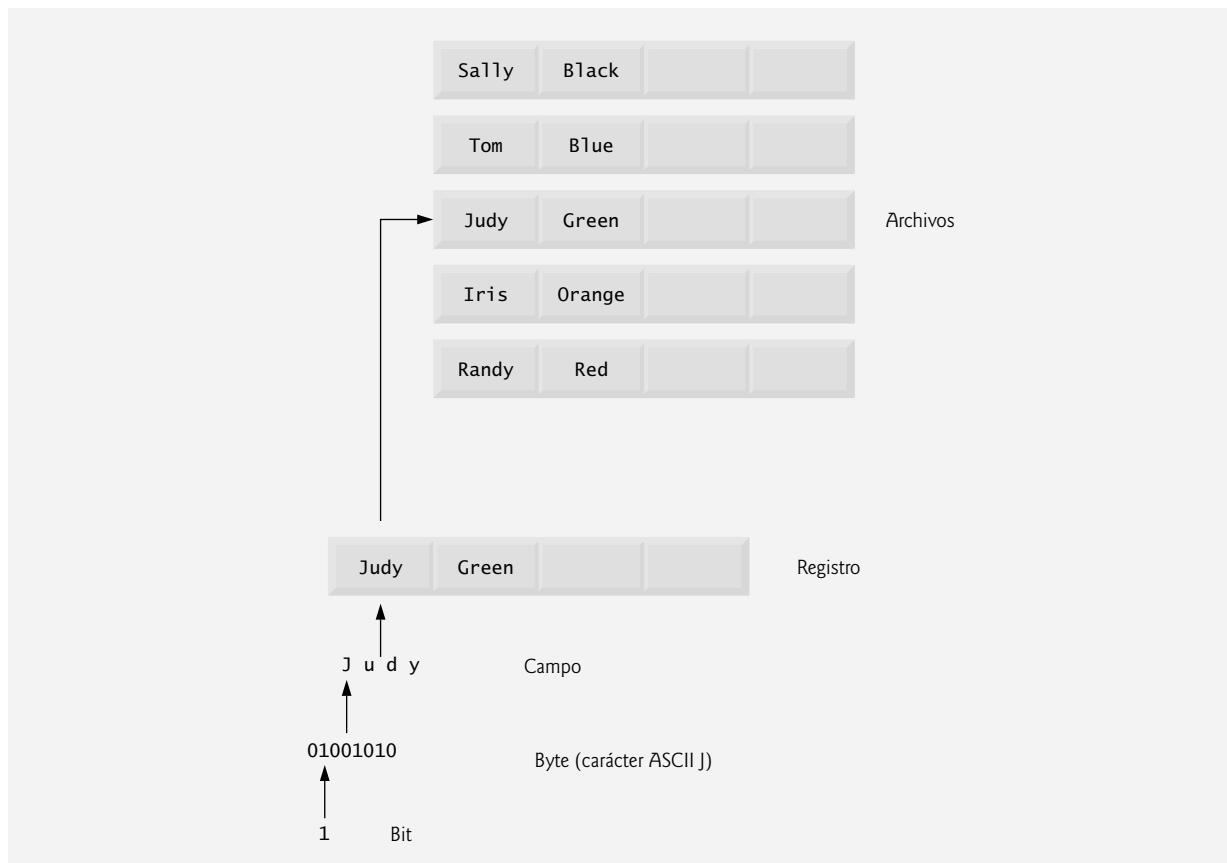
Finalmente, todos los elementos de datos que procesan las computadoras digitales se reducen a combinaciones de ceros y unos. Esto ocurre debido a que es simple y económico construir dispositivos electrónicos que puedan asumir uno de dos estados estables: un estado representa 0 y el otro representa 1. Es increíble que las impresionantes funciones realizadas por las computadoras impliquen solamente las manipulaciones más fundamentales de 0s y 1s.

El elemento más pequeño de datos que soportan las computadoras se conoce como **bit** (abreviatura de “**dígito binario**”; un dígito que puede suponer uno de dos valores). Cada elemento de datos, o bit, puede asumir el valor 0 o el valor 1. Los circuitos de computadora realizan varias manipulaciones simples de bits, como examinar o establecer el valor de un bit, o invertir su valor (de 1 a 0 o de 0 a 1).

Es muy difícil para los programadores trabajar con datos en el formato de bits de bajo nivel. Es preferible trabajar con datos en formatos como **dígitos decimales** (0-9), **letras** (A-Z y a-z) y **símbolos especiales** (por ejemplo, \$, @, %, &, \* y muchos otros). Los dígitos, letras y símbolos especiales se conocen como **caracteres**. El **conjunto de caracteres** de una computadora es el conjunto de todos los caracteres utilizados para escribir programas y representar elementos de datos de esa computadora. Las computadoras pueden procesar solamente 1s y 0s, por lo que cada carácter en el conjunto de caracteres de una computadora se representa como un patrón de 1s y 0s. Los **bytes** están compuestos de ocho bits. Los programadores crean programas y elementos de datos con caracteres; las computadoras manipulan y procesan estos caracteres como patrones de bits. Por ejemplo, C++ proporciona el tipo de datos `char`. Cada `char` por lo general ocupa un byte. C++ también proporciona el tipo de datos `wchar_t`, que puede ocupar más de un byte (para soportar conjuntos de caracteres más grandes, como el **conjunto de caracteres Unicode®**; para obtener más información acerca de Unicode®, visite el sitio [www.unicode.org](http://www.unicode.org)).

Así como los caracteres están compuestos de bits, los **campos** están compuestos de caracteres. Un campo es un grupo de caracteres que transmiten cierto significado. Por ejemplo, un campo que consiste de letras mayúsculas y minúsculas puede representar el nombre de una persona.

Los elementos de datos que son procesados por las computadoras forman una **jerarquía de datos** (figura 17.1), en la cual los elementos de datos se hacen más grandes y complejos en estructura, a medida que progresamos de bits a caracteres, de caracteres a campos, hacia agregados de datos más grandes.



**Figura 17.1** | Jerarquía de datos.

Generalmente, un **registro** (que puede ser representado como una clase en C++) está compuesto de varios **campos** (conocidos como miembros de datos en C++). Por ejemplo, en un sistema de nóminas el registro para un empleado específico podría incluir los siguientes campos:

1. Número de identificación del empleado
2. Nombre
3. Dirección
4. Sueldo por hora
5. Número de exenciones reclamadas
6. Ingresos desde inicio de año a la fecha
7. Monto de impuestos retenidos

Por lo tanto, un registro es un grupo de campos relacionados. En el ejemplo anterior, cada campo está asociado al mismo empleado. Un archivo es un grupo de registros relacionados.<sup>1</sup> El archivo de nómina de una compañía generalmente contiene un registro para cada empleado. Por ejemplo, un archivo de nómina para una pequeña compañía podría contener sólo 22 registros, mientras que un archivo de nómina para una compañía grande podría contener 100,000.

1. En general, un archivo puede contener datos arbitrarios en formatos arbitrarios. En algunos sistemas operativos, un archivo se ve simplemente como una colección de bytes. En dicho sistema operativo, cualquier organización de los bytes en un archivo (como organizar los datos en registros) es una vista creada por el programador de aplicaciones.

registros. Es común para una compañía tener muchos archivos, algunos de ellos contenido miles de millones, o incluso billones de caracteres de información.

Para facilitar la recuperación de registros específicos de un archivo, debe seleccionarse cuando menos un campo en cada registro como **clave de registro**. Una clave de registro sirve para identificar que un registro pertenece a una persona o entidad específica, y es única en cada registro. En el registro de nómina que describimos anteriormente, por lo general se elegirá el número de identificación de empleado como clave de registro.

Existen muchas formas de organizar los registros en un archivo. Un tipo común de organización se conoce como **archivo secuencial**, en el cual los registros se almacenan en orden, con base en el campo que es la clave de registro. En un archivo de nómina, los registros generalmente se colocan en orden, con base en el número de identificación de empleado. El registro del primer empleado en el archivo contiene el número de identificación de empleado más pequeño, y los registros subsiguientes contienen números cada vez mayores.

La mayoría de las empresas utilizan muchos archivos distintos para almacenar datos. Por ejemplo, una compañía podría tener archivos de nómina, de cuentas por cobrar (listas del dinero que deben los clientes), de cuentas por pagar (listas del dinero que se debe a los proveedores), archivos de inventarios (listas de información acerca de los artículos que maneja la empresa) y muchos otros tipos de archivos. A menudo, un grupo de archivos relacionados se almacena en una **base de datos**. A una colección de programas diseñada para crear y administrar bases de datos se le conoce como **sistema de administración de bases de datos (DBMS)**.

## 17.3 Archivos y flujos

C++ considera a cada archivo como una secuencia de bytes (figura 17.2). Cada archivo termina con un **marcador de fin de archivo** o con un número de bytes específico que se registra en una estructura de datos administrativa, mantenida por el sistema. Cuando se *abre* un archivo, se crea un objeto y se asocia un flujo a ese objeto. En el capítulo 15 vimos que los objetos `cin`, `cout`, `cerr` y `clog` se crean cuando se incluye `<iostream>`. Los flujos asociados con estos objetos proporcionan canales de comunicación entre un programa y un archivo o dispositivo específico. Por ejemplo, el objeto `cin` (objeto flujo de entrada estándar) permite a un programa introducir datos desde el teclado o desde otros dispositivos, el objeto `cout` (objeto flujo de salida estándar) permite a un programa enviar datos a la pantalla o a otros dispositivos, y los objetos `cerr` y `clog` (objetos flujo de error estándar) permiten a un programa enviar mensajes de error a la pantalla o a otros dispositivos.



**Figura 17.2** | La manera en que C++ ve a un archivo de  $n$  bytes.

Para llevar a cabo el procesamiento de archivos en C++, se deben incluir los archivos de encabezado `<iostream>` y `<fstream>`. El encabezado `<fstream>` incluye las definiciones para las plantillas de clases de flujos `basic_ifstream` (para las operaciones de entrada con archivos), `basic_ofstream` (para las operaciones de salida con archivos) y `basic_fstream` (para las operaciones de entrada y salida con archivos). Cada plantilla de clase tiene una especialización de plantilla predefinida que permite las operaciones de E/S con valores `char`. Además, la biblioteca `<fstream>` proporciona alias `typedef` para estas especializaciones de plantilla. Por ejemplo, la definición `typedef istream` representa a una especialización de `basic_ifstream` que permite la entrada de valores `char` desde un archivo. De manera similar, `typedef ofstream` representa una especialización de `basic_ofstream` que permite enviar valores `char` a archivos. Además, `typedef fstream` representa una especialización de `basic_fstream` que permite introducir valores `char` desde (y enviarlos hacia) archivos.

Para abrir los archivos, se crean objetos de estas especializaciones de plantillas de flujo. Estas plantillas se “derivan” de las plantillas de clases `basic_istream`, `basic_ostream` y `basic_iostream`, respectivamente. Por ende, todas las funciones miembro, operadores y manipuladores que pertenecen a estas plantillas (y que describimos en el capítulo 15) también se pueden aplicar a los flujos de archivos. En la figura 17.3 se sintetizan las relaciones de herencia de las clases de E/S que hemos visto hasta este punto.

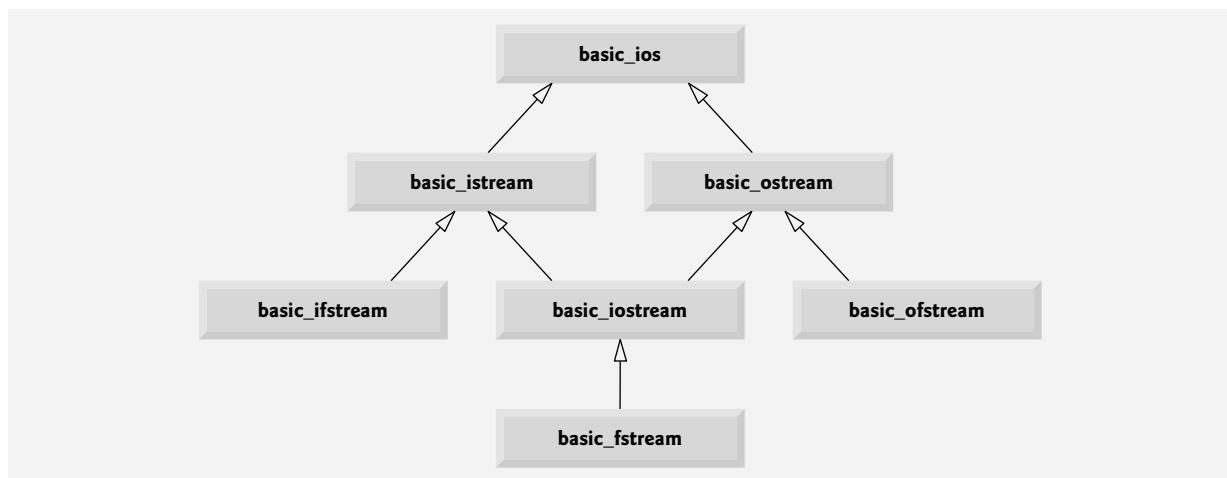


Figura 17.3 | Porción de la jerarquía de plantillas de E/S de flujos.

## 17.4 Creación de un archivo secuencial

C++ no impone una estructura sobre un archivo. Por ende, en un archivo de C++ no existe un concepto tal como el de un “registro”. En consecuencia, el programador debe estructurar los archivos de manera que cumplan con los requerimientos de la aplicación. En el siguiente ejemplo veremos cómo se puede imponer una estructura de registro simple sobre un archivo.

La figura 17.4 crea un archivo secuencial que podría utilizarse en un archivo de cuentas por cobrar, para ayudar a administrar el dinero que deben los clientes de crédito a una empresa. Para cada cliente, el programa obtiene su número de cuenta, nombre y saldo (es decir, el monto que el cliente debe a la empresa por los bienes y servicios recibidos en el pasado). Los datos que se obtienen para cada cliente constituyen un registro para ese cliente. El número de cuenta sirve como la clave de registro; es decir, el programa crea y da mantenimiento al archivo en orden por número de cuenta. Este programa supone que el usuario introduce los registros en orden por número de cuenta. En un sistema de cuentas por cobrar completo, sería conveniente contar con una herramienta para ordenar datos, para que el usuario pueda introducir los registros en cualquier orden; después los registros se ordenarían y escribirían en el archivo.

```

1 // Fig. 17.4: Fig17_04.cpp
2 // Creación de un archivo secuencial.
3 #include <iostream>
4 using std::cerr;
5 using std::cin;
6 using std::cout;
7 using std::endl;
8 using std::ios;
9
10 #include <fstream> // flujo de archivo
11 using std::ofstream; // flujo de archivo de salida
12
13 #include <cstdlib>
14 using std::exit; // prototipo de la función exit
15
16 int main()
17 {
18 // el constructor de ofstream abre el archivo
19 ofstream archivoClientesSalida("clientes.dat", ios::out);
20
21 // sale del programa si no puede crear el archivo
22 if (!archivoClientesSalida) // operador ! sobrecargado
23 {

```

Figura 17.4 | Creación de un archivo secuencial. (Parte I de 2).

```

24 cerr << "No se pudo abrir el archivo" << endl;
25 exit(1);
26 } // fin de if
27
28 cout << "Escriba la cuenta, nombre y saldo." << endl
29 << "Escriba fin de archivo para terminar la entrada.\n? ";
30
31 int cuenta;
32 char nombre[30];
33 double saldo;
34
35 // lee la cuenta, nombre y saldo de cin, y después los coloca en el archivo
36 while (cin >> cuenta >> nombre >> saldo)
37 {
38 archivoClientesSalida << cuenta << ' ' << nombre << ' ' << saldo << endl;
39 cout << "? ";
40 } // fin de while
41
42 return 0; // el destructor de ofstream cierra el archivo
43 } // fin de main

```

```

Escriba la cuenta, nombre y saldo.
Escriba fin de archivo para terminar la entrada.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z

```

**Figura 17.4** | Creación de un archivo secuencial. (Parte 2 de 2).

Vamos a examinar este programa. Como se dijo antes, para abrir los archivos se crean objetos `ifstream`, `ofstream` o `fstream`. En la figura 17.4, el archivo se va a abrir para salida, por lo que se crea un objeto `ofstream`. Se pasan dos argumentos al constructor del objeto: el **nombre de archivo** y el **modo de apertura de archivo** (línea 19). Para un objeto `ofstream`, el modo de apertura de archivo puede ser `ios::out` para enviar datos a un archivo, o `ios::app` para adjuntar datos al final de un archivo (sin modificar los datos que ya estén en el archivo). Los archivos existentes que se abren con el modo `ios::out` se **truncan**: se descartan todos los datos en el archivo. Si el archivo especificado no existe todavía, entonces el objeto `ofstream` crea el archivo, usando ese nombre de archivo.

En la línea 19 se crea un objeto `ofstream` llamado `archivoClientesSalida`, asociado con el archivo `clientes.dat` que se abre en modo de salida. Los argumentos "`clientes.dat`" e `ios::out` se pasan al constructor de `ofstream`, el cual abre el archivo (esto establece una "línea de comunicación" con el archivo). De manera predeterminada, los objetos `ofstream` se abren en modo de salida, por lo que en la línea 19 se podría haber utilizado la instrucción alterna

```
ofstream archivoClientesSalida("clientes.dat");
```

para abrir `clientes.dat` en modo de salida. En la figura 17.5 se listan los modos de apertura de archivos.

### Error común de programación 17.1



Tenga cuidado al abrir un archivo existente en modo de salida (`ios::out`), en especial si desea preservar el contenido del archivo, que se descartará sin advertencia.

Se puede crear un objeto `ofstream` sin necesidad de abrir un archivo específico; después se puede adjuntar un archivo al objeto. Por ejemplo, la instrucción

```
ofstream archivoClientesSalida;
```

crea un objeto `ofstream` llamado `archivoClientesSalida`. La función miembro `open` de `ofstream` abre un archivo y lo adjunta a un objeto `ofstream` existente, como se muestra a continuación:

```
archivoClientesSalida.open("clientes.dat", ios::out);
```

| Modo                     | Descripción                                                                                                                                                                                         |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ios::app</code>    | Añade toda la salida al final del archivo.                                                                                                                                                          |
| <code>ios::ate</code>    | Abre un archivo en modo de salida y se desplaza hasta el final del archivo (por lo general se utiliza para añadir datos a un archivo). Los datos se pueden escribir en cualquier parte del archivo. |
| <code>ios::in</code>     | Abre un archivo en modo de entrada.                                                                                                                                                                 |
| <code>ios::out</code>    | Abre un archivo en modo de salida.                                                                                                                                                                  |
| <code>ios::trunc</code>  | Descarta el contenido del archivo, si es que existe (también es la acción predeterminada para <code>ios::out</code> ).                                                                              |
| <code>ios::binary</code> | Abre un archivo en modo de entrada o salida binaria (es decir, que no es texto).                                                                                                                    |

Figura 17.5 | Modos de apertura de archivos.



### Error común de programación 17.2

*Si no se abre un archivo antes de tratar de hacer referencia a éste en un programa, se producirá un error.*

Después de crear un objeto `ofstream` y tratar de abrirlo, el programa prueba si la operación de apertura tuvo éxito. La instrucción `if` en las líneas 22 a 26 utiliza la función miembro `operator!` sobrecargada de `ios` para determinar si la operación `open` tuvo éxito. La condición devuelve un valor `true` si se establece el bit `failbit` o el bit `badbit` para el flujo en la operación `open`. Ciertos posibles errores son: tratar de abrir un archivo no existente en modo de lectura, tratar de abrir un archivo en modo de lectura o escritura sin permiso, y abrir un archivo en modo de escritura cuando no hay espacio disponible en disco.

Si la condición indica un intento fallido de abrir el archivo, en la línea 24 se imprime el mensaje de error "No se pudo abrir el archivo", y en la línea 25 se invoca a la función `exit` para terminar el programa. El argumento para `exit` se devuelve al entorno desde el que se invocó el programa. El argumento 0 indica que el programa terminó en forma normal; cualquier otro valor indica que el programa terminó debido a un error. El entorno de llamada (probablemente el sistema operativo) utiliza el valor devuelto por `exit` para responder al error en forma apropiada.

Otra función miembro de operador sobrecargada de `ios` (`operator void*`) convierte el flujo en un apuntador, para que se pueda evaluar como 0 (es decir, el apuntador nulo) o un número distinto de cero (es decir, cualquier otro valor de apuntador). Cuando se usa el valor de un apuntador como una condición, C++ convierte un apuntador nulo en el valor `bool false` y un apuntador no nulo en el valor `bool true`. Si se establecieron los bits `failbit` o `badbit` (vea el capítulo 15) para el flujo, se devuelve 0 (`false`). La condición en la instrucción `while` de las líneas 36 a 40 invoca a la función miembro `operator void*` en `cin` de manera implícita. La condición permanece como `true`, siempre y cuando no se haya establecido el bit `failbit` o el bit `badbit` para `cin`. Al introducir el indicador de fin de archivo se establece el bit `failbit` para `cin`. La función `operator void *` se puede utilizar para probar un objeto de entrada para el fin de archivo, en vez de llamar a la función miembro `eof` de manera explícita en el objeto de entrada.

Si en la línea 19 se abrió el archivo con éxito, el programa empieza a procesar los datos. En las líneas 28 y 29 se pide al usuario que introduzca varios campos para cada registro, o el indicador de fin de archivo cuando esté completa la entrada de datos. En la figura 17.6 se listan las combinaciones de teclado para introducir el fin de archivo en varios sistemas computacionales.

En la línea 36 se extrae cada conjunto de datos y se determina si se introdujo el fin de archivo. Al encontrar el fin de archivo, o cuando se introducen datos incorrectos, `operator void *` devuelve el apuntador nulo (que se convierte en el valor `bool false`) y la instrucción `while` termina. El usuario introduce el fin de archivo para informar al programa que ya no procese más datos. El indicador de fin de archivo se establece cuando el usuario introduce la combinación de teclas de fin de archivo. La instrucción `while` itera hasta que se establece el indicador de fin de archivo.

| Sistema computacional | Combinación de teclado                                          |
|-----------------------|-----------------------------------------------------------------|
| UNIX/Linux/Mac OS X   | <code>&lt;Ctrl-d&gt;</code> (en una línea por sí solo)          |
| Microsoft Windows     | <code>&lt;Ctrl-z&gt;</code> (algunas veces va seguido de Intro) |
| VAX (VMS)             | <code>&lt;Ctrl-z&gt;</code>                                     |

Figura 17.6 | Combinaciones de teclas de fin de archivo para varios sistemas computacionales populares.

En la línea 38 se escribe un conjunto de datos en el archivo `clientes.dat`, usando el operador de inserción de flujo `<<` y el objeto `archivoClientesSalida` asociado con el archivo al principio del programa. Los datos se pueden recuperar mediante un programa diseñado para leer el archivo (vea la sección 17.5). Observe que, debido a que el archivo creado en la figura 17.4 es simplemente un archivo de texto, puede verse en cualquier editor de texto.

Una vez que el usuario introduce el indicador de fin de archivo, `main` termina. Esto invoca de manera implícita a la función destructor del objeto `archivoClientesSalida`, que cierra el archivo `clientes.dat`. También podemos cerrar el objeto `ofstream` de manera explícita, usando la función miembro `close` en la instrucción

```
archivoClientesSalida.close();
```

### Tip de rendimiento 17.1

*Al cerrar los archivos de manera explícita cuando el programa ya no necesita hacer referencia a ellos, se puede reducir el uso de los recursos (en especial si el programa continúa ejecutándose después de cerrar los archivos).*

En la ejecución de ejemplo para el programa de la figura 17.4, el usuario introduce información para cinco cuentas y después indica que la entrada de datos está completa, al introducir el fin de archivo (se muestra `^Z` para Microsoft Windows). Esta ventana de diálogo no muestra cómo aparecen los registros de datos en el archivo. Para verificar que el programa haya creado el archivo con éxito, en la siguiente sección se muestra cómo crear un programa que lea este archivo e imprima su contenido.

## 17.5 Cómo leer datos de un archivo secuencial

Los archivos almacenan datos de manera que éstos se puedan obtener para procesarlos cuando sea necesario. En la sección anterior demostramos cómo crear un archivo para acceso secuencial. En esta sección veremos cómo leer datos secuencialmente desde un archivo.

En la figura 17.7 se leen registros del archivo de datos `clientes.dat` que creamos usando el programa de la figura 17.4, y se muestra el contenido de estos registros. Al crear un objeto `ifstream`, se abre un archivo en modo de entrada. El constructor de `ifstream` puede recibir el nombre del archivo y el modo de apertura del mismo como argumentos. En la línea 31 se crea un objeto `ifstream` llamado `archivoClientesEntrada`, y se asocia con el archivo `clientes.dat`. Los argumentos entre paréntesis se pasan a la función constructor de `ifstream`, la cual abre el archivo y establece una “línea de comunicación” con el mismo.

### Buena práctica de programación 17.1

*Abra un archivo en modo de entrada solamente (usando `ios::in`) si el contenido de éste no se debe modificar. Esto evita la modificación accidental del contenido del archivo, y es un ejemplo del principio del menor privilegio.*

```

1 // Fig. 17.7: Fig17_07.cpp
2 // Cómo leer e imprimir un archivo secuencial.
3 #include <iostream>
4 using std::cerr;
5 using std::cout;
6 using std::endl;
7 using std::fixed;
8 using std::ios;
9 using std::left;
10 using std::right;
11 using std::showpoint;
12
13 #include <fstream> // flujo de archivo
14 using std::ifstream; // flujo de archivo de entrada
15
16 #include <iomanip>
17 using std::setw;
18 using std::setprecision;
19
```

Figura 17.7 | Cómo leer e imprimir un archivo secuencial. (Parte I de 2).

```

20 #include <iostream>
21 using std::string;
22
23 #include <cstdlib>
24 using std::exit; // prototipo de la función exit
25
26 void imprimirLinea(int, const string, double); // prototipo
27
28 int main()
29 {
30 // el constructor de ifstream abre el archivo
31 ifstream archivoClientesEntrada("clientes.dat", ios::in);
32
33 // sale del programa si ifstream no pudo abrir el archivo
34 if (!archivoClientesEntrada)
35 {
36 cerr << "No se pudo abrir el archivo" << endl;
37 exit(1);
38 } // fin de if
39
40 int cuenta;
41 char nombre[30];
42 double saldo;
43
44 cout << left << setw(10) << "Cuenta" << setw(13)
45 << "Nombre" << "Saldo" << endl << fixed << showpoint;
46
47 // muestra cada registro en el archivo
48 while (archivoClientesEntrada >> cuenta >> nombre >> saldo)
49 imprimirLinea(cuenta, nombre, saldo);
50
51 return 0; // el destructor de ifstream cierra el archivo
52 } // fin de main
53
54 // muestra un solo registro del archivo
55 void imprimirLinea(int cuenta, const string nombre, double saldo)
56 {
57 cout << left << setw(10) << cuenta << setw(13) << nombre
58 << setw(7) << setprecision(2) << right << saldo << endl;
59 } // fin de la función imprimirLinea

```

| Cuenta | Nombre | Saldo  |
|--------|--------|--------|
| 100    | Jones  | 24.98  |
| 200    | Doe    | 345.67 |
| 300    | White  | 0.00   |
| 400    | Stone  | -42.16 |
| 500    | Rich   | 224.62 |

Figura 17.7 | Cómo leer e imprimir un archivo secuencial. (Parte 2 de 2).

Los objetos de la clase `ifstream` se abren en modo de entrada de manera predeterminada. Podríamos haber utilizado la instrucción

```
ifstream archivoClientesEntrada("clientes.dat");
```

para abrir `clientes.dat` en modo de entrada. Al igual que un objeto `ofstream`, un objeto `ifstream` se puede crear sin abrir un archivo específico, ya que se le puede adjuntar un archivo más adelante.

El programa utiliza la condición `!archivoClientEntrada` para determinar si el archivo se abrió con éxito antes de tratar de obtener datos del archivo. En la línea 48 se lee un conjunto de datos (es decir, un registro) del archivo. Después de que se ejecuta la línea anterior la primera vez, `cuenta` tiene el valor 100, `nombre` tiene el valor "Jones" y `saldo` tiene el valor 24.98. Cada vez que se ejecuta la línea 48, lee otro registro del archivo y lo coloca en las variables `cuenta`, `nombre` y `saldo`. En la línea 49 se muestran los registros, usando la función `imprimirLinea` (líneas 55 a 59), la cual

utiliza manipuladores de flujos parametrizados para dar formato a los datos que se van a mostrar. Al llegar al fin del archivo, la llamada implícita a `operator void *` en la condición `while` devuelve el apuntador nulo (que se convierte en el valor `bool false`), la función destructor de `ifstream` cierra el archivo y el programa termina.

Para obtener datos secuencialmente de un archivo, por lo general los programas empiezan a leer desde el principio del archivo y leen todos los datos en forma consecutiva, hasta encontrar los datos deseados. Tal vez sea necesario procesar el archivo secuencialmente varias veces (desde el principio del mismo) durante la ejecución de un programa. Tanto `istream` como `ostream` proporcionan funciones miembro para reposicionar el apuntador de posición del archivo (el número de byte del siguiente byte en el archivo que se va a leer o escribir). Estas funciones miembro son `seekg` ("seek get", "buscar obtener") para `istream` y `seekp` ("seek put", "buscar colocar") para `ostream`. Cada objeto `istream` tiene un "apuntador obtener", el cual indica el número de byte en el archivo a partir del cuál va a ocurrir la siguiente entrada, y cada objeto `ostream` tiene un "apuntador colocar", el cual indica el número de byte en el archivo en el que se debe colocar la siguiente salida. La instrucción

```
archivoClientesEntrada.seekg(0);
```

reposiciona el apuntador de posición del archivo y lo coloca al principio del archivo (ubicación 0) adjunto a `archivoClientesEntrada`. El argumento para `seekg` es comúnmente un entero `long`. Se puede especificar un segundo argumento para indicar la dirección de búsqueda, que puede ser `ios::beg` (la opción predeterminada) para un posicionamiento relativo al inicio de un flujo, `ios::cur` para un posicionamiento relativo a la posición actual en un flujo, o `ios::end` para un posicionamiento relativo al final de un flujo. El apuntador de posición del archivo es un valor entero que especifica la ubicación en el archivo como un número de bytes desde la ubicación inicial del archivo (a ésta también se le conoce como el **desplazamiento** desde el inicio del archivo). Algunos ejemplos de cómo posicionar el apuntador "obtener" de posición del archivo son:

```
// se posiciona en el n-ésimo byte de objetoArchivo (asumiendo ios::beg)
objetoArchivo.seekg(n);

// se posiciona n bytes hacia adelante en objetoArchivo
objetoArchivo.seekg(n, ios::cur);

// se posiciona n bytes hacia atrás desde el final de objetoArchivo
objetoArchivo.seekg(n, ios::end);

// se posiciona al final de objetoArchivo
objetoArchivo.seekg(0, ios::end);
```

Se pueden realizar las mismas operaciones mediante la función miembro `seekp` de `ostream`. Las funciones miembro `tellg` y `tellp` se proporcionan para devolver las posiciones actuales de los apuntadores "obtener" y "colocar", respectivamente. La siguiente instrucción asigna el valor del apuntador "obtener" de posición del archivo a la variable `ubicacion` de tipo `long`:

```
ubicacion = objetoArchivo.tellg();
```

En la figura 17.8 se permite a un gerente de créditos mostrar la información de la cuenta para los clientes con saldos en cero (es decir, los clientes que no deben dinero a la empresa), saldos de crédito (negativos) (es decir, son clientes a quienes les debe dinero la empresa), y saldos de débito (positivos) (es decir, los clientes que deben dinero a la empresa por los bienes y servicios recibidos en el pasado). El programa muestra un menú y permite al gerente de créditos introducir una de tres opciones para obtener la información de crédito. La opción 1 produce una lista de cuentas con saldos en cero. La opción 2 produce una lista de cuentas con saldos de crédito. La opción 3 produce una lista de cuentas con saldos de débito. La opción 4 termina la ejecución del programa. Si se introduce una opción inválida, se muestra el indicador para que el usuario introduzca otra opción.

```
1 // Fig. 17.8: Fig17_08.cpp
2 // Programa de solicitud de crédito.
3 #include <iostream>
4 using std::cerr;
5 using std::cin;
6 using std::cout;
7 using std::endl;
```

**Figura 17.8** | Programa de solicitud de crédito. (Parte 1 de 4).

```

8 using std::fixed;
9 using std::ios;
10 using std::left;
11 using std::right;
12 using std::showpoint;
13
14 #include <fstream>
15 using std::ifstream;
16
17 #include <iomanip>
18 using std::setw;
19 using std::setprecision;
20
21 #include <string>
22 using std::string;
23
24 #include <cstdlib>
25 using std::exit; // prototipo de la función exit
26
27 enum TipoSolicitud { SALDO_CERO = 1, SALDO_CREDITO, SALDO_DEBITO, TERMINAR };
28 int obtenerSolicitud();
29 bool debeMostrar(int, double);
30 void imprimirLinea(int, const string, double);
31
32 int main()
33 {
34 // el constructor de ifstream abre el archivo
35 ifstream archivoClientesSalida("clientes.dat", ios::in);
36
37 // sale del programa si ifstream no pudo abrir el archivo
38 if (!archivoClientesSalida)
39 {
40 cerr << "No se pudo abrir el archivo" << endl;
41 exit(1);
42 } // fin de if
43
44 int solicitud;
45 int cuenta;
46 char nombre[30];
47 double saldo;
48
49 // obtiene la solicitud del usuario (por ejemplo, saldo en cero, de crédito o débito)
50 solicitud = obtenerSolicitud();
51
52 // procesa la solicitud del usuario
53 while (solicitud != TERMINAR)
54 {
55 switch (solicitud)
56 {
57 case SALDO_CERO:
58 cout << "\nCuentas con saldos en cero:\n";
59 break;
60 case SALDO_CREDITO:
61 cout << "\nCuentas con saldos de credito:\n";
62 break;
63 case SALDO_DEBITO:
64 cout << "\nCuentas con saldos de debito:\n";
65 break;
66 } // fin de switch
67
68 // lee la cuenta, el nombre y el saldo del archivo
69 archivoClientesSalida >> cuenta >> nombre >> saldo;

```

Figura 17.8 | Programa de solicitud de crédito. (Parte 2 de 4).

```

70 // muestra el contenido del archivo (hasta eof)
71 while (!archivoClientesSalida.eof())
72 {
73 // muestra el registro
74 if (debeMostrar(solicitud, saldo))
75 imprimirLinea(cuenta, nombre, saldo);
76
77 // lee la cuenta, el nombre y el saldo del archivo
78 archivoClientesSalida >> cuenta >> nombre >> saldo;
79 } // fin de while interior
80
81 archivoClientesSalida.clear(); // restablece eof para la siguiente entrada
82 archivoClientesSalida.seekg(0); // se reposiciona al inicio del archivo
83 solicitud = obtenerSolicitud(); // obtiene una solicitud adicional del usuario
84 } // fin de while exterior
85
86
87 cout << "Fin de ejecucion." << endl;
88 return 0; // el destructor de ifstream cierra el archivo
89 } // fin de main
90
91 // obtiene la solicitud del usuario
92 int obtenerSolicitud()
93 {
94 int solicitud; // solicitud del usuario
95
96 // muestra las opciones de solicitud
97 cout << "\nEscriba la opcion" << endl
98 << " 1 - Listar cuentas con saldos en cero" << endl
99 << " 2 - Listar cuentas con saldos de credito" << endl
100 << " 3 - Listar cuentas con saldos de debito" << endl
101 << " 4 - Finalizar ejecucion" << fixed << showpoint;
102
103 do // introduce la solicitud del usuario
104 {
105 cout << "\n? ";
106 cin >> solicitud;
107 } while (solicitud < SALDO_CERO && solicitud > TERMINAR);
108
109 return solicitud;
110 } // fin de la función obtenerSolicitud
111
112 // determina si se va a mostrar el registro dado
113 bool debeMostrar(int tipo, double saldo)
114 {
115 // determina si se van a mostrar los saldos en cero
116 if (tipo == SALDO_CERO && saldo == 0)
117 return true;
118
119 // determina si se van a mostrar los saldos de crédito
120 if (tipo == SALDO_CREDITO && saldo < 0)
121 return true;
122
123 // determina si se van a mostrar los saldos de débito
124 if (tipo == SALDO_DEBITO && saldo > 0)
125 return true;
126
127 return false;
128 } // fin de la función debeMostrar
129
130 // muestra un solo registro del archivo
131 void imprimirLinea(int cuenta, const string nombre, double saldo)

```

Figura 17.8 | Programa de solicitud de crédito. (Parte 3 de 4).

```

132 {
133 cout << left << setw(10) << cuenta << setw(13) << nombre
134 << setw(7) << setprecision(2) << right << saldo << endl;
135 } // fin de la función imprimirLinea

```

```

Escriba la opcion
1 - Listar cuentas con saldos en cero
2 - Listar cuentas con saldos de credito
3 - Listar cuentas con saldos de debito
4 - Finalizar ejecucion
? 1

Cuentas con saldos en cero:
300 White 0.00

Escriba la opcion
1 - Listar cuentas con saldos en cero
2 - Listar cuentas con saldos de credito
3 - Listar cuentas con saldos de debito
4 - Finalizar ejecucion
? 2

Cuentas con saldos de credito:
400 Stone -42.16

Escriba la opcion
1 - Listar cuentas con saldos en cero
2 - Listar cuentas con saldos de credito
3 - Listar cuentas con saldos de debito
4 - Finalizar ejecucion
? 3

Cuentas con saldos de debito:
100 Jones 24.98
200 Doe 345.67
500 Rich 224.62

Escriba la opcion
1 - Listar cuentas con saldos en cero
2 - Listar cuentas con saldos de credito
3 - Listar cuentas con saldos de debito
4 - Finalizar ejecucion
? 4
Fin de ejecucion.

```

Figura 17.8 | Programa de solicitud de crédito. (Parte 4 de 4).

## 17.6 Actualización de archivos secuenciales

Los datos a los que se dan formato y se escriben en un archivo secuencial, como se muestra en la figura 17.4, no se pueden modificar sin el riesgo de destruir otros datos en el archivo. Por ejemplo, si hay que cambiar el nombre "White" por "Worthington", el nombre anterior no se puede sobrescribir sin que se corrompa el archivo. El registro para White se escribió en el archivo de la siguiente manera:

300 White 0.00

Si se vuelve a escribir este registro, empezando en la misma ubicación en el archivo y usando el nuevo nombre más largo, el registro sería:

300 Worthington 0.00

El nuevo registro contiene seis caracteres más que el registro original. Por lo tanto, los caracteres más allá de la segunda "o" en "Worthington" sobrescribirían el inicio del siguiente registro secuencial en el archivo. El problema es que, en el modelo de entrada/salida con formato que utiliza el operador de inserción de flujo `<<` y el operador de extracción de flujo `>>`, los campos (y por ende, los registros) pueden variar en cuanto a su tamaño. Por ejemplo, los valores 7, 14, -117, 2074 y 27383 son todos `int`, los cuales almacenan el mismo número de bytes de "datos puros" de manera interna (por

lo general, cuatro bytes en los equipos populares de 32 bits de la actualidad). Sin embargo, estos enteros se convierten en campos de distintos tamaños cuando se imprimen como texto con formato (secuencias de caracteres). Por lo tanto, el modelo de entrada/salida con formato no se utiliza comúnmente para actualizar registros en su posición, en el archivo.

Dicha actualización puede realizarse de una manera complicada. Por ejemplo, para modificar el nombre anterior, se podrían copiar los registros del archivo secuencial antes de 300 White 0.00 en un nuevo archivo, después se escribiría el registro actualizado en el nuevo archivo, y luego se copiarían los registros después de 300 White 0.00 en el nuevo archivo. Para ello se requiere procesar todos los registros en el archivo, para actualizar sólo un registro. Si se van a actualizar muchos registros del archivo en una sola pasada, esta técnica puede ser aceptable.

## 17.7 Archivos de acceso aleatorio

Hasta ahora hemos visto cómo crear archivos secuenciales y buscar en ellos para localizar información. Los archivos secuenciales son inapropiados para las **aplicaciones de acceso instantáneo**, en las que un registro específico se debe localizar inmediatamente. Las aplicaciones comunes de acceso instantáneo son los sistemas de reservación de aerolíneas, sistemas bancarios, sistemas de punto de venta, cajeros automáticos y otros tipos de **sistemas de procesamiento de transacciones** que requieran acceso rápido a datos específicos. Un banco podría tener cientos de miles (o incluso millones) de otros clientes, y a pesar de ello, cuando un cliente utiliza un cajero automático, el programa comprueba la cuenta de ese cliente en unos cuantos segundos o menos, para ver si tiene suficientes fondos. Este tipo de acceso instantáneo es posible mediante los **archivos de acceso aleatorio**. Los registros individuales de un archivo de acceso aleatorio se pueden utilizar de manera directa (y rápida), sin tener que buscar en otros registros.

Como hemos dicho, C++ no impone una estructura sobre un archivo. Por lo tanto, la aplicación que deseé utilizar archivos de acceso aleatorio debe crearlos. Se puede utilizar una variedad de técnicas. Tal vez el método más sencillo es requerir que todos los registros en un archivo sean de la misma longitud fija. Al utilizar registros de longitud fija y con el mismo tamaño, es más fácil para el programa calcular (como una función del tamaño de registro y la clave de registro) la ubicación exacta de cualquier registro relativo al inicio del archivo. Pronto veremos cómo esto facilita el acceso inmediato a registros específicos, incluso en archivos extensos.

En la figura 17.9 se ilustra la forma en que C++ ve a un archivo de acceso aleatorio, compuesto de registros de longitud fija (en este caso, cada registro tiene 100 bytes de longitud). Un archivo de acceso aleatorio es como un ferrocarril con muchos carros del mismo tamaño; algunos están vacíos y otros llenos.

Se pueden insertar datos en un archivo de acceso aleatorio sin destruir otros datos en el archivo. Los datos almacenados con anterioridad también se pueden actualizar o eliminar, sin necesidad de volver a escribir el archivo completo. En las siguientes secciones explicaremos cómo crear un archivo de acceso aleatorio, introducir datos en el archivo, leer los datos tanto en forma secuencial como aleatoria, actualizar los datos y eliminar datos que ya no sean necesarios.

## 17.8 Creación de un archivo de acceso aleatorio

La función miembro `write` de `ostream` envía un número fijo de bytes, empezando en una ubicación específica en memoria al flujo especificado. Cuando el flujo se asocia con un archivo, la función `write` escribe los datos en la ubicación en el archivo especificado mediante el apuntador “colocar” de posición del archivo. La función miembro `read` de `istream` recibe un número fijo de bytes del flujo especificado y los coloca en un área en memoria, que empieza en una dirección especificada. Si el flujo se asocia con un archivo, la función `read` introduce bytes en la ubicación en el archivo especificado por el apuntador “obtener” de posición del archivo.

### *Escritura de bytes mediante la función miembro `write` de `ostream`*

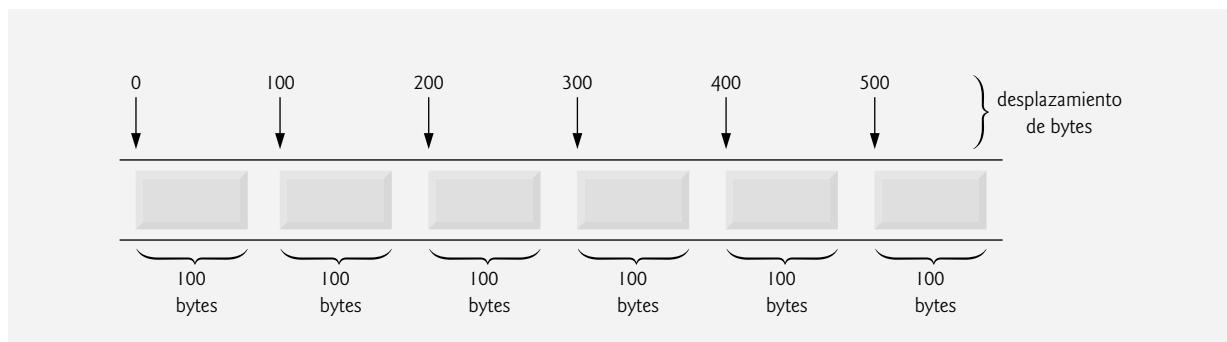
Al escribir el entero `numero` en un archivo, en vez de usar la instrucción

```
archivoSalida << numero;
```

que para un entero de cuatro bytes podría imprimir desde un dígito hasta 11 (10 dígitos más un signo, cada uno de los cuales requiere un solo byte de almacenamiento), podemos usar la instrucción

```
archivoSalida.write(reinterpret_cast< const char * >(&numero),
 sizeof(numero));
```

que siempre escribe la versión binaria de los cuatro bytes del entero (en un equipo con enteros de cuatro bytes). La función `write` trata a su primer argumento como un grupo de bytes, al ver el objeto en memoria como un `const char *`, el cual es un apuntador a un byte. Empezando desde esa ubicación, la función `write` envía como salida el número de bytes especificados por su segundo argumento: un entero de tipo `size_t`. Como veremos, la función `read` de `istream` se puede utilizar después para leer los cuatro bytes y colocarlos de vuelta en la variable entera `numero`.



**Figura 17.9** | Forma en que C++ ve a un archivo de acceso aleatorio.

### Conversión entre los tipos de apuntadores con el operador `reinterpret_cast`

Por desgracia, la mayoría de los apuntadores que pasamos a la función `write` como el primer argumento no son de tipo `const char *`. Para enviar como salida objetos de otros tipos, debemos convertir los apuntadores a esos objetos al tipo `const char *`; en caso contrario, el compilador no compilará las llamadas a la función `write`. C++ proporciona el operador `reinterpret_cast` para casos como éste en el que un apuntador de un tipo debe convertirse en un tipo de apuntador no relacionado. También podemos usar este operador de conversión para realizar conversiones entre los tipos de apuntadores y el tipo entero, y viceversa. Sin un operador `reinterpret_cast`, la instrucción `write` que envía como salida el entero `numero` no se compilará, ya que el compilador no permite pasar un apuntador de tipo `int *` (el tipo devuelto por la expresión `&numero`) a una función que espera un argumento de tipo `const char *`; en lo que respecta al compilador, estos tipos son incompatibles.

Una operación `reinterpret_cast` se lleva a cabo en tiempo de compilación, y no cambia el valor del objeto al cual apunta su operando. En vez de ello, solicita que el compilador reinterprete el operando como el tipo de destino (especificado en los signos `<` y `>` que siguen después de la palabra clave `reinterpret_cast`). En la figura 17.12 utilizamos a `reinterpret_cast` para convertir un apuntador `DatosCliente` en un `const char *`, que reinterpreta un objeto `DatosCliente` como bytes que se van a enviar a un archivo. Los programas de procesamiento de archivos de acceso aleatorio raras veces escriben un solo campo en un archivo. Por lo general, escriben un objeto de una clase a la vez, como demostramos en los siguientes ejemplos.



#### Tip para prevenir errores 17.1

Es fácil utilizar `reinterpret_cast` para realizar manipulaciones peligrosas que podrían conducir a errores graves en tiempo de ejecución.



#### Tip de portabilidad 17.1

El uso de `reinterpret_cast` es dependiente del compilador, y puede hacer que los programas se comporten de manera diferente en distintas plataformas. El operador `reinterpret_cast` no se debe utilizar, a menos que sea absolutamente necesario.



#### Tip de portabilidad 17.2

Un programa que lee datos sin formato (escritos por `write`) debe compilarse y ejecutarse en un sistema compatible con el programa que escribió los datos, ya que distintos sistemas pueden representar los datos internos de manera diferente.

### Programa de procesamiento de crédito

Considere el siguiente enunciado del problema:

Cree un programa de procesamiento de crédito que sea capaz de almacenar a lo más 100 registros de longitud fija para una empresa que puede tener hasta 100 clientes. Cada registro debe consistir de un número de cuenta que actúe como la clave de registro, un apellido paterno, un primer nombre y un saldo. El programa debe ser capaz de actualizar una cuenta, insertar una nueva cuenta, eliminar una cuenta e insertar todos los registros de las cuentas en un archivo de texto con formato para imprimirla.

En las siguientes secciones presentaremos las técnicas para crear este programa de procesamiento de crédito. La figura 17.12 ilustra cómo abrir un archivo de acceso aleatorio, definir el formato de los registros usando un objeto de la clase **DatosCliente** (figuras 17.10 y 17.11) y escribir datos en el disco, en formato binario. Este programa inicializa los 100 registros del archivo **credito.dat** con objetos vacíos, usando la función **write**. Cada objeto vacío contiene 0 para el número de cuenta, la cadena nula (representada por comillas vacías) para el apellido paterno y el primer nombre, y 0.0 para el saldo. Cada registro se inicializa con la cantidad de espacio vacío en donde se almacenarán los datos de la cuenta.

```

1 // Fig. 17.10: DatosCliente.h
2 // Definición de la clase DatosCliente, utilizada en las figuras 17.12 a 17.15.
3 #ifndef DATOSCLIENTE_H
4 #define DATOSCLIENTE_H
5
6 #include <string>
7 using std::string;
8
9 class DatosCliente
10 {
11 public:
12 // constructor predeterminado de DatosCliente
13 DatosCliente(int = 0, string = "", string = "", double = 0.0);
14
15 // funciones de acceso para numeroCuenta
16 void establecerNumeroCuenta(int);
17 int obtenerNumeroCuenta() const;
18
19 // funciones de acceso para apellidoPaterno
20 void establecerApellidoPaterno(string);
21 string obtenerApellidoPaterno() const;
22
23 // funciones de acceso para primerNombre
24 void establecerPrimerNombre(string);
25 string obtenerPrimerNombre() const;
26
27 // funciones de acceso para el saldo
28 void establecerSaldo(double);
29 double obtenerSaldo() const;
30 private:
31 int numeroCuenta;
32 char apellidoPaterno[15];
33 char primerNombre[10];
34 double saldo;
35 }; // fin de la clase DatosCliente
36
37 #endif

```

**Figura 17.10** | Archivo de encabezado de la clase **DatosCliente**.

Los objetos de la clase **string** no tienen tamaño uniforme, sino que utilizan la memoria asignada en forma dinámica para dar cabida a las cadenas de varias longitudes. Este programa debe mantener registros de longitud fija, por lo que la clase **DatosCliente** almacena el primer nombre y el apellido del cliente en arreglos **char** de longitud fija. Las funciones miembro **establecerApellidoPaterno** (figura 17.11, líneas 37 a 45) y **establecerPrimerNombre** (figura 17.11, líneas 54 a 62) copian los caracteres de un objeto **string** en el arreglo **char** correspondiente. Considere la función **establecerApellidoPaterno**. En la línea 40 se inicializa el apuntador **const char \* valorApellidoPaterno** con el resultado de una llamada a la función miembro **string** llamada **datos**, la cual devuelve un arreglo que contiene los caracteres del objeto **string**. [Nota: no se garantiza que este arreglo tenga terminación nula]. En la línea 41 se invoca a la función miembro **string** llamada **size** para obtener la longitud de **cadenaApellidoPaterno**. En la línea 42 se asegura que **longitud** sea menor de 15 caracteres, y después en la línea 43 se copian **longitud** caracteres de **valorApellidoPaterno** al arreglo **char** llamado **apellidoPaterno**. La función miembro **establecerPrimerNombre** realiza los mismos pasos para el primer nombre.

```

1 // Fig. 17.11: DatosCliente.cpp
2 // La clase DatosCliente almacena la información de crédito del cliente.
3 #include <string>
4 using std::string;
5
6 #include "DatosCliente.h"
7
8 // constructor predeterminado de DatosCliente
9 DatosCliente::DatosCliente(int valorNumeroCuenta,
10 string valorApellidoPaterno, string valorPrimerNombre, double valorSaldo)
11 {
12 establecerNumeroCuenta(valorNumeroCuenta);
13 establecerApellidoPaterno(valorApellidoPaterno);
14 establecerPrimerNombre(valorPrimerNombre);
15 establecerSaldo(valorSaldo);
16 } // fin del constructor de DatosCliente
17
18 // obtiene el valor del número de cuenta
19 int DatosCliente::obtenerNumeroCuenta() const
20 {
21 return numeroCuenta;
22 } // fin de la función obtenerNumeroCuenta
23
24 // establece el valor del número de cuenta
25 void DatosCliente::establecerNumeroCuenta(int valorNumeroCuenta)
26 {
27 numeroCuenta = valorNumeroCuenta; // debe validar
28 } // fin de la función establecerNumeroCuenta
29
30 // obtiene el valor del apellido paterno
31 string DatosCliente::obtenerApellidoPaterno() const
32 {
33 return apellidoPaterno;
34 } // fin de la función obtenerApellidoPaterno
35
36 // establece el valor del apellido paterno
37 void DatosCliente::establecerApellidoPaterno(string cadenaApellidoPaterno)
38 {
39 // copia a lo más 15 caracteres de la cadena a apellidoPaterno
40 const char *valorApellidoPaterno = cadenaApellidoPaterno.data();
41 int longitud = cadenaApellidoPaterno.size();
42 longitud = (longitud < 15 ? longitud : 14);
43 strncpy(apellidoPaterno, valorApellidoPaterno, longitud);
44 apellidoPaterno[longitud] = '\0'; // adjunta un carácter nulo a apellidoPaterno
45 } // fin de la función establecerApellidoPaterno
46
47 // obtiene el valor del primer nombre
48 string DatosCliente::obtenerPrimerNombre() const
49 {
50 return primerNombre;
51 } // fin de la función obtenerPrimerNombre
52
53 // establece el valor del primer nombre
54 void DatosCliente::establecerPrimerNombre(string cadenaPrimerNombre)
55 {
56 // copia a lo más 10 caracteres de la cadena a primerNombre
57 const char *valorPrimerNombre = cadenaPrimerNombre.data();
58 int longitud = cadenaPrimerNombre.size();
59 longitud = (longitud < 10 ? longitud : 9);
60 strncpy(primerNombre, valorPrimerNombre, longitud);
61 primerNombre[longitud] = '\0'; // adjunta un carácter nulo a primerNombre
62 } // fin de la función establecerPrimerNombre

```

**Figura 17.11** | La clase DatosCliente representa la información de crédito de un cliente. (Parte I de 2).

```

63 // obtiene el valor del saldo
64 double DatosCliente::obtenerSaldo() const
65 {
66 return saldo;
67 } // fin de la función obtenerSaldo
68
69 // establece el valor del saldo
70 void DatosCliente::establecerSaldo(double valorSaldo)
71 {
72 saldo = valorSaldo;
73 } // fin de la función establecerSaldo

```

**Figura 17.11** | La clase DatosCliente representa la información de crédito de un cliente. (Parte 2 de 2).

En la figura 17.12, la línea 18 crea un objeto `ofstream` para el archivo `credito.dat`. El segundo argumento para el constructor (`ios::out | ios::binary`) indica que vamos a abrir el archivo para salida en modo binario, lo cual es requerido si debemos escribir registros de longitud fija. En las líneas 31 y 32 se escribe el objeto `clienteEnBlanco` en el archivo `credito.dat` asociado con el objeto `ofstream` llamado `creditoSalida`. Recuerde que el operador `sizeof` devuelve el tamaño en bytes del objeto contenido entre paréntesis (vea el capítulo 8). El primer argumento para la función `write` en la línea 31 debe ser de tipo `const char *`. Sin embargo, el tipo de datos de `&clienteEnBlanco` es `DatosCliente *`. Para convertir `&clienteEnBlanco` en `const char *`, en la línea 31 se utiliza el operador de conversión `reinterpret_cast`, por lo que la llamada a `write` se compila sin generar un error de compilación.

```

1 // Fig. 17.12: Fig17_12.cpp
2 // Creación de un archivo de acceso aleatorio.
3 #include <iostream>
4 using std::cerr;
5 using std::endl;
6 using std::ios;
7
8 #include <fstream>
9 using std::ofstream;
10
11 #include <cstdlib>
12 using std::exit; // prototipo de la función exit
13
14 #include "DatosCliente.h" // definición de la clase DatosCliente
15
16 int main()
17 {
18 ofstream creditoSalida("credito.dat", ios::out | ios::binary);
19
20 // sale del programa si ofstream no pudo abrir el archivo
21 if (!creditoSalida)
22 {
23 cerr << "No se pudo abrir el archivo." << endl;
24 exit(1);
25 } // fin de if
26
27 DatosCliente clienteEnBlanco; // el constructor pone en ceros cada miembro de datos
28
29 // escribe 100 registros en blanco en el archivo
30 for (int i = 0; i < 100; i++)
31 creditoSalida.write(reinterpret_cast< const char * >(&clienteEnBlanco),
32 sizeof(DatosCliente));
33
34 return 0;
35 } // fin de main

```

**Figura 17.12** | Creación de un archivo de acceso aleatorio con 100 registros en blanco, en forma secuencial.

## 17.9 Cómo escribir datos al azar a un archivo de acceso aleatorio

En la figura 17.13 se escriben datos en el archivo `credito.dat` y se utiliza la combinación de las funciones `seekp` y `write` de `fstream` para almacenar datos en ubicaciones exactas en el archivo. La función `seekp` establece el apuntador “colocar” de posición del archivo en una posición específica en el archivo, y después `write` escribe los datos. Observe que en la línea 19 se incluye el archivo de encabezado `DatosCliente.h` definido en la figura 17.10, para que el programa pueda utilizar objetos `DatosCliente`.

```

1 // Fig. 17.13: Fig17_13.cpp
2 // Escritura en un archivo de acceso aleatorio.
3 #include <iostream>
4 using std::cerr;
5 using std::cin;
6 using std::cout;
7 using std::endl;
8 using std::ios;
9
10 #include <iomanip>
11 using std::setw;
12
13 #include <fstream>
14 using std::fstream;
15
16 #include <cstdlib>
17 using std::exit; // prototipo de la función exit
18
19 #include "DatosCliente.h" // definición de la clase DatosCliente
20
21 int main()
22 {
23 int numeroCuenta;
24 char apellidoPaterno[15];
25 char primerNombre[10];
26 double saldo;
27
28 fstream creditoSalida("credito.dat", ios::in | ios::out | ios::binary);
29
30 // sale del programa si fstream no puede abrir el archivo
31 if (!creditoSalida)
32 {
33 cerr << "No se pudo abrir el archivo." << endl;
34 exit(1);
35 } // fin de if
36
37 cout << "Escriba el numero de cuenta (de 1 a 100, 0 para terminar la entrada)\n? ";
38
39 // requiere que el usuario especifique el número de cuenta
40 DatosCliente cliente;
41 cin >> numeroCuenta;
42
43 // el usuario introduce información, la cual se copia en el archivo
44 while (numeroCuenta > 0 && numeroCuenta <= 100)
45 {
46 // el usuario introduce el apellido paterno, primer nombre y saldo
47 cout << "Escriba apellido paterno, primer nombre y saldo\n? ";
48 cin >> setw(15) >> apellidoPaterno;
49 cin >> setw(10) >> primerNombre;
50 cin >> saldo;
51

```

Figura 17.13 | Escritura en un archivo de acceso aleatorio. (Parte I de 2).

```

52 // establece los valores de numeroCuenta, apellidoPaterno, primerNombre y saldo del
 registro
53 cliente.establecerNumeroCuenta(numeroCuenta);
54 cliente.establecerApellidoPaterno(apellidoPaterno);
55 cliente.establecerPrimerNombre(primerNombre);
56 cliente.establecerSaldo(saldo);
57
58 // busca la posición en el archivo del registro especificado por el usuario
59 creditoSalida.seekp((cliente.obtenerNumeroCuenta() - 1) *
60 sizeof(DatosCliente));
61
62 // escribe la información especificada por el usuario en el archivo
63 creditoSalida.write(reinterpret_cast< const char * >(&cliente),
64 sizeof(DatosCliente));
65
66 // permite al usuario escribir otro número de cuenta
67 cout << "Escriba el numero de cuenta\n? ";
68 cin >> numeroCuenta;
69 } // fin de while
70
71 return 0;
72 } // fin de main

```

```

Escriba el numero de cuenta (de 1 a 100, 0 para terminar la entrada)
? 37
Escriba apellido paterno, primer nombre y saldo
? Barker Doug 0.00
Escriba el numero de cuenta
? 29
Escriba apellido paterno, primer nombre y saldo
? Brown Nancy -24.54
Escriba el numero de cuenta
? 96
Escriba apellido paterno, primer nombre y saldo
? Stone Sam 34.98
Escriba el numero de cuenta
? 88
Escriba apellido paterno, primer nombre y saldo
? Smith Dave 258.34
Escriba el numero de cuenta
? 33
Escriba apellido paterno, primer nombre y saldo
? Dunn Stacey 314.33
Escriba el numero de cuenta
? 0

```

**Figura 17.13 |** Escritura en un archivo de acceso aleatorio. (Parte 2 de 2).

En las líneas 59 y 60 se posiciona el apuntador “colocar” de posición del archivo para el objeto `creditoSalida` en la posición de byte calculada por

```
(cliente.obtenerNumeroCuenta() - 1) * sizeof(DatosCliente)
```

Como el número de cuenta está entre 1 y 100, se resta 1 del número de cuenta al calcular la posición de byte del registro. Así, para el registro 1, el apuntador de posición del archivo se establece en el byte 0 del archivo. Observe que en la línea 28 se utiliza el objeto `creditoSalida` de `fstream` para abrir el archivo `credito.dat` existente. El archivo se abre para entrada y salida en modo binario, para lo cual se combinan los modos de apertura de archivos `ios::in`, `ios::out` e `ios::binary`. Para combinar varios modos de apertura de archivos, se separa cada modo de apertura de los demás mediante el operador OR inclusivo a nivel de bits (`|`). Al abrir el archivo `credito.dat` existente de esta manera, aseguramos que este programa pueda manipular los registros escritos en el archivo mediante el programa de la figura 17.12, en vez de crear el archivo éste se reutiliza. En el capítulo 21, Bits, caracteres, cadenas estilo C y estructuras se describe el operador OR inclusivo a nivel de bits con detalle.

## 17.10 Cómo leer de un archivo de acceso aleatorio en forma secuencial

En las secciones anteriores, creamos un archivo de acceso aleatorio y escribimos datos en ese archivo. En esta sección, desarrollaremos un programa que lee el archivo en forma secuencial e imprime sólo esos registros que contienen datos. Estos programas producen un beneficio adicional. Vea si puede determinar cuál es; lo revelaremos al final de esta sección.

La función `read` de `istream` introduce un número especificado de bytes, desde la posición actual en el flujo especificado, hasta un objeto. Por ejemplo, en las líneas 57 y 58 de la figura 17.14 se lee el número de bytes especificado por `sizeof(DatosCliente)` del archivo asociado con el objeto `creditoEntrada` de `ifstream` y se almacenan los datos en el registro `cliente`. Observe que la función `read` requiere un primer argumento de tipo `char *`. Como `&cliente` es de tipo `DatosCliente *`, `&cliente` debe convertirse en `char *` utilizando el operador de conversión `reinterpret_cast`. Observe que en la línea 24 se incluye el archivo de encabezado `datosCliente.h` definido en la figura 17.10, para que el programa pueda usar objetos `DatosCliente`.

En la figura 17.14 se lee cada registro en el archivo `credito.dat` de manera secuencial, se comprueba cada registro para determinar si contiene datos y se muestran los resultados con formato para los registros que contienen datos. La condición en la línea 50 utiliza la función miembro `eof` de `ios` para determinar cuándo se llega al fin del archivo, y hace que la ejecución de la instrucción `while` termine. Además, si ocurre un error al leer del archivo, el ciclo termina debido a que `creditoEntrada` se evalúa como `false`. Los datos recibidos del archivo se imprimen mediante la función `imprimirLinea` (líneas 65 a 72), que recibe dos argumentos: un objeto `ostream` y una estructura `DatosCliente` a imprimir. El tipo de parámetro `ostream` es interesante, ya que cualquier objeto `ostream` (como `cout`) o cualquier objeto de una clase derivada de `ostream` (como un objeto de tipo `ofstream`) puede suministrarse como argumento. Esto significa que se puede utilizar la misma función, por ejemplo, para enviar datos al flujo de salida estándar y a un flujo de archivo, sin tener que escribir funciones separadas.

```

1 // Fig. 17.14: Fig17_14.cpp
2 // Lectura secuencial de un archivo de acceso aleatorio.
3 #include <iostream>
4 using std::cerr;
5 using std::cout;
6 using std::endl;
7 using std::fixed;
8 using std::ios;
9 using std::left;
10 using std::right;
11 using std::showpoint;
12
13 #include <iomanip>
14 using std::setprecision;
15 using std::setw;
16
17 #include <fstream>
18 using std::ifstream;
19 using std::ostream;
20
21 #include <cstdlib>
22 using std::exit; // prototipo de la función exit
23
24 #include "DatosCliente.h" // definición de la clase DatosCliente
25
26 void imprimirLinea(ostream&, const DatosCliente &); // prototipo
27
28 int main()
29 {
30 ifstream creditoEntrada("credito.dat", ios::in | ios::binary);
31
32 // sale del programa si ifstream no puede abrir el archivo
33 if (!creditoEntrada)

```

Figura 17.14 | Lectura secuencial de un archivo de acceso aleatorio. (Parte I de 2).

```

34 {
35 cerr << "No se pudo abrir el archivo." << endl;
36 exit(1);
37 } // fin de if
38
39 cout << left << setw(10) << "Cuenta" << setw(16)
40 << "Apellido" << setw(11) << "Nombre" << left
41 << setw(10) << right << "Saldo" << endl;
42
43 DatosCliente cliente; // crea un registro
44
45 // lee el primer registro del archivo
46 creditoEntrada.read(reinterpret_cast< char * >(&cliente),
47 sizeof(DatosCliente));
48
49 // lee todos los registros del archivo
50 while (creditoEntrada && !creditoEntrada.eof())
51 {
52 // muestra un registro
53 if (cliente.obtenerNumeroCuenta() != 0)
54 imprimirLinea(cout, cliente);
55
56 // lee el siguiente registro del archivo
57 creditoEntrada.read(reinterpret_cast< char * >(&cliente),
58 sizeof(DatosCliente));
59 } // fin de while
60
61 return 0;
62 } // fin de main
63
64 // muestra un solo registro
65 void imprimirLinea(ostream &salida, const DatosCliente ®istro)
66 {
67 salida << left << setw(10) << registro.obtenerNumeroCuenta()
68 << setw(16) << registro.obtenerApellidoPaterno()
69 << setw(11) << registro.obtenerPrimerNombre()
70 << setw(10) << setprecision(2) << right << fixed
71 << showpoint << registro.obtenerSaldo() << endl;
72 } // fin de la función imprimirLinea

```

| Cuenta | Apellido | Nombre | Saldo  |
|--------|----------|--------|--------|
| 29     | Brown    | Nancy  | -24.54 |
| 33     | Dunn     | Stacey | 314.33 |
| 37     | Barker   | Doug   | 0.00   |
| 88     | Smith    | Dave   | 258.34 |
| 96     | Stone    | Sam    | 34.98  |

Figura 17.14 | Lectura secuencial de un archivo de acceso aleatorio. (Parte 2 de 2).

¿Qué hay acerca de ese beneficio adicional que prometimos? Si examina la ventana de resultados, observará que los registros se listan en orden (por número de cuenta). Ésta es una consecuencia de la forma en que almacenamos estos registros en el archivo, usando las técnicas de acceso directo. En comparación con el ordenamiento de inserción que usamos en el capítulo 7, el ordenamiento mediante las técnicas de acceso directo es relativamente rápido. La velocidad se logra al hacer el archivo lo bastante grande como para que pueda contener todos los posibles registros que se podrían crear. Desde luego que esto significa que el archivo podría estar ocupado escasamente la mayor parte del tiempo, produciendo como resultado un desperdicio del almacenamiento. Éste es otro ejemplo de la conexión entre espacio y tiempo: al utilizar grandes cantidades de espacio, podemos desarrollar un algoritmo de ordenamiento mucho más rápido. Por fortuna, la continua reducción en el precio de las unidades de almacenamiento ha provocado que esto no sea tan problemático.

## 17.11 Ejemplo práctico: un programa para procesar transacciones

Ahora presentaremos un programa para procesar transacciones de un tamaño considerable (figura 17.15), en el que se utiliza un archivo de acceso aleatorio para lograr un procesamiento de acceso “instantáneo”. El programa mantiene la información de las cuentas de un banco. Actualiza las cuentas existentes, agrega nuevas cuentas, elimina cuentas y almacena un listado con formato de todas las cuentas actuales en un archivo de texto. Asumimos que se ha ejecutado el programa de la figura 17.12 para crear el archivo `credito.dat` y que se ha ejecutado el programa de la figura 17.13 para insertar los datos iniciales.

```

1 // Fig. 17.15: Fig17_15.cpp
2 // Este programa lee un archivo de acceso aleatorio en forma secuencial,
3 // actualiza los datos escritos anteriormente en el archivo, crea datos
4 // para colocarlos en el archivo, y elimina los datos previamente almacenados.
5 #include <iostream>
6 using std::cerr;
7 using std::cin;
8 using std::cout;
9 using std::endl;
10 using std::fixed;
11 using std::ios;
12 using std::left;
13 using std::right;
14 using std::showpoint;
15
16 #include <fstream>
17 using std::ofstream;
18 using std::ostream;
19 using std::fstream;
20
21 #include <iomanip>
22 using std::setw;
23 using std::setprecision;
24
25 #include <cstdlib>
26 using std::exit; // prototipo de la función exit
27
28 #include "DatosCliente.h" // definición de la clase DatosCliente
29
30 int escribirOpcion();
31 void crearArchivoTexto(fstream&);
32 void actualizarRegistro(fstream&);
33 void nuevoRegistro(fstream&);
34 void eliminarRegistro(fstream&);
35 void imprimirLinea(ostream&, const DatosCliente &);
36 int obtenerCuenta(const char * const);
37
38 enum Opciones { IMPRIMIR = 1, ACTUALIZAR, NUEVO, ELIMINAR, TERMINAR };
39
40 int main()
41 {
42 // abre el archivo para leer y escribir
43 fstream creditoEntSal("credito.dat", ios::in | ios::out | ios::binary);
44
45 // sale del programa si fstream no puede abrir el archivo
46 if (!creditoEntSal)
47 {
48 cerr << "No se pudo abrir el archivo." << endl;
49 exit (1);
50 } // fin de if

```

Figura 17.15 | Programa de cuentas bancarias. (Parte I de 5).

```
51 int opcion; // almacena la opción del usuario
52
53 // permite al usuario especificar una acción
54 while ((opcion = escribirOpcion()) != TERMINAR)
55 {
56 switch (opcion)
57 {
58 case IMPRIMIR: // crea un archivo de texto a partir del archivo de registros
59 crearArchivoTexto(creditoEntSal);
60 break;
61 case ACTUALIZAR: // actualiza el registro
62 actualizarRegistro(creditoEntSal);
63 break;
64 case NUEVO: // crea un registro
65 nuevoRegistro(creditoEntSal);
66 break;
67 case ELIMINAR: // elimina un registro existente
68 eliminarRegistro(creditoEntSal);
69 break;
70 default: // muestra un error si el usuario no selecciona una opción válida
71 cerr << "Opcion incorrecta" << endl;
72 break;
73 } // fin de switch
74
75 creditoEntSal.clear(); // restablece el indicador de fin de archivo
76 } // fin de while
77
78 return 0;
79 } // fin de main
80
81 // permite al usuario introducir la opción del menú
82 int escribirOpcion()
83 {
84 // muestra las opciones disponibles
85 cout << "\nEscriba su opcion" << endl
86 << "1 - almacenar un archivo de texto con formato de las cuentas" << endl
87 << " llamado \"imprimir.txt\" para imprimirlo" << endl
88 << "2 - actualizar una cuenta" << endl
89 << "3 - agregar una nueva cuenta" << endl
90 << "4 - eliminar una cuenta" << endl
91 << "5 - fin del programa\n? ";
92
93 int opcionMenu;
94 cin >> opcionMenu; // introduce la selección del menú que hizo el usuario
95 return opcionMenu;
96 } // fin de la función escribirOpcion
97
98 // crea un archivo de texto con formato para imprimirlo
99 void crearArchivoTexto(fstream &leerDelArchivo)
100 {
101 // crea un archivo de texto
102 ofstream archivoImprimirSalida("imprimir.txt", ios::out);
103
104 // sale del programa si ofstream no puede crear el archivo
105 if (!archivoImprimirSalida)
106 {
107 cerr << "No se pudo crear el archivo." << endl;
108 exit(1);
109 } // fin de if
110 }
```

Figura 17.15 | Programa de cuentas bancarias. (Parte 2 de 5).

```

112 archivoImprimirSalida << left << setw(10) << "Cuenta" << setw(16)
113 << "Apellido" << setw(11) << "Nombre" << right
114 << setw(10) << "Saldo" << endl;
115
116 // establece el apuntador de posición del archivo en el inicio de leerDelArchivo
117 leerDelArchivo.seekg(0);
118
119 // lee el primer registro del archivo de registros
120 DatosCliente cliente;
121 leerDelArchivo.read(reinterpret_cast< char * >(&cliente),
122 sizeof(DatosCliente));
123
124 // copia todos los registros del archivo de registros al archivo de texto
125 while (!leerDelArchivo.eof())
126 {
127 // escribe un solo registro en el archivo de texto
128 if (cliente.obtenerNumeroCuenta() != 0) // omite los registros vacíos
129 imprimirLinea(archivoImprimirSalida, cliente);
130
131 // lee el siguiente registro del archivo de registros
132 leerDelArchivo.read(reinterpret_cast< char * >(&cliente),
133 sizeof(DatosCliente));
134 } // fin de while
135 } // fin de la función crearArchivoTexto
136
137 // actualiza el saldo en el registro
138 void actualizarRegistro(fstream &actualizarArchivo)
139 {
140 // obtiene el número de la cuenta que se va a actualizar
141 int numeroCuenta = obtenerCuenta("Escriba la cuenta que se debe actualizar");
142
143 // desplaza el apuntador de posición del archivo al registro correcto en el archivo
144 actualizarArchivo.seekg((numeroCuenta - 1) * sizeof(DatosCliente));
145
146 // lee el primer registro del archivo
147 DatosCliente cliente;
148 actualizarArchivo.read(reinterpret_cast< char * >(&cliente),
149 sizeof(DatosCliente));
150
151 // actualiza el registro
152 if (cliente.obtenerNumeroCuenta() != 0)
153 {
154 imprimirLinea(cout, cliente); // muestra el registro
155
156 // solicita al usuario que especifique la transacción
157 cout << "\nEscriba el cargo (+) o pago (-): ";
158 double transaccion; // cargo o pago
159 cin >> transaccion;
160
161 // actualiza el saldo del registro
162 double saldoAnterior = cliente.obtenerSaldo();
163 cliente.establecerSaldo(saldoAnterior + transaccion);
164 imprimirLinea(cout, cliente); // muestra el registro
165
166 // desplaza el apuntador de posición del archivo al registro correcto en el archivo
167 actualizarArchivo.seekp((numeroCuenta - 1) * sizeof(DatosCliente));
168
169 // escribe el registro actualizado sobre el registro anterior en el archivo
170 actualizarArchivo.write(reinterpret_cast< const char * >(&cliente),
171 sizeof(DatosCliente));

```

Figura 17.15 | Programa de cuentas bancarias. (Parte 3 de 5).

```
172 } // fin de if
173 else // muestra un error si la cuenta no existe
174 cerr << "La cuenta #" << numeroCuenta
175 << " no tiene informacion." << endl;
176 } // fin de la función actualizarRegistro
177
178 // crea e inserta un registro
179 void nuevoRegistro(fstream &insertarEnArchivo)
180 {
181 // obtiene el número de cuenta que se debe crear
182 int numeroCuenta = obtenerCuenta("Escriba el nuevo numero de cuenta");
183
184 // desplaza el apuntador de posición del archivo al registro correcto en el archivo
185 insertarEnArchivo.seekg((numeroCuenta - 1) * sizeof(DatosCliente));
186
187 // lee un registro del archivo
188 DatosCliente cliente;
189 insertarEnArchivo.read(reinterpret_cast< char * >(&cliente),
190 sizeof(DatosCliente));
191
192 // crea un registro, si no es que ya existe
193 if (cliente.obtenerNumeroCuenta() == 0)
194 {
195 char apellidoPaterno[15];
196 char primerNombre[10];
197 double saldo;
198
199 // el usuario introduce el apellido paterno, primer nombre y saldo
200 cout << "Escriba apellido paterno, primer nombre y saldo\n? ";
201 cin >> setw(15) >> apellidoPaterno;
202 cin >> setw(10) >> primerNombre;
203 cin >> saldo;
204
205 // usa los valores para llenar los valores de la cuenta
206 cliente.establecerApellidoPaterno(apellidoPaterno);
207 cliente.establecerPrimerNombre(primerNombre);
208 cliente.establecerSaldo(saldo);
209 cliente.establecerNumeroCuenta(numeroCuenta);
210
211 // desplaza el apuntador de posición del archivo al registro correcto en el archivo
212 insertarEnArchivo.seekp((numeroCuenta - 1) * sizeof(DatosCliente));
213
214 // inserta el registro en el archivo
215 insertarEnArchivo.write(reinterpret_cast< const char * >(&cliente),
216 sizeof(DatosCliente));
217 } // fin de if
218 else // muestra un error si la cuenta ya existe
219 cerr << "La cuenta #" << numeroCuenta
220 << " ya contiene información." << endl;
221 } // fin de la función nuevoRegistro
222
223 // elimina un registro existente
224 void eliminarRegistro(fstream &eliminarDelArchivo)
225 {
226 // obtiene el número de cuenta que debe eliminar
227 int numeroCuenta = obtenerCuenta("Escriba la cuenta a eliminar");
228
229 // desplaza el apuntador de posición del archivo al registro correcto en el archivo
230 eliminarDelArchivo.seekg((numeroCuenta - 1) * sizeof(DatosCliente));
231 }
```

Figura 17.15 | Programa de cuentas bancarias. (Parte 4 de 5).

```

232 // lee el registro del archivo
233 DatosCliente cliente;
234 eliminarDelArchivo.read(reinterpret_cast< char * >(&cliente),
235 sizeof(DatosCliente));
236
237 // elimina el registro, si es que existe en el archivo
238 if (cliente.obtenerNumeroCuenta() != 0)
239 {
240 DatosCliente clienteEnBlanco; // crea un registro en blanco
241
242 // desplaza el apuntador de posición del archivo al registro correcto en el archivo
243 eliminarDelArchivo.seekp((numeroCuenta - 1) *
244 sizeof(DatosCliente));
245
246 // reemplaza el registro existente con uno en blanco
247 eliminarDelArchivo.write(
248 reinterpret_cast< const char * >(&clienteEnBlanco),
249 sizeof(DatosCliente));
250
251 cout << "La cuenta #" << numeroCuenta << " se elimino.\n";
252 } // fin de if
253 else // muestra un error si el registro no existe
254 cerr << "La cuenta #" << numeroCuenta << " esta vacia.\n";
255 } // fin de eliminarRegistro
256
257 // muestra un solo registro
258 void imprimirLinea(ostream &salida, const DatosCliente ®istro)
259 {
260 salida << left << setw(10) << registro.obtenerNumeroCuenta()
261 << setw(16) << registro.obtenerApellidoPaterno()
262 << setw(11) << registro.obtenerPrimerNombre()
263 << setw(10) << setprecision(2) << right << fixed
264 << showpoint << registro.obtenerSaldo() << endl;
265 } // fin de la función imprimirLinea
266
267 // obtiene el valor del número de cuenta del usuario
268 int obtenerCuenta(const char * const indicador)
269 {
270 int numeroCuenta;
271
272 // obtiene el valor del número de cuenta
273 do
274 {
275 cout << indicador << " (1 - 100): ";
276 cin >> numeroCuenta;
277 } while (numeroCuenta < 1 || numeroCuenta > 100);
278
279 return numeroCuenta;
280 } // fin de la función obtenerCuenta

```

**Figura 17.15** | Programa de cuentas bancarias. (Parte 5 de 5).

El programa tiene cinco opciones (la opción 5 es para terminar el programa). La opción 1 llama a la función `crearArchivoTexto` para almacenar una lista con formato de toda la información de las cuentas en un archivo de texto llamado `imprimir.txt`, el cual se puede imprimir. La función `crearArchivoTexto` (líneas 100 a 135) recibe un objeto `fstream` como un argumento a utilizar para introducir datos desde el archivo `credito.dat`. La función `crearArchivoTexto` invoca a la función miembro `read` de `istream` (líneas 132 y 133) y utiliza las técnicas de acceso a archivos secuenciales de la figura 17.14 para introducir datos desde `credito.dat`. La función `imprimirLinea`, que vimos en la sección 17.10, se utiliza para escribir los datos en el archivo `imprimir.txt`. Observe que `crearArchivoTexto` utiliza la función miembro `seekg` de `istream` (línea 117) para asegurar que el apuntador de posición del archivo esté en el inicio del archivo. Después de seleccionar la opción 1, el archivo `imprimir.txt` contiene lo siguiente:

| Cuenta | Apellido | Nombre | Saldo  |
|--------|----------|--------|--------|
| 29     | Brown    | Nancy  | -24.54 |
| 33     | Dunn     | Stacey | 314.33 |
| 37     | Barker   | Doug   | 0.00   |
| 88     | Smith    | Dave   | 258.34 |
| 96     | Stone    | Sam    | 34.98  |

La opción 2 llama a `actualizarRegistro` (líneas 138 a 176) para actualizar una cuenta. Esta función sólo actualiza un registro existente, por lo que primero determina si el registro especificado está vacío. En las líneas 148 y 149 se leen datos y se colocan en el objeto `cliente`, usando la función miembro `read` de `istream`. Después, en la línea 152 se compara el valor devuelto por `obtenerNumeroCuenta` de la estructura `cliente` con cero, para determinar si el registro contiene información. Si este valor es cero, en las líneas 174 y 175 se imprime un mensaje de error que indica que el registro está vacío. Si el registro contiene información, en la línea 154 se muestra el registro mediante la función `imprimirLinea`, en la línea 159 se introduce el monto de la transacción y en las líneas 162 a 171 se calcula el nuevo saldo y se vuelve a escribir el registro en el archivo. Un conjunto típico de resultados para la opción 2 es

```
Escriba la cuenta que se debe actualizar (1 - 100): 37
37 Barker Doug 0.00

Escriba el cargo (+) o pago (-): +87.99
37 Barker Doug 87.99
```

La opción 3 llama a la función `nuevoRegistro` (líneas 179 a 221) para agregar una nueva cuenta al archivo. Si el usuario escribe un número de cuenta para una cuenta existente, `nuevoRegistro` muestra un mensaje de error indicando que la cuenta existe (líneas 219 y 220). Esta función agrega una nueva cuenta de la misma forma que el programa de la figura 17.12. Un conjunto típico de resultados para la opción 3 es

```
Escriba el nuevo numero de cuenta (1 - 100): 22
Escriba apellido paterno, primer nombre y saldo
? Johnston Sarah 247.45
```

La opción 4 llama a la función `eliminarRegistro` (líneas 224 a 255) para eliminar un registro del archivo. En la línea 227 se pide al usuario que introduzca el número de cuenta. Sólo puede eliminarse un registro existente, por lo que si la cuenta especificada está vacía, en la línea 254 se muestra un mensaje de error. Si la cuenta existe, en las líneas 247 a 249 se reinicializa esa cuenta al copiar un registro vacío (`clienteEnBlanco`) al archivo. En la línea 251 se muestra un mensaje para informar al usuario que se ha eliminado el registro. Un conjunto típico de resultados para la opción 4 es

```
Escriba la cuenta a eliminar (1 - 100): 29
La cuenta #29 se elimino.
```

Observe que en la línea 43 se abre el archivo `credito.dat` mediante la creación de un objeto `fstream` para lectura y escritura, usando los modos `ios::in` e `ios::out` en conjunto mediante un OR.

## 17.12 Generalidades acerca de la serialización de objetos

En este capítulo y en el capítulo 15 se presentó el estilo orientado a objetos de las operaciones de entrada/salida. Sin embargo, nuestros ejemplos se concentraron en la E/S de los tipos fundamentales, en vez de los objetos de tipos definidos por el usuario. En el capítulo 11 mostramos cómo realizar operaciones de entrada y salida con objetos, mediante la sobrecarga de operadores. Realizamos operaciones de entrada con objetos mediante la sobrecarga del operador de extracción (`>>`) para el objeto `istream` apropiado. Realizamos operaciones de salida con objetos mediante la sobrecarga del operador de inserción de flujo (`<<`) para el objeto `ostream` apropiado. En ambos casos, sólo se realizaron operaciones de entrada o salida con los miembros de datos de un objeto y, en cada caso, se hicieron con un formato significativo sólo para los objetos de ese tipo específico. Las funciones miembro de un objeto no se envían o reciben con los datos de un objeto; en vez de ello, una copia de las funciones miembro de la clase permanece disponible en forma interna y es compartida por todos los objetos de la clase.

Cuando se escriben los miembros de datos de un objeto en un archivo en disco, se pierde la información del tipo del objeto. Sólo se almacenan los valores de los atributos del objeto, y no la información del tipo en el disco. Si el programa que lee estos datos conoce el tipo del objeto al que éstos corresponden, puede leer los datos y colocarlos en un objeto de ese tipo, como hicimos en nuestros ejemplos con archivos de acceso aleatorio.

Cuando almacenamos objetos de distintos tipos en el mismo archivo, ocurre un problema interesante. ¿Cómo podemos diferenciarlos (o sus colecciones de datos miembro) al leerlos e introducirlos en un programa? El problema es que, por lo general, los objetos no tienen campos de tipos (vimos este problema en el capítulo 13).

Una metodología utilizada por varios lenguajes de programación es lo que se conoce como **serialización de objetos**. Un **objeto serializado** es un objeto que se representa como una secuencia de bytes que incluye los datos del objeto, así como información acerca de su tipo y de los tipos de datos almacenados en el mismo. Una vez que se escribe un objeto serializado en un archivo, se puede leer del mismo y **deserializarse**; es decir, se pueden utilizar la información del tipo y los bytes que representan al objeto y sus datos para recrearlo en memoria. C++ no proporciona un mecanismo de serialización integrado; sin embargo, existen bibliotecas de C++ de código fuente abierto y de terceros que soportan la serialización de objetos. Las Bibliotecas Boost de C++ de código fuente abierto ([www.boost.org](http://www.boost.org)) ofrecen soporte para serializar objetos en los formatos de texto, binario y en lenguaje de marcado extensible (XML) ([www.boost.org/libs/serialization/doc/index.html](http://www.boost.org/libs/serialization/doc/index.html)). En el capítulo 24 veremos las generalidades acerca de las Bibliotecas Boost de C++.

## 17.13 Repaso

En este capítulo presentamos varias técnicas de procesamiento de archivos para manipular datos persistentes. El lector aprendió que los datos se almacenan en las computadoras en forma de 0s y 1s, y que las combinaciones de estos valores forman bytes, campos, registros y, en ciertos casos, archivos. Vimos una introducción a las diferencias entre los flujos basados en caracteres y basados en bytes, y a varias plantillas de clases de procesamiento de archivos en el archivo de encabezado `<fstream>`. Despues, el lector aprendió a utilizar el procesamiento de archivos secuenciales para manipular los registros almacenados en orden, mediante el campo clave de registro. También aprendió a utilizar archivos de acceso aleatorio para obtener y manipular al instante registros de longitud fija. Presentamos un ejemplo práctico considerable sobre el procesamiento de transacciones, que utiliza un archivo de acceso aleatorio para realizar un procesamiento con acceso “instantáneo”. Por último, vimos los conceptos básicos de la serialización de objetos. En el siguiente capítulo hablaremos sobre las operaciones comunes de manipulación de cadenas que proporciona la plantilla de clase `basic_string`. También presentaremos las herramientas de procesamiento de flujos de cadena, que permiten realizar operaciones de entrada y salida de cadenas hacia/desde la memoria.

## Resumen

### Sección 17.1 Introducción

- Los archivos se utilizan para la persistencia de los datos: la retención permanente de datos.
- Las computadoras almacenan archivos en dispositivos de almacenamiento secundario, como discos duros, CDs, DVDs, memoria flash y cintas magnéticas.

### Sección 17.2 Jerarquía de datos

- El elemento más pequeño de datos que soportan las computadoras se llama bit (abreviación de “dígito binario” en inglés: un dígito que puede asumir uno de dos valores, 0 o 1).
- Los dígitos, letras y símbolos especiales se conocen como caracteres.
- El conjunto de todos los caracteres utilizados para escribir programas y representar elementos de datos en una computadora específica se conoce como el conjunto de caracteres de esa computadora.
- Los bytes están compuestos de ocho bits.
- Así como los caracteres están compuestos de bits, los campos están compuestos de caracteres. Un campo es un grupo de caracteres que transmite cierto significado.
- Por lo general, un registro (es decir, una clase en C++) está compuesto de varios campos (es decir, datos miembro en C++).
- Por lo menos un campo en un registro se selecciona como clave de registro para identificar que un registro pertenece a una persona o entidad específica, que es distinta de todos los demás registros en el archivo.
- En un archivo secuencial, los registros comúnmente se almacenan en orden, con base en el campo clave de registro.
- A menudo, un grupo de archivos relacionados se almacena en una base de datos.
- Una colección de programas diseñados para crear y administrar bases de datos se denomina sistema de administración de bases de datos (DBMS).

### **Sección 17.3 Archivos y flujos**

- C++ ve a cada archivo como una secuencia de bytes.
- Cada archivo termina con un marcador de fin de archivo, o en un número de byte específico registrado en una estructura de datos administrativa, mantenida por el sistema.
- Al abrir un archivo, se crea un objeto y se asocia un flujo a ese objeto.
- Para realizar el procesamiento de archivos en C++, se deben incluir los archivos de encabezado `<iostream>` y `<fstream>`.
- El encabezado `<fstream>` incluye las definiciones para las plantillas de clases de flujos `basic_ifstream` (operaciones de entrada con archivos), `basic_ofstream` (operaciones de salida con archivos) y `basic_fstream` (operaciones de entrada y salida con archivos).
- Cada plantilla de clase tiene una especialización de plantilla predefinida que permite operaciones de E/S con caracteres. La biblioteca `<fstream>` proporciona alias con `typedef` para estas especializaciones de plantillas. La definición `typedef ifstream` representa una especialización de `basic_ifstream` que permite operaciones de entrada con valores `char` desde un archivo. La definición `typedef ofstream` representa una especialización de `basic_ofstream` que permite operaciones de salida con valores `char` hacia archivos. La definición `typedef fstream` representa una especialización de `basic_fstream` que permite operaciones de entrada/salida con valores `char` desde/hacia archivos.
- Las plantillas de procesamiento de archivos se derivan de las plantillas de clases `basic_istream`, `basic_ostream` y `basic_iostream`, respectivamente. Por ende, todas las funciones miembro, operadores y manipuladores que pertenecen a estas plantillas también se pueden aplicar a los flujos de archivos.

### **Sección 17.4 Creación de un archivo secuencial**

- C++ no impone una estructura sobre un archivo, por lo que debemos estructurar los archivos para que cumplan con los requerimientos de la aplicación.
- Un archivo se puede abrir en modo de salida cuando se crea un objeto `ofstream`. Se pasan dos argumentos al constructor del objeto: el nombre del archivo y el modo de apertura del archivo.
- Para un objeto `ofstream`, el modo de apertura del archivo puede ser `ios::out` para enviar datos a un archivo, o `ios::app` para adjuntar datos al final de un archivo. Los archivos existentes que se abren con el modo `ios::out` se truncan; todos los datos en el archivo se descartan. Si el archivo especificado no existe aún, entonces el objeto `ofstream` crea el archivo, usando ese nombre de archivo.
- De manera predeterminada, los objetos `ofstream` se abren en modo de salida, por lo que no se requiere el segundo argumento.
- Un objeto `ofstream` se puede crear sin abrir un archivo específico; se puede adjuntar un archivo al objeto más adelante, con la función miembro `open`.
- La función miembro operador `operator!` de `ios` determina si un flujo se abrió en forma correcta. Este operador se puede utilizar en una condición que devuelva un valor verdadero si se establece el bit `failbit` o el bit `badbit` para el flujo en la operación de apertura. Algunos posibles errores que producen un resultado verdadero son tratar de abrir un archivo inexistente en modo de lectura, tratar de abrir un archivo para leer o escribir sin permiso, y abrir un archivo para escribir cuando no hay espacio disponible en el disco.
- Otra función miembro operador sobrecargada de `ios` (`operator void *`) convierte el flujo en un apuntador, para que se pueda evaluar como 0 o un valor distinto de cero. Cuando se utiliza el valor de un apuntador como una condición, un apuntador nulo representa el valor `bool false` y un apuntador no nulo representa el valor `bool true`. Si se ha establecido el bit `failbit` o el bit `badbit` para el flujo, se devuelve 0 (`false`).
- Al introducir el indicador de fin de archivo, se establece el bit `failbit` para `cin`.
- La función `operator void *` se puede utilizar para evaluar un objeto de entrada para el fin de archivo, en vez de llamar a la función miembro `eof` de manera explícita en el objeto de entrada.
- Cuando se hace una llamada al destructor de un objeto flujo, se cierra el flujo correspondiente. También se puede cerrar el flujo de manera explícita, mediante la función miembro `close` del flujo.
- Al cerrar los archivos en forma explícita cuando ya no son necesarios, se puede reducir el uso de los recursos de un programa.

### **Sección 17.5 Cómo leer datos de un archivo secuencial**

- Los archivos almacenan datos, para que puedan obtenerse y procesarse cuando sea necesario.
- Al crear un objeto `ifstream`, se abre un archivo en modo de entrada. El constructor de `ifstream` puede recibir el nombre del archivo y el modo de apertura del mismo como argumentos.
- Debemos abrir un archivo en modo de entrada solamente (usando `ios::in`) si el contenido del archivo no se debe modificar. Esto evita la modificación accidental del contenido del archivo, y es un ejemplo del principio del menor privilegio.
- Los objetos de la clase `ifstream` se abren en modo de entrada de manera predeterminada, por lo que no se requiere el segundo argumento para el constructor.
- Al igual que un objeto `ofstream`, un objeto `ifstream` se puede crear sin tener que abrir un archivo específico, debido a que se le puede adjuntar uno más adelante.

- Para obtener los datos secuencialmente de un archivo, los programas generalmente empiezan a leer desde el inicio del archivo, y leen todos los datos en forma consecutiva hasta encontrar los datos deseados.
- Tanto `istream` como `ostream` proporcionan funciones miembro para reposicionar el apuntador de posición del archivo (el número de byte del siguiente byte en el archivo que se debe leer o escribir). Estas funciones miembro son `seekg` (“buscar obtener”) para `istream` y `seekp` (“buscar colocar”) para `ostream`. Cada objeto `istream` tiene un “apuntador obtener”, el cual indica el número de byte en el archivo desde el que se va a realizar la siguiente operación de entrada, y cada objeto `ostream` tiene un “apuntador colocar”, el cual indica el número de byte en el archivo en el que se deben colocar los datos de la siguiente operación de salida.
- Por lo general, el argumento para `seekg` es un entero largo. Se puede especificar un segundo argumento para indicar la dirección de búsqueda, que puede ser `ios::beg` (el valor predeterminado) para un posicionamiento relativo al inicio de un flujo, `ios::cur` para un posicionamiento relativo a la posición actual en un flujo, o `ios::end` para un posicionamiento relativo al final de un flujo.
- El apuntador de posición del archivo es un valor entero que especifica la ubicación en el archivo como un número de bytes a partir de la ubicación inicial del archivo (es decir, el desplazamiento desde el inicio del archivo).
- Las funciones miembro `tellg` y `tellp` se proporcionan para devolver las ubicaciones actuales de los apuntadores “obtener” y “colocar”, respectivamente.

### **Sección 17.6 Actualización de archivos secuenciales**

- Los datos a los que se da formato y se escriben en un archivo secuencial no se pueden modificar sin el riesgo de destruir otros datos en el archivo. El problema es que los registros pueden variar en tamaño.
- La actualización se puede realizar de una manera compleja. Los registros antes del que se va a actualizar se copian a un nuevo archivo, después el registro actualizado se escribe en el nuevo archivo, y luego los registros que siguen después del registro modificado se copian al nuevo archivo. Esto requiere el procesamiento de cada registro en el archivo para actualizar un solo registro. No obstante, si se van a actualizar muchos registros en una pasada del archivo, esta técnica puede ser aceptable.

### **Sección 17.7 Archivos de acceso aleatorio**

- Los archivos secuenciales son inapropiados para las aplicaciones de acceso instantáneo, en los que un registro específico se debe localizar de inmediato.
- Las aplicaciones de acceso comunes son: los sistemas de reservación de aerolíneas, sistemas bancarios, sistemas de punto de venta, cajeros automáticos y otros tipos de sistemas de procesamiento de transacciones, que requieren un acceso rápido a ciertos datos.
- El acceso instantáneo se logra mediante los archivos de acceso aleatorio. Se puede acceder a los registros individuales de un archivo de acceso aleatorio en forma directa (y rápida), sin tener que buscar en otros registros.
- El método más sencillo para dar formato a los archivos para un acceso aleatorio es requerir que todos los registros en un archivo tengan la misma longitud fija. Al utilizar registros de longitud fija que tengan el mismo tamaño, es más fácil para el programa calcular (como una función del tamaño del registro y la clave de registro) la ubicación exacta de cualquier registro, relativa al inicio del archivo.
- Se pueden insertar datos en un archivo de acceso aleatorio sin destruir los demás datos en el archivo.
- Los datos almacenados con anterioridad se pueden actualizar o eliminar sin tener que volver a escribir el archivo completo.

### **Sección 17.8 Creación de un archivo de acceso aleatorio**

- La función miembro `write` de `ostream` envía como salida un número fijo de bytes (empezando en una ubicación específica en memoria) al flujo especificado. La función `write` escribe los datos en la ubicación en el archivo especificada por el apuntador “colocar” de posición del archivo.
- La función miembro `read` de `istream` introduce un número fijo de bytes del flujo especificado hacia un área en la memoria, empezando en una dirección especificada. Si el flujo está asociado con un archivo, la función `read` introduce bytes en la ubicación en el archivo especificada por el apuntador “obtener” de posición del archivo.
- La función `write` trata su primer argumento como un grupo de bytes, al ver el objeto en memoria un valor `const char *`, el cual es un apuntador a un byte (recuerde que un `char` es un byte). Empezando a partir de esa ubicación, la función `write` envía a la salida el número de bytes especificado por su segundo argumento. Después, se puede utilizar la función `read` de `istream` para leer los bytes y colocarlos de vuelta en la memoria.
- C++ proporciona el operador `reinterpret_cast` para convertir un apuntador de un tipo a un tipo de apuntador no relacionado. Podemos utilizar también este operador de conversión para realizar conversiones entre los tipos de apuntadores y los tipos enteros, y viceversa.
- Una operación con `reinterpret_cast` se realiza en tiempo de compilación y no modifica el valor del objeto al que apunta su operando.
- Un programa que lee datos sin formato (escritos por `write`) debe compilarse y ejecutarse en un sistema compatible con el programa que escribió los datos, ya que distintos sistemas pueden representar internamente los datos de una forma diferente.

- Los objetos de la clase `string` no tienen un tamaño uniforme, sino que utilizan la memoria asignada en forma dinámica para dar cabida a las cadenas de varias longitudes.
- La función miembro `data` de `string` devuelve un arreglo que contiene los caracteres del objeto `string`. No se garantiza que este arreglo tenga terminación nula.
- La función miembro `size` de `string` obtiene la longitud de un objeto `string`.
- El modo de apertura de archivo `ios::binary` indica que un archivo se debe abrir en modo binario.

### **Sección 17.9 Cómo escribir datos al azar a un archivo de acceso aleatorio**

- Para combinar varios modos de apertura de archivos, hay que separar cada modo de apertura de los otros mediante el operador OR inclusivo a nivel de bits (`|`).

### **Sección 17.10 Cómo leer de un archivo de acceso aleatorio en forma secuencial**

- La función `read` de `istream` introduce un número especificado de bytes, desde la posición actual del flujo especificado, hacia un objeto.
- Una función que recibe un parámetro `ostream` puede recibir cualquier objeto `ostream` (como `cout`) o cualquier objeto de una clase derivada de `ostream` (como un objeto de tipo `stream`) como argumento. Esto significa que se puede utilizar la misma función, por ejemplo, para realizar operaciones de salida en el flujo de salida estándar y en un flujo de archivos, sin tener que especificar funciones separadas.

### **Sección 17.12 Generalidades acerca de la serialización de objetos**

- Cuando se envían datos miembro de un objeto a un archivo en disco, perdemos la información del tipo de ese objeto. Sólo almacenamos los valores de los atributos de los objetos, no la información del tipo, en el disco. Si el programa que lee estos datos conoce el tipo del objeto al que corresponden, puede leer los datos y colocarlos en un objeto de ese tipo.
- Varios lenguajes de programación soportan la serialización de objetos. Un objeto serializado es un objeto que se representa como una secuencia de bytes que incluye los datos del objeto, así como información acerca del tipo del objeto y los tipos de datos almacenados en él. Una vez que se ha escrito un objeto serializado en un archivo, se puede leer de éste y deserializarse; es decir, la información de tipos y los bytes que representan al objeto y sus datos se puede utilizar para recrear el objeto en memoria.
- C++ no proporciona un mecanismo de serialización integrado; sin embargo, hay bibliotecas de C++ de terceros y de código fuente abierto que soportan la serialización de objetos.
- Las Bibliotecas Boost de C++ de código fuente abierto proporcionan soporte para los objetos en los formatos de texto, binario y en lenguaje de marcado extensible (XML) ([www.boost.org/libs/serialization/doc/index.html](http://www.boost.org/libs/serialization/doc/index.html)).

## **Terminología**

|                                                               |                                                              |
|---------------------------------------------------------------|--------------------------------------------------------------|
| abrir un archivo                                              | <code>fstream</code>                                         |
| adjuntar datos a un archivo                                   | <code>&lt;fstream&gt;</code> , archivo de encabezado         |
| aplicación de acceso instantáneo                              | <code>Ifstream</code>                                        |
| apuntador de posición del archivo                             | <code>ios::app</code> , modo de apertura de archivo          |
| archivo                                                       | <code>ios::ate</code> , modo de apertura de archivo          |
| archivo de acceso aleatorio                                   | <code>ios::beg</code> , punto de inicio de búsqueda          |
| archivo secuencial                                            | <code>ios::binary</code> , modo de apertura de archivo       |
| base de datos                                                 | <code>ios::cur</code> , punto de inicio de búsqueda          |
| bit                                                           | <code>ios::end</code> , punto de inicio de búsqueda          |
| buscar dirección                                              | <code>ios::in</code> , modo de apertura de archivo           |
| byte                                                          | <code>ios::out</code> , modo de apertura de archivo          |
| campo                                                         | <code>ios::trunc</code> , modo de apertura de archivo        |
| campo de caracteres                                           | jerarquía de datos                                           |
| clave de registro                                             | modos de apertura de archivo                                 |
| <code>clog</code> (flujo de error estándar con búfer)         | nombre de archivo                                            |
| <code>close</code> , función miembro de <code>ofstream</code> | <code>ofstream</code>                                        |
| conjunto de caracteres                                        | <code>open</code> , función miembro de <code>ofstream</code> |
| <code>data</code> , función miembro de <code>string</code>    | <code>operator void *</code>                                 |
| desplazamiento desde el inicio de un archivo                  | persistencia de los datos                                    |
| dígito binario                                                | <code>read</code> , función miembro de <code>istream</code>  |
| dígito decimal                                                | registro                                                     |
| dispositivo de almacenamiento secundario                      | <code>seekg</code> , función miembro de <code>istream</code> |
| entrada/salida de objetos                                     | <code>seekp</code> , función miembro de <code>ostream</code> |
| fin de archivo                                                | símbolo especial                                             |

sistema de administración de bases de datos (DBMS)  
 sistema de procesamiento de transacciones  
 size, función de string

tellg, función miembro de istream  
 tellp, función miembro de ostream  
 truncar un archivo existente

## Ejercicios de autoevaluación

**17.1** Complete los siguientes enunciados:

- Básicamente, todos los elementos de datos procesados por una computadora se reducen a combinaciones de \_\_\_\_\_ y \_\_\_\_\_.
- El elemento más pequeño de datos que puede procesar una computadora se llama \_\_\_\_\_.
- Un \_\_\_\_\_ es un grupo de registros relacionados.
- Los dígitos, letras y símbolos especiales se conocen como \_\_\_\_\_.
- Un grupo de archivos relacionados se llama \_\_\_\_\_.
- La función miembro \_\_\_\_\_ de los flujos de archivos fstream, ifstream y ofstream cierra un archivo.
- La función miembro \_\_\_\_\_ de istream lee un carácter del flujo especificado.
- La función miembro \_\_\_\_\_ de los flujos de archivos fstream, ifstream y ostream abre un archivo.
- La función miembro \_\_\_\_\_ de istream se utiliza comúnmente cuando se leen datos de un archivo en aplicaciones de acceso aleatorio.
- Las funciones miembro \_\_\_\_\_ y \_\_\_\_\_ de istream y ostream establecen el apuntador de posición del archivo en una ubicación específica de un flujo de entrada o salida, respectivamente.

**17.2** Indique si cada uno de los siguientes enunciados es *verdadero* o *falso*. En caso de ser *falso*, explique por qué.

- La función miembro `read` no puede utilizarse para leer datos del objeto de entrada `cin`.
- El programador debe crear los objetos `cin`, `cout`, `cerr` y `clog` de manera explícita.
- Un programa debe llamar a la función `close` de manera explícita para cerrar un archivo asociado con un objeto `ifstream`, `ofstream` o `fstream`.
- Si el apuntador de posición del archivo apunta a una ubicación en un archivo secuencial que no sea el inicio del archivo, éste se debe cerrar y volver a abrir para leer datos desde el inicio del mismo.
- La función miembro `write` de ostream puede escribir en el flujo de salida estándar `cout`.
- Los datos en archivos secuenciales siempre se actualizan sin sobrescribir los datos aledaños.
- Es innecesario buscar en todos los registros de un archivo de acceso aleatorio para encontrar un registro específico.
- Los registros en los archivos de acceso aleatorio deben ser de una longitud uniforme.
- Las funciones miembro `seekp` y `seekg` deben buscar en forma relativa al inicio de un archivo.

**17.3** Asuma que cada uno de los siguientes enunciados se aplica al mismo programa.

- Escriba una instrucción que abra el archivo `maestant.dat` en modo de entrada; use un objeto `ifstream` llamado `maestAntEntrada`.
- Escriba una instrucción que abra el archivo `trans.dat` en modo de entrada; use un objeto `ifstream` llamado `transaccionEnt`.
- Escriba una instrucción que abra el archivo `maestnuevo.dat` en modo de salida (y de creación); use el objeto `ofstream` `maestNuevSalida`.
- Escriba una instrucción que lea un registro del archivo `maestant.dat`. El registro consiste en el entero `numeroCuenta`, la cadena `nombre` y el valor de punto flotante `saldoActual`; use el objeto `ifstream` `maestAntEntrada`.
- Escriba una instrucción que lea un registro del archivo `trans.dat`. El registro consiste en el entero `numCuenta` y el valor de punto flotante `montoDolares`; utilice el objeto `ifstream` `transaccionEnt`.
- Escriba una instrucción que escriba un registro en el archivo `maestnuev.dat`. El registro consiste en el entero `numCuenta`, la cadena `nombre` y el valor de punto flotante `saldoActual`; utilice el objeto `ofstream` `maestNuevSalida`.

**17.4** Busque el (los) error(es) y muestre cómo corregirlo(s) en cada uno de los siguientes enunciados.

- El archivo `porpagar.dat` al que hace referencia el objeto `ofstream` `porPagarSalida` no se ha abierto.  

```
porPagarSalida << cuenta << empresa << monto << endl;
```
- La siguiente instrucción debe leer un registro del archivo `porpagar.dat`. El objeto `ifstream` `porPagarEntrada` hace referencia a este archivo, y el objeto `ifstream` `porCobrarEntrada` hace referencia al archivo `porcobrar.dat`.  

```
porCobrarEntrada >> cuenta >> empresa >> monto;
```
- El archivo `herramientas.dat` debe abrirse para agregarle datos al archivo sin descartar los datos actuales.  

```
ofstream herramientasSalida("herramientas.dat", ios::out);
```

## Respuestas a los ejercicios de autoevaluación

- 17.1** a) 1s, 0s. b) bit. c) archivo. d) caracteres. e) base de datos. f) close. g) get. h) open. i) read. j) seekg, seekp.
- 17.2** a) Falso. La función read puede leer de cualquier objeto flujo de entrada derivado de `istream`.  
 b) Falso. Estos cuatro flujos se crean de manera automática para el programador. El encabezado `<iostream>` debe incluirse en un archivo para usarlos. Este encabezado incluye declaraciones para cada objeto flujo.  
 c) Falso. Los archivos se cerrarán cuando se ejecuten los destructores para los objetos `ifstream`, `ofstream` o `fstream`, cuando los objetos flujo quedan fuera de alcance, o antes de que termine la ejecución del programa, pero es una buena práctica de programación cerrar todos los archivos en forma explícita mediante `close`, una vez que ya no sean necesarios.  
 d) Falso. La función miembro `seekp` o `seekg` se puede utilizar para reposicionar el apuntador “obtener” o “colocar” de posición del archivo en el inicio del archivo.  
 e) Verdadero.  
 f) Falso. En la mayoría de los casos, los registros de archivos secuenciales no son de una longitud uniforme. Por lo tanto, es posible que al actualizar un registro se sobreescrbían otros datos.  
 g) Verdadero.  
 h) Falso. Por lo general, los registros en un archivo de acceso aleatorio son de longitud uniforme.  
 i) Falso. Es posible buscar desde el inicio del archivo, desde el final del archivo y desde la posición actual en el archivo.
- 17.3** a) `ifstream maestAntEntrada( "maestant.dat", ios::in );`  
 b) `ifstream transaccionEnt( "trans.dat", ios::in );`  
 c) `ofstream maestNuevSalida( "maestnuev.dat", ios::out );`  
 d) `maestNuevEntrada >> numeroCuenta >> nombre >> saldoActual;`  
 e) `transaccionEnt >> numCuenta >> montoDolares;`  
 f) `maestNuevEntrada << numCuenta << nombre << saldoActual;`
- 17.4** a) *Error:* el archivo `porpagar.dat` no se ha abierto antes de tratar de enviar datos al flujo.  
*Corrección:* use la función `open` de `ostream` para abrir `porpagar.dat` en modo de salida.  
 b) *Error:* se está utilizando el objeto `istream` incorrecto para leer un registro del archivo llamado `porpagar.dat`.  
*Corrección:* use el objeto `porPagarEnt` de `istream` para hacer referencia a `porpagar.dat`.  
 c) *Error:* el contenido del archivo se descarta, debido a que se abre en modo de salida (`ios::out`).  
*Corrección:* para agregar datos al archivo, ábralo para actualizar (`ios::ate`) o para adjuntar (`ios::app`).

## Ejercicios

- 17.5** Complete los siguientes enunciados:
- Las computadoras almacenan grandes cantidades de datos en dispositivos de almacenamiento secundario, tales como \_\_\_\_\_.
  - Un \_\_\_\_\_ está compuesto de varios campos.
  - Para facilitar la recuperación de registros específicos de un archivo, debe seleccionarse un campo en cada registro como \_\_\_\_\_.
  - La gran mayoría de información que se almacena en los sistemas computacionales se guarda en archivos \_\_\_\_\_.
  - Un grupo de caracteres relacionados que transmiten significado se llama \_\_\_\_\_.
  - Los objetos de flujo estándar declarados por el encabezado `<iostream>` son \_\_\_\_\_, \_\_\_\_\_ y \_\_\_\_\_.
  - La función miembro \_\_\_\_\_ de `ostream` envía un carácter al flujo especificado.
  - La función miembro \_\_\_\_\_ de `istream` reposiciona el apuntador de posición del archivo en un archivo.
- 17.6** Determine cuál de los siguientes enunciados es *verdadero* y cuál es *falso*. Si es *falso*, explique por qué.
- Las impresionantes funciones realizadas por las computadoras involucran esencialmente la manipulación de ceros y unos.
  - Las personas prefieren manipular bits en vez de caracteres y campos, debido a que los bits son más compactos.
  - Las personas especifican los programas y elementos de datos como caracteres; después, las computadoras manipulan y procesan estos caracteres como grupos de ceros y unos.
  - El código postal de 5 dígitos de una persona es un ejemplo de un campo numérico.
  - El domicilio de una persona se considera generalmente como un campo alfabético en las aplicaciones computacionales.

- f) Los elementos de datos que se representan en las computadoras forman una jerarquía de datos, en la cual los elementos de datos se hacen más grandes y complejos, a medida que progresamos de campos a caracteres, de caracteres a bits, etcétera.
- g) Una clave de registro identifica que un registro pertenece a un campo específico.
- h) La mayoría de las empresas almacenan toda su información en un solo archivo, para poder facilitar el procesamiento computacional de la información.
- i) Cuando un programa crea un archivo, éste es retenido automáticamente por la computadora para cuando se haga referencia a él en un futuro; por ello se dice que los archivos son persistentes.

**17.7** El ejercicio de autoevaluación 17.3 pide al lector que escriba una serie de instrucciones individuales. En realidad, estas instrucciones forman el núcleo de un tipo importante de programa para procesar archivos: un programa para asociar archivos. En el procesamiento de datos comercial, es común tener varios archivos en cada sistema de aplicaciones. Por ejemplo, en un sistema de cuentas por cobrar hay generalmente un archivo maestro, el cual contiene información detallada acerca de cada cliente, como su nombre, dirección, número telefónico, saldo deudor, límite de crédito, términos de descuento, acuerdos contractuales y posiblemente un historial condensado de compras recientes y pagos en efectivo.

A medida que ocurren las transacciones (es decir, a medida que se generan las ventas y llegan los pagos en efectivo), se introducen en un archivo. Al final de cada periodo de negocios (un mes para algunas compañías, una semana para otras y un día en algunos casos), el archivo de transacciones (llamado *trans.dat* en el ejercicio 17.3) se aplica al archivo maestro (llamado *maestant.dat* en el ejercicio 17.3) para actualizar el registro de compras y pagos de cada cuenta. Durante una actualización, el archivo maestro se rescribe como un nuevo archivo (*maestnue.dat*), el cual se utiliza al final del siguiente periodo de negocios para empezar de nuevo el proceso de actualización.

Los programas para asociar archivos deben tratar con ciertos problemas que no existen en programas de un solo archivo. Por ejemplo, no siempre ocurre una asociación. Si un cliente en el archivo maestro no ha realizado compras ni pagos en efectivo en el periodo actual de negocios, no aparecerá ningún registro para este cliente en el archivo de transacciones. De manera similar, un cliente que haya realizado compras o pagos en efectivo podría haberse mudado recientemente a esta comunidad, y tal vez la compañía no haya tenido la oportunidad de crear un registro maestro para este cliente.

Use las instrucciones del ejercicio 17.3 como base para escribir un programa completo para asociar archivos de cuentas por cobrar. Utilice el número de cuenta en cada archivo como la clave de registro para fines de asociar los archivos. Suponga que cada archivo es un archivo secuencial con registros almacenados en orden ascendente, por número de cuenta.

Cuando ocurra una coincidencia (es decir, que aparezcan registros con el mismo número de cuenta en el archivo maestro y en el archivo de transacciones), sume el monto en dólares del registro de transacciones al saldo actual en el registro maestro, y escriba el registro *maestnue.dat*. (Suponga que las compras se indican mediante montos positivos en el archivo de transacciones, y los pagos mediante montos negativos). Cuando haya un registro maestro para una cuenta específica, pero no haya un registro de transacciones correspondiente, simplemente escriba el registro maestro en *maestnue.dat*. Cuando haya un registro de transacciones pero no un registro maestro correspondiente, imprima el mensaje de error "Hay un registro de transacciones no asociado para ese numero de cliente ..." (utilice el número de cuenta del registro de transacciones).

**17.8** Despues de escribir el programa del ejercicio 17.7, escriba un programa simple para crear ciertos datos de prueba para verificar el programa. Utilice los siguientes datos de cuentas de ejemplo:

| Archivo maestro  |            |        |
|------------------|------------|--------|
| Número de cuenta | Nombre     | Saldo  |
| 100              | Alan Jones | 348.17 |
| 300              | Mary Smith | 27.19  |
| 500              | Sam Sharp  | 0.00   |
| 700              | Susy Green | -14.22 |

| Archivo de transacciones |                         |
|--------------------------|-------------------------|
| Número de cuenta         | Monto de la transacción |
| 100                      | 27.14                   |
| 300                      | 62.11                   |
| 400                      | 100.56                  |
| 900                      | 82.17                   |

**17.9** Ejecute el programa del ejercicio 17.7, usando los archivos de datos de prueba creados en el ejercicio 17.8. Imprima el nuevo archivo maestro. Compruebe que las cuentas se hayan actualizado en forma correcta.

**17.10** Es posible (en realidad, común) tener varios registros de transacciones con la misma clave de registro. Esto ocurre debido a que un cliente específico podría realizar varias compras y pagos en efectivo durante un periodo de negocios. Vuelva a escribir su programa de asociación de archivos de cuentas por cobrar del ejercicio 17.7 para proveer la posibilidad de manejar varios registros de transacciones con la misma clave de registro. Modifique los datos de prueba del ejercicio 17.8 para incluir los siguientes registros de transacciones adicionales:

| Número de cuenta | Monto en dólares |
|------------------|------------------|
| 300              | 83.89            |
| 700              | 80.78            |
| 700              | 1.53             |

**17.11** Escriba una serie de instrucciones que realicen cada una de las siguientes acciones. Suponga que hemos definido la clase **Persona** que contiene los siguientes datos miembros **private**:

```
char apellidoPaterno[15];
char primerNombre[15];
char edad[4];
```

y las funciones miembro **public**

```
// funciones de acceso para apellidoPaterno
void establecerApellidoPaterno(string);
string obtenerApellidoPaterno() const;

// funciones de acceso para primerNombre
void establecerPrimerNombre(string);
string obtenerPrimerNombre() const;

// funciones de acceso para edad
void establecerEdad(string);
string obtenerEdad() const;
```

Suponga también que cualquier archivo de acceso aleatorio se ha abierto en forma apropiada.

- Inicialice el archivo **nombreedad.dat** con 100 registros que almacenen los valores **apellidoPaterno = "noasignado"**, **primerNombre = ""** y **edad = "0"**.
- Introduzca 10 apellidos paternos, primeros nombres y edades, y escríbalos en el archivo.
- Actualice un registro que ya contenga información. Si el registro no contiene información, informe al usuario que **"No hay informacion"**.
- Elimine un registro que contenga información, reinicializando ese registro específico.

**17.12** Usted es el propietario de una ferretería y necesita llevar un inventario que le pueda indicar los distintos tipos de herramientas que tiene, cuántas de ellas tiene a la mano y el costo de cada una. Escriba un programa que inicialice el archivo de acceso aleatorio **ferreteria.dat** con 100 registros vacíos, que le permita introducir los datos relacionados con cada herramienta, listar todas sus herramientas, eliminar un registro para una herramienta que ya no tenga y que le permita actualizar **cualquier** información en el archivo. El número de identificación de cada herramienta deberá ser el número de registro. Utilice la siguiente información para empezar su archivo:

| Registro # | Nombre de la herramienta | Cantidad | Costo             |
|------------|--------------------------|----------|-------------------|
| 3          | Lijadora eléctrica       | 7        | 57.98             |
| 17         | Martillo                 | 76       | 11.99             |
| 24         | Serrucho                 | 21       | 11.00             |
| 39         | Podadora de césped       | 3        | 79.50             |
|            |                          |          | <i>(continúa)</i> |

| (continuación) |                          |          |       |
|----------------|--------------------------|----------|-------|
| Registro #     | Nombre de la herramienta | Cantidad | Costo |
| 56             | Sierra eléctrica         | 18       | 99.99 |
| 68             | Destornillador           | 106      | 6.99  |
| 77             | Mazo                     | 11       | 21.50 |
| 83             | Llave inglesa            | 34       | 7.50  |

**17.13 (Generador de palabras de números telefónicos)** Los teclados telefónicos estándar contienen los dígitos del 0 al 9. Cada uno de los números del 2 al 9 tiene tres letras asociadas, como se indica en la siguiente tabla:

| Dígito | Letras |
|--------|--------|
| 2      | A B C  |
| 3      | D E F  |
| 4      | G H I  |
| 5      | J K L  |
| 6      | M N O  |
| 7      | P R S  |
| 8      | T U V  |
| 9      | W X Y  |

A muchas personas se les dificulta memorizar números telefónicos, por lo que utilizan la correspondencia entre los dígitos y las letras para desarrollar palabras de siete letras que corresponden a sus números telefónicos. Por ejemplo, una persona cuyo número telefónico sea 686-3767 podría utilizar la correspondencia indicada en la tabla anterior para desarrollar la palabra de siete letras “NUMEROS”.

Las empresas intentan con frecuencia obtener números telefónicos que sean fáciles de recordar para sus clientes. Si una empresa puede anunciar una palabra simple para que sus clientes la marquen, entonces sin duda esa empresa recibirá unas cuantas llamadas más.

Cada número telefónico de siete dígitos corresponde a muchas palabras separadas de siete letras. Por desgracia, la mayoría de estas palabras representan yuxtaposiciones irreconocibles de letras. Sin embargo, es posible que el dueño de una carpintería se complazca en saber que el número telefónico de su taller, 683-2537, corresponde a “MUEBLES”. Un veterinario con el número telefónico 627-2682 se complacería en saber que ese número corresponde a las letras “MASCOTA”.

Escriba un programa que, dado un número de siete dígitos, escriba en un archivo todas las combinaciones posibles de palabras de siete letras que corresponden a ese número. Hay  $2,187$  ( $3$  elevado a la  $7$ ) combinaciones posibles. Evite los números telefónicos con los dígitos 0 y 1.

**17.14** Escriba un programa que utilice el operador `sizeof` para determinar los tamaños en bytes de los diversos tipos de datos de su sistema computacional. Escriba los resultados en el archivo `tamdatos.dat`, de manera que pueda imprimir los resultados más tarde. Estos resultados se deberán mostrar en un formato de dos columnas, con el nombre del tipo en la columna izquierda y el tamaño de ese tipo en la columna derecha, como se muestra a continuación:

|                    |    |
|--------------------|----|
| char               | 1  |
| unsigned char      | 1  |
| short int          | 2  |
| unsigned short int | 2  |
| int                | 4  |
| unsigned int       | 4  |
| long int           | 4  |
| unsigned long int  | 4  |
| float              | 4  |
| double             | 8  |
| long double        | 10 |

[Nota: los tamaños de los tipos de datos integrados en su computadora podrían ser distintos de los antes listados].



*Adapta las acciones  
a la palabra,  
la palabra a la acción;  
con esta práctica especial,  
para que no excedas la  
modestia de la naturaleza.*

—William Shakespeare

*La diferencia  
entre la palabra casi correcta  
y la palabra correcta  
es en realidad una cuestión  
extensa; es la diferencia  
entre el insecto del rayo  
y el rayo.*

—Mark Twain

*Ni una sola palabra.*

—Miguel de Cervantes

*Hice esta carta más extensa  
de lo usual, ya que no tengo  
el tiempo para hacerla más  
corta.*

—Blaise Pascal

# La clase `string` y el procesamiento de flujos de cadena

## OBJETIVOS

En este capítulo aprenderá a:

- Utilizar la clase `string` de la Biblioteca estándar de C++ para tratar a los objetos `string` como objetos completos.
- Realizar asignaciones, concatenaciones, comparaciones, búsquedas e intercambios con objetos `string`.
- Determinar las características de los objetos `string`.
- Buscar, reemplazar e insertar caracteres en objetos `string`.
- Convertir objetos `string` a cadenas estilo C y viceversa.
- Usar los iteradores de `string`.
- Realizar operaciones de entrada/salida entre objetos `string` y la memoria.

- 18.1** Introducción
- 18.2** Asignación y concatenación de objetos `string`
- 18.3** Comparación de objetos `string`
- 18.4** Subcadenas
- 18.5** Intercambio de objetos `string`
- 18.6** Características de los objetos `string`
- 18.7** Búsqueda de subcadenas y caracteres en un objeto `string`
- 18.8** Reemplazo de caracteres en un objeto `string`
- 18.9** Inserción de caracteres en un objeto `string`
- 18.10** Conversión a cadenas estilo C
- 18.11** Iteradores
- 18.12** Procesamiento de flujos de cadena
- 18.13** Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)

## 18.1 Introducción

La plantilla de clase `basic_string` proporciona operaciones comunes de manipulación de cadenas, como copiar, realizar búsquedas, etc. La definición de la plantilla y todas las herramientas de soporte se definen en el espacio de nombres (`namespace std`); éstas incluyen la instrucción `typedef`

```
typedef basic_string< char > string;
```

que crea el tipo alias `string` para `basic_string< char >`. También se proporciona una definición `typedef` para el tipo `wchar_t`. El tipo `wchar_t`<sup>1</sup> almacena caracteres (es decir, caracteres de dos bytes, de cuatro bytes, etc.) para soportar otros conjuntos de caracteres. Nosotros utilizaremos exclusivamente objetos `string` en este capítulo. Para utilizar objetos `string`, debemos incluir el archivo de encabezado `<string>`.

Un objeto `string` se puede inicializar con un argumento constructor tal como

```
string texto("Hola"); // crea una cadena a partir de un const char *
```

que crea un objeto `string` que contiene los caracteres en "Hola", o con dos argumentos constructores como en:

```
string nombre(8, 'x'); // cadena de 8 caracteres 'x'
```

que crea un objeto `string` que contiene ocho caracteres 'x'. La clase `string` también proporciona un constructor predeterminado (que crea una cadena vacía) y un constructor de copia. Una `cadena vacía` es un objeto `string` que no contiene caracteres.

Un objeto `string` también se puede inicializar mediante la sintaxis constructora alternativa en su definición, como en:

```
string mes = "Marzo"; // igual que: string mes("Marzo");
```

Recuerde que el operador = en la declaración anterior no es una asignación, sino una llamada implícita al constructor de la clase `string`, que realiza la conversión.

Observe que la clase `string` no proporciona conversiones de `int` o `char` a `string` en una definición `string`. Por ejemplo, las definiciones

```
string error1 = 'c';
string error2('u');
string error3 = 22;
string error4(8);
```

---

1. El tipo `wchar_t` se utiliza comúnmente para representar el código Unicode®, que tiene caracteres de 16 bits, pero no por ello el tamaño de `wchar_t` está fijo. El estándar Unicode describe una especificación para producir una codificación consistente de los caracteres y símbolos en todo el mundo. Para aprender más acerca del estándar Unicode, visite [www.unicode.org](http://www.unicode.org).

producen errores de sintaxis. Se permite la asignación de un solo carácter a un objeto `string` en una instrucción de asignación, como en

```
cadena1 = 'n';
```

### Error común de programación 18.1



*Tratar de convertir un `int` o `char` a un `string` mediante una inicialización en una declaración, o mediante un argumento constructor, es un error de compilación.*

A diferencia de las cadenas estilo C, los objetos `string` no necesariamente tienen terminación nula. [Nota: el documento del estándar de C++ sólo proporciona una descripción de la interfaz para la clase `string`; la implementación es dependiente de la plataforma]. La longitud de un objeto `string` se puede obtener mediante la función miembro `length` y mediante la función miembro `size`. El operador subíndice (`[]`), se puede utilizar con objetos `string` para acceder a los caracteres individuales y modificarlos. Al igual que las cadenas estilo C, los objetos `string` tienen un primer subíndice de 0 y un último subíndice de `length() - 1`.

La mayoría de las funciones miembro `string` reciben como argumentos una ubicación de subíndice inicial y el número de caracteres con el que deben operar.

El operador de extracción de flujo (`>>`) se sobrecarga para soportar objetos `string`. Las instrucciones

```
string objetoString;
cin >> objetoString;
```

declaran un objeto `string` y leen un objeto `string` del dispositivo de entrada estándar. La entrada se delimita por caracteres de espacio en blanco. Al encontrar un delimitador, se termina la operación de entrada. La función `getline` también se sobrecarga para objetos `string`. Asumiendo que `cadena1` es un objeto `string`, la instrucción

```
getline(cin, cadena1);
```

lee un objeto `string` del teclado y lo coloca en `cadena1`. La entrada se delimita mediante una nueva línea ('\n'), por lo que `getline` puede leer una línea de texto y colocarla en un objeto `string`.

## 18.2 Asignación y concatenación de objetos `string`

La figura 18.1 demuestra la asignación y concatenación de objetos `string`. En la línea 7 se incluye el encabezado `<string>` para la clase `string`. Los objetos `string` `cadena1`, `cadena2` y `cadena3` se crean en las líneas 12 a 14. En la línea 16 se asigna el valor de `cadena1` a `cadena2`. Una vez que se lleva a cabo la asignación, `cadena2` es una copia de `cadena1`. En la línea 17 se utiliza la función miembro `assign` para copiar `cadena1` en `cadena3`. Se hace una copia separada (es decir, `cadena1` y `cadena3` son objetos independientes). La clase `string` también proporciona una versión sobrecargada de la función miembro `assign` que copia un número especificado de caracteres, como en

```
cadenaDestino.assign(cadenaOrigen, inicio, numeroDeCaracteres);
```

en donde `cadenaOrigen` es el objeto `string` que se va a copiar, `inicio` es el subíndice inicial y `numeroDeCaracteres` es el número de caracteres a copiar.

En la línea 22 se utiliza el operador subíndice para asignar 'r' a `cadena3[2]` (con lo cual se forma "car") y para asignar 'r' a `cadena2[0]` (con lo cual se forma "rat"). Después se imprimen los objetos `string`.

```

1 // Fig. 18.1: Fig18_01.cpp
2 // Demostración de la asignación y concatenación de objetos string.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 int main()
11 {
12 string cadena1("cat");
13 string cadena2; // se inicializa con la cadena vacía

```

**Figura 18.1** | Demostración de la asignación y concatenación de objetos `string`. (Parte 1 de 2).

```

14 string cadena3; // se inicializa con la cadena vacía
15
16 cadena2 = cadena1; // asigna cadena1 a cadena2
17 cadena3.assign(cadena1); // asigna cadena1 a cadena3
18 cout << "cadena1: " << cadena1 << "\ncadena2: " << cadena2
19 << "\ncadena3: " << cadena3 << "\n\n";
20
21 // modifica cadena2 y cadena3
22 cadena2[0] = cadena3[2] = 'r';
23
24 cout << "Despues de modificar cadena2 y cadena3:\n" << "cadena1: "
25 << cadena1 << "\ncadena2: " << cadena2 << "\ncadena3: ";
26
27 // demostración de la función miembro at
28 for (int i = 0; i < cadena3.length(); i++)
29 cout << cadena3.at(i);
30
31 // declara cadena4 y cadena5
32 string cadena4(cadena1 + "apulta"); // concatenación
33 string cadena5;
34
35 // += sobrecargado
36 cadena3 += "peta"; // crea "carpeta"
37 cadena1.append("acumba"); // crea "catacumba"
38
39 // adjunta las ubicaciones de los subíndices 4 hasta el final de cadena1 para
40 // crear la cadena "cumba" (al principio, cadena5 estaba vacía)
41 cadena5.append(cadena1, 4, cadena1.length() - 4);
42
43 cout << "\n\nDespues de concatenar:\ncadena1: " << cadena1
44 << "\ncadena2: " << cadena2 << "\ncadena3: " << cadena3
45 << "\ncadena4: " << cadena4 << "\ncadena5: " << cadena5 << endl;
46
47 return 0;
48 } // fin de main

```

```

cadena1: cat
cadena2: cat
cadena3: cat

Despues de modificar cadena2 y cadena3:
cadena1: cat
cadena2: rat
cadena3: car

Despues de concatenar:
cadena1: catacumba
cadena2: rat
cadena3: carpeta
cadena4: catapulta
cadena5: cumba

```

**Figura 18.1 |** Demostración de la asignación y concatenación de objetos `string`. (Parte 2 de 2).

En las líneas 28 y 29 se imprime el contenido de `cadena3` un carácter a la vez, usando la función miembro `at`. La función miembro `at` proporciona un acceso comprobado (o comprobación de rango); es decir, al ir más allá del final del objeto `string` se lanza una excepción `out_of_range`. (Consulte el capítulo 16 para obtener una discusión detallada del manejo de excepciones). Observe que el operador subíndice (`[]`) no proporciona un acceso comprobado. Esto es consistente con su uso en los arreglos.

### Error común de programación 18.2



*Acceder al subíndice de un objeto `string` fuera de los límites de éste mediante el uso de la función `at` es un error lógico que produce una excepción `out_of_range`.*



### Error común de programación 18.3

Acceder a un elemento más allá del tamaño del objeto `string` usando el operador subíndice es un error lógico no reportado.

La cadena `cadena4` se declara (línea 32) y se inicializa con el resultado de la concatenación de `cadena1` y "apulta" mediante el uso del operador + sobrecargado, que para la clase `string` denota la concatenación. La línea 36 utiliza el operador de asignación de suma (+=) para concatenar `cadena3` y "pet". En la línea 37 se utiliza la función miembro `append` para concatenar `cadena1` y "acumba".

En la línea 41 se adjunta la cadena "cumba" a la cadena vacía `cadena5`. Esta función miembro recibe el objeto `string` (`cadena1`) de donde va a obtener los caracteres, el subíndice inicial en el objeto `string` (4) y el número de caracteres que se deben adjuntar (el valor devuelto por `cadena1.length() - 4`).

## 18.3 Comparación de objetos `string`

La clase `string` proporciona funciones miembro para comparar objetos `string`. En la figura 18.2 se demuestran las herramientas de comparación de la clase `string`.

El programa declara cuatro objetos `string` (líneas 12 a 15) e imprime cada objeto `string` (líneas 17 y 18). La condición en la línea 21 compara la igualdad entre `cadena1` y `cadena4`, usando el operador de igualdad sobrecargado. Si la condición es `true`, se imprime "cadena1 == cadena4". Si la condición es `false`, se evalúa la condición en la línea 25. Todas las funciones operador sobrecargadas de la clase `string` que se demuestran aquí, así como las que no se demuestran aquí (!=, <, >= y <=), devuelven valores `bool`.

```

1 // Fig. 18.2: Fig18_02.cpp
2 // Demostración de las herramientas de comparación de cadena.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 int main()
11 {
12 string cadena1("Probando las funciones de comparacion.");
13 string cadena2("Hola");
14 string cadena3("probador");
15 string cadena4(cadena2);
16
17 cout << "cadena1: " << cadena1 << "\ncadena2: " << cadena2
18 << "\ncadena3: " << cadena3 << "\ncadena4: " << cadena4 << "\n\n";
19
20 // comparación de cadena1 y cadena4
21 if (cadena1 == cadena4)
22 cout << "cadena1 == cadena4\n";
23 else // cadena1 != cadena4
24 {
25 if (cadena1 > cadena4)
26 cout << "cadena1 > cadena4\n";
27 else // cadena1 < cadena4
28 cout << "cadena1 < cadena4\n";
29 } // fin de else
30
31 // comparación de cadena1 y cadena2
32 int resultado = cadena1.compare(cadena2);
33
34 if (resultado == 0)
35 cout << "cadena1.compare(cadena2) == 0\n";
36 else // resultado != 0

```

Figura 18.2 | Comparación de objetos `string`. (Parte I de 2).

```

37 {
38 if (resultado > 0)
39 cout << "cadena1.compare(cadena2) > 0\n";
40 else // resultado < 0
41 cout << "cadena1.compare(cadena2) < 0\n";
42 } // fin de else
43
44 // comparación de cadena1 (elementos 1 a 4) y cadena3 (elementos 1 a 4)
45 resultado = cadena1.compare(1, 4, cadena3, 1, 4);
46
47 if (resultado == 0)
48 cout << "cadena1.compare(1, 4, cadena3, 1, 4) == 0\n";
49 else // resultado != 0
50 {
51 if (resultado > 0)
52 cout << "cadena1.compare(1, 4, cadena3, 1, 4) > 0\n";
53 else // resultado < 0
54 cout << "cadena1.compare(1, 4, cadena3, 1, 4) < 0\n";
55 } // fin de else
56
57 // comparación de cadena2 y cadena4
58 resultado = cadena4.compare(0, cadena2.length(), cadena2);
59
60 if (resultado == 0)
61 cout << "cadena4.compare(0, cadena2.length(), "
62 << "cadena2) == 0" << endl;
63 else // resultado != 0
64 {
65 if (resultado > 0)
66 cout << "cadena4.compare(0, cadena2.length(), "
67 << "cadena2) > 0" << endl;
68 else // resultado < 0
69 cout << "cadena4.compare(0, cadena2.length(), "
70 << "cadena2) < 0" << endl;
71 } // fin de else
72
73 // comparación de cadena2 y cadena4
74 resultado = cadena2.compare(0, 3, cadena4);
75
76 if (resultado == 0)
77 cout << "cadena2.compare(0, 3, cadena4) == 0" << endl;
78 else // resultado != 0
79 {
80 if (resultado > 0)
81 cout << "cadena2.compare(0, 3, cadena4) > 0" << endl;
82 else // resultado < 0
83 cout << "cadena2.compare(0, 3, cadena4) < 0" << endl;
84 } // fin de else
85
86 return 0;
87 } // fin de main

```

```

cadena1: Probando las funciones de comparacion.
cadena2: Hola
cadena3: probador
cadena4: Hola

cadena1 > cadena4
cadena1.compare(cadena2) > 0
cadena1.compare(1, 4, cadena3, 1, 4) == 0
cadena4.compare(0, cadena2.length(), cadena2) == 0
cadena2.compare(0, 3, cadena4) < 0

```

Figura 18.2 | Comparación de objetos **string**. (Parte 2 de 2).

En la línea 32 se utiliza la función miembro `compare` de `string` para comparar `cadena1` y `cadena2`. A la variable `resultado` se le asigna 0 si los objetos `string` son equivalentes, un número positivo si `cadena1` es lexicográficamente mayor que `cadena2`, o un número negativo si `cadena1` es lexicográficamente menor que `cadena2`. Debido a que una cadena que empieza con 'P' se considera lexicográficamente mayor que una cadena que empieza con 'H', a `resultado` se le asigna un valor mayor que 0, según lo que confirman los resultados. Un léxico es un diccionario. Cuando decimos que un objeto `string` es lexicográficamente menor que otro, queremos indicar que el método `compare` utiliza los valores numéricos de los caracteres (vea el apéndice B, Conjunto de caracteres ASCII) en cada objeto `string` para determinar que el primer objeto `string` es menor que el segundo.

En la línea 45 se utiliza una versión sobrecargada de la función miembro `compare` para comparar las porciones de `cadena1` y `cadena3`. Los primeros dos argumentos (1 y 4) especifican el subíndice inicial y la longitud de la porción de `cadena1` ("prueba") que se va a comparar con `cadena3`. El tercer argumento es el objeto `string` de comparación. Los dos últimos argumentos (1 y 4) son el subíndice inicial y la longitud de la porción del objeto `string` de comparación que se va a comparar (también "prueba"). El valor asignado a `resultado` es 0 para la igualdad, un número positivo si `cadena1` es lexicográficamente mayor que `cadena3`, o un número negativo si `cadena1` es lexicográficamente menor que `cadena3`. Como las dos piezas de objetos `string` que se están comparando aquí son idénticas, a `resultado` se le asigna 0.

En la línea 58 se utiliza otra versión sobrecargada de la función `compare` para comparar `cadena4` y `cadena2`. Los primeros dos argumentos son los mismos: el subíndice inicial y la longitud. El último argumento es el objeto `string` de comparación. El valor devuelto también es el mismo: 0 para la igualdad, un número positivo si `cadena4` es lexicográficamente mayor que `cadena2`, o un número negativo si `cadena4` es lexicográficamente menor que `cadena2`. Como las dos piezas de objetos `string` que se están comparando aquí son idénticas, a `resultado` se le asigna 0.

En la línea 74 se hace una llamada a la función miembro `compare` para comparar los primeros 3 caracteres en `cadena2` con `cadena4`. Como "Hol" es menor que "Hola", se devuelve un valor menor que cero.

## 18.4 Subcadenas

La clase `string` proporciona la función miembro `substr` para obtener una subcadena de un objeto `string`. El resultado es un nuevo objeto `string` que se copia del objeto `string` de origen. En la figura 18.3 se demuestra el uso de `substr`.

```

1 // Fig. 18.3: Fig18_03.cpp
2 // Demostración de la función miembro substr.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 int main()
11 {
12 string cadena1("El aeroplano aterrizó a tiempo.");
13
14 // obtiene la subcadena "plano" que
15 // empieza en el subíndice 7 y consiste de 5 caracteres
16 cout << cadena1.substr(7, 5) << endl;
17 return 0;
18 } // fin de main

```

plano

Figura 18.3 | Demostración de la función miembro `substr` de `string`.

El programa declara e inicializa un objeto `string` en la línea 12. En la línea 16 se utiliza la función miembro `substr` para obtener una subcadena de `cadena1`. El primer argumento especifica el subíndice inicial de la subcadena deseada; el segundo argumento especifica la longitud de la subcadena.

## 18.5 Intercambio de objetos `string`

La clase `string` proporciona la función miembro `swap` para intercambiar objetos `string`. En la figura 18.4 se intercambian dos objetos `string`. En las líneas 12 y 13 se declaran e inicializan los objetos `string` `primero` y `segundo`. Después se imprime cada uno de ellos. En la línea 18 se utiliza la función miembro `swap` para intercambiar los valores de `primero` y `segundo`. Los dos objetos `string` se imprimen de nuevo para confirmar que se hayan cambiado. La función miembro `swap` de `string` es útil para implementar programas que ordenan cadenas.

```

1 // Fig. 18.4: Fig18_04.cpp
2 // Uso de la función swap para intercambiar dos objetos string.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 int main()
11 {
12 string primero("uno");
13 string segundo("dos");
14
15 // imprime los objetos string
16 cout << "Antes de swap:\nprimero: " << primero << "\nsegundo: " << segundo;
17
18 primero.swap(segundo); // intercambia los objetos string
19
20 cout << "\n\nDespués de swap:\nprimero: " << primero
21 << "\nsegundo: " << segundo << endl;
22
23 return 0;
24 } // fin de main

```

Antes Swap:  
 Primero: uno  
 Segundo: dos

Después Swap:  
 Primero: dos  
 Segundo: uno

**Figura 18.4** | Uso de la función `swap` para intercambiar dos objetos `string`.

## 18.6 Características de los objetos `string`

La clase `string` proporciona funciones miembro para recopilar información acerca del tamaño de un objeto `string`, su longitud, capacidad, longitud máxima y otras características. El tamaño o la longitud de un objeto `string` es el número de caracteres actualmente almacenados en el objeto `string`. La capacidad de un objeto `string` es el número de caracteres que se pueden almacenar en el objeto `string` sin necesidad de asignar más memoria. La capacidad de un objeto `string` debe ser cuando menos igual a su tamaño actual, aunque puede ser mayor. La capacidad exacta de un objeto `string` depende de la implementación. El tamaño máximo es el mayor tamaño posible que puede tener un objeto `string`. Si se excede este valor, se lanza una excepción `length_error`. En la figura 18.5 se demuestran las funciones miembro de la clase `string` para determinar varias características de los objetos `string`.

```

1 // Fig. 18.5: Fig18_05.cpp
2 // Demostración de las funciones miembro relacionadas con el tamaño y la capacidad.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::cin;

```

**Figura 18.5** | Impresión de las características de los objetos `string`. (Parte I de 3).

```

7 using std::boolalpha;
8
9 #include <string>
10 using std::string;
11
12 void imprimirEstadisticas(const string &);
13
14 int main()
15 {
16 string cadena1; // cadena vacía
17
18 cout << "Estadísticas antes de la entrada:\n" << boolalpha;
19 imprimirEstadisticas(cadena1);
20
21 // solo se lee "sopita" de "sopita caliente"
22 cout << "\n\nEscriba una cadena: ";
23 cin >> cadena1; // delimitada por espacio en blanco
24 cout << "La cadena introducida fue: " << cadena1;
25
26 cout << "\nEstadísticas después de la entrada:\n";
27 imprimirEstadisticas(cadena1);
28
29 // lee "caliente"
30 cin >> cadena1; // delimitada por espacio en blanco
31 cout << "\n\nEl resto de la cadena es: " << cadena1 << endl;
32 imprimirEstadisticas(cadena1);
33
34 // adjunta 46 caracteres a cadena1
35 cadena1 += "1234567890abcdefghijklmnopqrstuvwxyz1234567890";
36 cout << "\n\ncadena1 es ahora: " << cadena1 << endl;
37 imprimirEstadisticas(cadena1);
38
39 // agrega 10 elementos a cadena1
40 cadena1.resize(cadena1.length() + 10);
41 cout << "\n\nEstadísticas después de cambiar el tamaño en base a (length + 10):\n";
42 imprimirEstadisticas(cadena1);
43
44 cout << endl;
45 return 0;
46 } // fin de main
47
48 // muestra las estadísticas de la cadena
49 void imprimirEstadisticas(const string &refString)
50 {
51 cout << "capacidad: " << refString.capacity() << "\ntamaño max: "
52 << refString.max_size() << "\ntamaño: " << refString.size()
53 << "\nlongitud: " << refString.length()
54 << "\nvacia: " << refString.empty();
55 } // fin de imprimirEstadisticas

```

```

Estadísticas antes de la entrada:
capacidad: 0
tamaño max: 4294967294
tamaño: 0
longitud: 0
vacia: true

Escriba una cadena: sopita caliente
La cadena introducida fue: sopita
Estadísticas después de la entrada:
capacidad: 15
tamaño max: 4294967294

```

**Figura 18.5** | Impresión de las características de los objetos `string`. (Parte 2 de 3).

```
tamanio: 6
longitud: 6
vacia: false

El resto de la cadena es: caliente
capacidad: 15
tamanio max: 4294967294
tamanio: 8
longitud: 8
vacia: false

cadena1 es ahora: caliente1234567890abcdefghijklmnopqrstuvwxyz1234567890
capacidad: 63
tamanio max: 4294967294
tamanio: 54
longitud: 54
vacia: false

Estadísticas después de cambiar el tamaño en base a (length + 10):
capacidad: 94
tamanio max: 4294967294
tamanio: 64
longitud: 64
vacia: false
```

**Figura 18.5** | Impresión de las características de los objetos `string`. (Parte 3 de 3).

El programa declara el objeto `string` vacío `cadena1` (línea 16) y lo pasa a la función `imprimirEstadísticas` (línea 19). La función `imprimirEstadísticas` (líneas 49 a 55) recibe una referencia a un objeto `const string` como argumento, e imprime la capacidad (usando la función miembro `capacity`), el tamaño máximo (usando la función miembro `max_size`), el tamaño (usando la función miembro `size`), la longitud (usando la función miembro `length`) y si el objeto `string` está o no vacío (usando la función miembro `empty`). La llamada inicial a `imprimirEstadísticas` indica que los valores iniciales para la capacidad, tamaño y longitud de `cadena1` son 0.

El tamaño y la longitud de 0 indican que no hay caracteres almacenados en un objeto `string`. Debido a que la capacidad inicial es 0, cuando se colocan caracteres en `cadena1` se asigna memoria para dar cabida a los nuevos caracteres. Recuerde que el tamaño y la longitud siempre son idénticos. En esta implementación, el tamaño máximo es 4294967293. El objeto `cadena1` está vacío, por lo que la función `empty` devuelve `true`.

En la línea 23 se introduce una cadena. En este ejemplo, se introduce "sopita caliente". Como un carácter de espacio es un delimitador, sólo se almacena "sopita" en `cadena1`; sin embargo, "caliente" permanece en el búfer de entrada. En la línea 27 se hace una llamada a la función `imprimirEstadísticas` para imprimir las estadísticas de `cadena1`. Observe en los resultados que la longitud es 6 y que la capacidad es 15.



### Tip de rendimiento 18.1

Para minimizar el número de veces que se asigna y desasigna la memoria, algunas implementaciones de la clase `string` proporcionan una capacidad predeterminada que es mayor que la longitud del objeto `string`.

En la línea 30 se lee "caliente" del búfer de entrada y se almacena en `cadena1`, con lo cual se reemplaza "sopita". En la línea 32 se pasa `cadena1` a `imprimirEstadísticas`.

En la línea 35 se utiliza el operador `+=` sobrecargado para concatenar una cadena de 46 caracteres de largo con `cadena1`. En la línea 37 se pasa `cadena1` a `imprimirEstadísticas`. Observe que la capacidad se ha incrementado a 63 elementos y la longitud es ahora 54.

En la línea 40 se utiliza la función miembro `resize` para incrementar la longitud de `cadena1` por 10 caracteres. Los elementos adicionales se establecen como caracteres nulos. Observe que en los resultados la capacidad no ha cambiado, y la longitud ahora es 60.

## 18.7 Búsqueda de subcadenas y caracteres en un objeto `string`

La clase `string` proporciona funciones miembro `const` para buscar subcadenas y caracteres en un objeto `string`. En la figura 18.6 se demuestran las funciones de búsqueda.

```

1 // Fig. 18.6: Fig18_06.cpp
2 // Demostración de las funciones miembro de búsqueda en objetos string.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 int main()
11 {
12 string cadena1("mediodia es 12 pm; medianoche no es.");
13 int ubicacion;
14
15 // encuentra "es" en las ubicaciones 9 y 33
16 cout << "Cadena original:\n" << cadena1
17 << "\n\n(find) \"es\" se encontró en: " << cadena1.find("es")
18 << "\n(rfind) \"es\" se encontró en: " << cadena1.rfind("es");
19
20 // encuentra 'e' en la ubicación 1
21 ubicacion = cadena1.find_first_of("liesop");
22 cout << "\n\n(find_first_of) encontró 'e' << cadena1[ubicacion]
23 << "' del grupo \"liesop\" en: " << ubicacion;
24
25 // encuentra 's' en la ubicación 34
26 ubicacion = cadena1.find_last_of("liesop");
27 cout << "\n\n(find_last_of) encontró 's' << cadena1[ubicacion]
28 << "' del grupo \"misop\" en: " << ubicacion;
29
30 // encuentra '1' en la ubicación 12
31 ubicacion = cadena1.find_first_not_of("medop ias");
32 cout << "\n\n(find_first_not_of) '' << cadena1[ubicacion]
33 << "' no está contenido en \"medop ias\" y se encontró en: "
34 << ubicacion;
35
36 // encuentra ';' en la ubicación 17
37 ubicacion = cadena1.find_first_not_of("12medop ias");
38 cout << "\n\n(find_first_not_of) '' << cadena1[ubicacion]
39 << "' no está contenido en \"12medop ias\" y se "
40 << "encontró en: " << ubicacion << endl;
41
42 // busca los caracteres que no estén en cadena1
43 ubicacion = cadena1.find_first_not_of(
44 "mediodia es 12 pm; medianoche no es.");
45 cout << "\n\n(find_first_not_of(\"mediodia es 12 pm; medianoche no es.\"))"
46 << " devuelve: " << ubicacion << endl;
47 return 0;
48 } // fin de main

```

Cadena original:

mediodia es 12 pm; medianoche no es.

(find) "es" se encontró en: 9  
 (rfind) "es" se encontró en: 33

(find\_first\_of) encontró 'e' del grupo "liesop" en: 1

(find\_last\_of) encontró 's' del grupo "misop" en: 34

(find\_first\_not\_of) '1' no está contenido en "medop ias" y se encontró en: 12

(find\_first\_not\_of) ';' no está contenido en "12medop ias" y se encontró en: 17

find\_first\_not\_of("mediodia es 12 pm; medianoche no es.") devuelve: -1

**Figura 18.6** | Demostración de las funciones find de string.

La cadena `cadena1` se declara y se inicializa en la línea 12. La línea 17 trata de buscar "es" en `cadena1`, usando la función `find`. Si se encuentra "es", se devuelve el subíndice de la ubicación inicial de esa cadena. Si no se encuentra el objeto `string`, se devuelve el valor `string::npos` (una constante `public static` definida en la clase `string`). Este valor es devuelto por las funciones de `string` relacionadas con `find` para indicar que no se encontró una subcadena o un carácter en el objeto `string`.

En la línea 18 se utiliza la función miembro `rfind` para realizar una búsqueda inversa en `cadena1` (es decir, de derecha a izquierda). Si se encuentra "es", se devuelve la ubicación del subíndice. Si no se encuentra la cadena, se devuelve `string::npos`. [Nota: el resto de las funciones de búsqueda presentadas en esta sección devuelven el mismo tipo, a menos que se indique lo contrario].

En la línea 21 se utiliza la función miembro `find_first_of` para localizar la primera ocurrencia en `cadena1` de cualquier carácter en "liesop". La búsqueda se realiza desde el inicio de `cadena1`. El carácter 'o' se encuentra en el elemento 1.

En la línea 26 se utiliza la función miembro `find_last_of` para buscar la última ocurrencia en `cadena1` de cualquier carácter en "liesop". La búsqueda se realiza desde el final de `cadena1`. El carácter 'o' se encuentra en el elemento 29.

En la línea 31 se utiliza la función miembro `find_first_not_of` para buscar el primer carácter en `cadena1` que no esté contenido en "medop ias". El carácter '1' se encuentra en el elemento 8. La búsqueda se realiza desde el inicio de `cadena1`.

En la línea 37 se utiliza la función miembro `find_first_not_of` para buscar el primer carácter que no esté contenido en "12medop ias". El carácter '.' se encuentra en el elemento 12. La búsqueda se realiza desde el final de `cadena1`.

En las líneas 43 y 44 se utiliza la función miembro `find_first_not_of` para buscar el primer carácter que no esté contenido en "mediodia es 12pm; medianoche no es.". En este caso, el objeto `string` en el que se realiza la búsqueda contiene cada carácter especificado en el argumento de cadena. Como no se encontró un carácter, se devuelve `string::npos` (que tiene el valor -1 en este caso).

## 18.8 Reemplazo de caracteres en un objeto `string`

En la figura 18.7 se demuestran las funciones miembro `string` para reemplazar y borrar caracteres. En las líneas 13 y 17 se declara e inicializa el objeto `string` `cadena1`. En la línea 23 se utiliza la función miembro `erase` de `string` para borrar todo, desde (e incluyendo a) el carácter en la posición 62 hasta el final de `cadena1`. [Nota: cada carácter de nueva línea ocupa un elemento en el objeto `string`].

En las líneas 29 a 36 se utiliza `find` para localizar cada ocurrencia del carácter de espacio. Después, cada espacio se reemplaza con un punto mediante una llamada a la función miembro `replace` de `string`. La función `replace` recibe tres argumentos: el subíndice del carácter en el objeto `string` en el que debe empezar el reemplazo, el número de caracteres a reemplazar y la cadena de reemplazo. La función miembro `find` devuelve `string::npos` cuando no se encuentra el carácter de búsqueda. En la línea 35 se suma 1 a `posicion` para seguir buscando en la ubicación del siguiente carácter.

En las líneas 40 a 48 se utiliza la función `find` para buscar cada punto y otra función `replace` sobrecargada para reemplazar cada periodo y su siguiente carácter con dos signos de punto y coma. Los argumentos que se pasan a esta versión de `replace` son el subíndice del elemento en el que comienza la operación de reemplazo, el número de caracteres a reemplazar, una cadena de caracteres de reemplazo de la cual se selecciona una subcadena para utilizarla como caracteres de reemplazo, el elemento en la cadena de caracteres en la que empieza la subcadena de reemplazo, y el número de caracteres en la cadena de caracteres de reemplazo que se debe utilizar.

```

1 // Fig. 18.7: Fig18_07.cpp
2 // Demostración de las funciones miembro erase y replace de string.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 int main()

```

Figura 18.7 | Demostración de las funciones `erase` y `replace`. (Parte 1 de 2).

```

11 {
12 // el compilador concatena todas las partes en una cadena
13 string cadena1("Los valores en cualquier subarbol izquierdo"
14 "\nson menos que el valor en el"
15 "\nnodo padre y los valores en"
16 "\ncualquier subarbol derecho son mayores"
17 "\nque el valor en el nodo padre");
18
19 cout << "Cadena original:\n" << cadena1 << endl << endl;
20
21 // elimina todos los caracteres de (e incluyendo a) la ubicación 72
22 // hasta el final de cadena1
23 cadena1.erase(72);
24
25 // imprime una nueva cadena
26 cout << "Cadena original despues de erase:\n" << cadena1
27 << "\n\nDespues del primer reemplazo:\n";
28
29 int posicion = cadena1.find(" "); // busca el primer espacio
30
31 // reemplaza todos los espacios con un punto
32 while (posicion != string::npos)
33 {
34 cadena1.replace(posicion, 1, ".");
35 posicion = cadena1.find(" ", posicion + 1);
36 } // fin de while
37
38 cout << cadena1 << "\n\nDespues del segundo reemplazo:\n";
39
40 posicion = cadena1.find("."); // busca el primer periodo
41
42 // reemplaza todos los puntos con dos signos de punto y coma
43 // NOTA: esto sobrescribirá los caracteres
44 while (posicion != string::npos)
45 {
46 cadena1.replace(posicion, 2, "xxxxx;yyy", 5, 2);
47 posicion = cadena1.find(".", posicion + 1);
48 } // fin de while
49
50 cout << cadena1 << endl;
51 return 0;
52 } // fin de main

```

Cadena original:

Los valores en cualquier subarbol izquierdo  
 son menos que el valor en el  
 nodo padre y los valores en  
 cualquier subarbol derecho son mayores  
 que el valor en el nodo padre

Cadena original despues de erase:

Los.valores.en.cualquier.subarbol.izquierdo  
 son.menos.que.el.valor.en.el

Despues del primer reemplazo:

Los.;valores;;n;;ualquier;;ubarbol;;zquierdo  
 son;;enos;;ue;;l;;alor;;n;;l

Despues del segundo reemplazo:

Los.;;alores;;n;;ualquier;;ubarbol;;zquierdo  
 son;;enos;;ue;;l;;alor;;n;;l

Figura 18.7 | Demostración de las funciones `erase` y `replace`. (Parte 2 de 2).

## 18.9 Inserción de caracteres en un objeto `string`

La clase `string` proporciona funciones miembro para insertar caracteres en un objeto `string`. La figura 18.8 demuestra las capacidades de la función `insert` de `string`.

El programa declara, inicializa y después imprime los objetos `string` `cadena1`, `cadena2`, `cadena3` y `cadena4`. En la línea 22 se utiliza la función miembro `insert` de `string` para insertar el contenido de `cadena2` antes del elemento 10 de `cadena1`.

En la línea 25 se utiliza `insert` para insertar `cadena4` antes del elemento 3 de `cadena3`. Los últimos dos argumentos especifican los elementos inicial y final de `cadena4` que se deben insertar. Al utilizar `string::npos`, se inserta todo el objeto `string`.

```

1 // Fig. 18.8: Fig18_08.cpp
2 // Demostración de las funciones miembro insert de la clase string.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 int main()
11 {
12 string cadena1("principio fin");
13 string cadena2("enmedio ");
14 string cadena3("12345678");
15 string cadena4("xx");
16
17 cout << "Cadenas iniciales:\n" << cadena1
18 << "\n" << cadena2 << "\n" << cadena3
19 << "\n" << cadena4 << "\n\n";
20
21 // inserta "enmedio" en la ubicación 10 en cadena1
22 cadena1.insert(10, cadena2);
23
24 // inserta "xx" en la ubicación 3 en cadena3
25 cadena3.insert(3, cadena4, 0, string::npos);
26
27 cout << "Cadenas despues de insert:\n" << cadena1
28 << "\n" << cadena2 << "\n" << cadena3
29 << "\n" << cadena4 << endl;
30
31 } // fin de main

```

```

Cadenas iniciales:
cadena1: principio fin
cadena2: enmedio
cadena3: 12345678
cadena4: xx

Cadenas despues de insert:
cadena1: principio enmedio fin
cadena2: enmedio
cadena3: 123xx45678
cadena4: xx

```

Figura 18.8 | Demostración de las funciones miembro `insert` de `string`.

## 18.10 Conversión a cadenas estilo C

La clase `string` proporciona funciones miembro para convertir objetos de la clase `string` en cadenas basadas en apuntador estilo C. Como dijimos antes, a diferencia de las cadenas basadas en apuntador, los objetos `string` no tienen

necesariamente una terminación nula. Estas funciones de conversión son útiles cuando una función dada recibe una cadena basada en apuntador como argumento. En la figura 18.9 se demuestra la conversión de objetos `string` en cadenas basadas en apuntador.

El programa declara un objeto `string`, un `int` y dos apunadores `char` (líneas 12 a 15). El objeto `string` `cadena1` se inicializa con "CADENAS", `ptr1` se inicializa con 0 y `longitud` se inicializa con la longitud de `cadena1`. La memoria de un tamaño suficiente como para contener una cadena basada en apuntador equivalente al objeto `string` `cadena1` se asigna en forma dinámica y se adjunta al apuntador `char` `ptr2`.

En la línea 18 se utiliza la función miembro `copy` de `string` para copiar el objeto `cadena1` en el arreglo `char` al que apunta `ptr2`. En la línea 19 se coloca en forma manual un carácter de terminación nula en el arreglo al que apunta `ptr2`.

En la línea 23 se utiliza la función `c_str` para obtener un apuntador `const char *` que apunte a una cadena estilo C con terminación nula, con el mismo contenido que `cadena1`. El apuntador se pasa al operador de inserción de flujo para la salida.

En la línea 29 se asigna al apuntador `const char * ptr1` un apuntador devuelto por la función miembro `data` de la clase `string`. Esta función miembro devuelve un arreglo de caracteres estilo C sin terminación nula. Observe que en este ejemplo no modificamos el objeto `string` `cadena1`. Si se modificara `cadena1` (es decir, que la memoria dinámica del objeto `string` cambie su dirección debido a la llamada a una función tal como `cadena1.insert( 0, "abcd" )`), `ptr1` podría volverse inválido, lo cual podría provocar resultados impredecibles.

```

1 // Fig. 18.9: Fig18_09.cpp
2 // Conversión a cadenas estilo C.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 int main()
11 {
12 string cadena1("CADENAS"); // constructor de string con char* arg
13 const char *ptr1 = 0; // inicializa *ptr1
14 int longitud = cadena1.length();
15 char *ptr2 = new char[longitud + 1]; // incluyendo el carácter nulo
16
17 // copia caracteres de cadena1 a la memoria asignada
18 cadena1.copy(ptr2, longitud, 0); // copia cadena1 a ptr2 char*
19 ptr2[longitud] = '\0'; // agrega el terminador nulo
20
21 cout << "el objeto string cadena1 es " << cadena1
22 << "\ncadena1 convertida a una cadena estilo C es "
23 << cadena1.c_str() << "\nptr1 es ";
24
25 // Asigna al apuntador ptr1 el valor const char * devuelto por
26 // la función data(). NOTA: ésta es una asignación potencialmente
27 // peligrosa. Si se modifica cadena1, el apuntador ptr1 se
28 // puede hacer inválido.
29 ptr1 = cadena1.data();
30
31 // imprime cada carácter usando un apuntador
32 for (int i = 0; i < longitud; i++)
33 cout << *(ptr1 + i); // usa aritmética de apunadores
34
35 cout << "\nptr2 es " << ptr2 << endl;
36 delete [] ptr2; // reclama la memoria asignada en forma dinámica
37 return 0;
38 } // fin de main

```

**Figura 18.9** | Conversión de objetos `string` en cadenas estilo C y arreglos de caracteres. (Parte 1 de 2).

```

el objeto string cadena1 es CADENAS
cadena1 convertida a una cadena estilo C es CADENAS
ptr1 es CADENAS
ptr2 es CADENAS

```

**Figura 18.9** | Conversión de objetos `string` en cadenas estilo C y arreglos de caracteres. (Parte 2 de 2).

En las líneas 32 y 33 se utiliza la aritmética de apuntadores para imprimir el arreglo de caracteres al que apunta `ptr1`. En las líneas 35 y 36 se imprime la cadena estilo C a la que apunta `ptr2` y se elimina (mediante `delete`) la memoria asignada para `ptr2`, para evitar una fuga de memoria.



### Error común de programación 18.4

*Si no termina el arreglo de caracteres devuelto por `data` con un carácter nulo, se pueden producir errores en tiempo de ejecución.*



### Buena práctica de programación 18.1

*Siempre que sea posible, utilice los objetos más robustos de la clase `string` en vez de cadenas basadas en apuntador estilo C.*

## 18.11 Iteradores

La clase `string` proporciona iteradores para realizar recorridos hacia adelante y hacia atrás de objetos `string`. Los iteradores proporcionan acceso a los caracteres individuales con una sintaxis similar a las operaciones de los apuntadores. No se comprueba el rango de los iteradores. Observe que en esta sección proporcionamos “ejemplos mecánicos” para demostrar el uso de los iteradores. En el capítulo 22 hablaremos sobre usos más robustos de los iteradores. En la figura 18.10 se demuestran los iteradores.

```

1 // Fig. 18.10: Fig18_10.cpp
2 // Uso de un iterador para imprimir un objeto string.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 int main()
11 {
12 string cadena1("Prueba de los iteradores");
13 string::const_iterator iterador1 = cadena1.begin();
14
15 cout << "cadena1 = " << cadena1
16 << "\n(Usando el iterador iterador1) cadena1 es: ";
17
18 // itera a través del objeto string
19 while (iterador1 != cadena1.end())
20 {
21 cout << *iterador1; // desreferencia el iterador para obtener un char
22 iterador1++; // avanza el iterador al siguiente char
23 } // fin de while
24
25 cout << endl;
26 return 0;
27 } // fin de main

```

```

cadena1 = Prueba de los iteradores
(Usando el iterador iterador1) cadena1 es: Prueba de los iteradores

```

**Figura 18.10** | Usando un iterador para imprimir un objeto `string`.

En las líneas 12 y 13 se declara el objeto `string cadena1` y el objeto `string::const_iterator iterador1`. Un `const_iterator` es un iterador que no puede modificar el objeto `string`; en este caso, el objeto `string` a través del cual está iterando. El iterador `iterador1` se inicializa al principio de `cadena1` con la función miembro `begin` de la clase `string`. Existen dos versiones de `begin`: una que devuelve un `iterator` para iterar a través de un objeto `string` no `const` y una versión `const` que devuelve un `const_iterator` para iterar a través de un objeto `const_string`. En la línea 15 se imprime `cadena1`.

En las líneas 19 a 23 se utiliza el iterador `iterador1` para “recorrer” `cadena1`. La función miembro `end` de la clase `string` devuelve un objeto `iterator` (o `const_iterator`) para la posición que está más allá del último elemento de `cadena1`. Para imprimir cada elemento, se desreferencia el iterador de igual forma que como se desreferencia a un apuntador, y el iterador se avanza una posición mediante el operador `++`.

La clase `string` proporciona las funciones miembro `rend` y `rbegin` para acceder a los caracteres individuales de un objeto `string` en forma inversa, desde el final de un objeto `string` hacia su inicio. Las funciones miembro `rend` y `rbegin` devuelven objetos `reverse_iterator` o `const_reverse_iterator` (dependiendo de si el objeto `string` es no `const` o `const`). En los ejercicios pedimos al lector que escriba un programa para demostrar estas herramientas. En el capítulo 22 utilizaremos más los iteradores y los iteradores inversos.



### Tip para prevenir errores 18.1

*Use la función miembro `at` de `string` (en vez de iteradores) cuando deseé obtener el beneficio de la comprobación de rangos.*



### Buena práctica de programación 18.2

*Cuando las operaciones en las que se involucra el iterador no deben modificar los datos que se están procesando, use un `const_iterator`. Éste es otro ejemplo de cómo emplear el principio del menor privilegio.*

## 18.12 Procesamiento de flujos de cadena

Además de la E/S de flujos estándar y la E/S de flujos de archivos, en C++ la E/S de flujos incluye herramientas para recibir datos de (y enviar datos a) objetos `string` en la memoria. A menudo, a estas herramientas se les conoce como **E/S en memoria o procesamiento de flujos de cadena**.

La entrada desde un objeto `string` está soportada por la clase `istringstream`. La salida hacia un objeto `string` está soportada por la clase `ostringstream`. Los nombres de las clases `istringstream` y `ostringstream` son en realidad alias definidos por las siguientes definiciones `typedef`:

```
typedef basic_istringstream< char > istringstream;
typedef basic_ostringstream< char > ostringstream;
```

Las plantillas de clase `basic_istringstream` y `basic_ostringstream` proporcionan la misma funcionalidad que las clases `istream` y `ostream`, además de otras funciones miembro específicas para el formato en memoria. Los programas que utilizan el formato en memoria deben incluir los archivos de encabezado `<sstream>` y `<iostream>`.

Una aplicación para estas técnicas es la validación de datos. Un programa puede leer toda una línea a la vez del flujo de entrada y colocarla en un objeto `string`. Después, una rutina de validación puede escudriñar el contenido del objeto `string` y corregir (o reparar) los datos, si es necesario. Luego el programa puede continuar recibiendo datos de entrada del objeto `string`, sabiendo que los datos de entrada se encuentran en el formato apropiado.

Enviar datos a un objeto `string` es una buena forma de aprovechar las poderosas herramientas de formato de la salida de los flujos de C++. Los datos se pueden preparar en un objeto `string` para imitar el formato de pantalla editado. Ese objeto `string` se podría escribir en un archivo en disco para preservar la imagen de la pantalla.

Un objeto `ostringstream` utiliza un objeto `string` para almacenar los datos de salida. La función miembro `str` de la clase `ostringstream` devuelve una copia de ese objeto `string`.

La figura 18.11 demuestra el uso de un objeto `ostringstream`. El programa crea el objeto `ostringstream cadenaSalida` (línea 15) y utiliza el operador de inserción de flujo para enviar una serie de objetos `string` y valores numéricos al objeto.

En las líneas 27 y 28 se envían los objetos `string cadena1`, `string cadena2`, `string cadena3`, `double doble1`, `string cadena4`, `int entero`, `string cadena5` y la dirección de `int entero`; todos al objeto `cadenaSalida` en memoria. En la línea 31 se utilizan el operador de inserción de flujo y la llamada `cadenaSalida.str()` para mostrar una copia del objeto `string` creado en las líneas 27 y 28. En la línea 34 se demuestra que se pueden adjuntar más datos al objeto `string` en memoria, con sólo enviar otra operación de inserción de flujo a `cadenaSalida`. En las líneas 35 y 36 se muestra el objeto `string cadenaSalida` después de adjuntar caracteres adicionales.

```

1 // Fig. 18.11: Fig18_11.cpp
2 // Uso de un objeto ostringstream asignado en forma dinámica.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 #include <sstream> // archivo de encabezado para el procesamiento de flujos de cadena
11 using std::ostringstream; // operadores de inserción de flujo
12
13 int main()
14 {
15 ostringstream cadenaSalida; // crea una instancia de ostringstream
16
17 string cadena1("Envio de varios tipos de datos ");
18 string cadena2("a un objeto ostringstream:");
19 string cadena3("\n" double: ");
20 string cadena4("\n" int: ");
21 string cadena5("\ndireccion de int: ");
22
23 double doble1 = 123.4567;
24 int entero = 22;
25
26 // envia objetos string, double e int al objeto ostringstream cadenaSalida
27 cadenaSalida << cadena1 << cadena2 << cadena3 << doble1
28 << cadena4 << entero << cadena5 << &entero;
29
30 // Llama a str para obtener el contenido string del objeto ostringstream
31 cout << "cadenaSalida contiene:\n" << cadenaSalida.str();
32
33 // agrega caracteres adicionales y llama a str para enviar el objeto string
34 cadenaSalida << "\nse agregaron mas caracteres";
35 cout << "\n\ndespues de las inserciones de flujo adicionales,\n"
36 << "cadenaSalida contiene:\n" << cadenaSalida.str() << endl;
37 return 0;
38 } // fin de main

cadenaSalida contiene:
Envio de varios tipos de datos a un objeto ostringstream:
 double: 123.457
 int: 22
direccion de int: 0012FDF0

despues de las inserciones de flujo adicionales,
cadenaSalida contiene:
Envio de varios tipos de datos a un objeto ostringstream:
 double: 123.457
 int: 22
direccion de int: 0012FDF0
se agregaron mas caracteres

```

Figura 18.11 | Uso de un objeto **ostringstream** asignado en forma dinámica.

Un objeto **istringstream** introduce datos desde un objeto **string** en memoria a las variables del programa. Los datos se almacenan en un objeto **istringstream** como caracteres. La operación de entrada desde el objeto **istringstream** funciona de manera idéntica a las operaciones de entrada desde cualquier archivo. El objeto **istringstream** interpreta el final del objeto **string** como el fin de archivo.

En la figura 18.12 se demuestra cómo introducir datos desde un objeto **istringstream**. En las líneas 15 y 16 se crea el objeto **string** **entrada** que contiene los datos, y el objeto **istringstream** **cadenaEntrada** que se construye para contener los datos en el objeto **string** **entrada**. Este objeto contiene los datos

Entrada prueba 123 4.7 A

que, cuando se leen como entrada para el programa, consisten en dos cadenas ("Entrada" y "prueba"), un `int` (123), un `double` (4.7) y un `char` ('A'). Estos caracteres se extraen en las variables `cadena1`, `cadena2`, `entero`, `doble1` y `caracter` en la línea 23.

```

1 // Fig. 18.12: Fig18_12.cpp
2 // Demostración de las operaciones de entrada desde un objeto istream.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 #include <sstream>
11 using std::istringstream;
12
13 int main()
14 {
15 string entrada("Entrada prueba 123 4.7 A");
16 istringstream cadenaEntrada(entrada);
17 string cadena1;
18 string cadena2;
19 int entero;
20 double doble1;
21 char caracter;
22
23 cadenaEntrada >> cadena1 >> cadena2 >> entero >> doble1 >> caracter;
24
25 cout << "Se extrajeron los siguientes elementos\n"
26 << "del objeto istream:" << "\nstring: " << cadena1
27 << "\nstring: " << cadena2 << "\n int: " << entero
28 << "\ndouble: " << doble1 << "\n char: " << caracter;
29
30 // intento de leer de un flujo vacío
31 long valor;
32 cadenaEntrada >> valor;
33
34 // prueba los resultados del flujo
35 if (cadenaEntrada.good())
36 cout << "\n\nel valor long es: " << valor << endl;
37 else
38 cout << "\n\ncadenaEntrada esta vacia" << endl;
39
40 return 0;
41 } // fin de main

```

```

Se extrajeron los siguientes elementos
del objeto istream:
string: Entrada
string: prueba
 int: 123
double: 4.7
 char: A

cadenaEntrada esta vacia

```

**Figura 18.12** | Demostración de las operaciones de entrada en un objeto `istringstream`.

Después los datos se imprimen en las líneas 25 a 28. El programa trata de leer de `cadenaEntrada` de nuevo en la línea 32. La condición `if` en la línea 35 utiliza la función `good` (sección 15.8) para probar si quedan datos. Como no quedan datos, la función devuelve `false` y se ejecuta la parte `else` de la instrucción `if...else`.

## 18.13 Repaso

Este capítulo presentó la clase **string** de la Biblioteca estándar de C++, que permite a los programas tratar a las cadenas como objetos completos. Hablamos sobre cómo realizar asignaciones, concatenaciones, comparaciones, búsquedas e intercambios con cadenas. También presentamos varios métodos para determinar las características de las cadenas, cómo buscar, reemplazar e insertar caracteres en una cadena, y cómo convertir cadenas a cadenas estilo C y viceversa. El lector también aprendió acerca de los iteradores de cadenas y cómo realizar operaciones de entrada y salida desde/hacia las cadenas en memoria. En el capítulo 19, Búsqueda y ordenamiento, hablaremos sobre el algoritmo de búsqueda binaria y el algoritmo de ordenamiento por combinación. También usaremos la notación “Big O” para analizar y comparar la eficiencia de varios algoritmos de búsqueda y ordenamiento.

## Resumen

### Sección 18.1 Introducción

- La plantilla de clase de C++ **basic\_string** proporciona operaciones comunes de manipulación de cadenas, como copiar, buscar, etcétera.
- La instrucción **typedef**

```
typedef basic_string< char > cadena;
```

crea el tipo de alias **string** para **basic\_string< char >**. También se proporciona una instrucción **typedef** para el tipo **wchar\_t**. Por lo general, este tipo almacena caracteres de dos bytes (16 bits) para soportar otros conjuntos de caracteres. El tamaño de **wchar\_t** no se fija en base al estándar.

- Para utilizar objetos **string**, incluya el archivo de encabezado **<string>** de la Biblioteca estándar de C++.
- La clase **string** no proporciona constructores que convierten de **int** o **char** a **string**.
- En una instrucción de asignación está permitido asignar un solo carácter a un objeto **string**.
- Los objetos **string** no necesariamente tienen terminación nula.
- La mayoría de las funciones miembro de **string** reciben como argumentos la ubicación del subíndice inicial y el número de caracteres con los que deben operar.

### Sección 18.2 Asignación y concatenación de objetos **string**

- La clase **string** proporciona el operador **operator=** sobrecargado y la función miembro **assign** para asignaciones de objetos **string**.
- El operador subíndice (**[]**) proporciona acceso de lectura/escritura para cualquier elemento de un objeto **string**.
- La función miembro **at** de **string** proporciona acceso comprobado; al ir más allá de cualquiera de los extremos del objeto **string** se lanza una excepción **out\_of\_range**. El operador subíndice (**[]**) no proporciona acceso comprobado.
- La clase **string** proporciona los operadores **+** y  **$\text{+=}$**  sobrecargados, y la función miembro **append** para realizar la concatenación de objetos **string**.

### Sección 18.3 Comparación de objetos **string**

- La clase **string** proporciona los operadores sobrecargados  **$\text{==}$** ,  **$\text{!=}$** ,  **$\text{<}$** ,  **$\text{>}$** ,  **$\text{<=}$**  y  **$\text{>=}$**  para las comparaciones de objetos **string**.
- La función miembro **compare** de **string** compara dos objetos **string** (o subcadenas) y devuelve 0 si los objetos **string** son iguales, un número positivo si el primer objeto **string** es lexicográficamente mayor que el segundo, o un número negativo si el primer objeto **string** es lexicográficamente menor que el segundo.

### Sección 18.4 Subcadenas

- La función miembro **substr** de **string** obtiene una subcadena de un objeto **string**.

### Sección 18.5 Intercambio de objetos **string**

- La función miembro **swap** de **string** intercambia el contenido de dos objetos **string**.

### Sección 18.6 Características de los objetos **string**

- Las funciones miembro **size** y **length** de **string** devuelven el tamaño o la longitud de un objeto **string** (es decir, el número de caracteres actualmente almacenados en el objeto **string**).
- La función miembro **capacity** de **string** devuelve el número total de caracteres que se pueden almacenar en el objeto **string** sin incrementar la cantidad de memoria asignada al mismo.
- La función miembro **max\_size** de **string** devuelve el tamaño máximo que puede tener un objeto **string**.
- La función miembro **resize** de **string** modifica la longitud de un objeto **string**.

**Sección 18.7 Búsqueda de subcadenas y caracteres en un objeto *string***

- Las funciones de búsqueda `find`, `rfind`, `find_first_of`, `find_last_of` y `find_first_not_of` localizan subcadenas o caracteres en un objeto `string`.

**Sección 18.8 Reemplazo de caracteres en un objeto *string***

- La función miembro `erase` de `string` elimina elementos de un objeto `string`.
- La función miembro `replace` de `string` reemplaza los caracteres en un objeto `string`.

**Sección 18.9 Inserción de caracteres en un objeto *string***

- La función miembro `insert` de `string` inserta caracteres en un objeto `string`.

**Sección 18.10 Conversión a cadenas estilo C**

- La función miembro `c_str` de `string` devuelve un apuntador `const char *` que apunta a una cadena de caracteres estilo C que contiene todos los caracteres en un objeto `string`.
- La función miembro `data` de `string` devuelve un apuntador `const char *` que apunta a un arreglo de caracteres estilo C sin terminación nula, que contiene todos los caracteres en un objeto `string`.

**Sección 18.11 Iteradores**

- La clase `string` proporciona las funciones miembro `end` y `begin` para iterar a través de elementos individuales.
- La clase `string` proporciona las funciones miembro `rend` y `rbegin` para acceder a los caracteres individuales de un objeto `string` en forma inversa, desde el final del objeto `string` hacia el inicio.

**Sección 18.12 Procesamiento de flujos de cadena**

- La entrada desde un objeto `string` está soportada por la clase `istringstream`. La salida hacia un objeto `string` está soportada por la clase `ostringstream`.
- La función miembro `str` de `ostringstream` devuelve una copia `string` de un objeto `string`.

**Terminología**

|                                                                                  |                                                                              |
|----------------------------------------------------------------------------------|------------------------------------------------------------------------------|
| <code>&lt;sstream&gt;</code> , archivo de encabezado                             | <code>find_first_of</code> , función miembro de la clase <code>string</code> |
| acceso comprobado                                                                | <code>find_last_of</code> , función miembro de la clase <code>string</code>  |
| <code>append</code> , función miembro de la clase <code>string</code>            | <code>getline</code> , función miembro de la clase <code>string</code>       |
| <code>assign</code> , función miembro de la clase <code>string</code>            | <code>insert</code> , función miembro de la clase <code>string</code>        |
| <code>at</code> , función miembro de la clase <code>string</code>                | <code>istringstream</code> , clase                                           |
| <code>basic_string</code> , plantilla de clase                                   | <code>iterator</code>                                                        |
| <code>begin</code> , función miembro de la clase <code>string</code>             | <code>length</code> , función miembro de la clase <code>string</code>        |
| <code>c_str</code> , función miembro de la clase <code>string</code>             | longitud de un objeto <code>string</code>                                    |
| cadena vacía                                                                     | <code>max_size</code> , función miembro de la clase <code>string</code>      |
| capacidad de una cadena                                                          | <code>ostringstream</code> , clase                                           |
| <code>capacity</code> , función miembro de la clase <code>string</code>          | procesamiento de flujos <code>stream</code>                                  |
| comparación lexicográfica                                                        | <code>rbegin</code> , función miembro de la clase <code>string</code>        |
| <code>compare</code> , función miembro de la clase <code>string</code>           | <code>rend</code> , función miembro de la clase <code>string</code>          |
| comprobación de rango                                                            | <code>replace</code> , función miembro de la clase <code>string</code>       |
| concatenación                                                                    | <code>resize</code> , función miembro de la clase <code>string</code>        |
| <code>const_iterator</code>                                                      | <code>reverse_iterator</code>                                                |
| <code>const_reverse_iterator</code>                                              | <code>rfind</code> , función miembro de la clase <code>string</code>         |
| <code>copy</code> , función miembro de la clase <code>string</code>              | <code>size</code> , función miembro de la clase <code>string</code>          |
| <code>data</code> , función miembro de la clase <code>string</code>              | <code>Str</code> , función miembro de la clase <code>ostringstream</code>    |
| E/S en memoria                                                                   | <code>string::npos</code> , constante                                        |
| <code>end</code> , función miembro de la clase <code>string</code>               | <code>substr</code> , función miembro de la clase <code>string</code>        |
| <code>erase</code> , función miembro de la clase <code>string</code>             | <code>swap</code> , función miembro de la clase <code>string</code>          |
| <code>find</code> , función miembro de la clase <code>string</code>              | tamaño máximo de un objeto <code>string</code>                               |
| <code>find_first_not_of</code> , función miembro de la clase <code>string</code> | <code>wchar_t</code> , tipo                                                  |

**Ejercicios de autoevaluación****18.1** Complete los siguientes enunciados:

- El encabezado \_\_\_\_\_ se debe incluir para la clase `string`.
- La clase `string` pertenece al namespace \_\_\_\_\_.

**766 Capítulo 18 La clase `string` y el procesamiento de flujos de cadena**

- c) La función \_\_\_\_\_ elimina caracteres de un objeto `string`.  
d) La función \_\_\_\_\_ encuentra la primera ocurrencia de cualquier carácter de un objeto `string`.
- 18.2** Indique si cada uno de los siguientes enunciados es *verdadero* o *falso*. En caso de ser *falso*, explique por qué.
- La concatenación de objetos `string` se puede realizar con el operador de asignación de suma, `+=`.
  - Los caracteres dentro de un objeto `string` empiezan en el índice 0.
  - El operador de asignación (`=`) copia un objeto `string`.
  - Una cadena estilo C es un objeto `string`.
- 18.3** Busque el (los) error(es) en cada una de las siguientes instrucciones, y explique cómo corregirlo(s).
- ```
string cadena1( 28 ); // construye cadena1
string cadena2( 'z' ); // construye cadena2
```
 - ```
// asume que se conoce el espacio de nombres std
const char *ptr = nombre.datos() // nombre es "joe bob"
ptr[3] = '-';
cout << ptr << endl;
```

**Respuestas a los ejercicios de autoevaluación**

- 18.1** a) `<string>`. b) `std`. c) `erase`. d) `find_first_of`.
- 18.2** a) Verdadero.  
b) Verdadero.  
c) Verdadero.  
d) Falso. Un objeto `string` proporciona muchos servicios distintos. Una cadena estilo C no proporciona servicios. Las cadenas estilo C tienen terminación nula; los objetos `string` no necesariamente tienen terminación nula. Las cadenas estilo C son apuntadores y los objetos `string` son objetos.
- 18.3** a) Los constructores para la clase `string` no existen para los argumentos enteros y de caracteres. Deben usarse otros constructores válidos; si es necesario, hay que convertir los argumentos en objetos `string`.  
b) La función `data` no agrega un terminador nulo. Además, el código intenta modificar un valor `const char`. Reemplace todas las líneas con el siguiente código:  
`cout << nombre.substr( 0, 3 ) + "-" + nombre.substr( 4 ) << endl;`

**Ejercicios**

- 18.4** Complete los siguientes enunciados:
- Las funciones miembro \_\_\_\_\_ y \_\_\_\_\_ de la clase `string` convierten objetos `string` en cadenas estilo C.
  - La función miembro \_\_\_\_\_ de la clase `string` se utiliza para la asignación.
  - \_\_\_\_\_ es el tipo de valor de retorno de la función `rbegin`.
  - La función miembro `string` de la clase \_\_\_\_\_ se utiliza para obtener una subcadena.
- 18.5** Indique si cada uno de los siguientes enunciados es *verdadero* o *falso*. En caso de ser *falso*, explique por qué.
- Los objetos `string` siempre tienen terminación nula.
  - La función miembro `max_size` de la clase `string` devuelve el tamaño máximo para un objeto `string`.
  - La función miembro `at` de la clase `string` puede lanzar una excepción `out_of_range`.
  - La función miembro `begin` de la clase `string` devuelve un objeto `iterator`.
- 18.6** Busque errores en las siguientes instrucciones y explique cómo corregirlos:
- `std::cout << s.data() << std::endl // s es "holo"`
  - `erase( s.rfind( "x" ), 1 ); // s es "xenon"`
  - ```
string& foo()
{
    string s( "Hola" );
    ... // otras instrucciones
    return;
} // fin de la función foo
```
- 18.7** (*Cifrado simple*) Cierta información en Internet se puede cifrar con un algoritmo simple conocido como “rot13”, el cual rota cada carácter 13 posiciones en el alfabeto. Así, ‘a’ corresponde a ‘n’, y ‘x’ corresponde a ‘k’. rot13 es un ejemplo del **cifrado de clave simétrica**. Con este tipo de cifrado, tanto el que cifra como el que descifra utilizan la misma clave.

- a) Escriba un programa que cifre un mensaje usando rot13.
- b) Escriba un mensaje que descifre el mensaje codificado usando 13 como la clave.
- c) Después de escribir los programas de los incisos (a) y (b), responda brevemente a la siguiente pregunta: si no conociera la clave para el inciso (b), ¿qué tan difícil cree usted que sería quebrantar el código? ¿Qué pasaría si tuviera acceso a un poder de cómputo considerable (por ejemplo, supercomputadoras)? En el ejercicio 18.26 le pediremos que escriba un programa para lograr esto.

18.8 Escriba un programa usando iteradores, que demuestre el uso de las funciones `rbegin` y `rend`.

18.9 Escriba un programa que lea varios objetos `string` e imprima sólo los que terminen con "r" o "ay". Sólo deben considerarse letras en minúscula.

18.10 Escriba un programa que demuestre cómo pasar un objeto `string`, por valor y por referencia.

18.11 Escriba un programa que introduzca por separado un primer nombre y un apellido, y que concatene los dos en un nuevo objeto `string`.

18.12 Escriba un programa para jugar al ahorcado. El programa debe elegir una palabra (que se codifica directamente en el programa, o se lee de un archivo de texto) y mostrar lo siguiente:

Adivine la palabra: XXXXX

Cada X representa una letra. El usuario trata de adivinar las letras en la palabra. Deberá mostrarse la respuesta apropiada (si o no) después de cada intento de adivinar. Después de cada intento incorrecto, muestre el diagrama con otra parte del cuerpo incluida. Después de siete intentos incorrectos, el usuario deberá colgarse. La pantalla debe tener la siguiente apariencia:



Después de cada intento de adivinar, muestre todos los intentos que hizo el usuario. Si el usuario adivina la palabra correctamente, el programa debe mostrar lo siguiente:

Felicidades!!! Adivino mi palabra. Desea jugar otra vez? si/no

18.13 Escriba un programa que introduzca un objeto `string` y que lo imprima en forma inversa. Convierta todos los caracteres en mayúsculas a minúsculas, y todos los caracteres en minúsculas a mayúsculas.

18.14 Escriba un programa que utilice las herramientas de comparación que presentamos en este capítulo para alfabetizar una serie de nombres de animales. Sólo se deben utilizar letras mayúsculas para la comparación.

18.15 Escriba un programa que cree un criptograma a partir de un objeto `string`. Un criptograma es un mensaje o palabra en donde cada letra se reemplaza por otra. Por ejemplo, el siguiente objeto `string`

El ave se llama graznido

podría codificarse para formar

st fds ks ttfgef rwfjxpmq

Observe que los espacios no se codifican. En este caso específico, 'E' se reemplazó con 's', cada 'a' se reemplazó con 'f', etc. Las letras mayúsculas se deben convertir en minúsculas en el criptograma. Use técnicas similares a las del ejercicio 18.7

18.16 Modifique el ejercicio 18.15 para permitir al usuario resolver el criptograma. El usuario deberá introducir dos caracteres a la vez: el primer carácter especifica una letra en el criptograma, y la segunda letra especifica la letra de reemplazo. Si la letra de reemplazo es correcta, reemplace la letra en el criptograma con la letra de reemplazo en minúscula.

18.17 Escriba un programa que introduzca un enunciado y cuente el número de palíndromos en éste. Un palíndromo es una palabra que se lee lo mismo al revés que al derecho. Por ejemplo, "agua" no es un palíndromo, pero "arenera" sí lo es.

18.18 Escriba un programa que cuente el número total de vocales en un enunciado. Imprima la frecuencia de cada vocal.

18.19 Escriba un programa que inserte los caracteres "*****" en la mitad exacta de un objeto `string`.

18.20 Escriba un programa que elimine las secuencias "por" y "POR" de un objeto `string`.

18.21 Escriba un programa que introduzca una línea de texto, reemplace todos los signos de puntuación con espacios y utilice la función `strtok` de la biblioteca de cadenas estilo C para dividir el objeto `string` en palabras individuales (tokens).

18.22 Escriba un programa que introduzca una línea de texto y la imprima al revés. Use iteradores en su solución.

18.23 Escriba una versión recursiva del ejercicio 18.22.

768 Capítulo 18 La clase `string` y el procesamiento de flujos de cadena

18.24 Escriba un programa que demuestre el uso de las funciones `erase` que reciben argumentos `iterator`.

18.25 Escriba un programa que genere lo siguiente del objeto `string` "abcdefghijklmnopqrstuvwxyz{":

```
a  
bcb  
cdedc  
defgfed  
efghihgfe  
fghijkjihgf  
ghijklmlkjhg  
hijklmnmonmlkjih  
ijklmnopqponmlkji  
jklnopqrstuvwxyz  
klmnopqrstuvwxyz  
lmnopqrstuvwxyz  
mnopqrstuvwxyz  
nopqrstuvwxyz{zyxwvutsrqponm
```

18.26 En el ejercicio 18.7 pedimos al lector que escribiera un algoritmo de cifrado simple. Escriba un programa que trate de descifrar un mensaje "rot13", usando la sustitución simple de frecuencias. (Suponga que no conoce la clave). Las letras más frecuentes en la frase cifrada deben reemplazarse con las letras de uso más común en español (a, e, i, o, u, s, t, r, etc.). Escriba las posibilidades en un archivo. ¿Qué hizo que fuera fácil quebrantar el código? ¿Cómo se puede mejorar el mecanismo de cifrado?

18.27 Escriba una versión de la rutina de ordenamiento por selección (figura 8.28) que ordene objetos `string`. Use la función `swap` en su solución.

18.28 Modifique la clase `Empleado` de las figuras 13.6 y 13.7, agregando una función utilitaria `private` llamada `esValido-NúmeroSeguroSocial`. Esta función miembro debe validar el formato de un número de seguro social (por ejemplo, `###-##-####`, en donde # es un dígito). Si el formato es válido, devuelve `true`; en caso contrario devuelve `false`.



*Con sollozos y lágrimas
él sorteó
los de mayor tamaño...*

—Lewis Carroll

*Intenta el final, y nunca
dejes lugar a dudas;
no hay nada tan difícil
que no pueda averiguarse
mediante la búsqueda.*

—Robert Herrick

*Está bloqueado
en mi memoria,
y tú deberás guardar
la llave.*

—William Shakespeare

*Una ley inmutable en
los negocios es que las
palabras son palabras,
las explicaciones son
explicaciones, las promesas
son promesas; pero sólo el
desempeño es la realidad.*

—Harold S. Green

Búsqueda y ordenamiento

OBJETIVOS

En este capítulo aprenderá a:

- Buscar un valor dado en un vector, usando la búsqueda binaria.
- Utilizar la notación “Big O” para expresar la eficiencia de un algoritmo y comparar el rendimiento de los algoritmos.
- Repasar la eficiencia de los algoritmos de ordenamiento por selección y de ordenamiento por inserción.
- Ordenar un vector, utilizando el algoritmo de ordenamiento por combinación recursivo.
- Determinar la eficiencia de varios algoritmos de búsqueda y ordenamiento.
- Enumerar los algoritmos de búsqueda y ordenamiento descritos en este texto.
- Comprender la naturaleza de los algoritmos de tiempo de ejecución constante, lineal y cuadrático.

- 19.1** Introducción
- 19.2** Algoritmos de búsqueda
 - 19.2.1** Eficiencia de la búsqueda lineal
 - 19.2.2** Búsqueda binaria
- 19.3** Algoritmos de ordenamiento
 - 19.3.1** Eficiencia del ordenamiento por selección
 - 19.3.2** Eficiencia del ordenamiento por inserción
 - 19.3.3** Ordenamiento por combinación (una implementación recursiva)
- 19.4** Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)

19.1 Introducción

La **búsqueda** de datos implica el determinar si un valor (conocido como la **clave de búsqueda**) está presente en los datos y, de ser así, hay que encontrar su ubicación. Dos algoritmos populares de búsqueda son la búsqueda lineal simple (presentado en la sección 7.7) y la búsqueda binaria, que es más rápida pero a la vez más compleja, y que presentaremos en este capítulo.

El **ordenamiento** coloca los datos en orden, por lo general ascendente o descendente, con base en una o más **claves de ordenamiento**. Una lista de nombres se podría ordenar en forma alfabética, las cuentas bancarias podrían ordenarse por número de cuenta, los registros de nóminas de empleados podrían ordenarse por número de seguro social, etcétera. Anteriormente vimos el ordenamiento por inserción (sección 7.8) y el ordenamiento por selección (sección 8.6). En este capítulo se presenta el ordenamiento por combinación, que es más eficiente pero más complejo. En la figura 19.1 se sintetizan los algoritmos de búsqueda y ordenamiento que veremos en los ejemplos y ejercicios de este libro. En este capítulo también se introduce la **notación Big O**, que se utiliza para estimar el tiempo de ejecución para un algoritmo en el peor caso; es decir, qué tan duro tendrá que trabajar un algoritmo para resolver un problema.

19.2 Algoritmos de búsqueda

Buscar un número telefónico, acceder a un sitio Web y comprobar la definición de una palabra en un diccionario son acciones que implican buscar entre grandes cantidades de datos. Todos los algoritmos de búsqueda logran el mismo objetivo: encontrar un elemento que coincida con una clave de búsqueda dada, si es que existe dicho elemento. Sin embargo, hay varias cosas que diferencian a unos algoritmos de búsqueda de otros. La principal diferencia es la cantidad de esfuerzo que requieren para completar la búsqueda. Una manera de describir este esfuerzo es mediante la notación Big O. Para los algoritmos de búsqueda y ordenamiento, esto es en especial dependiente del número de elementos de datos.

En el capítulo 7 hablamos sobre el algoritmo de búsqueda lineal, que es un algoritmo de búsqueda simple y fácil de implementar. Ahora hablaremos sobre la eficiencia del algoritmo de búsqueda lineal en base a la notación Big O. Después presentaremos un algoritmo de búsqueda que sea relativamente eficiente, pero más complejo y difícil de implementar.

19.2.1 Eficiencia de la búsqueda lineal

Suponga que un algoritmo simplemente evalúa si el primer elemento de un vector es igual al segundo elemento. Si el vector tiene 10 elementos, este algoritmo sólo requiere una comparación. Si el vector tiene 1000 elementos, sigue requiriendo una comparación. De hecho, el algoritmo es independiente del número de elementos en el vector. Se dice que este algoritmo tiene un **tiempo de ejecución constante**, el cual se representa en la notación Big O como $O(1)$. Un algoritmo que es $O(1)$ no necesariamente requiere sólo de una comparación. $O(1)$ sólo significa que el número de comparaciones es *constante*; no crece a medida que aumenta el tamaño del arreglo. Un algoritmo que evalúa si el primer elemento de un arreglo es igual a los siguientes tres elementos sigue siendo $O(1)$, aun y cuando requiera tres comparaciones.

Un algoritmo que evalúa si el primer elemento de un vector es igual a *cualquiera* de los demás elementos del vector requerirá cuando menos de $n - 1$ comparaciones, en donde n es el número de elementos en el vector. Si el vector tiene 10 elementos, este algoritmo requiere hasta nueve comparaciones. Si el vector tiene 1000 elementos, requiere hasta 999 comparaciones. A medida que n aumenta en tamaño, la parte de la expresión correspondiente a la n “domina”, y si le restamos uno no hay consecuencias. Big O está diseñado para resaltar estos términos dominantes e ignorar los términos

Capítulo	Algoritmo	Ubicación
<i>Algoritmos de búsqueda:</i>		
7	Búsqueda lineal	Sección 7.7
19	Búsqueda binaria	Sección 19.2.2
	Búsqueda lineal recursiva	Ejercicio 19.8
	Búsqueda binaria recursiva	Ejercicio 19.9
20	Búsqueda con árboles binarios	Sección 20.7
	Búsqueda lineal en una lista enlazada	Ejercicio 20.21
22	Función <code>binary_search</code> de la biblioteca estándar	Sección 22.5.6
<i>Algoritmos de ordenamiento:</i>		
7	Ordenamiento por inserción	Sección 7.8
8	Ordenamiento por selección	Sección 8.6
19	Ordenamiento por combinación recursivo	Sección 19.3.3
	Ordenamiento de burbuja	Ejercicios 19.5 y 19.6
	Ordenamiento de cubeta	Ejercicio 19.7
	Ordenamiento rápido (quicksort) recursivo	Ejercicio 19.10
20	Ordenamiento con árboles binarios	Sección 20.7
22	Función <code>sort</code> de la biblioteca estándar	Sección 22.5.6
	Ordenamiento basado en montículo (heapsort)	Sección 22.5.12

Figura 19.1 | Los algoritmos de búsqueda y ordenamiento de este libro.

que pierden importancia, a medida que n crece. Por esta razón, se dice que un algoritmo que requiere un total de $n - 1$ comparaciones (como el que describimos en este párrafo) es $O(n)$. Se considera que un algoritmo $O(n)$ tiene un **tiempo de ejecución lineal**. A menudo, $O(n)$ significa “en el orden de n ”, o dicho en forma más simple, “orden n ”.

Ahora, suponga que tiene un algoritmo que evalúa si *cualquier* elemento de un vector se duplica en cualquier otra parte del mismo. El primer elemento debe compararse con todos los demás elementos del vector. El segundo elemento debe compararse con todos los demás elementos, excepto con el primero (ya se comparó con éste). El tercer elemento debe compararse con todos los elementos, excepto los primeros dos. Al final, este algoritmo terminará realizando $(n - 1) + (n - 2) + \dots + 2 + 1$, o $n^2/2 - n/2$ comparaciones. A medida que n aumenta, el término n^2 domina y el término n se vuelve intrascendente. De nuevo, la notación Big O resalta el término n^2 , dejando a $n^2/2$. Pero como veremos pronto, los factores constantes se omiten en la notación Big O.

Big O se enfoca en la forma en que aumenta el tiempo de ejecución de un algoritmo, en relación con el número de elementos procesados. Suponga que un algoritmo requiere n^2 comparaciones. Con cuatro elementos, el algoritmo requiere 16 comparaciones; con ocho elementos, 64 comparaciones. Con este algoritmo, al duplicar el número de elementos se cuadriplica el número de comparaciones. Considere un algoritmo similar que requiere $n^2/2$ comparaciones. Con cuatro elementos, el algoritmo requiere ocho comparaciones; con ocho elementos, 32 comparaciones. De nuevo, al duplicar el número de elementos se cuadriplica el número de comparaciones. Ambos algoritmos aumentan como el cuadrado de n , por lo que Big O ignora la constante y ambos algoritmos se consideran como $O(n^2)$, lo cual se conoce como **tiempo de ejecución cuadrático** y se pronuncia como “en el orden de n al cuadrado”, o dicho en forma más simple, “orden n al cuadrado”.

Cuando n es pequeña, los algoritmos $O(n^2)$ (que se ejecutan en las computadoras personales de la actualidad, con miles de millones de operaciones por segundo) no afectan el rendimiento en forma considerable. Pero a medida que n aumenta, se empieza a notar la reducción en el rendimiento. Un algoritmo $O(n^2)$ que se ejecuta en un vector de un millón de elementos requeriría tres mil millones de “operaciones” (en donde cada una requeriría en realidad varias instrucciones de máquina para ejecutarse). Esto podría requerir varias horas para ejecutarse. Un vector de mil millones de elementos requeriría un trillón de operaciones, ¡un número tan grande que el algoritmo tardaría décadas! Por desgracia, los algoritmos $O(n^2)$ tienden a ser más fáciles de escribir. En este capítulo veremos algoritmos con medidas de Big O más favorables. Estos algoritmos eficientes comúnmente requieren un poco más de astucia y esfuerzo para crearlos, pero

su rendimiento superior bien vale la pena el esfuerzo adicional, en especial a medida que n aumenta y los algoritmos se combinan en programas más grandes.

El algoritmo de búsqueda lineal se ejecuta en un tiempo $O(n)$. El peor caso en este algoritmo es que se debe comprobar cada elemento para determinar si la clave de búsqueda existe en el vector. Si el tamaño del vector se duplica, el número de comparaciones que el algoritmo debe realizar también se duplica. Observe que la búsqueda lineal puede proporcionar un rendimiento sorprendente, si el elemento que coincide con la clave de búsqueda se encuentra en (o está cerca de) la parte frontal del vector. Pero buscamos algoritmos que tengan un buen desempeño, en promedio, en todas las búsquedas, incluyendo aquellas en las que el elemento que coincide con la clave de búsqueda se encuentra cerca del final del vector.

La búsqueda lineal es el algoritmo de búsqueda más fácil de programar, pero puede ser lento si se le compara con otros algoritmos de búsqueda. Si un programa necesita realizar muchas búsquedas en vectores grandes, puede ser mejor implementar un algoritmo distinto más eficiente, como la búsqueda binaria, el cual presentaremos en la siguiente sección.



Tip de rendimiento 19.1

Algunas veces los algoritmos más simples tienen un desempeño pobre. Su virtud es que son fáciles de programar, probar y depurar. Algunas veces se requieren algoritmos más complejos para obtener el máximo rendimiento.

19.2.2 Búsqueda binaria

El **algoritmo de búsqueda binaria** es más eficiente que el algoritmo de búsqueda lineal, pero primero requiere que se ordene el vector. Esto sólo vale la pena cuando se realizarán muchas búsquedas en el vector, una vez ordenado, o cuando la aplicación de búsqueda tiene requerimientos de rendimiento estrictos. La primera iteración de este algoritmo evalúa el elemento medio en el vector. Si éste coincide con la clave de búsqueda, el algoritmo termina. Suponiendo que el vector se ordene en forma ascendente, entonces si la clave de búsqueda es menor que el elemento de en medio, no puede coincidir con ningún elemento en la segunda mitad del vector, y el algoritmo continúa sólo con la primera mitad (es decir, el primer elemento hasta, pero sin incluir, el elemento de en medio). Si la clave de búsqueda es mayor que el elemento de en medio, no puede coincidir con ninguno de los elementos de la primera mitad del vector, y el algoritmo continúa sólo con la segunda mitad del vector (es decir, desde el elemento después del elemento de en medio, hasta el último elemento). Cada iteración evalúa el valor medio de la porción restante del vector. Si la clave de búsqueda no coincide con el elemento, el algoritmo elimina la mitad de los elementos restantes. Para terminar, el algoritmo encuentra un elemento que coincide con la clave de búsqueda o reduce el subvector hasta un tamaño de cero.

Como ejemplo, considere el siguiente vector ordenado de 15 elementos:

2 3 5 10 27 30 34 51 65 77 81 82 93 99

y la clave de búsqueda de 65. Un programa que implemente el algoritmo de búsqueda binaria primero comprobaría si el 51 es la clave de búsqueda (ya que 51 es el elemento de en medio del vector). La clave de búsqueda (65) es mayor que 51, por lo que este número se descarta junto con la primera mitad del vector (todos los elementos menores que 51). A continuación, el algoritmo comprueba si 81 (el elemento de en medio del resto del vector) coincide con la clave de búsqueda. La clave de búsqueda (65) es menor que 81, por lo que se descarta este número junto con los elementos mayores de 81. Después de sólo dos pruebas, el algoritmo ha reducido el número de valores a comprobar a tres (56, 65 y 77). Después el algoritmo comprueba el 65 (que coincide con la clave de búsqueda), y devuelve el índice (9) del elemento del vector que contiene el 65. En este caso, el algoritmo sólo requirió tres comparaciones para determinar si la clave de búsqueda coincidió con un elemento del vector. Un algoritmo de búsqueda lineal hubiera requerido 10 comparaciones. [Nota: en este ejemplo hemos optado por usar un vector con 15 elementos, para que siempre haya un elemento obvio en medio del vector. Con un número par de elementos, la parte media del vector se encuentra entre dos elementos. Implementamos el algoritmo para elegir el mayor de estos dos elementos].

En las figuras 19.2 y 19.3 se define la clase `BusquedaBinaria` y sus funciones miembro, respectivamente. La clase `BusquedaBinaria` es similar a `BusquedaLineal` (sección 7.7): tiene un constructor, una función de búsqueda (`busquedaBinaria`), una función `mostrarElementos`, dos miembros de datos `private` y una función utilitaria `private` (`mostrarSubElementos`). En las líneas 18 a 28 de la figura 19.3 se define el constructor. Una vez que se inicializa el vector con valores `int` aleatorios de 10 a 99 (líneas 24 y 25), en la línea 27 se hace una llamada a la función `sort` de la Biblioteca estándar con el vector `datos`. Recuerde que el algoritmo de búsqueda binaria sólo trabaja con un vector ordenado. La función `sort` requiere dos argumentos que especifiquen el rango de elementos a ordenar. Estos argumentos se especifican con iteradores (que vimos brevemente en la sección 10.9, y veremos con detalle en el capítulo 22). Las funciones miembro `begin` y `end` de `vector` devuelven iteradores que se pueden utilizar con la función `sort` para indicar que se deben ordenar todos los elementos, desde el principio hasta el final.

En las líneas 31 a 61 se define la función `busquedaBinaria`. La clave de búsqueda se pasa al parámetro `elementoBusqueda` (línea 31). En las líneas 33 a 35 se calcula el índice del extremo `inferior`, el índice del extremo `superior` y el índice `medio` de la parte del vector en la que el programa está buscando en un momento dado. Al principio de la función, el extremo `inferior` es 0, el extremo `superior` es el tamaño del vector menos 1, y el `medio` es el promedio de estos dos valores. En la línea 36 se inicializa la `ubicacion` del elemento encontrado en -1; el valor que se devolverá si la clave de búsqueda no se encuentra. En las líneas 38 a 58 se itera hasta que `inferior` es mayor que `superior` (esto ocurre cuando el elemento no se encuentra) o `ubicacion` no es igual a -1 (indicando que se encontró la clave de búsqueda). En la línea 50 se evalúa si el valor en el elemento `medio` es igual a `elementoBusqueda`. Si es `true`, en la línea 51 se asigna `medio` a `ubicacion`. Después el ciclo termina y `ubicacion` se devuelve al que hizo la llamada. Cada iteración del ciclo evalúa un solo valor (línea 50) y elimina la mitad de los valores restantes en el vector (línea 53 o 55).

```

1 // Fig 19.2: BusquedaBinaria.h
2 // Clase que contiene un vector de enteros aleatorios y una función
3 // que utiliza la búsqueda binaria para encontrar un entero.
4 #include <vector>
5 using std::vector;
6
7 class BusquedaBinaria
8 {
9 public:
10    BusquedaBinaria( int ); // el constructor inicializa el vector
11    int busquedaBinaria( int ) const; // realiza una búsqueda binaria en el vector
12    void mostrarElementos() const; // muestra los elementos del vector
13 private:
14    int tamano; // tamaño del vector
15    vector< int > datos; // vector of ints
16    void mostrarSubElementos( int, int ) const; // muestra el rango de valores
17 }; // fin de la clase BusquedaBinaria

```

Figura 19.2 | Definición de la clase `BusquedaBinaria`.

```

1 // Fig 19.3: BusquedaBinaria.cpp
2 // Definición de las funciones miembro de la clase BusquedaBinaria.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // prototipos para las funciones srand y rand
8 using std::rand;
9 using std::srand;
10
11 #include <ctime> // prototipo para la función time
12 using std::time;
13
14 #include <algorithm> // prototipo para la función sort
15 #include "BusquedaBinaria.h" // definición de la clase BusquedaBinaria
16
17 // el constructor inicializa el vector con valores int aleatorios y ordena el vector
18 BusquedaBinaria::BusquedaBinaria( int tamanoVector )
19 {
20    tamano = ( tamanoVector > 0 ? tamanoVector : 10 ); // valida tamanoVector
21    srand( time( 0 ) ); // siembra usando el tiempo actual
22
23    // llena el vector con valores int aleatorios en el rango 10 a 99
24    for ( int i = 0; i < tamano; i++ )
25        datos.push_back( 10 + rand() % 90 ); // 10 a 99
26
27    std::sort( datos.begin(), datos.end() ); // ordena los datos
28 } // fin del constructor BusquedaBinaria

```

Figura 19.3 | Definición de la función miembro de la clase `BusquedaBinaria`. (Parte I de 2).

```

29 // realiza una búsqueda binaria en los datos
30 int BusquedaBinaria::busquedaBinaria( int elementoBusqueda ) const
31 {
32     int inferior = 0; // extremo inferior del área de búsqueda
33     int superior = tamano - 1; // extremo superior del área de búsqueda
34     int medio = ( inferior + superior + 1 ) / 2; // elemento medio
35     int ubicacion = -1; // devuelve el valor; -1 si no lo encuentra
36
37     do // itera para buscar el elemento
38     {
39         // imprime el resto de los elementos del vector en el que se busca
40         mostrarSubElementos( inferior, superior );
41
42         // imprime espacios para alineación
43         for ( int i = 0; i < medio; i++ )
44             cout << "    ";
45
46         cout << " * " << endl; // indica el elemento medio actual
47
48         // si encuentra el elemento en el medio
49         if ( elementoBusqueda == datos[ medio ] )
50             ubicacion = medio; // ubicacion es el medio actual
51         else if ( elementoBusqueda < datos[ medio ] ) // el medio es demasiado alto
52             superior = medio - 1; // elimina la mitad superior
53         else // el elemeto medio es demasiado inferior
54             inferior = medio + 1; // elimina la mitad inferior
55
56         medio = ( inferior + superior + 1 ) / 2; // recalcula el elemento medio
57     } while ( ( inferior <= superior ) && ( ubicacion == -1 ) );
58
59     return ubicacion; // devuelve la ubicación de la clave de búsqueda
60 } // fin de la función busquedaBinaria
61
62 // muestra los valores en el vector
63 void BusquedaBinaria::mostrarElementos() const
64 {
65     mostrarSubElementos( 0, tamano - 1 );
66 } // fin de la función mostrarElementos
67
68 // muestra ciertos valores en el vector
69 void BusquedaBinaria::mostrarSubElementos( int inferior, int superior ) const
70 {
71     for ( int i = 0; i < inferior; i++ ) // imprime espacios para alineación
72         cout << "    ";
73
74     for ( int i = inferior; i <= superior; i++ ) // imprime los elementos que quedan en el vector
75         cout << datos[ i ] << " ";
76
77     cout << endl;
78 } // fin de la función mostrarSubElementos
79

```

Figura 19.3 | Definición de la función miembro de la clase BusquedaBinaria. (Parte 2 de 2).

En las líneas 25 a 41 de la figura 19.4 se itera hasta que el usuario introduzca el valor -1. Para cada otro número que introduce el usuario, el programa realiza una búsqueda binaria en los datos para determinar si coinciden con un elemento en el vector. La primera línea de salida de este programa es el vector de valores `int`, en orden ascendente. Cuando el usuario instruye al programa que busque el número 38, el programa primero evalúa el elemento medio, que es 67 (como lo indica el símbolo `*`). La clave de búsqueda es menor que 67, por lo que el programa elimina la segunda mitad del vector y evalúa el elemento medio, empezando desde la primera mitad del vector. La clave de búsqueda es igual a 38, por lo que el programa devuelve el índice 3.

```

1 // Fig 19.4: Fig19_04.cpp
2 // Programa de prueba de BusquedaBinaria.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 #include "BusquedaBinaria.h" // definición de la clase BusquedaBinaria
9
10 int main()
11 {
12     int busquedaInt; // clave de búsqueda
13     int posicion; // ubicación de la clave de búsqueda en el vector
14
15     // crea el vector y lo imprime
16     BusquedaBinaria vectorBusqueda ( 15 );
17     vectorBusqueda.mostrarElementos();
18
19     // obtiene la entrada del usuario
20     cout << "\nEscriba un valor entero (-1 para terminar): ";
21     cin >> busquedaInt; // lee un valor int del usuario
22     cout << endl;
23
24     // introduce un entero en forma repetida; -1 termina el programa
25     while ( busquedaInt != -1 )
26     {
27         // usa la búsqueda binaria para tratar de encontrar el entero
28         posicion = vectorBusqueda.busquedaBinaria( busquedaInt );
29
30         // el valor de retorno de -1 indica que no se encontró el entero
31         if ( posicion == -1 )
32             cout << "El entero " << busquedaInt << " no se encontró.\n";
33         else
34             cout << "El entero " << busquedaInt
35             << " se encontró en la posición " << posicion << ".\n";
36
37         // obtiene la entrada del usuario
38         cout << "\n\nEscriba un valor entero (-1 para terminar): ";
39         cin >> busquedaInt; // lee un valor int del usuario
40         cout << endl;
41     } // fin de while
42
43     return 0;
44 } // fin de main

```

```

27 30 32 33 36 40 57 60 73 76 83 88 94 96 99
Escriba un valor entero (-1 para terminar): 33
27 30 32 33 36 40 57 60 73 76 83 88 94 96 99
*
27 30 32 33 36 40 57
*
El entero 33 se encontró en la posición 3.

Escriba un valor entero (-1 para terminar): 96
27 30 32 33 36 40 57 60 73 76 83 88 94 96 99
*
73 76 83 88 94 96 99
*
94 96 99
*
El entero 96 se encontró en la posición 13.

```

Figura 19.4 | Programa de prueba de BusquedaBinaria. (Parte I de 2).

```
Escriba un valor entero (-1 para terminar): 25
27 30 32 33 36 40 57 60 73 76 83 88 94 96 99
*
27 30 32 33 36 40 57
*
27 30 32
*
27
*
El entero 25 no se encontró.
```

```
Escriba un valor entero (-1 para terminar): -1
```

Figura 19.4 | Programa de prueba de BusquedaBinaria. (Parte 2 de 2).

Eficiencia de la búsqueda binaria

En el peor de los casos, el proceso de buscar en un vector ordenado de 1023 elementos sólo requiere 10 comparaciones cuando se utiliza una búsqueda binaria. Al dividir 1023 entre 2 en forma repetida (ya que después de cada comparación podemos eliminar la mitad del vector) y redondear (porque también eliminamos el elemento medio), se producen los valores 511, 255, 127, 63, 31, 15, 7, 3, 1 y 0. El número 1023 ($2^{10} - 1$) se divide entre 2 sólo 10 veces para obtener el valor 0, que indica que no hay más elementos para probar. La división entre 2 equivale a una comparación en el algoritmo de búsqueda binaria. Por ende, un vector de 1,048,575 ($2^{20} - 1$) elementos requiere un máximo de 20 comparaciones para encontrar la clave, y un vector de más de mil millones de elementos requiere un máximo de 30 comparaciones para encontrar la clave. Ésta es una enorme mejora en el rendimiento, en comparación con la búsqueda lineal. Para un vector de mil millones de elementos, ésta es una diferencia entre un promedio de 500 millones de comparaciones para la búsqueda lineal, ¡y un máximo de sólo 30 comparaciones para la búsqueda binaria! El número máximo de comparaciones necesarias para la búsqueda binaria de cualquier vector ordenado es el exponente de la primera potencia de 2 mayor que el número de elementos en el vector, que se representa como $\log_2 n$. Todos los logaritmos crecen aproximadamente a la misma proporción, por lo que en notación Big O se puede omitir la base. Esto produce un valor Big O de $O(\log n)$ para una búsqueda binaria, que también se conoce como **tiempo de ejecución logarítmico** y se pronuncia “en el orden de $\log n$ ”, o en forma más simple como “orden $\log n$ ”.

19.3 Algoritmos de ordenamiento

El ordenamiento de datos (es decir, colocar los datos en cierto orden específico, como ascendente o descendente) es una de las aplicaciones computacionales más importantes. Un banco ordena todos los cheques por número de cuenta, de manera que pueda preparar instrucciones bancarias individuales al final de cada mes. Las compañías telefónicas ordenan sus listas de cuentas por apellido paterno y luego por primer nombre, para facilitar el proceso de buscar números telefónicos. Casi cualquier organización debe ordenar datos, y a menudo cantidades masivas de ellos. El ordenamiento de datos es un problema intrigante, que requiere un uso intensivo de la computadora, y ha atraído un enorme esfuerzo de investigación.

Un punto importante a comprender acerca del ordenamiento es que el resultado final (el vector ordenado) será el mismo, sin importar qué algoritmo se utilice para ordenarlo. La elección del algoritmo sólo afecta al tiempo de ejecución y el uso que haga el programa de la memoria. En capítulos anteriores presentamos el ordenamiento por selección y el ordenamiento por inserción: algoritmos simples de implementar pero ineficientes. En la siguiente sección examinaremos la eficiencia de estos dos algoritmos, usando la notación Big O. El último algoritmo (ordenamiento por combinación, que presentaremos en este capítulo) es mucho más rápido pero más difícil de implementar.

19.3.1 Eficiencia del ordenamiento por selección

El ordenamiento por selección es un algoritmo de ordenamiento fácil de implementar, pero ineficiente. En la primera iteración del algoritmo se selecciona el elemento más pequeño en el vector, y se intercambia con el primer elemento. En la segunda iteración se selecciona el segundo elemento más pequeño (que viene siendo el elemento más pequeño de los elementos restantes) y se intercambia con el segundo elemento. El algoritmo continúa hasta que en la última iteración se selecciona el segundo elemento más grande y se intercambia con el índice del segundo al último, dejando el elemento más grande en el último índice. Después de la i -ésima iteración, los i elementos más pequeños del vector se ordenarán en forma ascendente, en los primeros i elementos del vector.

El algoritmo de ordenamiento itera $n - 1$ veces, intercambiando cada vez el elemento restante más pequeño en su posición ordenada. Para localizar el elemento restante más pequeño se requieren $n - 1$ comparaciones durante la primera iteración, $n - 2$ durante la segunda iteración, después $n - 3, \dots, 3, 2, 1$. Esto produce un total de $n(n - 1)/2$ o $(n^2 - n)/2$ comparaciones. En notación Big O, los términos más pequeños se eliminan y las constantes se ignoran, lo cual nos deja un valor Big O final de $O(n^2)$.

19.3.2 Eficiencia del ordenamiento por inserción

El ordenamiento por inserción es otro algoritmo de ordenamiento simple, pero eficiente. En la primera iteración de este algoritmo se toma el segundo elemento en el vector y, si es menor que el primero, se intercambian. En la segunda iteración se analiza el tercer elemento y se inserta en la posición correcta, respecto a los primeros dos elementos, de manera que los tres elementos estén ordenados. En la i -ésima iteración de este algoritmo, los primeros i elementos en el vector original estarán ordenados.

El algoritmo de ordenamiento por inserción itera $n - 1$ veces, insertando un elemento en la posición apropiada en los elementos ordenados hasta ese momento. Por cada iteración, para determinar en dónde se debe insertar el elemento tal vez haya que compararlo con cada uno de los siguientes elementos en el vector. En el peor caso se requerirán $n - 1$ comparaciones. Cada instrucción de repetición individual se ejecuta en un tiempo igual a $O(n)$. Para determinar la notación Big O, las instrucciones anidadas indican que debemos multiplicar el número de comparaciones. Para cada iteración de un ciclo exterior, habrá cierto número de iteraciones en el ciclo interior. En este algoritmo, para cada $O(n)$ iteraciones del ciclo exterior, habrá $O(n)$ iteraciones del ciclo interior, con lo cual se produce un valor Big O de $O(n * n)$, o de $O(n^2)$.

19.3.3 Ordenamiento por combinación (una implementación recursiva)

El ordenamiento por combinación es un algoritmo de ordenamiento eficiente, pero en concepto es más complejo que los ordenamientos de selección y de inserción. Para ordenar un vector, el algoritmo de ordenamiento por combinación lo divide en dos subvectores de igual tamaño, ordena cada subvector y después los combina en un vector más grande. Con un número impar de elementos, el algoritmo crea los dos subvectores de tal forma que uno tenga más elementos que el otro.

La implementación del ordenamiento por combinación en este ejemplo es recursiva. El caso base es un vector con un elemento que, desde luego, está ordenado, por lo que el ordenamiento por combinación regresa de inmediato cuando se le llama con un vector de un elemento. El paso recursivo divide a un vector de dos o más elementos en dos subvectores de igual tamaño, ordena en forma recursiva cada subvector y después los combina en un vector ordenado de mayor tamaño. [De nuevo, si hay un número impar de elementos, un subvector tiene un elemento más grande que el otro].

Suponga que el algoritmo ya ha combinado vectores más pequeños para crear los vectores ordenados A:

4 10 34 56 77

y B:

5 30 51 52 93

El ordenamiento por combinación combina estos dos vectores en un arreglo ordenado de mayor tamaño. El valor más pequeño en A es 4 (que se encuentra en el elemento cero de A). El valor más pequeño en B es 5 (que se encuentra en el índice cero de B). Para poder determinar el elemento más pequeño en el vector más grande, el algoritmo compara 4 y 5. El valor de A es más pequeño, por lo que el 4 se convierte en el valor del primer elemento en el vector combinado. El algoritmo continúa, para lo cual compara 10 (el valor del segundo elemento en A) con 5 (el valor del primer elemento en B). El valor de B es más pequeño, por lo que 5 se convierte en el valor del segundo elemento en el vector más grande. El algoritmo continúa comparando 10 con 30, en donde 10 se convierte en el valor del tercer elemento en el vector, y así en lo sucesivo.

En la figura 19.5 se define la clase `OrdenamientoCombinacion`, y en las líneas 31 a 34 de la figura 19.6 se define la función `ordenar`. En la línea 33 se hace una llamada a la función `ordenarSubVector` con 0 y `tamano - 1` como argumentos. Estos argumentos corresponden a los índices inicial y final del vector que se va a ordenar, con lo cual `ordenarSubVector` opera en todo el vector. La función `ordenarSubVector` se define en las líneas 37 a 61. En la línea 40 se evalúa el caso base. Si el tamaño del vector es 1, significa que ya está ordenado, por lo que la función simplemente regresa de inmediato. Si el tamaño del vector es mayor que 1, la función divide el vector en dos, llama en forma recursiva a la función `ordenarSubVector` para ordenar los dos subvectores y después los combina. En la línea 55 se hace una llamada recursiva a la función `ordenarSubVector` en la primera mitad del vector, y en la línea 56 se llama en forma recursiva a la función `ordenarSubVector` en la segunda mitad del vector. Cuando regresan estas dos llamadas a las funciones, cada mitad del vector se ha ordenado. En la línea 59 se hace una llamada a la función `combinar` (líneas 64 a 108) en las dos mitades del vector, para combinar los dos vectores ordenados en un vector ordenado de mayor tamaño.

```

1 // Fig 19.5: OrdenamientoCombinacion.h
2 // Clase que crea un vector lleno de enteros aleatorios.
3 // Proporciona una función para ordenar el vector con el ordenamiento por combinación.
4 #include <vector>
5 using std::vector;
6
7 // Definición de la clase OrdenamientoCombinacion
8 class OrdenamientoCombinacion
9 {
10 public:
11     OrdenamientoCombinacion( int ); // el constructor inicializa el vector
12     void ordenar(); // ordena el vector usando el ordenamiento por combinación
13     void mostrarElementos() const; // muestra los elementos del vector
14 private:
15     int tamaño; // tamaño del vector
16     vector< int > datos; // vector de valores int
17     void ordenarSubVector( int, int ); // ordena el subvector
18     void combinar( int, int, int, int ); // combina dos vectores ordenados
19     void mostrarSubVector( int, int ) const; // muestra el subvector
20 }; // fin de la clase OrdenamientoCombinacion

```

Figura 19.5 | Definición de la clase OrdenamientoCombinacion.

```

1 // Fig 19.6: OrdenamientoCombinacion.cpp
2 // Definición de las funciones miembro de la clase OrdenamientoCombinacion.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <vector>
8 using std::vector;
9
10 #include <cstdlib> // prototipos para las funciones srand y rand
11 using std::rand;
12 using std::srand;
13
14 #include <ctime> // prototipo para la función time
15 using std::time;
16
17 #include "OrdenamientoCombinacion.h" // definición de la clase OrdenamientoCombinacion
18
19 // el constructor llena el vector con enteros aleatorios
20 OrdenamientoCombinacion::OrdenamientoCombinacion( int tamañoVector )
21 {
22     tamaño = ( tamañoVector > 0 ? tamañoVector : 10 ); // valida tamañoVector
23     srand( time( 0 ) ); // siempre el generador de números aleatorios usando la hora actual
24
25     // llena el vector con valores int aleatorios en el rango de 10 a 99
26     for ( int i = 0; i < tamaño; i++ )
27         datos.push_back( 10 + rand() % 90 );
28 } // fin del constructor de OrdenamientoCombinacion
29
30 // divide el vector, ordena los subvectores y los combina en un vector ordenado
31 void OrdenamientoCombinacion::ordenar()
32 {
33     ordenarSubVector( 0, tamaño - 1 ); // ordena todo el vector en forma recursiva
34 } // fin de la función ordenar
35
36 // función recursiva para ordenar los subvectores
37 void OrdenamientoCombinacion::ordenarSubVector( int inferior, int superior )

```

Figura 19.6 | Definición de las funciones miembro de la clase OrdenamientoCombinacion. (Parte I de 3).

```

38  {
39      // prueba el caso base; el tamaño del vector es igual a 1
40      if ( ( superior - inferior ) >= 1 ) // si no es el caso base
41      {
42          int medio1 = ( inferior + superior ) / 2; // calcula el elemento medio del vector
43          int medio2 = medio1 + 1; // calcula el siguiente elemento más arriba
44
45          // imprime el paso de división
46          cout << " division: ";
47          mostrarSubVector( inferior, superior );
48          cout << endl << " ";
49          mostrarSubVector( inferior, medio1 );
50          cout << endl << " ";
51          mostrarSubVector( medio2, superior );
52          cout << endl << endl;
53
54          // divide el vector a la mitad; ordena cada mitad (llamadas recursivas)
55          ordenarSubVector( inferior, medio1 ); // primera mitad del vector
56          ordenarSubVector( medio2, superior ); // segunda mitad del vector
57
58          // combina dos vectores ordenados después de que regresan las llamadas de división
59          combinar( inferior, medio1, medio2, superior );
60      } // fin de if
61  } // fin de la función ordenarSubVector
62
63 // combina dos subvectores ordenados en un subvector ordenado
64 void OrdenamientoCombinacion::combinar( int izquierdo, int medio1, int medio2, int derecho )
65 {
66     int indiceIzq = izquierdo; // índice en el subvector izquierdo
67     int indiceDer = medio2; // índice en el subvector derecho
68     int indiceCombinado = izquierdo; // índice en vector de trabajo temporal
69     vector< int > combinado( tamano ); // vector de trabajo
70
71     // imprime dos subvectores antes de combinar
72     cout << "combinacion:" ;
73     mostrarSubVector( izquierdo, medio1 );
74     cout << endl << " ";
75     mostrarSubVector( medio2, derecho );
76     cout << endl;
77
78     // combina los vectores hasta llegar al final de uno de ellos
79     while ( indiceIzq <= medio1 && indiceDer <= derecho )
80     {
81         // coloca el menor de dos elementos actuales en el resultado
82         // y se desplaza al siguiente espacio en el vector
83         if ( datos[ indiceIzq ] <= datos[ indiceDer ] )
84             combinado[ indiceCombinado++ ] = datos[ indiceIzq++ ];
85         else
86             combinado[ indiceCombinado++ ] = datos[ indiceDer++ ];
87     } // fin de while
88
89     if ( indiceIzq == medio2 ) // si está al final del vector izquierdo
90     {
91         while ( indiceDer <= derecho ) // copia en el resto del vector derecho
92             combinado[ indiceCombinado++ ] = datos[ indiceDer++ ];
93     } // fin de if
94     else // al final del vector derecho
95     {
96         while ( indiceIzq <= medio1 ) // copia en el resto del vector izquierdo
97             combinado[ indiceCombinado++ ] = datos[ indiceIzq++ ];
98     } // fin de else

```

Figura 19.6 | Definición de las funciones miembro de la clase OrdenamientoCombinacion. (Parte 2 de 3).

```

99
100 // copia los valores de vuelta al vector original
101 for ( int i = izquierdo; i <= derecho; i++ )
102     datos[ i ] = combinado[ i ];
103
104 // imprime el vector combinado
105 cout << " ";
106 mostrarSubVector( izquierdo, derecho );
107 cout << endl << endl;
108 } // fin de la función combinar
109
110 // muestra los elementos en el vector
111 void OrdenamientoCombinacion::mostrarElementos() const
112 {
113     mostrarSubVector( 0, tamaño - 1 );
114 } // fin de la función mostrarElementos
115
116 // muestra ciertos valores en el vector
117 void OrdenamientoCombinacion::mostrarSubVector( int inferior, int superior ) const
118 {
119     // imprime espacios para alineación
120     for ( int i = 0; i < inferior; i++ )
121         cout << " ";
122
123     // imprime elementos izquierdo en el vector
124     for ( int i = inferior; i <= superior; i++ )
125         cout << " " << datos[ i ];
126 } // fin de la función mostrarSubVector

```

Figura 19.6 | Definición de las funciones miembro de la clase `OrdenamientoCombinacion`. (Parte 3 de 3).

En las líneas 79 a 87 de la función `combinar` se itera hasta que el programa llega al final de uno de los subvectores. En la línea 83 se evalúa cuál elemento al inicio de los vectores es menor. Si el elemento en el vector izquierdo es menor, en la línea 86 se le coloca en posición en el vector combinado. Cuando se completa el ciclo `while` (línea 87), se coloca todo un subvector completo en el vector combinado, pero el otro subvector sigue conteniendo datos. En la línea 89 se evalúa si el vector izquierdo ha llegado al final. De ser así, en las líneas 91 y 92 se llena el vector combinado con los elementos del vector derecho. Si el vector izquierdo no ha llegado al final, entonces el vector derecho debe haber llegado al final, y en las líneas 96 y 97 se llena el vector combinado con los elementos del vector izquierdo. Por último, en las líneas 101 y 102 se copia el vector combinado en el vector original. En la figura 19.7 se crea y utiliza un objeto `OrdenamientoCombinacion`. Los resultados de este programa muestran las divisiones y combinaciones realizadas por el ordenamiento por combinación, mostrando el progreso del ordenamiento en cada paso del algoritmo.

Eficiencia del ordenamiento por combinación

El ordenamiento por combinación es un algoritmo mucho más eficiente que el ordenamiento por inserción o por selección (aunque puede ser difícil de creer si analizamos los resultados de la figura 19.7, en donde se puede ver que realiza muchas operaciones). Considere la primera llamada (no recursiva) a la función `ordenarSubVector`. Esto produce dos llamadas recursivas a la función `ordenarSubVector` con subvectores que tienen un tamaño aproximado a la mitad del vector original, y una sola llamada a la función `combinar`. Esta llamada a la función `combinar` requiere, en el peor caso, $n - 1$ comparaciones para llenar el vector original, que es $O(n)$.

```

1 // Fig 19.07: Fig20_07.cpp
2 // Programa de prueba de OrdenamientoCombinacion.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6

```

Figura 19.7 | Programa de prueba de `OrdenamientoCombinacion`. (Parte 1 de 3).

```

7 #include "OrdenamientoCombinacion.h" // definición de la clase OrdenamientoCombinacion
8
9 int main()
10 {
11     // crea un objeto para realizar el ordenamiento por combinación
12     OrdenamientoCombinacion vectorOrdenamiento( 10 );
13
14     cout << "Vector desordenado:" << endl;
15     vectorOrdenamiento.mostrarElementos(); // imprime vector desordenado
16     cout << endl << endl;
17
18     vectorOrdenamiento.ordenar(); // ordena el vector
19
20     cout << "Vector ordenado:" << endl;
21     vectorOrdenamiento.mostrarElementos(); // imprime el vector ordenado
22     cout << endl;
23     return 0;
24 } // fin de main

```

Vector desordenado:

30 47 22 67 79 18 60 78 26 54

division: 30 47 22 67 79 18 60 78 26 54
 30 47 22 67 79
 18 60 78 26 54

division: 30 47 22 67 79
 30 47 22
 67 79

division: 30 47 22
 30 47
 22

division: 30 47
 30
 47

combinacion: 30
 47
 30 47

combinacion: 30 47
 22
 22 30 47

division: 67 79
 67
 79

combinacion: 67
 79
 67 79

combinacion: 22 30 47
 67 79
 22 30 47 67 79

division: 18 60 78 26 54
 18 60 78
 26 54

division: 18 60 78
 18 60
 78

(continúa...)

Figura 19.7 | Programa de prueba de OrdenamientoCombinacion. (Parte 2 de 3).

```

division:          18 60
                  18
                  60

combinacion:      18
                  60
                  18 60

combinacion:      18 60
                  78
                  18 60 78

division:          26 54
                  26
                  54

combinacion:      26
                  54
                  26 54

combinacion:      18 60 78
                  26 54
                  18 26 54 60 78

combinacion: 22 30 47 67 79
                  18 26 54 60 78
                  18 22 26 30 47 54 60 67 78 79

Vector ordenado:
  18 22 26 30 47 54 60 67 78 79

```

Figura 19.7 | Programa de prueba de OrdenamientoCombinacion. (Parte 3 de 3).

(Recuerde que cada elemento del vector se selecciona comparando un elemento de cada uno de los subvectores). Las dos llamadas a la función `ordenarSubVector` producen cuatro llamadas recursivas más a la función `ordenarSubVector` (cada una con un subvector de un tamaño aproximado a una cuarta parte del tamaño del vector original) y dos llamadas a la función `combinar`. Cada una de estas dos llamadas a la función `combinar` requiere, en el peor caso, $n/2 - 1$ comparaciones, para un número total de comparaciones de $O(n)$. Este proceso continúa, y cada llamada a `ordenarSubVector` genera dos llamadas adicionales a `ordenarSubVector` y una llamada a `combinar`, hasta que el algoritmo haya dividido el vector en subvectores de un elemento. En cada nivel, se requieren $O(n)$ comparaciones para combinar los subvectores. Cada nivel divide el tamaño de los vectores a la mitad, por lo que al duplicar el tamaño del vector se requiere un nivel más. Al cuadruplicar el tamaño del vector se requieren dos niveles más. Este patrón es logarítmico y produce $\log_2 n$ niveles. Esto resulta en una eficiencia total de $O(n \log n)$. En la figura 19.8 se sintetizan muchos de los algoritmos de búsqueda y ordenamiento que se cubren en este libro, y se listan las notaciones Big O para cada uno de ellos. En la figura 19.9 se listan los valores Big O que hemos cubierto en este capítulo, junto con varios valores para n , de manera que se resalten las diferencias en las tasas de crecimiento.

Algoritmo	Ubicación	Big O
<i>Algoritmos de búsqueda</i>		
Búsqueda lineal	Sección 7.7	$O(n)$
Búsqueda binaria	Sección 19.2.2	$O(\log n)$
Búsqueda lineal recursiva	Ejercicio 19.8	$O(n)$
Búsqueda binaria recursiva	Ejercicio 19.9	$O(\log n)$

Figura 19.8 | Algoritmos de búsqueda y ordenamiento con valores Big O. (Parte 1 de 2).

Algoritmo	Ubicación	Big O
<i>Algoritmos de búsqueda</i>		
Ordenamiento por inserción	Sección 7.8	$O(n^2)$
Ordenamiento por selección	Sección 8.6	$O(n^2)$
Ordenamiento por combinación	Sección 19.3.3	$O(n \log n)$
Ordenamiento de burbuja	Ejercicio 16.3 y 16.4	$O(n^2)$
Ordenamiento rápido (quicksort)	Ejercicio 19.10	Peor caso: $O(n^2)$ Caso promedio: $O(n \log n)$

Figura 19.8 | Algoritmos de búsqueda y ordenamiento con valores Big O. (Parte 2 de 2).

n	Valor decimal aproximado	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
2^{10}	1000	10	2^{10}	$10 \cdot 2^{10}$	2^{20}
2^{20}	1,000,000	20	2^{20}	$20 \cdot 2^{20}$	2^{40}
2^{30}	1,000,000,000	30	2^{30}	$30 \cdot 2^{30}$	2^{60}

Figura 19.9 | Número aproximado de comparaciones para las notaciones Big O comunes.

19.4 Repaso

En este capítulo vimos cómo realizar búsquedas y ordenar datos. Hablamos sobre el algoritmo de búsqueda binaria, que es más rápido pero más complejo que la búsqueda lineal (sección 7.7). El algoritmo de búsqueda binaria sólo funciona con un arreglo ordenado, pero cada iteración de la búsqueda binaria deja fuera a la mitad de los elementos en el arreglo. También vimos el algoritmo de ordenamiento por combinación, que es más eficiente que el ordenamiento por inserción (sección 7.8) o el ordenamiento por selección (sección 8.6). Además presentamos la notación Big O, que nos ayuda a expresar la eficiencia de un algoritmo. La notación Big O mide el tiempo de ejecución para un algoritmo en el peor caso. El valor Big O es útil para comparar algoritmos y seleccionar el más eficiente. En el siguiente capítulo aprenderá acerca de las estructuras dinámicas de datos que pueden aumentar o reducir su tamaño en tiempo de ejecución.

Resumen

Sección 19.1 Introducción

- La búsqueda de datos implica determinar si una clave de búsqueda está presente en los datos y, de ser así, encontrar su ubicación.
- El ordenamiento implica poner los datos en orden.
- Una manera de describir la eficiencia de un algoritmo es mediante la notación Big O (O), que indica qué tanto trabajo tiene que realizar un algoritmo para resolver un problema.

Sección 19.2 Algoritmos de búsqueda

- La principal diferencia entre los algoritmos de búsqueda es la cantidad de esfuerzo que requieren para poder devolver un resultado.

Sección 19.2.1 Eficiencia de la búsqueda lineal

- Para los algoritmos de búsqueda y ordenamiento, Big O describe cómo varía la cantidad de esfuerzo de un algoritmo específico, dependiendo de cuántos elementos haya en los datos.
- Se dice que un algoritmo que es $O(1)$ tiene un tiempo de ejecución constante. Esto no significa que el algoritmo requiera sólo una comparación; sólo significa que el número de comparaciones no aumenta a medida que se incrementa el tamaño del vector.

- Se considera que un algoritmo $O(n)$ tiene un tiempo de ejecución lineal.
- La notación Big O está diseñada para resaltar los factores dominantes, e ignorar los términos que pierden importancia con valores altos de n .
- La notación Big O representa la proporción de crecimiento de los tiempos de ejecución de los algoritmos, por lo que se ignoran las constantes.
- El algoritmo de búsqueda lineal se ejecuta en un tiempo $O(n)$.
- En el peor caso para la búsqueda lineal, se debe comprobar cada elemento para determinar si el elemento de búsqueda existe. Esto ocurre si la clave de búsqueda es el último elemento en el vector, o si no está presente.

Sección 19.2.2 Búsqueda binaria

- El algoritmo de búsqueda binaria es más eficiente que el algoritmo de búsqueda lineal, pero primero requiere que el vector esté ordenado. Esto sólo vale la pena cuando se van a realizar muchas búsquedas en el vector, una vez ordenado; o cuando la aplicación de búsqueda tiene requerimientos de rendimiento estrictos.
- La primera iteración de la búsqueda binaria evalúa el elemento medio del vector. Si es igual a la clave de búsqueda, el algoritmo devuelve su ubicación. Si la clave de búsqueda es menor que el elemento medio, la búsqueda binaria continúa con la primera mitad del vector. Si la clave de búsqueda es mayor que el elemento medio, la búsqueda binaria continúa con la segunda mitad del vector. En cada iteración de la búsqueda binaria se evalúa el valor medio del resto del vector y, si no se encuentra el elemento, se elimina la mitad de los elementos restantes.
- La búsqueda binaria es más eficiente que la búsqueda lineal, ya que en cada comparación elimina la mitad de los elementos del vector a considerar.
- La búsqueda binaria se ejecuta en un tiempo $O(\log n)$, ya que cada paso elimina la mitad de los elementos restantes.
- Si el tamaño del vector se duplica, la búsqueda binaria sólo requiere una comparación adicional para completarse con éxito.

Sección 19.3.1 Eficiencia del ordenamiento por selección

- El ordenamiento por selección es un algoritmo de ordenamiento simple, pero ineficiente.
- En la primera iteración del algoritmo por selección, se selecciona el elemento más pequeño en el vector y se intercambia con el primer elemento. En la segunda iteración del ordenamiento por selección, se selecciona el segundo elemento más pequeño (que viene siendo el elemento restante más pequeño) y se intercambia con el segundo elemento. El ordenamiento por selección continúa hasta que en la última iteración se selecciona el segundo elemento más grande, y se intercambia con el antepenúltimo elemento, dejando el elemento más grande en el último índice. En la i -ésima iteración del ordenamiento por selección, los i elementos más pequeños de todo el vector se ordenan en los primeros i elementos.

Sección 19.3.2 Eficiencia del ordenamiento por inserción

- El algoritmo de ordenamiento por selección se ejecuta en un tiempo $O(n^2)$.
- En la primera iteración del ordenamiento por inserción, se toma el segundo elemento en el arreglo y, si es menor que el primer elemento, se intercambian. En la segunda iteración del ordenamiento por inserción, se analiza el tercer elemento y se inserta en la posición correcta, respecto a los primeros dos elementos. Después de la i -ésima iteración del ordenamiento por inserción, quedan ordenados los primeros i elementos del arreglo original. Sólo se requieren $n - 1$ inserciones.
- El algoritmo de ordenamiento por inserción se ejecuta en un tiempo $O(n^2)$.

Sección 19.3.3 Ordenamiento por combinación (una implementación recursiva)

- El ordenamiento por combinación es un algoritmo de ordenamiento que es más rápido, pero más complejo de implementar, que el ordenamiento por selección y el ordenamiento por inserción.
- Para ordenar un vector, el algoritmo de ordenamiento por combinación lo divide en dos subvectores de igual tamaño, ordena cada subvector en forma recursiva y combina los subvectores en un vector más grande.
- El caso base del ordenamiento por combinación es un vector con un elemento. Un arreglo de un elemento ya está ordenado, por lo que el ordenamiento por combinación regresa de inmediato, cuando se llama con un vector de un elemento. La parte de este algoritmo que corresponde al proceso de combinar recibe dos vectores ordenados (éstos podrían ser vectores de un elemento) y los combina en un vector ordenado más grande.
- Para realizar la combinación, el ordenamiento por combinación analiza el primer elemento en cada vector, que también es el elemento más pequeño en el vector. El ordenamiento por combinación recibe el más pequeño de estos elementos y lo coloca en el primer elemento del vector ordenado más grande. Si aún hay elementos en el subvector, el ordenamiento por combinación analiza el segundo elemento en ese subvector (que ahora es el elemento más pequeño restante) y lo compara con el primer elemento en el otro subvector. El ordenamiento por combinación continúa con este proceso, hasta que se llena el arreglo más grande.
- En el peor caso, la primera llamada al ordenamiento por combinación tiene que realizar $O(n)$ comparaciones para llenar las n posiciones en el arreglo final.

- La porción del algoritmo de ordenamiento por combinación que corresponde al proceso de combinar se realiza en dos subvectores, cada uno de un tamaño aproximado a $n/2$. Para crear cada uno de estos subvectores, se requieren $n/2 - 1$ comparaciones para cada subvector, o un total de $O(n)$ comparaciones. Este patrón continúa a medida que cada nivel trabaja hasta en el doble de esa cantidad de arreglos, pero cada uno equivale a la mitad del tamaño del vector anterior.
- De manera similar a la búsqueda binaria, esta acción de partir los subvectores a la mitad produce un total de $\log n$ niveles, en donde cada nivel requiere $O(n)$ comparaciones, para una eficiencia total de $O(n \log n)$.

Terminología

algoritmos de búsqueda eficientes	$O(n \log n)$
algoritmos de búsqueda ineficientes	$O(n)$
algoritmos de ordenamiento eficientes	$O(n^2)$
algoritmos de ordenamiento ineficientes	orden 1
Big O, notación	orden $\log n$
búsqueda binaria	orden n
búsqueda de datos	orden n al cuadrado
búsqueda lineal	ordenamiento de burbuja (ejercicio)
clave de búsqueda	ordenamiento por combinación (implementación recursiva)
combinar dos vectores	ordenamiento por selección
dividir el vector en el ordenamiento por combinación	ordenamiento rápido (quicksort)
eficiencia de la búsqueda binaria	ordenar datos
eficiencia de la búsqueda lineal	sort, función de la biblioteca estándar
eficiencia del ordenamiento por combinación	tiempo de ejecución constante
eficiencia del ordenamiento por inserción	tiempo de ejecución cuadrático
eficiencia del ordenamiento por selección	tiempo de ejecución lineal
iterador de acceso aleatorio	tiempo de ejecución logarítmico
$O(1)$	tiempo de ejecución para un algoritmo en el peor caso
$O(\log n)$	

Ejercicios de autoevaluación

- 19.1** Complete los siguientes enunciados:
- Una aplicación de ordenamiento por selección debe requerir un tiempo aproximado de _____ veces más para ejecutarse en un arreglo de 128 elementos, en comparación con un arreglo de 32 elementos.
 - La eficiencia del ordenamiento por combinación es de _____.
- 19.2** ¿Qué aspecto clave de la búsqueda binaria y del ordenamiento por combinación es responsable de la parte logarítmica de sus respectivos valores Big O?
- 19.3** ¿En qué sentido es superior el ordenamiento por inserción al ordenamiento por combinación? ¿En qué sentido es superior el ordenamiento por combinación al ordenamiento por inserción?
- 19.4** En el texto decimos que, una vez que el ordenamiento por combinación divide el arreglo en dos subarreglos, después ordena estos dos subarreglos y los combina. ¿Por qué alguien podría quedar desconcertado al decir nosotros que “después ordena estos dos subarreglos”?

Respuestas a los ejercicios de autoevaluación

- 19.1** a) 16, ya que un algoritmo $O(n^2)$ requiere 16 veces más de tiempo para ordenar hasta cuatro veces más información.
b) $O(n \log n)$.
- 19.2** Ambos algoritmos incorporan la acción de “dividir a la mitad” (reducir algo de cierta forma a la mitad). La búsqueda binaria elimina del proceso una mitad del arreglo después de cada comparación. El ordenamiento por combinación divide el arreglo a la mitad, cada vez que se llama.
- 19.3** El ordenamiento por inserción es más fácil de comprender y de programar que el ordenamiento por combinación. El ordenamiento por combinación es mucho más eficiente [$O(n \log n)$] que el ordenamiento por inserción [$O(n^2)$].
- 19.4** En cierto sentido, en realidad no ordena estos dos subarreglos. Simplemente sigue dividiendo el arreglo original a la mitad, hasta que obtiene un subarreglo de un elemento, que desde luego, está ordenado. Despues construye los dos subarreglos originales al combinar estos arreglos de un elemento para formar subarreglos más grandes, los cuales se mezclan, y así en lo sucesivo.

Ejercicios

[Nota: la mayoría de los ejercicios que se muestran aquí son duplicados de los ejercicios de los capítulos 7 y 8. Incluimos estos ejercicios de nuevo como conveniencia para los lectores que estudian los algoritmos de búsqueda y ordenamiento en este capítulo].

19.5 (Ordenamiento de burbuja) Implemente el ordenamiento de burbuja, otra técnica de ordenamiento simple, pero ineficiente. Se le llama ordenamiento de burbuja o de hundimiento, debido a que los valores más pequeños van “subiendo como burbujas” gradualmente, hasta llegar a la parte superior del vector (es decir, hacia el primer elemento) como las burbujas de aire que se elevan en el agua, mientras que los valores más grandes se hunden en el fondo (final) del vector. Esta técnica utiliza ciclos anidados para realizar varias pasadas a través del vector. Cada pasada compara pares sucesivos de elementos. Si un par se encuentra en orden ascendente (o los valores son iguales), el ordenamiento de burbuja deja los valores como están. Si un par se encuentra en orden descendente, el ordenamiento de burbuja intercambia sus valores en el arreglo.

En la primera pasada se comparan los valores de los primeros dos elementos del vector, y se intercambian sus valores si es necesario. Despues se comparan los valores de los elementos segundo y tercero en el vector. Al final de esta pasada se comparan los valores de los últimos dos elementos en el arreglo y se intercambian, en caso de ser necesario. Despues de una pasada, el valor más grande estará en el último elemento. Despues de dos pasadas, los dos valores más grandes se encontrarán en los últimos dos elementos. Explique por qué el ordenamiento de burbuja es un algoritmo $O(n^2)$.

19.6 (Ordenamiento de burbuja mejorado) Realice las siguientes modificaciones simples para mejorar el rendimiento del ordenamiento de burbuja que desarrolló en el ejercicio 19.5:

- Despues de la primera pasada, se garantiza que el valor más grande estará en el elemento con la numeración más alta del vector; despues de la segunda pasada, los dos valores más altos estarán “acomodados”; y así en lo sucesivo. En vez de realizar nueve comparaciones (para un vector con 10 elementos) en cada pasada, modifique el ordenamiento de burbuja para que realice sólo las ocho comparaciones necesarias en la segunda pasada, siete en la tercera, y así en lo sucesivo.
- Los datos en el vector tal vez se encuentren ya en el orden apropiado, o casi apropiado, así que ¿para qué realizar nueve pasadas (en un elemento con 10 vectores) si basta con menos? Modifique el ordenamiento para comprobar al final de cada pasada si se han realizado intercambios. Si no se ha realizado ninguno, los datos ya deben estar en el orden apropiado, por lo que el programa debe terminar. Si se han realizado intercambios, por lo menos se necesita una pasada más.

19.7 (Ordenamiento de cubeta) Un ordenamiento de cubeta comienza con un vector unidimensional de enteros positivos que se deben ordenar, y un vector bidimensional de enteros, en el que las filas están indexadas de 0 a 9 y las columnas de 0 a $n - 1$, en donde n es el número de valores a ordenar. Cada fila del vector bidimensional se conoce como una *cubeta*. Escriba una clase llamada `OrdenamientoCubeta`, que contenga una función llamada `ordenar` y que opere de la siguiente manera:

- Coloque cada valor del vector unidimensional en una fila del vector de cubeta, con base en el dígito de las unidades (el de más a la derecha) del valor. Por ejemplo, el número 97 se coloca en la fila 7, el 3 se coloca en la fila 3 y el 100 se coloca en la fila 0. A este procedimiento se le llama *pasada de distribución*.
- Itere a través del vector de cubeta fila por fila, y copie los valores de vuelta al vector original. A este procedimiento se le llama *pasada de recopilación*. El nuevo orden de los valores anteriores en el vector unidimensional es 100, 3 y 97.
- Repita este proceso para cada posición de dígito subsiguiente (decenas, centenas, miles, etcétera).

En la segunda pasada (el dígito de las decenas) se coloca el 100 en la fila 0, el 3 en la fila 0 (ya que 3 no tiene dígito de decenas) y el 97 en la fila 9. Despues de la pasada de recopilación, el orden de los valores en el arreglo unidimensional es 100, 3 y 97. En la tercera pasada (dígito de las centenas), el 100 se coloca en la fila 1, el 3 en la fila 0 y el 97 en la fila 0 (despues del 3). Despues de esta última pasada de recopilación, el vector original se encuentra en orden.

Observe que el vector bidimensional de cubetas tiene 10 veces la longitud del vector entero que se está ordenando. Esta técnica de ordenamiento proporciona un mejor rendimiento que el ordenamiento de burbuja, pero requiere mucha más memoria; el ordenamiento de burbuja requiere espacio sólo para un elemento adicional de datos. Esta comparación es un ejemplo de la concesión entre espacio y tiempo: el ordenamiento de cubeta utiliza más memoria que el ordenamiento de burbuja, pero su rendimiento es mejor. Esta versión del ordenamiento de cubeta requiere copiar todos los datos de vuelta al vector original en cada pasada. Otra posibilidad es crear un segundo vector de cubeta bidimensional, e intercambiar en forma repetida los datos entre los dos vectores de cubeta.

19.8 (Búsqueda lineal recursiva) Modifique el ejercicio 7.33 para utilizar la función recursiva `busquedaLinealRecursiva` para realizar una búsqueda lineal en el vector. La función debe recibir la clave de búsqueda y el índice inicial como argumentos. Si se encuentra la clave de búsqueda, se devuelve su índice en el arreglo; en caso contrario, se devuelve -1. Cada llamada al método recursivo debe comprobar el valor de un elemento en el arreglo.

19.9 (Búsqueda binaria recursiva) Modifique la figura 19.3 para que utilice la función recursiva `busquedaBinariaRecursiva` para realizar una búsqueda binaria en el vector. La función debe recibir la clave de búsqueda, el índice inicial y el índice final como argumentos. Si se encuentra la clave de búsqueda, se devuelve su índice en el vector. Si no se encuentra, se devuelve -1.

19.10 (Quicksort) La técnica de ordenamiento recursiva llamada “quicksort” utiliza el siguiente algoritmo básico para un vector unidimensional de valores:

- Paso de particionamiento:* tomar el primer elemento del vector desordenado y determinar su ubicación final en el vector ordenado (es decir, todos los valores a la izquierda del elemento en el arreglo son menores que el elemento, y todos los valores a la derecha del elemento en el arreglo son mayores; a continuación le mostraremos cómo hacer esto). Ahora tenemos un valor en su ubicación apropiada y dos subvectores desordenados.
- Paso recursivo:* llevar a cabo el *paso de particionamiento* en cada subvector desordenado. Cada vez que se realiza el *paso de particionamiento* en un subvector, se coloca otro valor en su ubicación final en el vector ordenado, y se crean dos subvectores desordenados. Cuando un subvector consiste de un elemento, el valor de ese elemento se encuentra en su ubicación final (debido a que un vector de un elemento ya está ordenado).

El algoritmo básico parece lo bastante suficiente, pero ¿cómo determinamos la posición final del primer elemento de cada subvector? Como ejemplo, considere el siguiente conjunto de valores (el valor en negritas es el elemento de particionamiento; se colocará en su ubicación final en el vector ordenado):

37 2 6 4 89 8 10 12 68 45

Empezando desde el elemento de más a la derecha del vector, se compara cada elemento con **37** hasta que se encuentra un elemento menor que **37**; después se intercambian el **37** y ese elemento. El primer elemento menor que **37** es 12, por lo que se intercambian el **37** y el 12. El nuevo vector es

12 2 6 4 89 8 10 **37** 68 45

El elemento 12 está en cursivas, para indicar que se acaba de intercambiar con el **37**.

Empezando desde la parte izquierda del vector, pero con el elemento que está después de 12, se compara el valor de cada elemento con **37** hasta encontrar un elemento mayor que **37**; después se intercambian el **37** y ese elemento. El primer elemento mayor que **37** es 89, por lo que se intercambian el **37** y el 89. El nuevo vector es

12 2 6 4 **37** 8 10 89 68 45

Empezando desde la derecha, pero con el elemento antes del 89, se compara el valor de cada elemento con **37** hasta encontrar un elemento menor que **37**; después se intercambian el **37** y ese elemento. El valor del primer elemento menor que **37** es 10, por lo que se intercambian **37** y 10. El nuevo vector es

12 2 6 4 10 8 **37** 89 68 45

Empezando desde la izquierda, pero con el valor del elemento que está después de 10, se compara cada elemento con **37** hasta encontrar un elemento mayor que **37**; después se intercambian el **37** y ese elemento. No hay más elementos mayores que **37**, por lo que al comparar el **37** consigo mismo, sabemos que se ha colocado en su ubicación final en el vector ordenado. Cada valor a la izquierda de **37** es más pequeño, y cada valor a la derecha de **37** es más grande.

Una vez que se ha aplicado la partición en el vector anterior, hay dos subvectores desordenados. El subvector con valores menores que **37** contiene 12, 2, 6, 4, 10 y 8. El subvector con valores mayores que **37** contiene 89, 68 y 45. El ordenamiento continúa en forma recursiva, y ambos subvectores se partitionan de la misma forma que el vector original.

Con base en la anterior discusión, escriba la función recursiva `ayudanteQuicksort` para ordenar un vector entero unidimensional. La función debe recibir como argumentos un índice inicial y un índice final en el vector original que se está ordenando.

20



Estructuras de datos

*De muchas cosas
a las que estoy atado,
no he podido liberarme;
y muchas de las que me
liberé, han vuelto a mí.*

—Lee Wilson Dodd

*'¿Podría caminar un poco
más rápido?' dijo una
merluza a un caracol,
'hay una marsopa
acerándose mucho a
nosotros y está pisándome
la cola.'*

—Lewis Carroll

Siempre hay lugar en la cima.

—Daniel Webster

Empujen; sigan moviéndose.

—Thomas Morton

Daré vuelta a una nueva hoja.

—Miguel de Cervantes

OBJETIVOS

En este capítulo aprenderá a:

- Formar estructuras de datos enlazadas mediante el uso de apunadores, clases autorreferenciadas y recursividad.
- Crear y manipular estructuras dinámicas de datos como listas enlazadas, colas, pilas y árboles binarios.
- Usar árboles de búsqueda binaria, para la búsqueda y clasificación a alta velocidad
- Comprender varias aplicaciones importantes de las estructuras de datos enlazadas.
- Comprender cómo crear estructuras de datos reutilizables con plantillas de clases, herencia y composición.

- 20.1** Introducción
- 20.2** Clases autorreferenciadas
- 20.3** Asignación dinámica de memoria y estructuras de datos
- 20.4** Listas enlazadas
- 20.5** Pilas
- 20.6** Colas
- 20.7** Árboles
- 20.8** Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)
Sección especial: Construya su propio compilador

20.1 Introducción

En capítulos anteriores, hemos estudiado **estructuras de datos** de tamaño fijo, como los arreglos unidimensionales y de dos dimensiones. En este capítulo presentaremos las **estructuras de datos dinámicas**, que crecen y se reducen durante la ejecución. Las **listas enlazadas** son colecciones de elementos de datos “alineados en una fila” lógicamente; pueden insertarse y eliminarse elementos en cualquier parte de una lista enlazada. Las **pilas** son importantes en los compiladores y sistemas operativos; pueden insertarse y eliminarse elementos sólo en un extremo de una pila: su **parte superior**. Las **colas** representan líneas de espera; se insertan elementos en la parte final (conocida como **rabo**) de una cola y se eliminan elementos de su parte inicial (conocida como **cabeza**). Los **árboles binarios** facilitan la búsqueda y ordenamiento de los datos a alta velocidad, la eliminación eficiente de elementos de datos duplicados, la representación de directorios del sistema de archivos y la compilación de expresiones en lenguaje máquina. Estas estructuras de datos tienen muchas otras aplicaciones interesantes.

Hablaremos sobre varias estructuras de datos populares e importantes, e implementaremos programas para crearlas y manipularlas. Utilizaremos clases, plantillas de clases, herencia y composición para crear y empaquetar estas estructuras de datos, para reutilizarlas y darles mantenimiento.

El estudio de este capítulo es una sólida preparación para el capítulo 22, Biblioteca de plantillas estándar (STL, por sus siglas en inglés, Standard Template Library). La STL es una parte importante de la Biblioteca estándar de C++. Proporciona contenedores, iteradores para recorrer esos contenedores y algoritmos para procesar los elementos de esos contenedores. Aquí veremos que la STL ha tomado cada una de las estructuras de datos que describimos en este capítulo, y las ha empaquetado en clases de plantillas. El código de la STL está cuidadosamente escrito para ser portable, eficiente y extensible. Una vez que comprenda los principios y la construcción de las estructuras de datos como se presentan en este capítulo, el lector podrá hacer el mejor uso de las estructuras de datos pre-empaquetadas, iteradores y algoritmos en la STL, un conjunto a nivel mundial de componentes reutilizables.

Los ejemplos de este capítulo son programas prácticos que pueden utilizarse en cursos más avanzados y en aplicaciones industriales. Los programas emplean una extensa manipulación de apuntadores. Los ejercicios incluyen una vasta colección de aplicaciones útiles.

Le recomendamos que trate de realizar el proyecto principal descrito en la sección especial titulada *Construya su propio compilador*. Ha estado utilizando un compilador de C++ para traducir sus programas en lenguaje máquina y poder ejecutar estos programas en su computadora. En este proyecto creará su propio compilador. Este compilador leerá un archivo de instrucciones escritas en un lenguaje de alto nivel simple pero poderoso, similar a las primeras versiones del popular lenguaje BASIC. Su compilador traducirá estas instrucciones en un archivo de instrucciones de Lenguaje Máquina Simpletron (LMS); éste es el lenguaje que aprendió en la sección especial del capítulo 8, *Construya su propia computadora*. ¡Su programa Simulador de Simpletron ejecutará entonces el programa LMS producido por su compilador! Al implementar este proyecto mediante el uso de una metodología orientada a objetos, usted recibirá una maravillosa oportunidad para poner en práctica la mayor parte de lo que ha aprendido en este libro. La sección especial lo lleva cuidadosamente a través de las especificaciones del lenguaje de alto nivel, y describe los algoritmos que usted necesitará para convertir cada tipo de instrucción en lenguaje de alto nivel, a instrucciones de lenguaje máquina. Si disfruta de los retos, tal vez pueda tratar de realizar las diversas mejoras tanto al compilador como al Simulador Simpletron, las cuales se sugieren en los ejercicios.

20.2 Clases autorreferenciadas

Una clase autorreferenciada contiene miembro apuntador que apunta a otro objeto del mismo tipo de clase. Por ejemplo, la definición:

```
class Nodo
{
public:
    Nodo( int ); // constructor
    void establecerDatos( int ); // establece el miembro de datos
    int obtenerDatos() const; // obtiene el miembro de datos
    void establecerSiguientePtr( Nodo * ); // establece el apuntador al siguiente Nodo
    Nodo *obtenerSiguientePtr() const; // obtiene apuntador al siguiente Nodo
private:
    int datos; // datos almacenados en este Nodo
    Nodo *siguientePtr; // apuntador a otro objeto del mismo tipo
}; // fin de la clase Nodo
```

define un tipo llamado **Nodo**. Este tipo tiene dos datos miembro **private**: el miembro entero **datos** y el miembro apuntador **siguientePtr**. El miembro **siguientePtr** apunta a un objeto de tipo **Nodo**: otro objeto del mismo tipo que el que se está declarando aquí; es por ello que se utiliza el término “clase autorreferenciada”. El miembro **siguientePtr** es un **enlace**; es decir, **siguientePtr** puede “vincular” a un objeto de tipo **Nodo** con otro objeto del mismo tipo. El tipo **Nodo** también tiene cinco funciones miembro: un constructor que recibe un entero para inicializar datos miembro, una función **establecerDatos** para establecer el valor de los datos miembro, una función **obtenerDatos** para devolver el valor del miembro **datos**, una función **establecerSiguientePtr** para establecer el valor del miembro **siguientePtr** y una función **obtenerSiguientePtr** para devolver el valor del miembro **siguientePtr**.

Los objetos de clases autorreferenciadas se pueden enlazar entre sí para formar estructuras de datos útiles como listas, colas, pilas y árboles. En la figura 20.1 se muestran dos objetos de una clase autorreferenciada, enlazados entre sí para formar una lista. Observe que se coloca una barra diagonal inversa [que representa un apuntador nulo(0)] en el miembro de enlace del segundo objeto de la clase autorreferenciada para indicar que el enlace no apunta a otro objeto. La barra diagonal inversa es ilustrativa; no corresponde al carácter de barra diagonal inversa en C++. Por lo general, un apuntador nulo indica el final de una estructura de datos tal como el carácter nulo indica el final de una cadena.

Error común de programación 20.1



Si no se establece el enlace en el último nodo de una estructura de datos enlazada con el valor nulo(0), se produce un error lógico (posiblemente fatal).

20.3 Asignación dinámica de memoria y estructuras de datos

Para crear y mantener estructuras dinámicas de datos se requiere la asignación dinámica de memoria, la cual permite que un programa obtenga más memoria en tiempo de ejecución, para almacenar nuevos nodos. Cuando el programa ya no necesita la memoria, ésta se puede liberar para que se pueda reutilizar al asignar otros objetos en el futuro. El límite para la asignación dinámica de memoria puede ser tan grande como la cantidad de memoria física disponible en la computadora, o la cantidad de memoria virtual disponible en un sistema con memoria virtual. A menudo, los límites son mucho más pequeños ya que la memoria disponible de la computadora debe compartirse entre muchos programas.

El operador **new** recibe como argumento el tipo del objeto que se va a asignar en forma dinámica y devuelve un apuntador a un objeto de ese tipo. Por ejemplo, la instrucción

```
Nodo *nuevoPtr = new Nodo( 10 ); // crea un Nodo con el valor 10
```

asigna **sizeof(Nodo)** bytes, ejecuta el constructor de **Nodo** y asigna la dirección del nuevo **Nodo** a **nuevoPtr**. Si no hay memoria disponible, **new** lanza una expresión **bad_alloc**. El valor 10 se pasa al constructor de **Nodo**, el cual inicializa el miembro de datos del **Nodo** con 10.



Figura 20.1 | Dos objetos de una clase autorreferenciada enlazados entre sí.

El operador `delete` ejecuta el destructor de `Nodo` y desasigna la memoria asignada con `new`; esta memoria se devuelve al sistema, de manera que se pueda reasignar en el futuro. Para liberar la memoria asignada en forma dinámica por la instrucción `new` anterior, utilice la instrucción

```
delete nuevoPtr;
```

Observe que `nuevoPtr` en sí no se elimina; en vez de ello, se elimina el espacio al que apunta `nuevoPtr`. Si el apuntador `nuevoPtr` tiene el valor 0 (apuntador nulo), la instrucción anterior no tiene efecto. No es un error eliminar mediante `delete` un apuntador nulo.

Las siguientes secciones hablan sobre listas, pilas, colas y árboles. Las estructuras de datos que presentaremos en este capítulo se crean y mantienen mediante la asignación dinámica de memoria, las clases autorreferenciadas, las plantillas de clases y las plantillas de funciones.

20.4 Listas enlazadas

Una lista enlazada es una colección lineal de objetos de una clase autorreferenciada, conocidos como **nodos**, que están conectados por **enlaces de apuntador**; es por ello que se utiliza el término lista “enlazada”. Un programa accede a una lista enlazada mediante un apuntador al primer nodo en la lista. El programa accede a cada nodo subsiguiente a través del miembro apuntador de enlace almacenado en el nodo anterior. Por convención, el apuntador de enlace en el último nodo de una lista se establece en el valor nulo (0) para marcar el final de la lista. Los datos se almacenan en forma dinámica en una lista enlazada; se crea cada nodo según sea necesario. Un nodo puede contener datos de cualquier tipo, incluyendo objetos de otras clases. Si los nodos contienen apuntadores de clase base a objetos de clase base y objetos de clase derivada relacionados por la herencia, podemos tener una lista enlazada de dichos nodos y procesarlos en forma polimórfica mediante llamadas a funciones `virtual`. Las pilas y las colas son también **estructuras de datos lineales** y, como veremos, son versiones restringidas de las listas enlazadas. Los árboles son **estructuras de datos no lineales**.

Pueden almacenarse listas de datos en los arreglos, pero las listas enlazadas ofrecen varias ventajas. Una lista enlazada es apropiada cuando el número de elementos de datos que se van a representar en un momento dado es impredecible. Las listas enlazadas son dinámicas, por lo que la longitud de una lista puede incrementarse o reducirse, según sea necesario. Sin embargo, el tamaño de un arreglo “convencional” en C++ no puede alterarse, ya que el tamaño del arreglo se fija en tiempo de compilación. Los arreglos “convencionales” pueden llenarse. Las listas enlazadas se llenan sólo cuando el sistema no tiene suficiente memoria para satisfacer las peticiones de asignación dinámica de almacenamiento.

Tip de rendimiento 20.1



Un arreglo puede declararse de manera que contenga más elementos que el número de elementos esperados, pero esto puede desperdiciar memoria. Las listas enlazadas pueden proporcionar una mejor utilización de la memoria en estas situaciones. Las listas enlazadas permiten al programa adaptarse en tiempo de ejecución. Observe que la plantilla de clase `vector` (presentada en la sección 7.11) implementa una estructura de datos basada en arreglo que puede cambiar su tamaño en forma dinámica.

Para mantener las listas enlazadas en orden, se inserta cada nuevo elemento en el punto apropiado en la lista. Los elementos existentes de una lista no necesitan moverse.

Tip de rendimiento 20.2



La inserción y la eliminación en un arreglo ordenado puede llevar mucho tiempo; todos los elementos que van después del elemento insertado o eliminado deben desplazarse apropiadamente. Una lista enlazada permite operaciones de inserción eficientes en cualquier parte de la lista.

Tip de rendimiento 20.3



Los elementos de un arreglo se almacenan en forma contigua en la memoria. Esto permite un acceso inmediato a cualquier elemento del arreglo, ya que la dirección de cualquier elemento puede calcularse directamente con base en su posición relativa al inicio del arreglo. Las listas enlazadas no permiten dicho “acceso directo” inmediato a sus elementos. Por lo tanto, puede ser más trabajoso acceder a los elementos individuales en una lista enlazada que acceder a los elementos individuales en un arreglo. La selección de una estructura de datos se basa comúnmente en el rendimiento de las operaciones específicas utilizadas por un programa, y el orden en el que se mantienen los elementos de datos en la estructura de datos. Por ejemplo, generalmente es más eficiente insertar un elemento en una lista enlazada que en un arreglo ordenado.

Por lo general, los nodos de las listas enlazadas no se almacenan contiguamente en memoria. Sin embargo, en sentido lógico los nodos de una lista enlazada parecen estar contiguos. La figura 20.2 muestra una lista enlazada con varios nodos.

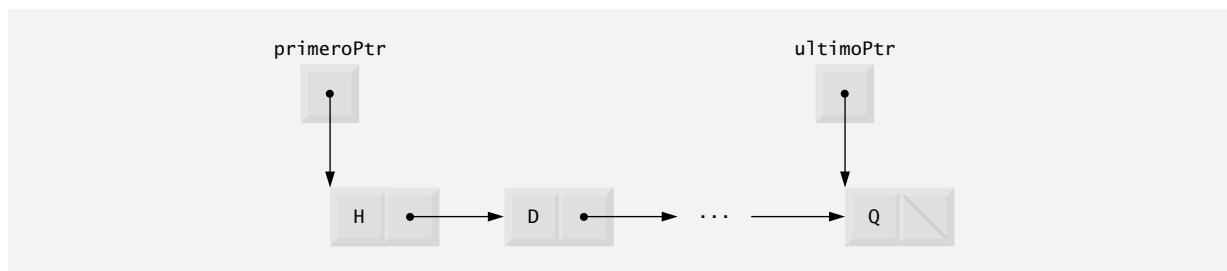


Figura 20.2 | Representación gráfica de una lista.



Tip de rendimiento 20.4

El uso de la asignación dinámica de memoria (en vez de arreglos de tamaño fijo) para las estructuras de datos que aumentan y reducen su tamaño en tiempo de ejecución puede ahorrar memoria. Sin embargo, hay que tener en cuenta que los apuntadores ocupan espacio, y que la asignación dinámica de memoria incurre en la sobrecarga de las llamadas a funciones.

Implementación de una lista enlazada

El programa de las figuras 20.3 a 20.5 utiliza una plantilla de clase `Lista` (en el capítulo 14 obtendrá información sobre las plantillas de clase) para manipular una lista de valores enteros y una lista de valores de punto flotante. El programa controlador (figura 20.5) proporciona cinco opciones: 1) Insertar un valor al inicio de la lista, 2) insertar un valor al final de la lista, 3) eliminar un valor del inicio de la lista, 4) eliminar un valor del final de la lista y 5) terminar el procesamiento de la lista. A continuación se muestra una discusión detallada del programa. En el ejercicio 20.20 se pedirá al lector que implemente una función recursiva que imprima una lista enlazada al revés, y en el ejercicio 20.21 se pedirá al lector que implemente una función recursiva que busque un elemento de datos específico en una lista enlazada.

El programa utiliza las plantillas de clase `NodoLista` (figura 20.3) y `Lista` (figura 20.4). En cada objeto `Lista` hay encapsulada una lista enlazada de objetos `NodoLista`. La plantilla de clase `NodoLista` (figura 20.3) contiene los datos miembro `private datos` y `siguientePtr` (líneas 19 y 20), un constructor para inicializar estos miembros y una función `obtenerDatos` para devolver los datos en un nodo. El dato miembro `datos` almacena un valor de tipo `TIPONODO`, el parámetro de tipo que se pasa a la plantilla de clase. El miembro `siguientePtr` almacena un apuntador al siguiente objeto `NodoLista` en la lista enlazada. Observe que en la línea 13 de la definición de la plantilla de clase `NodoLista` se declara la clase `Lista< TIPONODO >` como una función `friend`. Esto hace a todas las funciones miembro de una especialización dada de la plantilla de clase `Lista` amigas de la especialización correspondiente de la plantilla de clase `NodoLista`, de manera que pueden acceder a los miembros de datos `private` de los objetos `NodoLista` de ese tipo. Como el parámetro `TIPONODO` de la plantilla `NodoLista` se utiliza como el argumento de plantilla para `Lista` en la declaración `friend`, los objetos `NodoLista` especializados con un tipo específico se pueden procesar sólo mediante una `Lista` especializada con el mismo tipo (por ejemplo, una `Lista` de valores `int` administra objetos `NodoLista` que almacenan valores `int`).

```

1 // Fig. 20.3: NodoLista.h
2 // Definición de la plantilla de clase NodoLista.
3 #ifndef NODOLISTA_H
4 #define NODOLISTA_H
5
6 // declaración anticipada de la clase Lista, requerida para anunciar que la clase
7 // Lista existe y poder utilizarla en la declaración friend de la línea 13
8 template< typename TIPONODO > class Lista;
9
10 template< typename TIPONODO >
11 class NodoLista
12 {
13     friend class Lista< TIPONODO >; // hace de Lista una amiga
14 }
```

Figura 20.3 | Definición de la plantilla de clase `NodoLista`. (Parte 1 de 2).

```

15 public:
16     NodoLista( const TIPONODO & ); // constructor
17     TIPONODO obtenerDatos() const; // devuelve los datos en el nodo
18 private:
19     TIPONODO datos; // datos
20     NodoLista< TIPONODO > *siguientePtr; // siguiente nodo en la lista
21 }; // fin de la clase NodoLista
22
23 // constructor
24 template< typename TIPONODO >
25 NodoLista< TIPONODO >::NodoLista( const TIPONODO &info )
26     : datos( info ), siguientePtr( 0 )
27 {
28     // cuerpo vacío
29 } // fin del constructor de NodoLista
30
31 // devuelve una copia de los datos en el nodo
32 template< typename TIPONODO >
33 TIPONODO NodoLista< TIPONODO >::obtenerDatos() const
34 {
35     return datos;
36 } // fin de la función obtenerDatos
37
38 #endif

```

Figura 20.3 | Definición de la plantilla de clase `NodoLista`. (Parte 2 de 2).

```

1 // Fig. 20.4: Lista.h
2 // Definición de la plantilla de clase Lista.
3 #ifndef LISTA_H
4 #define LISTA_H
5
6 #include <iostream>
7 using std::cout;
8
9 #include "NodoLista.h" // definición de la clase NodoLista
10
11 template< typename TIPONODO >
12 class Lista
13 {
14 public:
15     Lista(); // constructor
16     ~Lista(); // destructor
17     void insertarAlFrente( const TIPONODO & );
18     void insertarAlFinal( const TIPONODO & );
19     bool eliminarDelFrente( TIPONODO & );
20     bool eliminarDelFinal( TIPONODO & );
21     bool estaVacia() const;
22     void imprimir() const;
23 private:
24     NodoLista< TIPONODO > *primeroPtr; // apuntador al primer nodo
25     NodoLista< TIPONODO > *ultimoPtr; // apuntador al último nodo
26
27     // función utilitaria para asignar un nuevo nodo
28     NodoLista< TIPONODO > *obtenerNuevoNodo( const TIPONODO & );
29 }; // fin de la clase Lista
30
31 // constructor predeterminado
32 template< typename TIPONODO >
33 Lista< TIPONODO >::Lista()

```

Figura 20.4 | Definición de la plantilla de clase `Lista`. (Parte I de 4).

```

34     : primeroPtr( 0 ), ultimoPtr( 0 )
35 {
36     // cuerpo vacío
37 } // fin del constructor de Lista
38
39 // destructor
40 template< typename TIPONODO >
41 Lista< TIPONODO >::~Lista()
42 {
43     if ( !estaVacia() ) // la Lista no está vacía
44     {
45         cout << "Destruyendo nodos ... \n";
46
47         NodoLista< TIPONODO > *actualPtr = primeroPtr;
48         NodoLista< TIPONODO > *tempPtr;
49
50         while ( actualPtr != 0 ) // elimina los nodos restantes
51     {
52             tempPtr = actualPtr;
53             cout << tempPtr->datos << '\n';
54             actualPtr = actualPtr->siguientePtr;
55             delete tempPtr;
56         } // fin de while
57     } // fin de if
58
59     cout << "Se destruyeron todos los nodos\n\n";
60 } // fin del destructor de Lista
61
62 // inserta un nodo al frente de la lista
63 template< typename TIPONODO >
64 void Lista< TIPONODO >::insertarAlFrente( const TIPONODO &valor )
65 {
66     NodoLista< TIPONODO > *nuevoPtr = obtenerNuevoNodo( valor ); // nuevo nodo
67
68     if ( estaVacia() ) // la Lista está vacía
69         primeroPtr = ultimoPtr = nuevoPtr; // la nueva lista sólo tiene un nodo
70     else // la Lista no está vacía
71     {
72         nuevoPtr->siguientePtr = primeroPtr; // apunta el nuevo nodo al nodo que antes era
73         el primero
74         primeroPtr = nuevoPtr; // orienta primeroPtr hacia el nuevo nodo
75     } // fin de else
76 } // fin de la función insertarAlFrente
77
78 // inserta un nodo al final de la lista
79 template< typename TIPONODO >
80 void Lista< TIPONODO >::insertarAlFinal( const TIPONODO &valor )
81 {
82     NodoLista< TIPONODO > *nuevoPtr = obtenerNuevoNodo( valor ); // nuevo nodo
83
84     if ( estaVacia() ) // la Lista está vacía
85         primeroPtr = ultimoPtr = nuevoPtr; // la nueva lista sólo tiene un nodo
86     else // la Lista no está vacía
87     {
88         ultimoPtr->siguientePtr = nuevoPtr; // actualiza el nodo que antes era el último
89         ultimoPtr = nuevoPtr; // nuevo último nodo
90     } // fin de else
91 } // fin de la función insertarAlFinal
92
93 // elimina un nodo de la parte frontal de la lista
94 template< typename TIPONODO >

```

Figura 20.4 | Definición de la plantilla de clase Lista. (Parte 2 de 4).

```

94  bool Lista< TIPONODO >::eliminarDelFrente( TIPONODO &valor )
95  {
96      if ( estaVacia() ) // la Lista está vacía
97          return false; // la eliminación no tuvo éxito
98      else
99      {
100         NodoLista< TIPONODO > *tempPtr = primeroPtr; // contiene tempPtr a eliminar
101
102         if ( primeroPtr == ultimoPtr )
103             primeroPtr = ultimoPtr = 0; // no hay nodos después de la eliminación
104         else
105             primeroPtr = primeroPtr->siguientePtr; // apunta al nodo que antes era el segundo
106
107         valor = tempPtr->datos; // devuelve los datos que se van a eliminar
108         delete tempPtr; // reclama el nodo que antes era el primero
109         return true; // la eliminación tuvo éxito
110     } // fin de else
111 } // fin de la función eliminarDelFrente
112
113 // elimina un nodo de la parte final de la lista
114 template< typename TIPONODO >
115 bool Lista< TIPONODO >::eliminarDelFinal( TIPONODO &valor )
116 {
117     if ( estaVacia() ) // la Lista está vacía
118         return false; // la eliminación no tuvo éxito
119     else
120     {
121         NodoLista< TIPONODO > *tempPtr = ultimoPtr; // contiene tempPtr a eliminar
122
123         if ( primeroPtr == ultimoPtr ) // la Lista tiene un elemento
124             primeroPtr = ultimoPtr = 0; // no hay nodos después de la eliminación
125         else
126         {
127             NodoLista< TIPONODO > *actualPtr = primeroPtr;
128
129             // localiza el penúltimo elemento
130             while ( actualPtr->siguientePtr != ultimoPtr )
131                 actualPtr = actualPtr->siguientePtr; // se desplaza al siguiente nodo
132
133             ultimoPtr = actualPtr; // elimina el último nodo
134             actualPtr->siguientePtr = 0; // ahora éste es el último nodo
135         } // fin de else
136
137         valor = tempPtr->datos; // devuelve el valor del nodo que antes era el último
138         delete tempPtr; // reclama el nodo que antes era el último
139         return true; // la eliminación tuvo éxito
140     } // fin de else
141 } // fin de la función eliminarDelFinal
142
143 // ¿está la Lista vacía?
144 template< typename TIPONODO >
145 bool Lista< TIPONODO >::estaVacia() const
146 {
147     return primeroPtr == 0;
148 } // fin de la función estaVacia
149
150 // devuelve el apuntador al nodo recién asignado
151 template< typename TIPONODO >
152 NodoLista< TIPONODO > *Lista< TIPONODO >::obtenerNuevoNodo(
153     const TIPONODO &valor )
154 {

```

Figura 20.4 | Definición de la plantilla de clase `Lista`. (Parte 3 de 4).

```

155     return new NodoLista< TIPONODO >( valor );
156 } // fin de la función obtenerNuevoNodo
157
158 // muestra el contenido de la Lista
159 template< typename TIPONODO >
160 void Lista< TIPONODO >::imprimir() const
161 {
162     if ( estaVacia() ) // la Lista está vacía
163     {
164         cout << "La lista esta vacia\n\n";
165         return;
166     } // fin de if
167
168     NodoLista< TIPONODO > *actualPtr = primeroPtr;
169
170     cout << "La lista es: ";
171
172     while ( actualPtr != 0 ) // obtiene los datos del elemento
173     {
174         cout << actualPtr->datos << ' ';
175         actualPtr = actualPtr->siguientePtr;
176     } // fin de while
177
178     cout << "\n\n";
179 } // fin de la función imprimir
180
181 #endif

```

Figura 20.4 | Definición de la plantilla de clase Lista. (Parte 4 de 4).

```

1 // Fig. 20.5: Fig20_05.cpp
2 // Programa de prueba de la clase Lista.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 #include <string>
9 using std::string;
10
11 #include "Lista.h" // definición de la clase Lista
12
13 // función para evaluar una Lista
14 template< typename T >
15 void probarLista( Lista< T > &objetoLista, const string &nombreTipo )
16 {
17     cout << "Prueba de una Lista de valores tipo " << nombreTipo << "\n";
18     instrucciones(); // muestra las instrucciones
19
20     int opcion; // almacena la opción del usuario
21     T valor; // almacena el valor de entrada
22
23     do // realiza las acciones seleccionadas por el usuario
24     {
25         cout << "? ";
26         cin >> opcion;
27
28         switch ( opcion )
29         {
30             case 1: // inserta al principio

```

Figura 20.5 | Manipulación de una lista enlazada. (Parte 1 de 4).

```

31         cout << "Escriba " << nombreTipo << ": ";
32         cin >> valor;
33         objetoLista.insertarAlFrente( valor );
34         objetoLista.imprimir();
35         break;
36     case 2: // inserta al final
37         cout << "Escriba " << nombreTipo << ": ";
38         cin >> valor;
39         objetoLista.insertarAlFinal( valor );
40         objetoLista.imprimir();
41         break;
42     case 3: // elimina del principio
43         if ( objetoLista.eliminarDelFrente( valor ) )
44             cout << valor << " se elimino de la lista\n";
45
46         objetoLista.imprimir();
47         break;
48     case 4: // elimina del final
49         if ( objetoLista.eliminarDelFinal( valor ) )
50             cout << valor << " se elimino de la lista\n";
51
52         objetoLista.imprimir();
53         break;
54     } // fin de switch
55 } while ( opcion != 5 ); // fin de do...while
56
57 cout << "Fin de la prueba de la lista\n\n";
58 } // fin de la función probarLista
59
60 // muestra las instrucciones del programa al usuario
61 void instrucciones()
62 {
63     cout << "Escriba una de las siguientes opciones:\n"
64         << " 1 para insertar al principio de la lista\n"
65         << " 2 para insertar al final de la lista\n"
66         << " 3 para eliminar del principio de la lista\n"
67         << " 4 para eliminar del final de la lista\n"
68         << " 5 para terminar el procesamiento de la lista\n";
69 } // fin de la función instrucciones
70
71 int main()
72 {
73     // prueba Lista de valores int
74     Lista< int > listaEnteros;
75     probarLista( listaEnteros, "entero" );
76
77     // prueba Lista de valores double
78     Lista< double > listaDouble;
79     probarLista( listaDouble, "double" );
80     return 0;
81 } // fin de main

```

Prueba de una Lista de valores tipo entero
 Escriba una de las siguientes opciones:
 1 para insertar al principio de la lista
 2 para insertar al final de la lista
 3 para eliminar del principio de la lista
 4 para eliminar del final de la lista
 5 para terminar el procesamiento de la lista
 ? 1
 Escriba entero: 1
 La lista es: 1

Figura 20.5 | Manipulación de una lista enlazada. (Parte 2 de 4).

```

? 1
Escriba entero: 2
La lista es: 2 1

? 2
Escriba entero: 3
La lista es: 2 1 3

? 2
Escriba entero: 4
La lista es: 2 1 3 4

? 3
2 se elimino de la lista
La lista es: 1 3 4

? 3
1 se elimino de la lista
La lista es: 3 4

? 4
4 se elimino de la lista
La lista es: 3

? 4
3 se elimino de la lista
La lista esta vacia

? 5
Fin de la prueba de la lista

Prueba de una Lista de valores tipo double
Escriba una de las siguientes opciones:
 1 para insertar al principio de la lista
 2 para insertar al final de la lista
 3 para eliminar del principio de la lista
 4 para eliminar del final de la lista
 5 para terminar el procesamiento de la lista
? 1
Escriba double: 1.1
La lista es: 1.1

? 1
Escriba double: 2.2
La lista es: 2.2 1.1

? 2
Escriba double: 3.3
La lista es: 2.2 1.1 3.3

? 2
Escriba double: 4.4
La lista es: 2.2 1.1 3.3 4.4

? 3
2.2 se elimino de la lista
La lista es: 1.1 3.3 4.4

? 3
1.1 se elimino de la lista
La lista es: 3.3 4.4

```

(continúa...)

Figura 20.5 | Manipulación de una lista enlazada. (Parte 3 de 4).

```
? 4
4.4 se elimino de la lista
La lista es: 3.3

? 4
3.3 se elimino de la lista
La lista esta vacia

? 5
Fin de la prueba de la lista

Se destruyeron todos los nodos

Se destruyeron todos los nodos
```

Figura 20.5 | Manipulación de una lista enlazada. (Parte 4 de 4).

En las líneas 24 y 25 de la plantilla de clase `Lista` (figura 20.4) se declaran los datos miembro `private primeroPtr` (un apuntador al primer `NodoLista` en una `Lista`) y `ultimoPtr` (un apuntador al último `NodoLista` en una `Lista`). El constructor predeterminado (líneas 32 a 37) inicializa ambos apuntadores con 0 (nulo). El destructor (líneas 40 a 60) asegura que todos los objetos `NodoLista` en un objeto `Lista` se destruyan cuando se destruya ese objeto `Lista`. Las funciones primarias de `Lista` son `insertarAlFrente` (líneas 63 a 75), `insertarAlFinal` (líneas 78 a 90), `eliminarDelFrente` (líneas 93 a 111) y `eliminarDelFinal` (líneas 114 a 141).

La función `estaVacia` (líneas 144 a 148) se conoce como función predicado: no altera la `Lista`; en vez de ello, determina si está vacía (es decir, que el apuntador al primer nodo de la `Lista` es nulo). Si la `Lista` está vacía se devuelve `true`; en caso contrario se devuelve `false`. La función `imprimir` (líneas 159 a 179) muestra el contenido de la `Lista`. La función utilitaria `obtenerNuevoNodo` (líneas 151 a 156) devuelve un objeto `NodoLista` asignado en forma dinámica. Esta función se llama desde las funciones `insertarAlFrente` e `insertarAlFinal`.



Tip para prevenir errores 20.1

Asigne el valor nulo (0) al miembro de enlace de un nuevo nodo. Los apuntadores se deben inicializar antes de poder utilizarlos.

El programa controlador (figura 20.5) utiliza la plantilla de función `probarLista` para permitir al usuario manipular objetos de la clase `Lista`. En las líneas 74 y 78 se crean objetos `Lista` para los tipos `int` y `double`, respectivamente. En las líneas 75 y 79 se invoca la plantilla de función `probarLista` con tres objetos `Lista`.

La función miembro `insertarAlFrente`

En las siguientes páginas hablaremos con detalle sobre cada una de las funciones miembro de la clase `Lista`. La función `insertarAlFrente` (figura 20.4, líneas 63 a 75) coloca un nuevo nodo al frente de la lista. Esta función consiste en varios pasos:

1. Llama a la función `obtenerNuevoNodo` (línea 66) y le pasa un `valor`, que es una referencia constante al valor del nodo que se va a insertar.
2. La función `obtenerNuevoNodo` (líneas 151 a 156) utiliza el operador `new` para crear un nuevo nodo en la lista y devuelve un apuntador a este nodo recién asignado, que se asigna a `nuevoPtr` en `insertarAlFrente` (línea 66).
3. Si la lista está vacía (línea 68), entonces tanto `primeroPtr` como `ultimoPtr` se establecen en `nuevoPtr` (línea 69).
4. Si la lista no está vacía (línea 70), entonces el nodo al que apunta `nuevoPtr` se pasa a la lista cuando se copia `primeroPtr` a `nuevoPtr->siguientePtr` (línea 72), de manera que el nuevo nodo apunta a lo que solía ser el primer nodo de la lista, y cuando se copia `nuevoPtr` a `primeroPtr` (línea 73), de manera que `primeroPtr` apunta ahora al nuevo primer nodo de la lista.

En la figura 20.6 se ilustra la función `insertarAlFrente`. La parte (a) de la figura muestra la lista y el nuevo nodo antes de la operación `insertarAlFrente`. Las flechas punteadas en la parte (b) ilustran el *paso 4* de la operación `insertarAlFrente`, que permite que el nodo que contiene 12 se convierta en el nuevo frente de la lista.

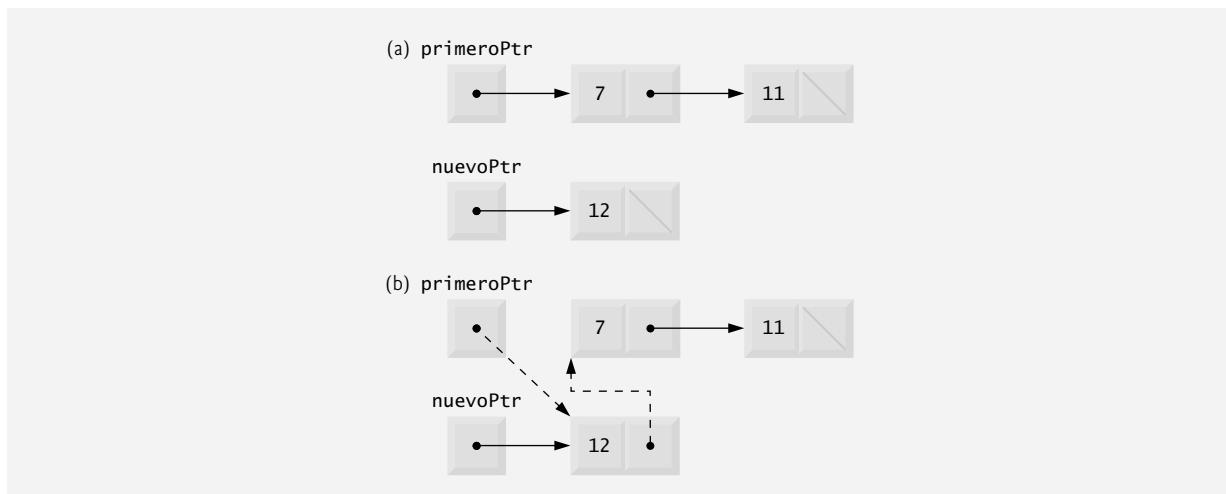


Figura 20.6 | Representación gráfica de la operación `insertarAlFrente`.

La función miembro `insertarAlFinal`

La función `insertarAlFinal` (figura 20.4, líneas 78 a 90) coloca un nuevo nodo al final de la lista. Esta función consiste en varios pasos:

1. Llama a la función `obtenerNuevoNodo` (línea 81) y le pasa un `valor`, que es una referencia constante al valor del nodo que se va a insertar.
2. La función `obtenerNuevoNodo` (líneas 151 a 156) utiliza el operador `new` para crear un nuevo nodo de la lista y devuelve un apuntador a este nodo recién asignado, el cual se asigna a `nuevoPtr` en `insertarAlFinal` (línea 81).
3. Si la lista está vacía (línea 83), entonces tanto `primeroPtr` como `ultimoPtr` se establecen con `nuevoPtr` (línea 84).
4. Si la lista no está vacía (línea 85), entonces el nodo al que apunta `nuevoPtr` se pasa a la lista cuando se copia `nuevoPtr` a `ultimoPtr -> siguientePtr` (línea 87), de manera que lo que solía ser el último nodo de la lista apunta al nuevo nodo, y cuando se copia `nuevoPtr` a `ultimoPtr` (línea 88), de manera que `ultimoPtr` ahora apunta al nuevo último nodo de la lista.

En la figura 20.7 se ilustra una operación `insertarAlFrente`. La parte (a) de la figura muestra la lista y el nuevo nodo antes de la operación. Las flechas punteadas en la parte (b) ilustran el *paso 4* de la función `insertarAlFrente` que permite agregar un nuevo nodo al final de una lista que no está vacía.

La función miembro `eliminarDelFrente`

La función `eliminarDelFrente` (figura 20.4, líneas 93 a 111) elimina del nodo frontal de la lista y copia el valor del nodo al parámetro de referencia. La función devuelve `false` si se hace un intento por eliminar un nodo de una lista vacía (líneas 96 y 97) y devuelve `true` si la eliminación tuvo éxito. Esta función consiste en varios pasos:

1. Asigna a `tempPtr` la dirección a la que apunta `primeroPtr` (línea 100). En un momento dado, `tempPtr` se utilizará para eliminar el nodo que se va a quitar.
2. Si `primeroPtr` es igual a `ultimoPtr` (línea 102), es decir, si la lista sólo tiene un elemento antes del intento de remoción, entonces se establecen `primeroPtr` y `ultimoPtr` con cero (línea 103) para separar ese nodo de la lista (dejando la lista vacía).
3. Si la lista tiene más de un nodo antes de la eliminación, entonces `ultimoPtr` se deja como está y se establece `primeroPtr` con `primeroPtr -> siguientePtr` (línea 105); es decir, se modifica `primeroPtr` para que apunte a lo que solía ser el segundo nodo anterior a la eliminación (y que ahora es el nuevo primer nodo).
4. Una vez que estas manipulaciones de apuntadores estén completas, se copia al parámetro de referencia `valor` el miembro `datos` del nodo que se va a quitar (línea 107).
5. Ahora elimina (`delete`) el nodo al que apunta `tempPtr` (línea 108).

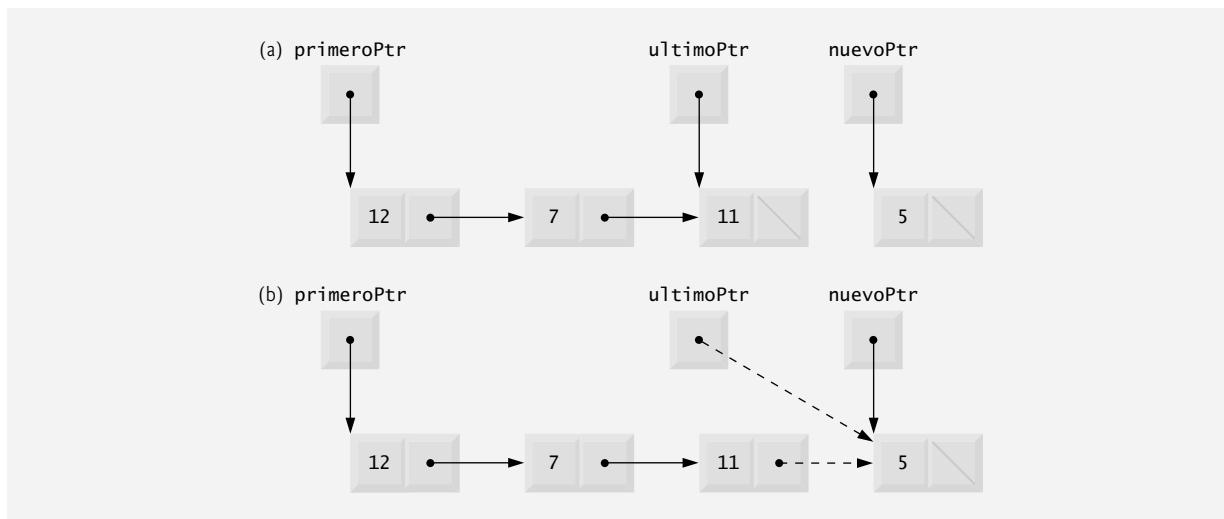


Figura 20.7 | Representación gráfica de la operación `insertarAlFinal`.

6. Devuelve `true`, indicando que la eliminación tuvo éxito (línea 109).

En la figura 20.8 se ilustra la función `eliminarDelFrente`. La parte (a) muestra la lista antes de la operación de eliminación. La parte (b) muestra las manipulaciones de apuntadores actuales para eliminar el nodo frontal de una lista no vacía.

La función miembro `eliminarDelFinal`

La función `eliminarDelFinal` (figura 20.4, líneas 114 a 141) elimina el nodo posterior de la lista y copia el valor del nodo al parámetro de referencia. La función devuelve `false` si se hace un intento por eliminar un nodo de una lista vacía (líneas 117 y 118), y devuelve `true` si la eliminación tiene éxito. Esta función consiste en varios pasos:

1. Asigna a `tempPtr` la dirección a la que apunta `ultimoPtr` (línea 121). En cierto momento, se utilizará `tempPtr` para eliminar el nodo que se va a quitar.

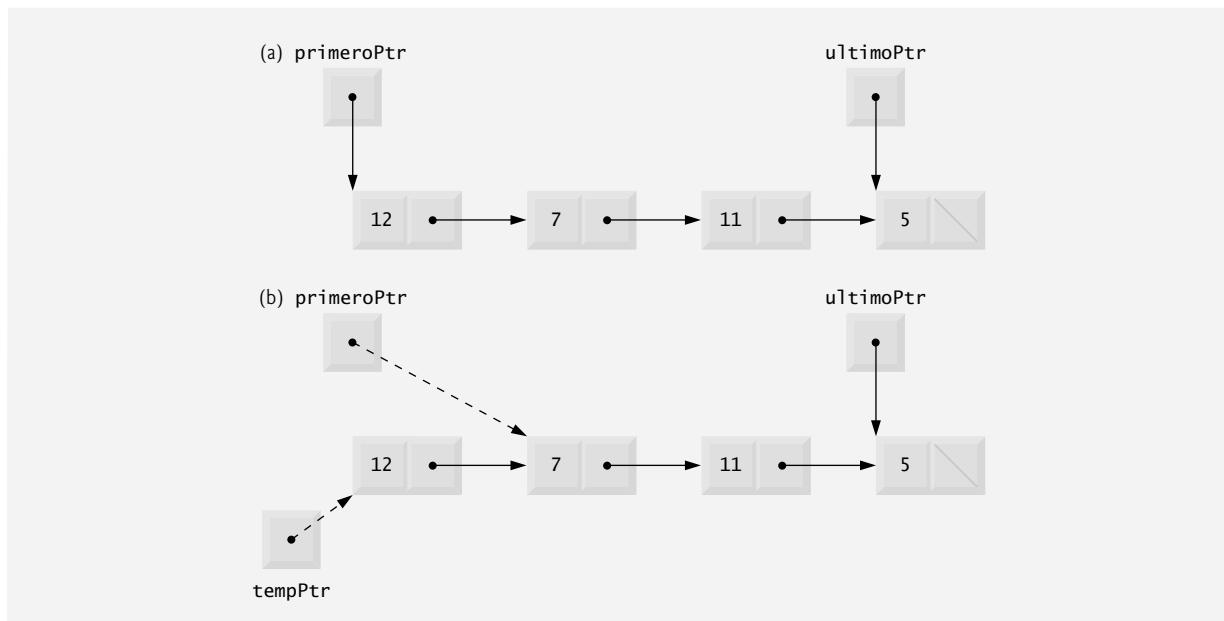


Figura 20.8 | Representación gráfica de la operación `eliminarDelFrente`.

2. Si `primeroPtr` es igual a `ultimoPtr` (línea 123), es decir, si la lista sólo tiene un elemento antes del intento de remoción, entonces se establecen `primeroPtr` y `ultimoPtr` con cero (línea 124) para sacar ese nodo de la lista (dejándola vacía).
3. Si la lista tiene más de un nodo antes de la eliminación, entonces se asigna a `actualPtr` la dirección a la que apunta `primeroPtr` (línea 127), para prepararse para “recorrer la lista”.
4. Ahora se “recorre la lista” con `actualPtr` hasta que apunte al nodo antes del último. Este nodo se convertirá en el último nodo, una vez que se complete la operación de eliminación. Esto se hace con un ciclo `while` (líneas 130 y 131) que sigue reemplazando a `actualPtr` por `actualPtr->siguientePtr`, mientras que `actualPtr->siguientePtr` no sea `ultimoPtr`.
5. Asigna `ultimoPtr` a la dirección a la que apunta `actualPtr` (línea 133) para sacar el nodo final de la lista,
6. Establece `actualPtr->siguientePtr` a cero (línea 134) en el nuevo último nodo de la lista.
7. Una vez que están completas todas las manipulaciones de apuntadores, se copia al parámetro referencia `valor` el miembro `datos` del nodo que se va a eliminar (línea 137).
8. Ahora se elimina (`delete`) el nodo al que apunta `tempPtr` (línea 138).
9. Devuelve `true` (línea 139), indicando que la eliminación tuvo éxito.

En la figura 20.9 se ilustra `eliminarDelFinal`. La parte (a) de la figura muestra la lista antes de la operación de eliminación. La parte (b) de la figura muestra las manipulaciones de apuntadores actuales.

La función miembro imprimir

La función `imprimir` (líneas 159 a 179) determina primero si la lista está vacía (línea 162). De ser así, imprime “La lista esta vacia” y regresa (líneas 164 y 165). En caso contrario, itera a través de la lista e imprime el valor en cada nodo. La función inicializa `actualPtr` como una copia de `primeroPtr` (línea 168), y después imprime la cadena “La lista es: ” (línea 170). Mientras que `actualPtr` no sea nulo (línea 172), se imprime el valor de `actualPtr->datos` (línea 174) y a `actualPtr` se le asigna el valor de `actualPtr->siguientePtr` (línea 175). Observe que si el vínculo en el último nodo de la lista no es nulo, el algoritmo de impresión tratará erróneamente de imprimir más allá del final de la lista. El algoritmo de impresión es idéntico para las listas enlazadas, pilas y colas (ya que basamos cada una de estas estructuras de datos en la misma infraestructura de la lista enlazada).

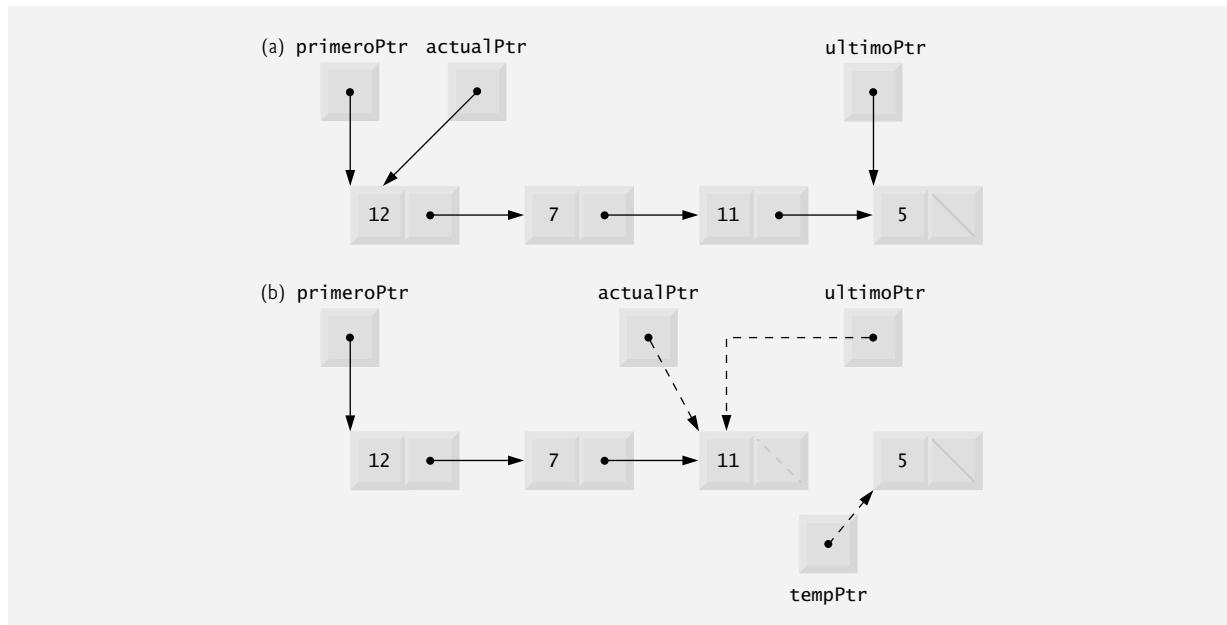


Figura 20.9 | Representación gráfica de la operación `eliminarDelFinal`.

Listas lineales y circulares de enlace simple y enlace doble

El tipo de lista enlazada que hemos estado viendo es una **lista de enlace simple**: la lista empieza con un apuntador al primer nodo, y cada nodo contiene un apuntador al siguiente nodo “en secuencia”. Esta lista termina con un nodo cuyo miembro apuntador tiene el valor 0. Una lista de enlace simple puede recorrerse sólo en una dirección.

Una **lista circular de enlace simple** (figura 20.10) empieza con un apuntador al primer nodo, y cada nodo contiene un apuntador al siguiente nodo. El “último nodo” no contiene un apuntador 0; en vez de ello, el apuntador en el último nodo apunta de vuelta al primer nodo, con lo cual se cierra el “círculo”.

Una **lista de enlace doble** (figura 20.11) permite recorridos tanto hacia adelante como hacia atrás. Dicha lista se implementa comúnmente con dos “apuntadores iniciales”: uno que apunta al primer elemento de la lista para permitir el recorrido desde la parte inicial hasta la parte final de la lista, y uno que apunta al último elemento para permitir un recorrido desde la parte final hasta la parte inicial. Cada nodo tiene un apuntador al siguiente nodo en la lista, en dirección hacia adelante, y un apuntador al siguiente nodo en la lista, en dirección hacia atrás. Por ejemplo, si su lista contiene un directorio telefónico alfabetizado, la búsqueda de alguien cuyo nombre empiece con una letra cerca del principio del alfabeto podría empezar desde la parte frontal de la lista. La búsqueda de alguien cuyo nombre empiece con una letra cerca del final del alfabeto podría empezar desde la parte final de la lista.

En una **lista circular con enlace doble** (figura 20.12), el apuntador hacia adelante del último nodo apunta al primer nodo, y el apuntador hacia atrás del primer nodo apunta al último nodo, con lo cual se cierra el “círculo”.



Figura 20.10 | Lista circular de enlace simple.

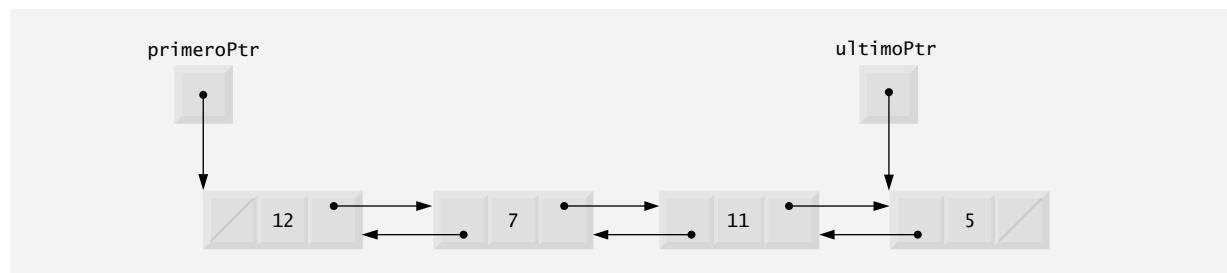


Figura 20.11 | Lista con enlace doble.

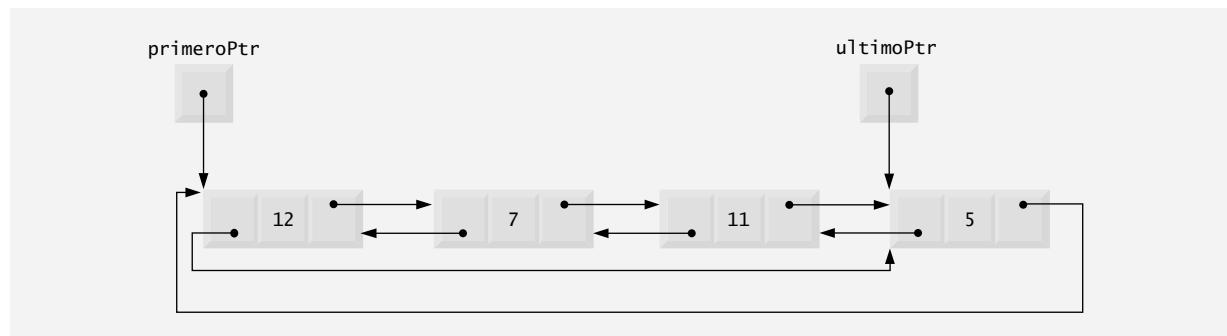


Figura 20.12 | Lista circular con enlace doble.

20.5 Pilas

En el capítulo 14, Plantillas, explicamos la noción de una plantilla de la clase pila con una implementación subyacente tipo arreglo. En esta sección utilizamos una implementación subyacente tipo lista enlazada basada en apuntadores. También veremos las pilas en el capítulo 22, Biblioteca de plantillas estándar (STL).

Una estructura de datos tipo pila permite agregar nodos a la pila y eliminarlos de ésta sólo desde su parte superior. Por esta razón, a una pila se le conoce como estructura de datos UEPS (último en entrar, primero en salir). Una manera de implementar una pila es como una versión restringida de una lista enlazada. En dicha implementación, el miembro de enlace en el último nodo de la pila se establece con el valor nulo (cero) para indicar el fondo de la pila.

Las funciones miembro básicas para manipular una pila son `push` (empujar) y `pop` (sacar). La función `push` inserta un nuevo nodo a la parte superior de la pila. La función `pop` elimina un nodo de la parte superior de la pila, almacena el valor que sacó en una variable de referencia que se pasa a la función que hizo la llamada, y devuelve `true` si la operación `pop` tuvo éxito (`false` en caso contrario).

Las pilas tienen muchas aplicaciones interesantes. Por ejemplo, cuando se hace una llamada a una función, la función llamada debe saber cómo regresar a su invocador, por lo que la dirección de retorno se mete en una pila. Si ocurre una serie de llamadas a funciones, los valores de retorno sucesivos se meten en la pila en el orden, último en entrar, primero en salir, para que cada función pueda regresar a su invocador. Las pilas soportan llamadas recursivas a funciones de la misma manera que las llamadas no recursivas convencionales. En la sección 6.11 se describe con detalle la pila de llamadas a funciones.

Las pilas proporcionan la memoria para, y almacenan los valores de las variables automáticas en cada invocación de una función. Cuando la función regresa a su invocador o lanza una excepción, se hace una llamada al destructor (si lo hay) para cada objeto local llamado, el espacio para las variables automáticas de esa función se saca de la pila y esas variables ya no son conocidas para el programa.

Los compiladores utilizan pilas para evaluar expresiones y generar código en lenguaje máquina. Los ejercicios en este capítulo exploran varias aplicaciones de las pilas, incluyendo el utilizarlas para desarrollar un compilador funcional completo.

Tomaremos ventaja de la estrecha relación entre las listas y las pilas para implementar una clase de pila, principalmente mediante la reutilización de una clase de lista. Primero vamos a implementar la clase de pila a través de la herencia `private` de la clase de lista. Despues implementaremos una clase de pila con la misma funcionalidad por medio de la composición, incluyendo un objeto lista como un miembro `private` de una clase de pila. Desde luego que todas las estructuras de datos en este capítulo, incluyendo estas dos clases de pilas, se implementan como plantillas para fomentar su reutilización a futuro.

El programa de las figuras 20.13 y 20.14 crea una plantilla de clase `Pila` (figura 20.13), principalmente a través de la herencia `private` (línea 9) de la plantilla de clase `Lista` de la figura 20.4. Queremos que la `Pila` tenga las funciones miembro `push` (líneas 13 a 16), `pop` (líneas 19 a 22), `estaPilaVacia` (líneas 25 a 28) e `imprimirPila` (líneas 31 a 34). Observe que, en esencia, estas funciones son las funciones `insertarAlFrente`, `eliminarDelFrente`, `estaVacia` e `imprimir` de la plantilla de clase `Lista`. Desde luego, la plantilla de clase `Lista` contiene otras funciones miembro

```

1 // Fig. 20.13: Pila.h
2 // Definicion de la plantilla de clase Pila, derivada de la clase Lista.
3 #ifndef PILA_H
4 #define PILA_H
5
6 #include "Lista.h" // definición de la clase Lista
7
8 template< typename TIPOPILA >
9 class Pila : private Lista< TIPOPILA >
10 {
11 public:
12     // push llama a la función insertarAlFrente de Lista
13     void push( const TIPOPILA &datos )
14     {
15         insertarAlFrente( datos );
16     } // fin de la función push
17
18     // pop llama a la función eliminarDelFrente de Lista
19     bool pop( TIPOPILA &datos )
20     {

```

Figura 20.13 | Definición de la plantilla de clase `Pila`. (Parte I de 2).

```

21     return eliminarDelFrente( datos );
22 } // fin de la función pop
23
24 // estaPilaVacia llama a la función estaVacia de Lista
25 bool estaPilaVacia() const
26 {
27     return estaVacia();
28 } // fin de la función estaPilaVacia
29
30 // imprimirPila llama a la función imprimir de Lista
31 void imprimirPila() const
32 {
33     imprimir();
34 } // fin de la función imprimir
35 }; // fin de la clase Pila
36
37 #endif

```

Figura 20.13 | Definición de la plantilla de clase `Pila`. (Parte 2 de 2).

(como `insertarAlFinal` y `eliminarDelFinal`) que no es conveniente que estén accesibles a través de la interfaz `public` para la clase `Pila`. Por lo tanto, al indicar que la plantilla de clase `Pila` va a heredar de la plantilla de clase `Lista`, especificamos una herencia `private`. Esto hace a todas las funciones miembro de la plantilla de clase `Lista` `private` en la plantilla de clase `Pila`. Luego, al implementar las funciones miembro de `Pila` hacemos que cada una de éstas haga una llamada a la función miembro apropiada de la clase `Lista`: `push` llama a `insertarAlFrente` (línea 15), `pop` llama a `eliminarDelFrente` (línea 21), `estaPilaVacia` llama a `estaVacia` (línea 27) e `imprimirPila` llama a `imprimir` (línea 33). A esto se le conoce como **delegación**.

La plantilla de clase tipo pila se utiliza en `main` (figura 20.14) para instanciar la pila entera `pilaInt` de tipo `Pila< int >` (línea 11). Los enteros del 0 al 2 se meten en `pilaInt` (líneas 16 a 20), y después se sacan de `pilaInt` (líneas 25 a 30). El programa utiliza la plantilla de clase `Pila` para crear la `pilaDouble` de tipo `Pila< double >` (línea 32). Los valores 1.1, 2.2 y 3.3 se meten en `pilaDouble` (líneas 38 a 43), y después se sacan de `pilaDouble` (líneas 48 a 53).

```

1 // Fig. 20.14: Fig20_14.cpp
2 // Programa de prueba de la plantilla de clase Pila.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Pila.h" // definición de la clase Pila
8
9 int main()
10{
11    Pila< int > pilaInt; // crea una Pila de valores int
12
13    cout << "procesando una Pila de valores enteros" << endl;
14
15    // mete los enteros a pilaInt
16    for ( int i = 0; i < 3; i++ )
17    {
18        pilaInt.push( i );
19        pilaInt.imprimirPila();
20    } // fin de for
21
22    int enteroSacado; // almacena el valor int sacado de la pila
23
24    // saca enteros de pilaInt
25    while ( !pilaInt.estaPilaVacia() )

```

Figura 20.14 | Un programa de prueba de una pila simple. (Parte 1 de 2).

```

26  {
27      pilaInt.pop( enteroSacado );
28      cout << enteroSacado << " se saco de la pila" << endl;
29      pilaInt.imprimirPila();
30  } // fin de while
31
32  Pila< double > pilaDouble; // crea Pila de valores double
33  double valor = 1.1;
34
35  cout << "procesando una Pila de valores double" << endl;
36
37  // mete los valores de punto flotante a pilaDouble
38  for ( int j = 0; j < 3; j++ )
39  {
40      pilaDouble.push( valor );
41      pilaDouble.imprimirPila();
42      valor += 1.1;
43  } // fin de for
44
45  double doubleSacado; // almacena el valor double sacado de la pila
46
47  // saca los valores de punto flotante de pilaDouble
48  while ( !pilaDouble.estaPilaVacia() )
49  {
50      pilaDouble.pop( doubleSacado );
51      cout << doubleSacado << " se saco de la pila" << endl;
52      pilaDouble.imprimirPila();
53  } // fin de while
54
55  return 0;
56 } // fin de main

```

procesando una Pila de valores enteros
La lista es: 0

La lista es: 1 0

La lista es: 2 1 0

2 se saco de la pila
La lista es: 1 0

1 se saco de la pila
La lista es: 0

0 se saco de la pila
La lista esta vacia

procesando una Pila de valores double
La lista es: 1.1

La lista es: 2.2 1.1

La lista es: 3.3 2.2 1.1

3.3 se saco de la pila
La lista es: 2.2 1.1

2.2 se saco de la pila
La lista es: 1.1

1.1 se saco de la pila
La lista esta vacia

Se destruyeron todos los nodos

Se destruyeron todos los nodos

Figura 20.14 | Un programa de prueba de una pila simple. (Parte 2 de 2).

Otra manera de implementar una plantilla de clase `Pila` es reutilizando la plantilla de clase `Lista` mediante la composición. La figura 20.15 es una nueva implementación de la plantilla de clase `Pila`, la cual contiene un objeto `Lista < TIPOPILA >` llamado `listaPila` (línea 38). Esta versión de la plantilla de clase `Pila` utiliza la clase `Lista` de la figura 20.4. Para probar esta clase utilice el programa controlador de la figura 20.14, pero incluya el nuevo archivo de encabezado: `PilaComposicion.h` en la línea 6 de ese archivo. El resultado del programa es idéntico para ambas versiones de la clase `Pila`.

```

1 // Fig. 20.15: Pilacomposición.h
2 // Definicion de la plantilla de clase Pila con un objeto Lista compuesto.
3 #ifndef PILACOMPOSICION_H
4 #define PILACOMPOSICION_H
5
6 #include "Lista.h" // definición de la clase Lista
7
8 template< typename TIPOPILA >
9 class Pila
10 {
11 public:
12     //sin constructor; el constructor de Lista realiza la inicializacion
13
14     // push llama a la función miembro insertarAlFrente del objeto listaPila
15     void push( const TIPOPILA &datos )
16     {
17         listaPila.insertarAlFrente( datos );
18     } // fin de la función push
19
20     // pop llama a la función miembro eliminarDelFrente del objeto listaPila
21     bool pop( TIPOPILA &datos )
22     {
23         return listaPila.eliminarDelFrente( datos );
24     } // fin de la función pop
25
26     // estaPilaVacia llama a la función miembro estaVacia del objeto listaPila
27     bool estaPilaVacia() const
28     {
29         return listaPila.estaVacia();
30     } // fin de la función estaPilaVacia
31
32     // imprimirPila llama a la función miembro imprimir del objeto listaPila
33     void imprimirPila() const
34     {
35         listaPila.imprimir();
36     } // fin de la función imprimirPila
37 private:
38     Lista< TIPOPILA > listaPila; // objeto Lista compuesto
39 }; // fin de la clase Pila
40
41 #endif

```

Figura 20.15 | Plantilla de clase `Pila` con un objeto `Lista` compuesto.

20.6 Colas

Una cola es similar a la fila para pagar en un supermercado: el cajero atiende primero a la persona que se encuentra hasta adelante, y los demás clientes entran a la fila sólo por su parte final y esperan a que se les atienda. Los nodos de una cola se eliminan sólo desde el principio (cabeza) de la misma y se insertan sólo al final (cola) de ésta. Por esta razón, a una cola se le conoce como estructura de datos PEPS (primero en entrar, primero en salir). Las operaciones para insertar y eliminar se conocen como `enqueue` (agregar a la cola) y `dequeue` (retirar de la cola).

Las colas tienen muchas aplicaciones en los sistemas computacionales. Las computadoras que tienen un procesador sólo pueden atender a un usuario a la vez. Las entradas para los otros usuarios se colocan en una cola. Cada entrada

avanza gradualmente al frente de la cola, a medida que los usuarios reciben atención. La entrada al frente es la siguiente en recibir atención.

Las colas también se utilizan para dar soporte al uso de la **cola de impresión**. Por ejemplo, una sola impresora puede compartirse entre todos los usuarios de la red. Muchos usuarios pueden enviar trabajos a la impresora, incluso cuando ésta ya se encuentre ocupada. Estos trabajos de impresión se colocan en una cola hasta que la impresora esté disponible. Un programa conocido como **spooler** administra la cola para asegurarse que, a medida que se complete cada trabajo de impresión, se envíe el siguiente trabajo a la impresora.

En las redes computacionales, los paquetes de información también esperan en colas. Cada vez que un paquete llega a un nodo de la red, debe enrutararse hacia el siguiente nodo en la red a través de la ruta hacia el destino final del paquete. El nodo enrutador envía un paquete a la vez, por lo que los paquetes adicionales se ponen en una cola hasta que el enrutador pueda enviarlos.

Un servidor de archivos en una red computacional se encarga de las peticiones de acceso a los archivos de muchos clientes distribuidos en la red. Los servidores tienen una capacidad limitada para dar servicio a las peticiones de los clientes. Cuando se excede esa capacidad, las peticiones de los clientes esperan en colas.

El programa de las figuras 20.16 y 20.17 crea una plantilla de clase **Cola** (figura 20.16) a través de la herencia **private** (línea 9) de la plantilla de clase **Lista** de la figura 20.4. Queremos que la **Cola** tenga las funciones miembro **enqueue** (líneas 13 a 16), **dequeue** (líneas 19 a 22), **estaColaVacia** (líneas 25 a 28) e **imprimirCola** (líneas 31 a 34). Observe que, en esencia, éstas son las funciones **insertarAlFinal**, **eliminarDelFrente**, **estaVacia** e **imprimir** de la plantilla de clase **Lista**. Desde luego, la plantilla de clase **Lista** contiene otras funciones miembro (es decir, **insertar-**

```

1 // Fig. 20.16: Cola.h
2 // Definición de la plantilla de clase Cola, derivada de la clase Lista.
3 #ifndef COLA_H
4 #define COLA_H
5
6 #include "Lista.h" // definición de la clase Lista
7
8 template< typename TIPOCOLA >
9 class Cola: private Lista< TIPOCOLA >
10 {
11 public:
12     // enqueue llama a la función miembro insertarAlFinal de Lista
13     void enqueue( const TIPOCOLA &datos )
14     {
15         insertarAlFinal( datos );
16     } // fin de la función enqueue
17
18     // dequeue llama a la función miembro eliminarDelFrente de Lista
19     bool dequeue( TIPOCOLA &datos )
20     {
21         return eliminarDelFrente( datos );
22     } // fin de la función dequeue
23
24     // estaListaVacia llama a la función miembro estaVacia de Lista
25     bool estaListaVacia() const
26     {
27         return estaVacia();
28     } // fin de la función estaListaVacia
29
30     // imprimirCola llama a la función miembro imprimir de Lista
31     void imprimirCola() const
32     {
33         imprimir();
34     } // fin de la función imprimirCola
35 }; // fin de la clase Cola
36
37 #endif

```

Figura 20.16 | Definición de la plantilla de clase **Cola**.

`AlFrente` y `eliminarDelFinal`) que no deseamos que estén accesibles a través de la interfaz `public` para la clase `Cola`. Por lo tanto, cuando indicamos que la plantilla de clase `Cola` va a heredar la plantilla de clase `Lista`, especificamos la herencia `private`. Esto hace a todas las funciones miembro de la plantilla de clase `Lista private` en la plantilla de clase `Cola`. Al implementar las funciones miembro de `Cola`, hacemos que cada una de éstas haga una llamada a la función miembro apropiada de la clase de lista: `enqueue` llama a `insertarAlFinal` (línea 15), `dequeue` llama a `eliminarDelFrente` (línea 21), `estaColaVacia` llama a `estaVacia` (línea 27) e `imprimirCola` llama a `imprimir` (línea 33). De nuevo, a esto se le conoce como delegación.

En la figura 20.17 se utiliza la plantilla de clase `Cola` para instanciar la cola de enteros `colaInt` de tipo `Cola < int >` (línea 11). Los enteros del 0 al 2 se agregan a `colaInt` (líneas 16 a 20), y luego se sacan de `colaInt` en el orden “primero en entrar, primero en salir” (líneas 25 a 30). A continuación, el programa instancia la cola `colaDouble` de tipo `Cola < double >` (línea 32). Los valores 1.1, 2.2 y 3.3 se agregan a `colaDouble` (líneas 38 a 43), y después se sacan de `colaDouble` en orden “primero en entrar, primero en salir” (líneas 48 a 53).

```

1 // Fig. 20.17: Fig20_17.cpp
2 // Programa de prueba de la plantilla de clase Cola.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Cola.h" // definición de la clase Cola
8
9 int main()
10 {
11     Cola< int > colaInt; // crea una Cola de enteros
12
13     cout << "procesando una Cola de valores enteros" << endl;
14
15     // agrega los enteros a colaInt
16     for ( int i = 0; i < 3; i++ )
17     {
18         colaInt.enqueue( i );
19         colaInt.imprimirCola();
20     } // fin de for
21
22     int enteroSacado; // almacena el entero sacado de la cola
23
24     // saca enteros de colaInt
25     while ( !colaInt.estaListaVacia() )
26     {
27         colaInt.dequeue( enteroSacado );
28         cout << enteroSacado << " se saco de la cola" << endl;
29         colaInt.imprimirCola();
30     } // fin de while
31
32     Cola< double > colaDouble; // crea una Cola de valores double
33     double valor = 1.1;
34
35     cout << "procesando una Cola de valores double" << endl;
36
37     // agrega los valores de punto flotante a colaDouble
38     for ( int j = 0; j < 3; j++ )
39     {
40         colaDouble.enqueue( valor );
41         colaDouble.imprimirCola();
42         valor += 1.1;
43     } // fin de for
44
45     double doubleSacado; // almacena el valor double sacado de la cola

```

Figura 20.17 | Programa para procesar una cola. (Parte 1 de 2).

```

46 // saca los valores de punto flotante de colaDouble
47 while ( !colaDouble.estaListaVacia() )
48 {
49     colaDouble.dequeue( doubleSacado );
50     cout << doubleSacado << " se saco de la cola" << endl;
51     colaDouble.imprimirCola();
52 }
53 // fin de while
54
55 return 0;
56 } // fin de main

```

procesando una Cola de valores enteros
La lista es: 0

La lista es: 0 1

La lista es: 0 1 2

0 se saco de la cola
La lista es: 1 2

1 se saco de la cola
La lista es: 2

2 se saco de la cola
La lista esta vacia

procesando una Cola de valores double
La lista es: 1.1

La lista es: 1.1 2.2

La lista es: 1.1 2.2 3.3

1.1 se saco de la cola
La lista es: 2.2 3.3

2.2 se saco de la cola
La lista es: 3.3

3.3 se saco de la cola
La lista esta vacia

Se destruyeron todos los nodos

Se destruyeron todos los nodos

Figura 20.17 | Programa para procesar una cola. (Parte 2 de 2).

20.7 Árboles

Las listas enlazadas, pilas y colas son estructuras lineales de datos. Un árbol es una estructura de datos bidimensional no lineal. Los nodos de un árbol contienen dos o más enlaces. En esta sección hablaremos sobre los **árboles binarios** (figura 20.18): los árboles cuyos nodos contienen dos enlaces (de los cuales ninguno, uno o ambos pueden ser null).

Terminología básica

Para esta discusión, refiérase a los nodos A, B, C y D en la figura 20.18. El **nodo raíz** (nodo B) es el primer nodo en un árbol. Cada enlace en el nodo raíz hace referencia a un **hijo** (nodos A y D). El **hijo izquierdo** (nodo A) es el nodo raíz del **subárbol izquierdo** (que sólo contiene el nodo A), y el **hijo derecho** (nodo D) es el nodo raíz del **subárbol derecho** (que contiene los nodos D y C). Los hijos de un nodo específico se llaman **hermanos** (por ejemplo, los nodos A y D son hermanos). Un nodo sin hijos se llama **nodo hoja** (por ejemplo, los nodos A y C son nodos hoja). Generalmente, los científicos computacionales dibujan árboles desde el nodo raíz hacia abajo; exactamente lo opuesto a la manera en que crecen los árboles naturales.

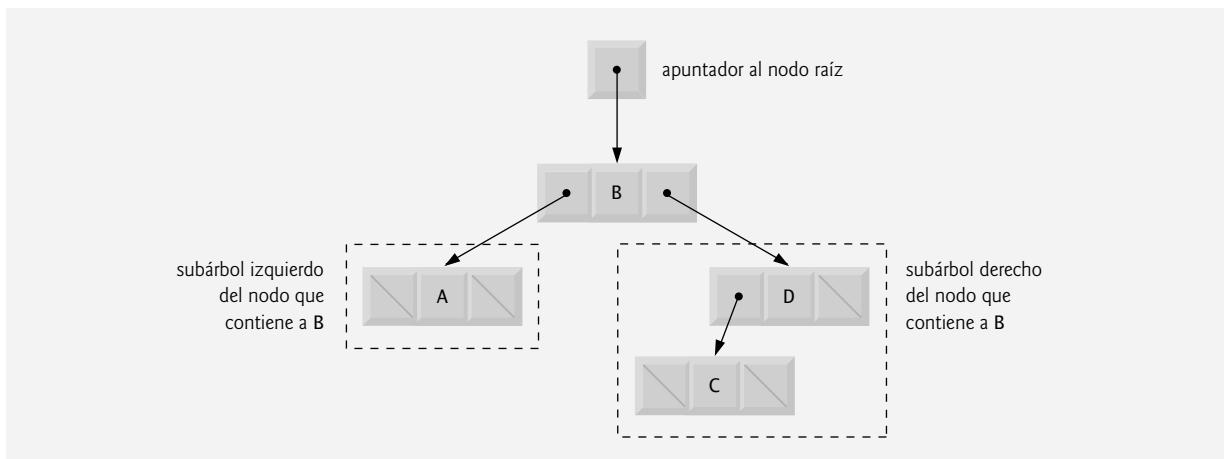


Figura 20.18 | Representación gráfica de un árbol binario.

Árboles de búsqueda binaria

Un **árbol de búsqueda binaria** (sin valores de nodos duplicados) tiene la característica de que los valores en cualquier subárbol izquierdo son menores que el valor en su **nodo padre**, y los valores en cualquier subárbol derecho son mayores que el valor en su nodo padre. En la figura 20.19 se muestra un árbol de búsqueda binaria con 9 valores. Observe que la forma del árbol de búsqueda binaria que corresponde a un conjunto de datos puede variar, dependiendo del orden en el que se inserten los valores en el árbol.

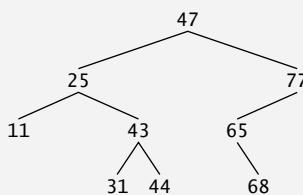


Figura 20.19 | Un árbol de búsqueda binario.

Implementación del programa del árbol de búsqueda binaria

El programa de las figuras 20.20 a 20.22 crea un árbol de búsqueda binaria y lo recorre (es decir, avanza a través de todos sus nodos) de tres maneras: usando los recorridos recursivos, **recorridos inorder**, **preorden** y **postorden** recursivos. En breve explicaremos estos algoritmos de recorrido.

Empezaremos nuestra discusión con el programa controlador (figura 20.22) y después continuaremos con las implementaciones de las clases **NodoArbol** (figura 20.20) y **Arbol** (figura 20.21). La función **main** (figura 20.22) empieza instanciando el árbol entero **arbolInt** de tipo **Arbol< int >** (línea 15). El programa pide 10 enteros, cada uno de los cuales se inserta en el árbol binario mediante una llamada a **insertarNodo** (línea 24). Despues el programa realiza recorridos preorden, inorder y postorden (explicaremos éstos en breve) de **arbolInt** (líneas 28, 31 y 34, respectivamente). Luego el programa instancia el árbol de punto flotante **arbolDouble** de tipo **Arbol< double >** (línea 36). El programa pide 10 valores **double**, cada uno de los cuales se inserta en el árbol binario mediante una llamada a **insertarNodo** (línea 46). A continuación, el programa realiza recorridos preorden, inorder y postorden de **arbolDouble** (líneas 50, 53 y 56, respectivamente).

Ahora trataremos la definición de plantillas de clase. Empezaremos con la definición de la plantilla de clase **ArbolNodo** (figura 20.20) que declara **Arbol <TIPONODO>** como su amiga (línea 13). Esto hace que todas las funciones miembro de una plantilla de clase de **Arbol** dada (figura 20.21) sean amigas de la plantilla clase especializada **ArbolNodo**, así que ellas pueden accesar los miembros **private** de los objetos **ArbolNodo**. Debido a que el parámetro **TIPONODO** de la plantilla para **ArbolNodo** se usa como el argumento de plantilla para **Arbol** en la declaración **friend**, **ArbolNodos** especializados con un tipo particular se pueden procesar solamente con un **Arbol** especializado del mismo tipo (es decir, un **Arbol** de valores enteros, administra objetos **ArbolNodo** que guardan valores enteros).

```

1 // Fig. 20.20: NodoArbol.h
2 // Definición de la plantilla de clase NodoArbol.
3 #ifndef NODOARBOL_H
4 #define NODOARBOL_H
5
6 // declaración anticipada de la clase Arbol
7 template< typename TIPONODO > class Arbol;
8
9 // definición de la plantilla de clase NodoArbol
10 template< typename TIPONODO >
11 class NodoArbol
12 {
13     friend class Arbol< TIPONODO >;
14 public:
15     // constructor
16     NodoArbol( const TIPONODO &d )
17         : izquierdoPtr( 0 ), // apuntador al subárbol izquierdo
18           datos( d ), // datos del nodo del árbol
19           derechoPtr( 0 ) // apuntador al subárbol derecho
20     {
21         // cuerpo vacío
22     } // fin del constructor de NodoArbol
23
24     // devuelve una copia de los datos del nodo
25     TIPONODO obtenerDatos() const
26     {
27         return datos;
28     } // fin de la función obtenerDatos
29 private:
30     NodoArbol< TIPONODO > *izquierdoPtr; // apuntador al subárbol izquierdo
31     TIPONODO datos;
32     NodoArbol< TIPONODO > *derechoPtr; // apuntador al subárbol derecho
33 }; // fin de la clase NodoArbol
34
35 #endif

```

Figura 20.20 | Definición de la plantilla de clase `NodoArbol`.

```

1 // Fig. 20.21: Arbol.h
2 // Definición de la plantilla de clase Arbol.
3 #ifndef ARBOL_H
4 #define ARBOL_H
5
6 #include <iostream>
7 using std::cout;
8 using std::endl;
9
10 #include "NodoArbol.h"
11
12 // definición de la plantilla de clase Arbol
13 template< typename TIPONODO > class Arbol
14 {
15 public:
16     Arbol(); // constructor
17     void insertarNodo( const TIPONODO & );
18     void recorridoPreOrden() const;
19     void recorridoInOrden() const;
20     void recorridoPostOrden() const;
21 private:
22     NodoArbol< TIPONODO > *raizPtr;

```

Figura 20.21 | Definición de la plantilla de clase `Arbol`. (Parte I de 3).

```

23 // funciones utilitarias
24 void ayudanteInsertarNodo( NodoArbol< TIPONODO > **, const TIPONODO & );
25 void ayudantePreOrden( NodoArbol< TIPONODO > * ) const;
26 void ayudanteInOrden( NodoArbol< TIPONODO > * ) const;
27 void ayudantePostOrden( NodoArbol< TIPONODO > * ) const;
28 } // fin de la clase Arbol
29
30 // constructor
31 template< typename TIPONODO >
32 Arbol< TIPONODO >::Arbol()
33 {
34     raizPtr = 0; // indica que al principio el árbol está vacío
35 } // fin del constructor de Arbol
36
37 // inserta el nodo en el Arbol
38 template< typename TIPONODO >
39 void Arbol< TIPONODO >::insertarNodo( const TIPONODO &valor )
40 {
41     ayudanteInsertarNodo( &raizPtr, valor );
42 } // fin de la función insertarNodo
43
44 // función utilitaria llamada por insertarNodo; recibe un apuntador
45 // a un apuntador, para que la función pueda modificar el valor del apuntador
46 template< typename TIPONODO >
47 void Arbol< TIPONODO >::ayudanteInsertarNodo(
48     NodoArbol< TIPONODO > **ptr, const TIPONODO &valor )
49 {
50     // el subárbol está vacío; crea nuevo NodoArbol que contiene el valor
51     if ( *ptr == 0 )
52         *ptr = new NodoArbol< TIPONODO >( valor );
53     else // el subárbol no está vacío
54     {
55         // los datos a insertar son menores que los datos en el nodo actual
56         if ( valor < ( *ptr )->datos )
57             ayudanteInsertarNodo( &( ( *ptr )->izquierdoPtr ), valor );
58         else
59         {
60             // los datos a insertar son mayores que los datos en el nodo actual
61             if ( valor > ( *ptr )->datos )
62                 ayudanteInsertarNodo( &( ( *ptr )->derechoPtr ), valor );
63             else // se ignora el valor de datos duplicado
64                 cout << valor << " dup" << endl;
65         } // fin de else
66     } // fin de else
67 } // fin de else
68 } // fin de la función ayudanteInsertarNodo
69
70 // empieza el recorrido preorden del Arbol
71 template< typename TIPONODO >
72 void Arbol< TIPONODO >::recorridoPreOrden() const
73 {
74     ayudantePreOrden( raizPtr );
75 } // fin de la función recorridoPreOrden
76
77 // función utilitaria para realizar el recorrido preorden del Arbol
78 template< typename TIPONODO >
79 void Arbol< TIPONODO >::ayudantePreOrden( NodoArbol< TIPONODO > *ptr ) const
80 {
81     if ( ptr != 0 )
82     {
83         cout << ptr->datos << ' '; // procesa el nodo
84         ayudantePreOrden( ptr->izquierdoPtr ); // recorre el subárbol izquierdo

```

Figura 20.21 | Definición de la plantilla de clase Arbol. (Parte 2 de 3).

```

85     ayudantePreOrden( ptr->derechoPtr ); // recorre el subárbol derecho
86 } // fin de if
87 } // fin de la función ayudantePreOrden
88
89 // empieza el recorrido inorden del Arbol
90 template< typename TIPONODO >
91 void Arbol< TIPONODO >::recorridoInOrden() const
92 {
93     ayudanteInOrden( raizPtr );
94 } // fin de la función recorridoInOrden
95
96 // función utilitaria para realizar el recorrido inorden del Arbol
97 template< typename TIPONODO >
98 void Arbol< TIPONODO >::ayudanteInOrden( NodoArbol< TIPONODO > *ptr ) const
99 {
100    if ( ptr != 0 )
101    {
102        ayudanteInOrden( ptr->izquierdoPtr ); // recorre el subárbol izquierdo
103        cout << ptr->datos << ' '; // procesa el nodo
104        ayudanteInOrden( ptr->derechoPtr ); // recorre el subárbol derecho
105    } // fin de if
106 } // fin de la función ayudanteInOrden
107
108 // empieza el recorrido postorden del Arbol
109 template< typename TIPONODO >
110 void Arbol< TIPONODO >::recorridoPostOrden() const
111 {
112     ayudantePostOrden( raizPtr );
113 } // fin de la función recorridoPostOrden
114
115 // función utilitaria para realizar el recorrido postorden del Arbol
116 template< typename TIPONODO >
117 void Arbol< TIPONODO >::ayudantePostOrden(
118     NodoArbol< TIPONODO > *ptr ) const
119 {
120    if ( ptr != 0 )
121    {
122        ayudantePostOrden( ptr->izquierdoPtr ); // recorre el subárbol izquierdo
123        ayudantePostOrden( ptr->derechoPtr ); // recorre el subárbol derecho
124        cout << ptr->datos << ' '; // procesa el nodo
125    } // fin de if
126 } // fin de la función ayudantePostOrden
127
128 #endif

```

Figura 20.21 | Definición de la plantilla de clase Arbol. (Parte 3 de 3).

En las líneas 30-32 se declaran datos **private** de **ArbolNodo**, los valores para los **datos** del nodo, y apunadores **izquierdoPtr** (a los nodos del subárbol izquierdo) y **derechoPtr** (a los nodos del subárbol derecho). El constructor (líneas 16-22) establece los datos al valor como argumento del constructor y establece los apunadores **izquierdoPtr** y **derechoPtr** a cero (inicializando este nodo como nodo hoja). La función miembro obtenerDatos (líneas 25-28) devuelve el valor dato.

```

1 // Fig. 20.22: Fig20_22.cpp
2 // Programa de prueba de la clase Arbol.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::fixed;

```

Figura 20.22 | Creación y recorrido de un árbol binario. (Parte I de 3).

```

7 #include <iomanip>
8 using std::setprecision;
9
10
11 #include "Tree.h" //Definición de la clase Arbol
12
13 int main()
14 {
15     Tree< int > intTree; // Crea un Arbol de valores del tipo int
16     int intValue;
17
18     cout << "Escriba 10 values enteros:\n";
19
20     // insertar 10 enteros a intTree
21     for ( int i = 0; i < 10; i++ )
22     {
23         cin >> intValue;
24         intTree.insertNode( intValue );
25     } // fin del for
26
27     cout << "\nRecorrido Preorden\n";
28     intTree.preOrderTraversal();
29
30     cout << "\nRecorrido inOrden\n";
31     intTree.inOrderTraversal();
32
33     cout << "\nRecorrido Postorden\n";
34     intTree.postOrderTraversal();
35
36     Tree< double > doubleTree; // crea Arbol con valores tipo double
37     double doubleValue;
38
39     cout << fixed << setprecision( 1 )
40         << "\n\n\nEscriba 10 valores del tipo double:\n";
41
42     // insertar 10 valores del tipo double en doubleTree
43     for ( int j = 0; j < 10; j++ )
44     {
45         cin >> doubleValue;
46         doubleTree.insertNode( doubleValue );
47     } // fin del for
48
49     cout << "\nRecorrido Preorden\n";
50     doubleTree.preOrderTraversal();
51
52     cout << "\nRecorrido inOrden\n";
53     doubleTree.inOrderTraversal();
54
55     cout << "\nRecorrido Postorden\n";
56     doubleTree.postOrderTraversal();
57
58     cout << endl;
59     return 0;
60 } // fin main

```

Escriba 10 valores enteros:
50 25 75 12 33 67 88 6 13 68

Recorrido preorden
50 25 12 6 13 33 75 67 68 88
Recorrido inorden
6 12 13 25 33 50 67 68 75 88

Figura 20.22 | Creación y recorrido de un árbol binario. (Parte 2 de 3).

```

Recorrido postorden
6 13 12 33 25 68 67 88 75 50

Escriba 10 valores double:
39.2 16.5 82.7 3.3 65.2 90.8 1.1 4.4 89.5 92.5

Recorrido preorden
39.2 16.5 3.3 1.1 4.4 82.7 65.2 90.8 89.5 92.5

Recorrido inorden
1.1 3.3 4.4 16.5 39.2 65.2 82.7 89.5 90.8 92.5

Recorrido postorden
1.1 4.4 3.3 16.5 65.2 89.5 92.5 90.8 82.7 39.2

```

Figura 20.22 | Creación y recorrido de un árbol binario. (Parte 3 de 3).

La plantilla de clase `Arbol1` (figura 20.21) tiene un dato `private` llamado `raizPtr` (línea 22), un apuntador al nodo raíz del árbol. En las líneas 17 a 20 de la plantilla de clase declaran las funciones miembro `public insertarNodo` (que inserta un nuevo nodo en el árbol) y `recorridoPreOrden`, `recorridoInOrden` y `recorridoPostOrden`, cada una de las cuales recorre el árbol en la forma designada. Cada una de estas funciones miembro llama a su propia función utilitaria recursiva separada para realizar las operaciones apropiadas en la representación interna del árbol, por lo que el programa no tiene que acceder a los datos `private` subyacentes para realizar estas funciones. Recuerde que la recursividad requiere que el programador pase un apuntador que represente el siguiente subárbol a procesar. El constructor de `Arbol1` inicializa `raizPtr` con cero para indicar que, al principio, el árbol está vacío.

La función utilitaria `ayudanteInsertarNodo` de la clase `Arbol1` (líneas 47 a 68) se llama mediante `insertarNodo` (líneas 39 a 43) para insertar un nodo en el árbol de manera recursiva. *Un nodo sólo se puede insertar como un nodo hoja en un árbol de búsqueda binaria.* Si el árbol está vacío se crea un nuevo `NodoArbol1`, se inicializa y se inserta en el árbol (líneas 53 y 54).

Si el árbol no está vacío, el programa compara el valor a insertar con el valor de datos en el nodo raíz. Si el valor a insertar es menor (línea 57), el programa llama en forma recursiva a `ayudanteInsertarNodo` (línea 58) para insertar el valor en el subárbol izquierdo. Si el valor a insertar es mayor (línea 62), el programa llama en forma recursiva a `ayudanteInsertarNodo` (línea 64) para insertar el valor en el subárbol derecho. Si el valor a insertar es idéntico al valor de datos en el nodo raíz, el programa imprime el mensaje " dup" (línea 65) y regresa sin insertar el valor duplicado en el árbol. Observe que `insertarNodo` pasa la dirección de `raizPtr` a `ayudanteInsertarNodo` (línea 42) para que pueda modificar el valor almacenado en `raizPtr` (es decir, la dirección del nodo raíz). Para recibir un apuntador a `raizPtr` (que también es un apuntador), el primer argumento de `ayudanteNodoRaiz` se declara como apuntador a un apuntador a `NodoRaiz`.

Cada una de las funciones miembro `recorridoInOrden` (líneas 90 a 94), `recorridoPreOrden` (líneas 71 a 75) y `recorridoPostOrden` (líneas 109 a 113) recorre el árbol e imprime los valores de los nodos. Para los fines de la siguiente discusión, utilizaremos el árbol de búsqueda binaria de la figura 20.23.

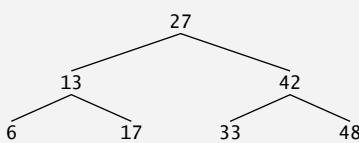


Figura 20.23 | Un árbol de búsqueda binaria.

Algoritmo de recorrido inorden

La función `recorridoInOrden` invoca a la función utilitaria `ayudanteInOrden` para realizar el recorrido inorden del árbol binario. Los pasos para un recorrido inorden son:

1. Recorrer el subárbol izquierdo con un recorrido inorden. (Esto se realiza mediante la llamada en `ayudanteInOrden` en la línea 102).
2. Procesa el valor en el nodo; es decir, imprime el valor del nodo (línea 103).
3. Recorre el subárbol derecho con un recorrido inorden. (Esto se realiza mediante la llamada a `ayudanteInOrden` en la línea 104).

El valor en un nodo no se procesa hasta que se procesen los valores en su subárbol izquierdo, debido a que cada llamada a `ayudanteInOrden` llama inmediatamente a `ayudanteInOrden` de nuevo, con el apuntador al subárbol izquierdo. El recorrido inorden del árbol en la figura 20.23 es:

6 13 17 27 33 42 48

Observe que el recorrido inorden de un árbol de búsqueda binaria imprime los valores del nodo en orden ascendente. El proceso de crear un árbol de búsqueda binaria también ordena los datos; por ende, a este proceso se le conoce como **ordenamiento de árbol binario**.

Algoritmo de recorrido preorden

La función `recorridoPreOrden` invoca a la función utilitaria `ayudantePreOrden` para realizar el recorrido preorden del árbol binario. Los pasos para un recorrido preorden son:

1. Procesar el valor en el nodo (línea 83).
2. Recorrer el subárbol izquierdo con un recorrido preorden. (Esto se realiza mediante la llamada a `ayudantePreOrden` en la línea 84).
3. Recorrer el subárbol derecho con un recorrido preorden. (Esto se realiza mediante la llamada a `ayudantePreOrden` en la línea 85).

El valor en cada nodo se procesa a medida que se visita cada nodo. Una vez que se procesa el valor en un nodo dado, se procesan los valores en el subárbol izquierdo. Después se procesan los valores en el subárbol derecho. El recorrido preorden del árbol en la figura 20.23 es:

27 13 6 17 42 33 48

Algoritmo de recorrido postorden

La función `recorridoPostOrden` invoca a la función utilitaria `ayudantePostOrden` para realizar el recorrido postorden del árbol binario. Los pasos para un recorrido postorden son:

1. Recorrer el subárbol izquierdo con un recorrido postorden. (Esto se realiza mediante la llamada a `ayudantePostOrden` en la línea 122).
2. Recorrer el subárbol derecho con un recorrido postorden. (Esto se realiza mediante la llamada a `ayudantePostOrden` en la línea 123).
3. Procesar el valor en el nodo (línea 124).

El valor en cada nodo no se imprime sino hasta que se impriman los valores de sus hijos. El `recorridoPostOrden` del árbol en la figura 20.23 es:

6 17 13 33 48 42 27

Eliminación de duplicados

El árbol de búsqueda binaria facilita la **eliminación de duplicados**. A medida que se va creando el árbol, se reconoce un intento de insertar un valor duplicado, ya que éste seguirá las mismas decisiones de “ir a la izquierda” o “ir a la derecha” en cada comparación, como se hizo con el valor original cuando se insertó en el árbol. Por ende, el duplicado se comparará en un momento dado con un nodo que contenga el mismo valor. El valor duplicado se puede descartar en este punto.

El proceso de buscar en un árbol binario un valor que coincida con una clave también es rápido. Si el árbol está balanceado, entonces cada ramificación contiene aproximadamente la mitad del número de nodos en el árbol. Cada comparación de un nodo con la clave de búsqueda elimina la mitad de los nodos. A esto se le conoce como algoritmo $O(\log n)$ (en el capítulo 19 se describe la notación Big O). Así, un árbol de búsqueda binaria con n elementos requeriría un máximo de $\log_2 n$ comparaciones para encontrar una coincidencia o para determinar que no existe una. Por ejemplo, esto significa que al buscar en un árbol de búsqueda binaria de 1000 elementos (balanceado), no se necesitan realizar más de 10 comparaciones, ya que $2^{10} > 1000$. Al buscar en un árbol de búsqueda binario de 1,000,000 elementos (balanceado), no se necesitan realizar más de 20 comparaciones, ya que $2^{20} > 1,000,000$.

Generalidades acerca de los ejercicios de árboles binarios

En los ejercicios se presentan algoritmos para otras operaciones más con árboles binarios, como eliminar un elemento de un árbol binario, imprimir un árbol binario en formato de árbol bidimensional y realizar un recorrido de un árbol binario por orden de nivel. El recorrido por orden de nivel de un árbol binario visita los nodos del árbol fila por fila, empezando

con el nivel del nodo raíz. En cada nivel del árbol, los nodos se visitan de izquierda a derecha. Otros ejercicios con árboles binarios incluyen el permitir que un árbol de búsqueda binaria contenga valores duplicados, insertar valores de cadena en un árbol binario y determinar cuántos niveles contiene un árbol binario.

20.8 Repaso

En este capítulo aprendió que las listas enlazadas son colecciones de elementos de datos que están “enlazados en una cadena”. También aprendió que un programa puede realizar inserciones y eliminaciones en cualquier parte de una lista enlazada (aunque nuestra implementación sólo realizaba inserciones y eliminaciones en los extremos de la lista). Demostramos que las estructuras de datos tipo pila y cola son versiones restringidas de las listas. Para las pilas vimos que las inserciones y eliminaciones sólo se realizan en la parte superior. Para las colas que representan filas de espera, vimos que las inserciones se realizan en la parte final (cola) y las eliminaciones se realizan en la parte inicial (cabeza). También presentamos la estructura de datos tipo árbol binario. Vimos un árbol de búsqueda binaria que facilita la búsqueda y el ordenamiento de datos de alta velocidad, y la eliminación eficiente de duplicados. En este capítulo aprendió a crear estas estructuras de datos para su reutilización (como plantillas) y su capacidad de mantenimiento. En el siguiente capítulo introduciremos las estructuras (**struct**), que son similares a las clases, y hablaremos sobre la manipulación de bits, caracteres y cadenas estilo C.

Resumen

Sección 20.1 Introducción

- Las estructuras dinámicas de datos aumentan y reducen su tamaño durante la ejecución.
- Las listas enlazadas son colecciones de elementos de datos “alineados en una fila”; las inserciones y eliminaciones se realizan en cualquier parte de una lista enlazada.
- Las pilas son importantes en los compiladores y sistemas operativos: las inserciones y eliminaciones se realizan sólo en un extremo de la pila: su parte superior (cima).
- Las colas representan líneas de espera; las inserciones se realizan en la parte posterior (también conocida como rabo) de una cola, y las eliminaciones se realizan desde la parte frontal (también conocida como cabeza).
- Los árboles binarios facilitan la búsqueda y el ordenamiento de datos de alta velocidad, la eliminación eficiente de elementos de datos duplicados, la representación de directorios del sistema de archivos y la compilación de expresiones en lenguaje máquina.

Sección 20.2 Clases autorreferenciadas

- Una clase autorreferenciada contiene un miembro apuntador que apunta a un objeto del mismo tipo de clase.
- Los objetos de clases autorreferenciadas pueden enlazarse entre sí para formar estructuras de datos útiles, como listas, colas, pilas y árboles.

Sección 20.3 Asignación dinámica de memoria y estructuras de datos

- El límite para la asignación dinámica de memoria puede ser tan grande como la cantidad de memoria física disponible en la computadora, o la cantidad de memoria virtual disponible en un sistema de memoria virtual.

Sección 20.4 Listas enlazadas

- Una lista enlazada es una colección lineal de objetos de clases autorreferenciadas, llamados nodos, que se conectan mediante enlaces apuntadores; de aquí que se le llame lista “enlazada”.
- Para acceder a una lista enlazada se utiliza un apuntador al primer nodo de la lista. Para acceder a cada nodo subsiguiente se utiliza el miembro apuntador de enlace almacenado en el nodo anterior.
- Las listas enlazadas, pilas y colas son estructuras de datos lineales. Los árboles son estructuras de datos no lineales.
- Una lista enlazada es apropiada cuando el número de elementos de datos que se van a representar en un momento dado es impredecible.
- Las listas enlazadas son dinámicas, por lo que la longitud de una lista puede aumentar o disminuir según sea necesario.
- Una lista con enlace simple empieza con un apuntador al primer nodo, y cada nodo contiene un apuntador al siguiente nodo “en secuencia”.
- Una lista circular con enlace simple empieza con un apuntador al primer nodo, y cada nodo contiene un apuntador al siguiente nodo. El “último nodo” no contiene un apuntador nulo, sino que apunta de vuelta al primer nodo, con lo cual se cierra el “círculo”.
- Una lista con enlace doble permite recorridos hacia adelante y hacia atrás.
- Una lista con enlace doble se implementa comúnmente con dos “apuntadores iniciales”: uno que apunta al primer elemento de la lista para permitir un recorrido desde la parte frontal hasta la parte final, y uno que apunta al último elemento para

- permitir un recorrido desde la parte final hasta la parte frontal. Cada nodo tiene un apuntador al siguiente nodo en la lista en dirección hacia adelante, y un apuntador al siguiente nodo en dirección hacia atrás.
- En una lista circular con enlace doble, el apuntador hacia adelante del último nodo apunta al primer nodo, y el apuntador hacia atrás del primer nodo apunta al último nodo, con lo cual se cierra el “círculo”.

Sección 20.5 Pilas

- Una estructura de datos tipo pila permite agregar nodos a la lista y eliminarlos desde la parte superior.
- Una pila se conoce como estructura de datos UEPS (último en entrar, primero en salir).
- Las funciones miembro básicas utilizadas para manipular una pila son push y pop. La función push inserta un nuevo nodo en la parte superior de la pila. La función pop elimina un nodo de la parte superior de la pila.

Sección 20.6 Colas

- Una cola es similar a una línea para pagar en el supermercado: la primera persona en la línea es atendida primero, y otros clientes entran al final de la línea y esperan a ser atendidos.
- Los nodos de una cola se eliminan sólo desde su cabeza y se insertan sólo en su rabo.
- Una cola se conoce como estructura de datos PEPS (primero en entrar, primero en salir). Las operaciones de insertar y eliminar se conocen como enqueue y dequeue.

Sección 20.7 Árboles

- Los árboles binarios son árboles cuyos nodos contienen dos enlaces (de los cuales ninguno, uno o ambos pueden ser nulos).
- El nodo raíz es el primer nodo en un árbol.
- Cada enlace en el nodo raíz se refiere a un hijo. El hijo izquierdo es el nodo raíz del subárbol izquierdo, y el hijo derecho es el nodo raíz del subárbol derecho.
- Los hijos de un solo nodo se llaman hermanos. Un nodo sin hijos se llama nodo hoja.
- Un árbol de búsqueda binario (sin valores de nodos duplicados) tiene la característica de que los valores en cualquier subárbol izquierdo son menores que el valor en su nodo padre, y los valores en cualquier subárbol derecho son mayores que el valor en su nodo padre.
- Un nodo sólo se puede insertar como un nodo hoja en un árbol de búsqueda binaria.
- Un recorrido inorden de un árbol binario realiza un recorrido inorden del subárbol izquierdo, procesa el valor en el nodo raíz y después realiza un recorrido inorden del subárbol derecho. El valor en un nodo no se procesa sino hasta que se procesen los valores en su subárbol izquierdo.
- Un recorrido preorden procesa el valor en el nodo raíz, realiza un recorrido preorden del subárbol izquierdo y después realiza un recorrido preorden del subárbol derecho. El valor en cada nodo se procesa a medida que se va encontrando.
- Un recorrido postorden realiza un recorrido postorden del subárbol izquierdo, realiza un recorrido postorden del subárbol derecho y después procesa el valor en el nodo raíz. El valor en cada nodo no se procesa sino hasta que se procesen los valores de ambos subárboles.
- El árbol de búsqueda binaria ayuda a eliminar datos duplicados. A medida que se crea el árbol, se reconocerá un intento de insertar un valor duplicado y éste puede descartarse.
- El recorrido por orden de nivel de un árbol binario visita los nodos del árbol fila por fila, empezando en el nivel del nodo raíz. En cada nivel del árbol, los nodos se visitan de izquierda a derecha.

Terminología

árbol binario	hermanos
árbol de búsqueda binaria	hijo derecho
cabeza de una cola	hijo izquierdo
cola	insertar un nodo
cola de impresión	lista circular con enlace doble
delegación	lista circular con enlace simple
dequeue	lista con enlace doble
eliminación de duplicados	lista con enlace simple
enlace	lista enlazada
enlace apuntador	nodo
enqueue	nodo hijo
estructura autorreferenciada	nodo hoja
estructura de datos	nodo padre
estructura de datos lineal	nodo raíz
estructura de datos no lineal	ordenamiento de árboles binarios
estructuras dinámicas de datos	parte superior de una pila

pila	recorrido postorden de un árbol binario
pop	recorrido preorder de un árbol binario
primero en entrar, primero en salir (PEPS)	spooler
push	subárbol derecho
rabo de una cola	subárbol izquierdo
recorrido inorden de un árbol binario	último en entrar, primero en salir (UEPS)
recorrido por orden de nivel	

Ejercicios de autoevaluación

20.1 Llene los espacios en blanco en cada uno de los siguientes enunciados:

- a) Una clase auto _____ se utiliza para formar estructuras de datos dinámicas que pueden crecer y reducirse en tiempo de ejecución.
- b) El operador _____ se utiliza para asignar memoria en forma dinámica y construir un objeto; este operador devuelve un apuntador al objeto.
- c) Una _____ es una versión restringida de una lista enlazada, en la que pueden insertarse y eliminarse nodos solamente desde el principio de la lista, y los valores de los nodos se devuelven en orden “último en entrar, primero en salir”.
- d) Una función que no altera una lista enlazada, sino que sólo la analiza para determinar si está vacía, es un ejemplo de una función _____.
- e) A una cola se le conoce como estructura de datos _____, ya que los primeros nodos que se insertan son los primeros que se eliminan.
- f) El apuntador al siguiente nodo en una lista enlazada se conoce como un _____.
- g) El operador _____ se utiliza para destruir un objeto y liberar la memoria asignada en forma dinámica.
- h) Una _____ es una versión restringida de una lista enlazada, en la que pueden insertarse nodos al final de la lista y eliminarse solamente desde el principio.
- i) Un _____ es una estructura de datos bidimensional no lineal, que contiene nodos con dos o más enlaces.
- j) A una pila se le conoce como estructura de datos _____, ya que el último nodo insertado es el primero que se elimina.
- k) Los nodos de un árbol _____ contienen dos miembros de enlace.
- l) El primer nodo de un árbol es el nodo _____.
- m) Cada enlace en el nodo de un árbol apunta a un _____ o _____ de ese nodo.
- n) El nodo de un árbol que no tiene hijos se llama nodo _____.
- o) Los cuatro algoritmos de recorrido que mencionamos en el texto para los árboles de búsqueda binaria son _____, _____, _____ y _____.

20.2 ¿Cuáles son las diferencias entre una lista enlazada y una pila?

20.3 ¿Cuáles son las diferencias entre una pila y una cola?

20.4 Tal vez un título más apropiado para este capítulo hubiera sido “Estructuras de datos reutilizables”. Escriba sus comentarios acerca de cómo contribuyen cada una de las siguientes entidades o conceptos a la reutilización de las estructuras de datos:

- a) clases
- b) plantillas de clases
- c) herencia
- d) herencia **private**
- e) composición

20.5 Proporcione manualmente los recorridos inorden, preorder y postorden del árbol de búsqueda binaria de la figura 20.24.

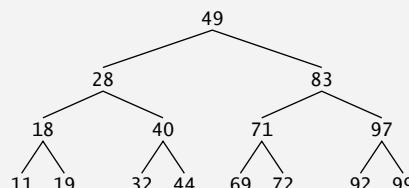


Figura 20.24 | Un árbol de búsqueda binaria con 15 nodos.

Respuestas a los ejercicios de autoevaluación

20.1 a) referenciada. b) new. c) pila. d) predicado. e) PEPS (primero en entrar, primero en salir). f) enlace. g) delete. h) cola. i) árbol. j) UEPS (último en entrar, primero en salir). k) binario. l) raíz. m) hijo o subárbol. n) hoja. o) inorden, preorden, postorden y orden de nivel.

20.2 Es posible insertar y eliminar un nodo en cualquier lugar de una lista enlazada. Los nodos en una pila pueden insertarse solamente en la parte superior y eliminarse desde la parte superior de una pila.

20.3 Una estructura de datos tipo cola permite eliminar nodos sólo desde su cabeza, e insertarlos sólo en su rabo. Una cola se conoce como estructura de datos PEPS (primero en entrar, primero en salir). Una estructura de datos tipo pila permite agregar nodos a la pila y eliminarlos de ésta sólo desde su parte superior. Una pila se conoce como estructura de datos UEPS (último en entrar, primero en salir).

- 20.4**
 - a) Las clases nos permiten crear tantas instancias de todos los objetos de estructura de datos de cierto tipo (es decir, clase) como sea necesario.
 - b) Las plantillas de clases nos permiten instanciar clases relacionadas, cada una de ellas basada en distintos parámetros de tipo; así, podemos generar tantos objetos de cada clase de plantilla como sea necesario.
 - c) La herencia nos permite reutilizar código de una clase base en una clase derivada, de manera que la estructura de datos de la clase derivada también sea una estructura de datos de la clase base (esto es, con herencia `public`).
 - d) La herencia privada nos permite reutilizar partes del código de una clase base para formar una estructura de datos de clase derivada; debido a que la herencia es `private`, todas las funciones miembro `public` de la clase base se convierten en `private` en la clase derivada. Esto nos permite evitar que los clientes de la estructura de datos de clase derivada accedan a las funciones miembro de la clase base que no son aplicables a la clase derivada.
 - e) La composición nos permite reutilizar código al hacer que una estructura de datos de un objeto de una clase sea miembro de una clase compuesta; si hacemos al objeto de la clase un miembro `private` de la clase compuesta, entonces las funciones miembro `public` del objeto de la clase no están disponibles a través de la interfaz del objeto.

20.5 El recorrido inorden es:

11 18 19 28 32 40 44 49 69 71 72 83 92 97 99

El recorrido preorden es:

49 28 18 11 19 40 32 44 83 71 69 72 97 92 99

El recorrido postorden es:

11 19 18 32 44 40 28 69 72 71 92 99 97 83 49

Ejercicios

20.6 Escriba un programa para concatenar dos objetos de lista enlazada de caracteres. El programa deberá incluir la función `concatenar`, que reciba referencias a ambos objetos lista como argumentos y que concatene la segunda lista con la primera.

20.7 Escriba un programa para combinar dos objetos de lista ordenada de enteros en un solo objeto de lista ordenada de enteros. La función `combinar` debe recibir referencias a cada uno de los objetos lista que se van a combinar, y debe devolver una referencia a objeto lista en el que se colocarán los elementos combinados.

20.8 Escriba un programa para insertar 25 enteros aleatorios de 0 a 100 en orden, en un objeto lista enlazada. El programa deberá calcular la suma de los elementos y el promedio de punto flotante de los elementos.

20.9 Escriba un programa para crear un objeto lista enlazada de 10 caracteres, y que luego cree un segundo objeto lista que contenga una copia de la primera lista, pero en orden inverso.

20.10 Escriba un programa que reciba una línea de texto como entrada y que utilice un objeto pila para imprimir la línea en orden inverso.

20.11 Escriba un programa que utilice un objeto pila para determinar si una cadena es una palíndroma (es decir, que la cadena se deletree en forma idéntica, tanto al revés como al derecho). El programa debe ignorar espacios y puntuación.

20.12 Los compiladores utilizan pilas para ayudar en el proceso de evaluar expresiones y generar código en lenguaje máquina. En este ejercicio y en el siguiente, investigaremos cómo los compiladores evalúan expresiones aritméticas que consisten solamente de constantes, operadores y paréntesis.

Los humanos generalmente escriben expresiones como $3 + 4 \cdot 7 / 9$, en donde el operador ($+$ o $/$ aquí) se escribe entre sus operandos; a esta notación se le conoce como **notación infijo**. Las computadoras “prefieren” la **notación postfijo**, en donde el operador se escribe a la derecha de sus dos operandos. Las anteriores expresiones infijo aparecerían en notación postfijo como $3\ 4\ +\ 7\ 9\ /$, respectivamente.

Para evaluar una expresión infijo compleja, un compilador primero convertiría la expresión en notación postfijo y evaluaría la versión postfijo de la expresión. Cada uno de estos algoritmos requiere solamente de una pasada de izquierda a derecha de la expresión. Cada algoritmo utiliza un objeto pila para dar soporte a su operación y, en cada algoritmo, la pila se utiliza para un propósito distinto.

En este ejercicio, usted escribirá una versión en C++ del algoritmo de conversión infijo a postfijo. En el siguiente ejercicio, usted escribirá una versión en C++ del algoritmo de evaluación de expresiones postfijo. En un ejercicio posterior, descubrirá que el código que escriba en este ejercicio podrá ayudarle a implementar un compilador completamente funcional.

Escriba un programa para convertir una expresión aritmética infijo ordinaria (suponga que se escribe una expresión válida) con enteros de un solo dígito, como:

$(6 + 2) * 5 - 8 / 4$

a una expresión postfijo. La versión postfijo de la expresión infijo anterior es:

$6\ 2\ +\ 5\ *\ 8\ 4\ /\ -$

El programa debe leer la expresión y colocarla en el arreglo de caracteres `infijo`, y utilizar las versiones modificadas de las funciones de la clase `pila`, implementadas en este capítulo para ayudar a crear la expresión postfijo en el arreglo de caracteres `postfijo`. El algoritmo para crear una expresión postfijo es el siguiente:

1) Meter un paréntesis izquierdo '(' en la pila.

2) Agregar un paréntesis derecho ')' al final de `infijo`.

3) Mientras que la pila no esté vacía, leer `infijo` de izquierda a derecha y hacer lo siguiente:

Si el carácter actual en `infijo` es un dígito, copiarlo al siguiente elemento de `postfijo`.

Si el carácter actual en `infijo` es un paréntesis izquierdo, meterlo a la pila.

Si el carácter actual en `infijo` es un operador,

Sacar los operadores (si los hay) de la parte superior de la pila, mientras tengan igual o mayor precedencia que el operador actual, e insertar en `postfijo` los operadores que se sacaron.

Meter en la pila el carácter actual en `infijo`.

Si el carácter actual en `infijo` es un paréntesis derecho:

Sacar operadores de la parte superior de la pila e insertarlos en `postfijo`, hasta que haya un paréntesis izquierdo en la parte superior de la pila.

Sacar (y descartar) el paréntesis izquierdo de la pila.

Las siguientes operaciones aritméticas se permiten en una expresión:

- + suma
- resta
- * multiplicación
- / división
- ^ exponenciación
- % módulo

[Nota: asumimos la asociatividad de izquierda a derecha para todos los operadores, para los fines de este ejercicio]. La pila debe mantenerse con nodos de pila que contengan, cada uno, un miembro de datos y un apuntador al siguiente nodo de la pila.

Algunas de las herramientas funcionales que puede ser conveniente proporcionar son:

- a) la función `convertirAPostfijo`, que convierte la expresión infijo a notación postfijo
- b) la función `esOperador`, el cual determina si c es un operador
- c) la función `precedencia`, que determina si la precedencia de `operador1` es menor, igual o mayor que la precedencia de `operador2` (la función devuelve -1, 0 y 1, respectivamente)
- d) la función `push`, que mete un valor en la pila
- e) la función `pop`, que saca un valor de la pila
- f) la función `parteSuperiorPila`, que devuelve el valor de la parte superior de la pila sin sacarlo de la misma
- g) la función `estaVacia`, que determina si la pila está vacía
- h) la función `imprimirPila`, que imprime la pila

20.13 Escriba un programa para evaluar una expresión postfijo (asuma que es válida) tal como:

$6\ 2\ +\ 5\ *\ 8\ 4\ /\ -$

El programa debe leer una expresión postfijo que consista de dígitos y operadores, para después colocarla en un arreglo de caracteres. Utilizando versiones modificadas de las funciones de la clase `pila` implementadas anteriormente en este capítulo, el programa deberá explorar la expresión y evaluarla. El algoritmo es el siguiente:

- 1) Adjuntar el carácter nulo ('\0') al final de la expresión postfijo. Al encontrar el carácter nulo, ya no habrá nada más qué procesar.

- 2) Mientras no se encuentre el carácter '\0', leer la expresión de izquierda a derecha.

Si el carácter actual es un dígito,

Meter su valor entero en la pila (el valor entero de un carácter tipo dígito es su valor en el conjunto de caracteres de la computadora menos el valor de '0' en el conjunto de caracteres de la computadora).

En caso contrario, si el carácter actual es un *operador*:

Sacar los dos elementos superiores de la pila y colocarlos en las variables *x* y *y*.

Calcular y *operador x*.

Meter el resultado del cálculo en la pila.

- 3) Al encontrar el carácter nulo en la expresión, sacar el valor superior de la pila. Éste es el resultado de la expresión postfijo.

[Nota: en el *paso 2* anterior, si el operador es '/', el valor superior de la pila es 2 y el siguiente elemento en la pila es 8, entonces sacar 2 y colocarlo en *x*, sacar 8 y colocarlo en *y*, evaluar $8 / 2$ y meter el resultado (4) de vuelta en la pila. Esta nota también se aplica al operador '-']. Las operaciones aritméticas permitidas en una expresión son:

- + suma
- resta
- * multiplicación
- / división
- \wedge exponenciación
- % módulo

[Nota: vamos a suponer la asociatividad de izquierda a derecha para todos los operadores, para los fines de este ejercicio]. La pila debe mantenerse con nodos de la pila que contengan un miembro de datos *int* y un apuntador al siguiente nodo de la pila. Tal vez usted pueda proporcionar las siguientes herramientas funcionales:

- a) la función *evaluarExpresionPostfijo*, que evalúa la expresión postfijo.
- b) la función *calcular*, que evalúa la expresión *op1 operador op2*.
- c) la función *push*, que mete un valor en la pila.
- d) la función *pop*, que saca un valor de la pila.
- e) la función *estaVacia*, que determina si la pila está vacía.
- f) la función *imprimirPila*, la cual imprime la pila.

20.14 Modifique el programa evaluador de expresiones postfijo del ejercicio 20.13, de manera que pueda procesar operandos enteros mayores que 9.

20.15 (*Simulación de supermercado*) Escriba un programa que simule una línea para pagar en un supermercado. La línea es un objeto cola. Los clientes (es decir, los objetos cliente) llegan en intervalos enteros aleatorios de 1 a 4 minutos. Además, a cada cliente se le atiende en intervalos enteros aleatorios de 1 a 4 minutos. Obviamente, los ritmos necesitan balancearse. Si el ritmo promedio de llegadas es mayor que el ritmo promedio de atención, la cola crecerá infinitamente. Incluso con ritmos "balanceados", el factor aleatorio puede aún provocar largas líneas. Ejecute la simulación del supermercado durante un día de 12 horas (720 minutos), utilizando el siguiente algoritmo:

- 1) Elegir un entero aleatorio entre 1 y 4 para determinar el minuto en el que debe llegar el primer cliente.

- 2) Al momento en que llegue el cliente:

Determinar el tiempo de atención del cliente (entero aleatorio de 1 a 4).

Empezar a atender al cliente.

Programar la hora de llegada del siguiente cliente (se suma un entero aleatorio de 1 a 4 al tiempo actual).

- 3) Para cada minuto del día:

Si llega el siguiente cliente,

Decirlo así.

Poner al cliente en la cola.

Programar la hora de llegada del siguiente cliente.

Si se terminó de atender al último cliente,

Decirlo así.

Sacar de la cola al siguiente cliente al que se va a atender.

Determinar el tiempo requerido para dar servicio al cliente

(se suma un entero aleatorio del 1 al 4 al tiempo actual).

Ahora ejecute su simulación durante 720 minutos y responda a cada una de las siguientes preguntas:

- a) ¿Cuál es el máximo número de clientes en la cola, en cualquier momento dado?
- b) ¿Cuál es el tiempo de espera más largo que experimenta un cliente?
- c) ¿Qué ocurre si el intervalo de llegada se cambia de 1 a 4 minutos por un intervalo de 1 a 3 minutos?

20.16 Modifique el programa de las figuras 20.20 a 20.22 para permitir que el objeto árbol binario contenga valores duplicados.

20.17 Escriba un programa con base en las figuras 20.20 a 20.22, que reciba como entrada una línea de texto, divida la oración en palabras separadas (tal vez quiera utilizar función `strtok` de la biblioteca), inserte las palabras en un árbol de búsqueda binaria e imprima los recorridos inorden, preorden y postorden del árbol. Use una metodología de POO.

20.18 En este capítulo vimos que la eliminación de duplicados es un proceso bastante simple cuando se crea un árbol de búsqueda binaria. Describa cómo llevaría a cabo la eliminación de duplicados utilizando sólo un arreglo unidimensional. Compare el rendimiento de la eliminación de valores duplicados con base en arreglos y el rendimiento de la eliminación de duplicados con base en árboles de búsqueda binaria.

20.19 Escriba una función llamada `profundidad` que reciba un árbol binario y determine cuántos niveles tiene.

20.20 (*Imprimir una lista en forma inversa mediante recursividad*) Escriba una función miembro llamada `imprimirListaAlReves` que imprima en forma recursiva los elementos en un objeto lista enlazada, en orden inverso. Escriba un programa de prueba para crear una lista ordenada de enteros e imprimir la lista en orden inverso.

20.21 (*Buscar en una lista en forma recursiva*) Escriba una función llamada `buscarLista` que busque en forma recursiva en un objeto lista enlazada un valor específico. La función deberá devolver un apuntador al valor, si es que lo encuentra; en caso contrario, deberá devolver `null`. Use su función en un programa de prueba para crear una lista de enteros. El programa deberá pedir al usuario un valor a localizar en la lista.

20.22 (*Eliminación en árboles binarios*) En este ejercicio hablaremos sobre cómo eliminar elementos de los árboles de búsqueda binaria. El algoritmo de eliminación no es tan simple como el de inserción. Al eliminar un elemento puede haber tres casos: que el elemento esté contenido en un nodo hoja (es decir, que no tenga hijos), que esté contenido en un nodo que tenga un hijo, o que esté contenido en un nodo con dos hijos.

Si el elemento que se va a eliminar está contenido en un nodo hoja, este nodo se elimina y al apuntador en el nodo padre se le asigna el valor nulo.

Si el elemento que se va a eliminar está contenido en un nodo con un hijo, al apuntador en el nodo padre se le asigna el nodo hijo y se elimina el nodo que contenga el elemento de datos. Esto hace que el nodo hijo ocupe el lugar del nodo eliminado en el árbol.

El último caso es el más difícil. Cuando se elimina un nodo con dos hijos, otro nodo en el árbol debe tomar su lugar. Sin embargo, el apuntador en el nodo padre no puede simplemente asignarse de manera que apunte a uno de los hijos del nodo que se va a eliminar. En la mayoría de los casos, el árbol de búsqueda binaria resultante no se adhiere a la siguiente característica de los árboles de búsqueda binaria (sin valores duplicados): *los valores en cualquier subárbol izquierdo son menores que el valor en el nodo padre, y los valores en cualquier subárbol derecho son mayores que el valor en el nodo padre*.

¿Cuál nodo debe utilizarse como *nodo de reemplazo* para mantener esta característica? Debe ser el nodo que contenga el valor más grande en el árbol, pero que sea menor que el valor en el nodo que se va a eliminar, o el nodo que contenga el valor más pequeño en el árbol, pero que sea mayor que el valor en el nodo que se va a eliminar. Consideremos el nodo con el valor más pequeño. En un árbol de búsqueda binaria, el valor más grande que sea menor que el valor de un parente se encuentra en el subárbol izquierdo del nodo padre y se garantiza que estará contenido en el nodo que se encuentre más a la derecha del subárbol. Para localizar este nodo hay que avanzar por el subárbol izquierdo hacia abajo y a la derecha, hasta que el apuntador al hijo derecho del nodo actual sea nulo. Ahora estamos apuntando al nodo de reemplazo, que es un nodo hoja o un nodo con un hijo a su izquierda. Si el nodo de reemplazo es un nodo hoja, los pasos para llevar a cabo la eliminación son los siguientes:

- 1) Almacenar el apuntador al nodo que se va a eliminar en una variable apuntador temporal (este apuntador se utiliza para eliminar la memoria asignada en forma dinámica).
- 2) Hacer que el apuntador en el parente del nodo que se va a eliminar apunte al nodo de reemplazo.
- 3) Asignar al apuntador en el parente del nodo de reemplazo el valor nulo.
- 4) Hacer que el apuntador al subárbol derecho en el nodo de reemplazo apunte al subárbol derecho del nodo que se va a eliminar.
- 5) Eliminar el nodo al que apunta la variable apuntador temporal.

Los pasos de eliminación para el caso de un nodo de reemplazo con un hijo izquierdo son similares a los pasos para un nodo de reemplazo sin hijos, sólo que el algoritmo debe también desplazar al hijo hacia la posición del nodo de reemplazo en el árbol. Si el nodo de reemplazo es un nodo con un hijo izquierdo, los pasos para llevar a cabo la eliminación son los siguientes:

- 1) Almacenar el apuntador al nodo que se va a eliminar en una variable apuntador temporal.
- 2) Hacer que el apuntador en el parente del nodo que se va a eliminar apunte al nodo de reemplazo.
- 3) Hacer que el apuntador en el parente del nodo de reemplazo apunte al hijo izquierdo del nodo de reemplazo.
- 4) Hacer que el apuntador al subárbol derecho en el nodo de reemplazo apunte al subárbol derecho del nodo que se va a eliminar.
- 5) Eliminar el nodo al que apunta la variable apuntador temporal.

Escriba la función miembro `eliminarNodo`, que debe recibir como argumentos un apuntador al nodo raíz del objeto y el valor a eliminar. La función debe localizar en el árbol el nodo que contenga el valor a eliminar, y debe utilizar los algoritmos aquí descritos para eliminar el nodo. La función debe imprimir un mensaje que indique si el valor se eliminó. Modifique el programa de las figuras 20.20 a 20.22 para utilizar esta función. Después de eliminar un elemento, llame a las funciones `inOrden`, `preOrden` y `postOrden` para confirmar que la operación de eliminación se haya llevado a cabo correctamente.

20.23 (Búsqueda en un árbol binario) Escriba la función miembro `busquedaArbolBinario`, para tratar de localizar un valor especificado en un objeto árbol de búsqueda binaria. La función debe recibir como argumentos un apuntador al nodo raíz del árbol binario y una clave de búsqueda a localizar. Si se encuentra el nodo que contiene la clave de búsqueda, la función debe devolver un apuntador a ese nodo; en caso contrario, la función debe devolver un apuntador nulo.

20.24 (Recorrido de un árbol binario por orden de nivel) El programa de las figuras 20.20 a 20.22 demostró el uso de tres métodos recursivos para recorrer un árbol binario: los recorridos inorden, preorden y postorden. En este ejercicio presentamos el *recorrido por orden de nivel* de un árbol binario, en el cual los valores de los nodos se imprimen nivel por nivel, empezando en el nivel del nodo raíz. Los nodos en cada nivel se imprimen de izquierda a derecha. El recorrido por orden de nivel no es un algoritmo recursivo. Utiliza un objeto cola para controlar la impresión en pantalla de los nodos. El algoritmo es el siguiente:

- 1) Insertar el nodo raíz en la cola.
- 2) Mientras haya nodos restantes en la cola:
 - Obtener el siguiente nodo en la cola
 - Imprimir el valor del nodo
 - Si el apuntador al hijo izquierdo del nodo no es nulo
 - Insertar el nodo del hijo izquierdo en la cola
 - Si el apuntador al hijo derecho del nodo no es nulo
 - Insertar el nodo del hijo derecho en la cola

Escriba la función miembro `ordenNivel` para llevar a cabo un recorrido por orden de nivel de un objeto árbol binario. Modifique el programa de las figuras 20.20 a 20.22 para utilizar esta función. [Nota: también necesitará modificar e incorporar las funciones de procesamiento de colas de la figura 20.16 en este programa].

20.25 (Imprimir árboles) Escriba una función miembro recursiva llamada `mostrarArbol` para desplegar un objeto árbol binario en la pantalla. La función deberá mostrar el árbol fila por fila, con la parte superior del mismo a la izquierda de la pantalla y la parte inferior hacia la derecha de la pantalla. Cada fila se debe mostrar en forma vertical. Por ejemplo, el árbol binario que aparece en la figura 20.24 debe mostrarse en pantalla como se indica en la figura 20.25. Observe que el nodo hoja que se encuentra más a la derecha en el árbol aparece en la parte superior de la pantalla, en la columna que está más a la derecha, y el nodo raíz aparece a la izquierda de la pantalla. Cada columna de salida empieza cinco espacios a la derecha de la columna anterior. La función `mostrarArbol` debe recibir un argumento llamado `totalEspacios`, el cual representa el número de espacios que anteceden al valor que va a mostrarse en pantalla (esta variable debe empezar en cero, para que el nodo raíz se muestre a la izquierda de la pantalla). La función utiliza un recorrido inorden modificado para mostrar el árbol en pantalla; empieza en el nodo que está más a la derecha en el árbol y avanza en retroceso hacia la izquierda. El algoritmo es el siguiente:

		99
	97	92
83		72
	71	69
49		44
	40	32
28		19
	18	11

Figura 20.25 | Resultados del uso de `mostrarArbol` para mostrar el árbol de la figura 20.24.

Mientras el apuntador al nodo actual no sea nulo:

- Llamar en forma recursiva a `mostrarArbol` con el subárbol derecho del nodo actual y `totalEspacios + 5`
- Utilizar una instrucción `for` para contar de 1 a `totalEspacios` e imprimir espacios
- Mostrar el valor en el nodo actual
- Hacer que el apuntador al nodo actual apunte al subárbol izquierdo del nodo actual
- Incrementar `totalEspacios` en 5.

Sección especial: construya su propio compilador

En los ejercicios 8.18 y 8.19 presentamos el Lenguaje máquina Simpletron (LMS) y usted implementó un simulador de computadora Simpletron para ejecutar programas escritos en LMS. En esta sección crearemos un compilador que convierta los programas escritos en un lenguaje de programación de alto nivel a LMS. Esta sección “enlaza” entre sí todo el proceso de programación. Usted escribirá programas en este nuevo lenguaje de alto nivel, los compilará en el compilador que va a construir y los ejecutará en el simulador que construyó en el ejercicio 8.19. Usted deberá hacer todo el esfuerzo posible por implementar su compilador con un enfoque orientado a objetos.

20.26 (El lenguaje Simple) Antes de empezar a construir el compilador, hablaremos sobre un lenguaje de alto nivel simple pero poderoso, similar a las primeras versiones del popular lenguaje BASIC. Llamaremos a este lenguaje *Simple*. Cada *instrucción* de Simple consiste de un *número de línea* y de una *instrucción* de Simple. Los números de línea deben aparecer en orden ascendente. Cada instrucción empieza con uno de los siguientes *comandos* de Simple: `rem`, `input`, `let`, `print`, `goto`, `if...goto` y `end` (vea la figura 20.26). Todos los comandos excepto `end` pueden utilizarse en forma repetida. Simple evalúa solamente las expresiones de enteros que utilizan los operadores `+`, `-`, `*` y `/`. Estos operadores tienen la misma precedencia que en C++. Pueden utilizarse paréntesis para cambiar el orden de evaluación de una expresión.

Nuestro compilador de Simple reconoce solamente letras en minúscula. Todos los caracteres en un archivo de Simple deben estar en minúsculas (las letras mayúsculas producen un error de sintaxis, a menos que aparezcan en una instrucción `rem`, en cuyo caso se ignoran). Un *nombre de variable* es una sola letra. Simple no permite el uso de nombres descriptivos para las variables, por lo que éstas deben explicarse en comentarios para indicar su uso en un programa. Simple utiliza solamente variables enteras. Simple no tiene declaraciones de variables; con sólo mencionar el nombre de una variable en un programa, ésta se declara y se inicializa con cero automáticamente. La sintaxis de Simple no permite la manipulación de cadenas (leer una cadena, escribir una cadena, comparar cadenas, etcétera). Si se encuentra una cadena en un programa de Simple (después de un comando distinto de `rem`), el compilador genera un error de sintaxis. La primera versión de nuestro compilador supone que los programas de Simple se introducen correctamente. En el ejercicio 20.29 pedimos al lector que modifique el compilador para llevar a cabo la comprobación de errores de sintaxis.

Simple utiliza la instrucción `if...goto` condicional y la instrucción `goto` incondicional para alterar el flujo de control durante la ejecución del programa. Si la condición en la instrucción `if...goto` es verdadera, el control se transfiere a una línea específica del programa. Los siguientes operadores relacionales y de igualdad son válidos en una instrucción `if...goto`: `<`, `>`, `<=`, `>=`, `==` y `!=`. La precedencia de estos operadores es la misma que en C++.

Consideremos ahora varios programas para demostrar las características de Simple. El primer programa (figura 20.27) lee dos enteros del teclado, almacena los valores en las variables `a` y `b`, calcula e imprime su suma (almacenada en la variable `c`).

El programa de la figura 20.28 determina e imprime el mayor de dos enteros. Los enteros se introducen desde el teclado y se almacenan en `s` y `t`. La instrucción `if...goto` evalúa la condición `s >= t`. Si es verdadera, el control se transfiere a la línea 90 y se muestra el valor de `s` en pantalla; en caso contrario se muestra `t` y el control se transfiere a la instrucción `end` de la línea 99, en donde termina el programa.

Comando	Instrucción de ejemplo	Descripción
<code>rem</code>	<code>50 rem este es un comentario</code>	Cualquier texto después del comando <code>rem</code> es para fines de documentación solamente, por lo que el compilador lo ignora.
<code>input</code>	<code>30 input x</code>	Mostrar un signo de interrogación para pedir al usuario que introduzca un entero. Leer ese entero desde el teclado y almacenarlo en <code>x</code> .
<code>let</code>	<code>80 let u = 4 * (j - 56)</code>	Asignar a <code>u</code> el valor de <code>4 * (j - 56)</code> . Observe que puede aparecer una expresión arbitrariamente compleja a la derecha del signo de igual.
<code>print</code>	<code>10 print w</code>	Mostrar el valor de <code>w</code> .
<code>goto</code>	<code>70 goto 45</code>	Transferir el control del programa a la línea 45.
<code>If...goto</code>	<code>35 if i == z goto 80</code>	Comparar si <code>i</code> y <code>z</code> son iguales y transferir el control del programa a la línea 80 si la condición es verdadera; en caso contrario, continuar la ejecución con la siguiente instrucción.
<code>end</code>	<code>99 end</code>	Terminar la ejecución del programa.

Figura 20.26 | Comandos de Simple.

```

1 10 rem  determinar e imprimir la suma de dos enteros
2 15 rem
3 20 rem  introducir los dos enteros
4 30 input a
5 40 input b
6 45 rem
7 50 rem  sumar los enteros y almacenar el resultado en c
8 60 let c = a + b
9 65 rem
10 70 rem  imprimir el resultado
11 80 print c
12 90 rem  terminar la ejecución del programa
13 99 end

```

Fig. 20.27 | Programa de Simple que determina la suma de dos enteros.

```

1 10 rem  determinar e imprimir el mayor de dos enteros
2 20 input s
3 30 input t
4 32 rem
5 35 rem  evaluar si s >= t
6 40 if s >= t goto 90
7 45 rem
8 50 rem  t es mayor que s, por lo que se imprime t
9 60 print t
10 70 goto 99
11 75 rem
12 80 rem  s es mayor o igual que t, por lo que se imprime s
13 90 print s
14 99 end

```

Fig. 20.28 | Programa de Simple que encuentra el mayor de dos enteros.

Simple no cuenta con una instrucción de repetición (como las instrucciones `for`, `while` o `do...while` de C++). Sin embargo, Simple puede simular cada una de las instrucciones de repetición de C++ mediante el uso de las instrucciones `if...goto`. En la figura 20.29 se utiliza un ciclo controlado por centinela para calcular los cuadrados de varios enteros. Cada entero se introduce desde el teclado y se almacena en la variable `j`. Si el valor introducido es el valor centinela -9999, el control se transfiere a la línea 99, en donde termina el programa. En caso contrario, a `k` se le asigna el cuadrado de `j`, `k` se muestra en pantalla y el control se pasa a la línea 20, en donde se introduce el siguiente entero.

Utilizando los programas de ejemplo de las figuras 20.27, 20.28 y 20.29 como guía, escriba un programa de Simple para realizar cada una de las siguientes acciones:

- Introducir tres enteros, determinar su promedio e imprimir el resultado.
- Usar un ciclo controlado por centinela para introducir 10 enteros, calcular e imprimir su suma.
- Usar un ciclo controlado por centinela para introducir siete enteros, algunos positivos y otros negativos, calcular e imprimir su promedio.
- Introducir una serie de enteros, determinar e imprimir el mayor. El primer entero introducido indica cuántos números deben procesarse.
- Introducir 10 enteros e imprimir el menor.
- Calcular e imprimir la suma de los enteros pares del 2 al 30.
- Calcular e imprimir el producto de los enteros impares del 1 al 9.

```

1 10 rem  calcular los cuadrados de varios enteros
2 20 input j
3 23 rem
4 25 rem  evaluar el valor centinela
5 30 if j == -9999 goto 99
6 33 rem
7 35 rem  calcular el cuadrado de j y asignar el resultado a k
8 40 let k = j * j
9 50 print k
10 53 rem
11 55 rem  iterar para obtener el siguiente valor de j
12 60 goto 20
13 99 end

```

Figura 20.29 | Calcular los cuadrados de varios enteros.

20.27 (*Construcción de un compilador; prerequisitos: completar los ejercicios 8.18, 8.19, 21.12, 21.13 y 21.26*) Ahora que hemos presentado el lenguaje Simple (ejercicio 20.26), hablaremos sobre cómo construir un compilador de Simple. Primero debemos considerar el proceso mediante el cual un programa de Simple se convierte a LMS y se ejecuta por el simulador Simpletron (vea la figura 20.30). El compilador lee un archivo que contiene un programa de Simple y lo convierte en código de LMS. Este código se envía a un archivo en disco, en el que las instrucciones de LMS aparecen una en cada línea. Después el archivo de LMS se carga en el simulador Simpletron y los resultados se envían a un archivo en disco y a la pantalla. Observe que el programa de Simpletron desarrollado en el ejercicio 8.19 acepta su entrada mediante el teclado. Este programa debe modificarse para que lea desde un archivo y así pueda ejecutar los programas producidos por nuestro compilador.

El compilador de Simple realiza dos *pasadas* del programa de Simple para convertirlo en LMS. En la primera pasada se construye una *tabla de símbolos* (objeto) en la que cada *número de línea* (objeto), *nombre de variable* (objeto) y *constante* (objeto) del programa de Simple se guarda con su tipo y ubicación correspondiente en el código final de LMS (la tabla de símbolos se describe detalladamente a continuación). En la primera pasada también se produce(n) el (los) objeto(s) de instrucción correspondientes para cada una de las instrucciones de Simple (objeto, etcétera). Como veremos, si el programa de Simple contiene instrucciones que transfieren el control a una línea que se encuentra más adelante en el programa, la primera pasada produce un programa de LMS que contiene algunas instrucciones “no terminadas”. En la segunda pasada del compilador se localizan y completan las instrucciones no terminadas, y se envía el programa de LMS a un archivo.

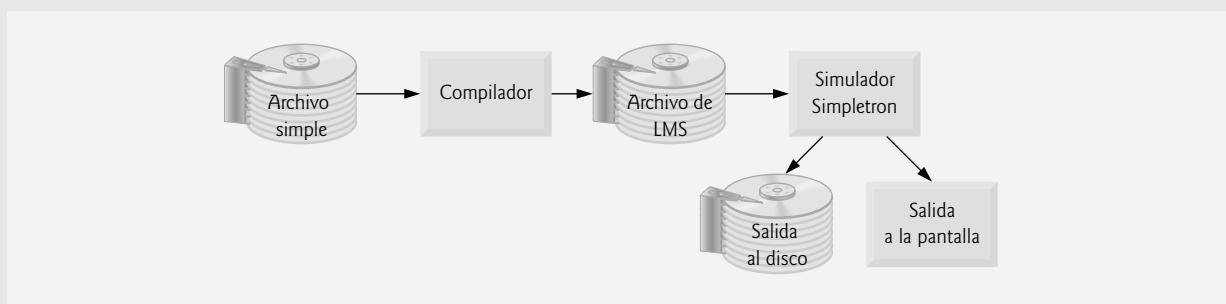


Figura 20.30 | Cómo escribir, compilar y ejecutar un programa en lenguaje Simple.

Primera pasada

El compilador empieza leyendo una instrucción del programa de Simple y la coloca en memoria. La línea debe separarse en sus *tokens* individuales (es decir, “piezas” de una instrucción) para su procesamiento y compilación (para facilitar esta tarea, puede utilizar la función `strtok` de la biblioteca estándar). Recuerde que todas las instrucciones empiezan con un número de línea, seguido de un comando. A medida que el compilador divide una instrucción en tokens, si éste es un número de línea, una variable o una constante, se coloca en la tabla de símbolos. Un número de línea se coloca en la tabla de símbolos solamente si es el primer token en una instrucción. El objeto `tablaDeSimbolos` es un arreglo de objetos `entradaTabla` que representan a cada uno de los símbolos en el programa. No hay restricción en cuanto al número de símbolos que pueden aparecer en el programa. Por lo tanto, la `tablaDeSimbolos` para un programa específico podría ser extensa. Por ahora, haga que la `tablaDeSimbolos` sea un arreglo de 100 elementos. Usted podrá incrementar o decrementar su tamaño una vez que el programa esté ejecutándose.

Cada objeto `entradaTabla` contiene tres miembros. El miembro `símbolo` es un entero que contiene la representación ASCII de una variable (recuerde que los nombres de las variables son caracteres individuales), un número de línea o una constante. El miembro `tipo` es uno de los siguientes caracteres que indican el tipo de ese símbolo: 'C' para constante, 'L' para número de línea y 'V' para variable. El miembro `ubicacion` contiene la ubicación de memoria Simpletron (00 a 99) a la que hace referencia el símbolo. La memoria Simpletron es un arreglo de 100 enteros en donde se almacenan instrucciones y datos de LMS. Para un número de línea, la ubicación es el elemento en el arreglo de memoria Simpletron en el que empiezan las instrucciones de LMS para la instrucción de Simple. Para una variable o constante, la ubicación es el elemento en el arreglo de memoria Simpletron en el que se almacena esa variable o constante. Las variables y constantes se asignan desde el final de la memoria Simpletron hacia atrás. La primera variable o constante se almacena en la ubicación 99, la siguiente en la ubicación 98 y así, sucesivamente.

La tabla de símbolos juega una parte integral para convertir los programas de Simple a LMS. En el capítulo 8 aprendimos que una instrucción de LMS es un entero de cuatro dígitos, compuesto de dos partes: el *código de la operación* y el *operando*. El código de operación se determina mediante los comandos en Simple. Por ejemplo, el comando `input` de Simple corresponde al código de operación 10 de LMS (lectura), y el comando `print` de Simple corresponde al código de operación 11 de LMS (escritura). El operando está en una ubicación de memoria que contiene los datos sobre los cuales el código de operación lleva a cabo su tarea (por ejemplo, el código de operación 10 lee un valor desde el teclado y lo guarda en la ubicación de memoria especificada por el operando). El compilador busca en la `tablaDeSimbolos` para determinar la ubicación de memoria Simpletron para cada símbolo, de manera que pueda utilizarse la ubicación correspondiente para completar las instrucciones de LMS.

La compilación de cada instrucción de Simple se basa en su comando. Por ejemplo, después de insertar el número de línea de una instrucción `rem` en la tabla de símbolos, el compilador ignora el resto de la instrucción ya que un comentario es sólo para fines de documentación. Las instrucciones `input`, `print`, `goto` y `end` corresponden a las instrucciones `leer`, `escribir`, `bifurcar` (hacia una ubicación específica) y `parar` de LMS. Las instrucciones que contienen estos comandos de Simple se convierten directamente a LMS. (observe que una instrucción `goto` puede contener una referencia no resuelta, si el número de línea especificado hace referencia a una instrucción que se encuentre más adelante en el archivo del programa de Simple; a esto se le conoce algunas veces como referencia adelantada).

Cuando una instrucción `goto` se compila con una referencia no resuelta, la instrucción de LMS debe *marcarse* para indicar que la segunda pasada del compilador debe completar la instrucción. Las banderas se almacenan en un arreglo de 100 elementos llamado `banderas` de tipo `int`, en donde cada elemento se inicializa con `-1`. Si la ubicación de memoria a la que hace referencia un número de línea en el programa de Simple no se conoce todavía (es decir, que no se encuentra en la tabla de símbolos), el número de línea se almacena en el arreglo `banderas`, en el elemento con el mismo subíndice que la instrucción incompleta. El operando de la instrucción incompleta se establece en `00` temporalmente. Por ejemplo, una instrucción de ramificación incondicional (que hace una referencia adelantada) se deja como `+4000` hasta la segunda pasada del compilador. En breve describiremos la segunda pasada del compilador.

La compilación de las instrucciones `if...goto` y `let` es más complicada que las otras instrucciones; son las únicas instrucciones que producen más de una instrucción de LMS. Para una instrucción `if...goto`, el compilador produce código para evaluar la condición y ramificar hacia otra línea, en caso de ser necesario. El resultado de la ramificación podría ser una referencia no resuelta. Cada uno de los operadores relacionales y de igualdad pueden simularse mediante el uso de las instrucciones `branch zero` y `branch negative` de LMS (o posiblemente una combinación de ambas).

Para una instrucción `let`, el compilador produce código para evaluar una expresión aritmética arbitrariamente compleja que consiste de variables y/o constantes enteras. Las expresiones deben separar cada operando y operador con espacios. En los ejercicios 20.12 y 20.13 se presentaron el algoritmo de conversión de infijo a postfijo y el algoritmo de evaluación de expresiones postfijo que utilizan los compiladores para evaluar expresiones. Antes de proseguir con su compilador, debe completar cada uno de estos ejercicios. Cuando un compilador encuentra una expresión, la convierte de notación infijo a notación postfijo y después evalúa la expresión postfijo.

¿Cómo es que el compilador produce el lenguaje máquina para evaluar una expresión que contiene variables? El algoritmo de evaluación de expresiones postfijo contiene un “gancho” en donde el compilador puede generar instrucciones de LMS, en vez de evaluar la expresión. Para permitir este “gancho” en el compilador, el algoritmo de evaluación postfijo debe modificarse para buscar en la tabla de símbolos cada símbolo que vaya encontrando (y posiblemente insertarlo), determinar la ubicación de memoria correspondiente de ese símbolo y *meter la ubicación de memoria a la pila* (en vez del símbolo). Cuando se encuentra un operador en la expresión postfijo, las dos ubicaciones de memoria que se encuentran en la parte superior de la pila se sacan y se produce el lenguaje máquina para llevar a cabo la operación, utilizando las ubicaciones de memoria como operandos. El resultado de cada subexpresión se almacena en una ubicación temporal en memoria y se mete de vuelta a la pila, de manera que pueda continuar la evaluación de la expresión postfijo. Al completarse esta evaluación postfija, la ubicación de memoria que contiene el resultado es la única ubicación que queda en la pila. Esta ubicación se saca y se generan las instrucciones de LMS para asignar el resultado a la variable que está a la izquierda de la instrucción `let`.

Segunda pasada

En la segunda pasada del compilador se llevan a cabo dos tareas: resolver cualquier referencia no resuelta y enviar el código de LMS a un archivo. La resolución de las referencias ocurre así:

- Buscar en el arreglo `banderas` una referencia no resuelta (es decir, un elemento con un valor distinto de `-1`).
- Localizar el objeto en el arreglo `tablaDeSímbolos` que contenga el símbolo almacenado en el arreglo `banderas` (asegúrese que el tipo del símbolo sea '`L`' para un número de línea).
- Insertar la ubicación de memoria del campo `ubicación` en la instrucción con la referencia no resuelta (recuerde que una instrucción que contiene una referencia no resuelta tiene el operando `00`).
- Repetir los pasos (a), (b) y (c) hasta que se llegue al final del arreglo `banderas`.

Una vez completo el proceso de resolución, todo el arreglo que contiene el código de LMS se envía a un archivo en disco, con una instrucción de LMS por línea. El simulador Simpletron puede leer este archivo para ejecutarlo (una vez que se modifique el simulador para que lea su entrada desde un archivo). El compilar su primer programa de Simple en un archivo de LMS y luego ejecutar ese archivo le dará un verdadero sentido de satisfacción personal.

Un ejemplo completo

En el siguiente ejemplo mostramos la conversión completa de un programa de Simple a LMS, como lo deberá realizar el compilador de Simple. Considere un programa de Simple que recibe un entero y suma los valores desde 1 hasta ese entero. El programa y las instrucciones de LMS producidas por la primera pasada del compilador de Simple se muestran en la figura 20.31. La tabla de símbolos construida por la primera pasada se muestra en la figura 20.32.

Programa de Simple	Ubicación e instrucción de LMS	Descripción
5 rem sumar 1 a x	ninguna	rem se ignora
10 input x	00 +1099	leer x y colocar su valor en la ubicación 99
15 rem verificar si y == x	ninguna	rem se ignora
20 if y == x goto 60	01 +2098 02 +3199 03 +4200	cargar y (98) en el acumulador restar x (99) al acumulador ramificar a ubicación no resuelta si el resultado es cero
25 rem incrementar y	ninguna	rem se ignora
30 let y = y + 1	04 +2098 05 +3097 06 +2196 07 +2096 08 +2198	cargar y en el acumulador sumar 1 (97) al acumulador almacenar en ubicación temporal 96 cargar de ubicación temporal 96 almacenar acumulador en y
35 rem sumar y al total	ninguna	rem se ignora
40 let t = t + y	09 +2095 10 +3098 11 +2194 12 +2094 13 +2195	cargar t (95) en el acumulador sumar y al acumulador almacenar en ubicación temporal 94 cargar de ubicación temporal 94 almacenar acumulador en t
45 rem ciclo con y	ninguna	rem se ignora
50 goto 20	14 +4001	ramificar a ubicación 01
55 rem mostrar resultado	ninguna	rem se ignora
60 print t	15 +1195	mostrar t en la pantalla
99 end	16 +4300	terminar la ejecución

Figura 20.31 | Las instrucciones de LMS producidas después de la primera pasada del compilador.

Símbolo	Tipo	Ubicación
5	L	00
10	L	00
'x'	V	99
15	L	01
20	L	01
'y'	V	98
25	L	04
30	L	04
1	C	97
35	L	09
40	L	09
't'	V	95
45	L	14
50	L	14
55	L	15
60	L	15
99	L	16

Figura 20.32 | Tabla de símbolos para el programa de la figura 20.31.

La mayoría de las instrucciones de Simple se convierten directamente a instrucciones de LMS individuales. Las excepciones en este programa son los comentarios, la instrucción `if...goto` en la línea 20 y las instrucciones `let`. Los comentarios no se traducen a lenguaje máquina. Sin embargo, el número de línea para un comentario se coloca en la tabla de símbolos, en caso de que se haga referencia al número de línea en una instrucción `goto` o `if...goto`. En la línea 20 del programa se especifica que, si la condición `y == x` es verdadera, el control del programa se transfiere a la línea 60. Ya que la línea 60 aparece más adelante en el programa, la primera pasada del compilador todavía no ha colocado el valor 60 en la tabla de símbolos (se colocan números de línea en la tabla de símbolos solamente cuando aparecen como el primer token en una instrucción). Por lo tanto, no es posible en este momento determinar el operando de la instrucción `branch zero` de LMS que se encuentra en la ubicación 03 del arreglo de instrucciones de LMS. El compilador coloca 60 en la ubicación 03 del arreglo banderas para indicar que la instrucción va a completarse en la segunda pasada.

Debemos llevar el registro de la ubicación de la siguiente instrucción en el arreglo de LMS, ya que no hay una correspondencia de uno a uno entre las instrucciones de Simple y las instrucciones de LMS. Por ejemplo, la instrucción `if...goto` de la línea 20 se compila en tres instrucciones de LMS. Cada vez que se produce una instrucción, debemos incrementar el *contador de instrucciones* a la siguiente ubicación en el arreglo de LMS. Hay que tener en cuenta que el tamaño de la memoria Simpletron podría presentar un problema para los programas de Simple con muchas instrucciones, variables y constantes. Es posible que el compilador se quede sin memoria. Para probar este caso, su programa debe contener un *contador de datos* para llevar un registro de la ubicación en la que se almacenará la siguiente variable o constante en el arreglo de LMS. Si el valor del contador de instrucciones es mayor que el valor del contador de datos, significa que el arreglo de LMS está lleno. En este caso, el proceso de compilación debe terminar y el compilador debe imprimir un mensaje de error, indicando que se agotó la memoria durante la compilación. Esto sirve para enfatizar que, aunque el programador está libre de la carga que representa para el compilador tener que administrar la memoria, el mismo compilador debe determinar cuidadosamente la colocación de instrucciones y datos en ella, y debe comprobar que no haya errores como el agotamiento de la memoria durante el proceso de compilación.

El proceso de compilación, paso a paso

Ahora analicemos el proceso de compilación para el programa de Simple que aparece en la figura 20.31. El compilador lee la primera línea del programa:

```
5 rem sumar 1 a x
```

en memoria. El primer token en la instrucción (el número de línea) se determina mediante el uso de `strtok` (en los capítulos 8 y 19 se describe el uso de las funciones de manipulación de cadenas estilo C de C++.) El token devuelto por `strtok` se convierte en un entero mediante el uso de `atoi`, de manera que el símbolo 5 puede localizarse en la tabla de símbolos. Si el símbolo no se encuentra, se inserta en la tabla de símbolos. Como nos encontramos en el inicio del programa y ésta es la primera línea, todavía no hay símbolos en la tabla. Por lo tanto, se inserta el 5 en la tabla de símbolos con el tipo L (número de línea) y se le asigna la primera ubicación en el arreglo de LMS (00). Aunque esta línea es un comentario, de todas formas se asigna un espacio en la tabla de símbolos para el número de línea (en caso que se haga referencia al mismo en una instrucción `goto` o `if...goto`). No se genera ninguna instrucción de LMS para una instrucción `rem`, por lo que no se incrementa el contador de instrucciones.

La instrucción:

```
10 input x
```

se divide en tokens. El número de línea 10 se coloca en la tabla de símbolos con el tipo L y se le asigna la primera ubicación en el arreglo de LMS (00 pues, como un comentario empezó el programa, el contador de instrucciones sigue siendo 00). El comando `input` indica que el siguiente token es una variable (sólo puede aparecer una variable en una instrucción `input`). Como `input` corresponde directamente a un código de operación de LMS, el compilador sólo tiene que determinar la ubicación de `x` en el arreglo de LMS. El símbolo `x` no se encuentra en la tabla de símbolos, por lo que se inserta en ésta como la representación ASCII de `x`, se le asigna el tipo V y la ubicación 99 en el arreglo de LMS (el almacenamiento de datos empieza en la ubicación 99 y se van asignando ubicaciones en forma regresiva). Ahora puede generarse el código LMS para esta instrucción. El código de operación 10 (código de operación de lectura de LMS) se multiplica por 100 y se agrega la ubicación de `x` (según lo determinado en la tabla de símbolos) para completar la instrucción. Después la instrucción se almacena en el arreglo de LMS, en la ubicación 00. El contador de instrucciones se incrementa en 1, ya que se produjo una sola instrucción de LMS.

La instrucción:

```
15 rem verificar si y == x
```

se divide en tokens. Se busca en la tabla de símbolos el número de línea 15 (el cual no se encuentra). El número de línea se inserta con el tipo L y se le asigna la siguiente ubicación en el arreglo, 01 (recuerde que las instrucciones `rem` no producen código, por lo que no se incrementa el contador de instrucciones).

La instrucción:

```
20 if y == x goto 60
```

se divide en tokens. El número de línea 20 se inserta en la tabla de símbolos y se le asigna el tipo L en la siguiente ubicación en el arreglo de LMS (01). El comando `if` indica que se va a evaluar una condición. La variable y no se encuentra en la tabla de símbolos, por lo que se inserta, se le asigna el tipo V y la ubicación 98. A continuación se generan las instrucciones de LMS para evaluar la condición. Como no hay un equivalente directo en LMS para la instrucción `if...goto`, ésta debe simularse mediante un cálculo en el que se utilicen x y y, y después debe hacerse una bifurcación con base en el resultado. Si y es igual a x el resultado de restar x y y es cero, por lo que puede utilizarse la instrucción `branch zero` con el resultado del cálculo para simular la instrucción `if...goto`. El primer paso requiere que se cargue y (de la ubicación 98 del arreglo de LMS) en el acumulador. Esto produce la instrucción 01 +2098. Luego, se resta x del acumulador. Esto produce la instrucción 02 +3199. El valor en el acumulador puede ser cero, positivo o negativo. Como el operador es ==, queremos utilizar la operación `branch zero`. Primero se busca en la tabla de símbolos la ubicación de la ramificación (60 en este caso), la cual no se encuentra. Por lo tanto, 60 se coloca en el arreglo banderas en la ubicación 03, y se genera la instrucción 03 +4200 (no podemos agregar la ubicación de la ramificación, ya que todavía no hemos asignado una ubicación a la línea 60 en el arreglo de LMS.) El contador de instrucciones se incrementa en 04.

El compilador continúa con la instrucción:

```
25 rem incrementar y
```

El número de línea 25 se inserta en la tabla de símbolos con el tipo L y se le asigna la ubicación 04 en el arreglo de LMS. No se incrementa el contador de instrucciones.

Cuando la instrucción:

```
30 let y = y + 1
```

se divide en tokens, el número de línea 30 se inserta en la tabla de símbolos con el tipo L y se le asigna la ubicación 04 en el arreglo de LMS. El comando `let` indica que la línea es una instrucción de asignación. Primero se insertan todos los símbolos de la línea en la tabla de símbolos (si no es que están ya ahí). El entero 1 se agrega a la tabla de símbolos con el tipo C y se le asigna la ubicación 97 de LMS. A continuación, el lado derecho de la asignación se convierte de notación infijo a postfixo. Luego se evalúa la expresión postfixo (y 1 +). El símbolo y se encuentra ya en la tabla de símbolos y su ubicación correspondiente en memoria se mete a la pila. El símbolo 1 también se encuentra ya en la tabla de símbolos y su ubicación correspondiente en memoria se mete a la pila. Al llegar al operador +, el evaluador de expresiones postfixo saca el elemento superior de la pila y lo coloca en el operando derecho del operador, saca de nuevo el elemento superior de la pila, lo coloca en el operando izquierdo del operador y produce las siguientes instrucciones de LMS:

```
04 +2098 (load y)
05 +3097 (add 1)
```

El resultado de la expresión se almacena en una ubicación temporal en memoria (96) mediante la instrucción:

```
06 +2196 (almacenar temporalmente)
```

y la ubicación temporal se mete en la pila. Ahora que se ha evaluado la expresión, el resultado debe almacenarse en y (es decir, la variable del lado izquierdo de =). Entonces la ubicación temporal se carga en el acumulador y éste se almacena en y mediante las instrucciones:

```
07 +2096 (cargar temporalmente)
08 +2198 (store y)
```

Observe que las instrucciones de LMS parecen ser redundantes. Hablaremos sobre esta cuestión en breve.

Cuando la instrucción:

```
35 rem sumar y al total
```

se divide en tokens, el número de línea 35 se inserta en la tabla de símbolos con el tipo L y se le asigna la ubicación 09.

La instrucción:

```
40 let t = t + y
```

es similar a la línea 30. La variable t se inserta en la tabla de símbolos con el tipo V y se le asigna la ubicación 95 en el arreglo de LMS. Las instrucciones siguen la misma lógica y formato que la línea 30, y se generan las instrucciones 09 +2095, 10 +3098, 11 +2194, 12 +2094 y 13 +2195. Observe que el resultado de t + y se asigna a la ubicación temporal 94 antes de asignarse a t (95). Una vez más, el lector observará que las instrucciones en las ubicaciones de memoria 11 y 12 parecen ser redundantes. De nuevo, hablaremos sobre esta cuestión en breve.

La instrucción:

```
45 rem ciclo con y
```

es un comentario, por lo que la línea 45 se agrega a la tabla de símbolos con el tipo L y se le asigna la ubicación 14 en el arreglo de LMS.

La instrucción:

```
50 goto 20
```

transfiere el control a la línea 20. El número de línea 50 se inserta en la tabla de símbolos con el tipo L y se le asigna la ubicación 14 en el arreglo de LMS. El equivalente de `goto` en LMS es la instrucción de *bifurcación incondicional* (40), la cual transfiere el control a una ubicación específica en el arreglo de LMS. El compilador busca en la tabla de símbolos la línea 20 y encuentra que a ésta le corresponde la ubicación 01 en el arreglo de LMS. El código de operación (40) se multiplica por 100 y se le agrega la ubicación 01 para producir la instrucción 14 +4001.

La instrucción:

```
55 rem mostrar resultado
```

es un comentario, por lo que la línea 55 se inserta en la tabla de símbolos con el tipo L y se le asigna la ubicación 15 en el arreglo de LMS.

La instrucción:

```
60 print t
```

es una instrucción de salida. El número de línea 60 se inserta en la tabla de símbolos con el tipo L y se le asigna la ubicación 15 en el arreglo de LMS. El equivalente de `print` en LMS es el código de operación 11 (*write*). La ubicación de t se determina a partir de la tabla de símbolos y se agrega al resultado de la multiplicación del código de operación por 100.

La instrucción:

```
99 end
```

es la línea final del programa. El número de línea 99 se almacena en la tabla de símbolos con el tipo L y se le asigna la ubicación 16 en el arreglo de LMS. El comando `end` produce la instrucción de LMS +4300 (43 significa *parar* en LMS), la cual se escribe como instrucción final en el arreglo de memoria de LMS.

Esto completa la primera pasada del compilador. Ahora consideremos la segunda pasada. En el arreglo `banderas` se buscan valores distintos de -1. La ubicación 03 contiene 60, por lo que el compilador sabe que la instrucción 03 está incompleta. El compilador completa la instrucción buscando en la tabla de símbolos el número 60, determinando su ubicación y agregándola a la instrucción incompleta. En este caso la búsqueda determina que la línea 60 corresponde a la ubicación 15 en el arreglo de LMS, por lo que se produce la instrucción completa 03 +4215 que sustituye a 03 +4200. Ahora el programa de Simple se ha compilado con éxito.

Para crear el compilador, tendrá que llevar a cabo cada una de las siguientes tareas:

- Modifique el programa simulador de Simpletron que escribió en el ejercicio 8.19 para que reciba la entrada de un archivo especificado por el usuario (vea el capítulo 17). El simulador debe mostrar sus resultados en un archivo en disco, con el mismo formato que el de la pantalla. Convierta el simulador para que sea un programa orientado a objetos. En especial, haga que cada parte del hardware sea un objeto. Ordene los tipos de instrucciones en una jerarquía de clases por medio de la herencia. Despues ejecute el programa en forma polimórfica, indicando a cada instrucción que se ejecute a sí misma con un mensaje `ejecutarInstruccion`.
- Modifique el algoritmo de conversión de expresiones infijo a postfix del ejercicio 20.12 para procesar operandos enteros con varios dígitos y operandos de nombres de variables con una sola letra. [Sugerencia: puede utilizarse la función `strtok` de la Biblioteca estándar de C ++ para localizar cada constante y variable en una expresión, y las constantes pueden convertirse de cadenas a enteros mediante el uso de la función `atoi` de la biblioteca estándar (`<csdtlib>`)]. [Nota: la representación de datos de la expresión postfix debe alterarse para dar soporte a los nombres de variables y constantes enteras].
- Modifique el algoritmo de evaluación de expresiones postfix para procesar operandos enteros con varios dígitos y operandos de nombres de variables. Además, el algoritmo deberá ahora implementar el “gancho” que se describió anteriormente, de manera que se produzcan instrucciones de LMS en vez de evaluar directamente la expresión. [Sugerencia: puede utilizarse la función `strtok` de la Biblioteca estándar para localizar cada constante y variable en una expresión, y las constantes pueden convertirse de cadenas a enteros mediante el uso de la función `atoi` de la Biblioteca estándar]. [Nota: la representación de datos de la expresión postfix debe alterarse para dar soporte a los nombres de variables y constantes enteras].
- Construya el compilador. Incorpore las partes (b) y (c) para evaluar las expresiones en instrucciones `let`. Su programa debe contener una función que realice la primera pasada del compilador y una función que realice la segunda pasada del compilador. Ambas funciones pueden llamar a otras funciones para realizar sus tareas. Haga que su compilador esté lo más orientado a objetos que sea posible.

20.28 (*Optimización del compilador de Simple*) Cuando se compila un programa y se convierte en LMS, se genera un conjunto de instrucciones. Ciertas combinaciones de instrucciones a menudo se repiten a sí mismas, por lo general en tercias conocidas como *producciones*. Una producción normalmente consiste de tres instrucciones tales como *load*, *add* y *store*. Por ejemplo, en la figura 20.33 se muestran cinco de las instrucciones de LMS que se produjeron en la compilación del programa de la figura 20.31. Las primeras tres instrucciones son la producción que suma 1 a y. Observe que las instrucciones 06 y 07 almacenan el valor del acumulador en la ubicación temporal 96 y cargan el valor de vuelta en el acumulador, de manera que la instrucción 08 pueda almacenar el valor en la ubicación 98. A menudo una producción va seguida de una instrucción *load* para la misma ubicación en la que fue guardada. Este código puede *optimizarse* mediante la eliminación de la instrucción *store* y la instrucción *load* que le sigue, las cuales operan en la misma ubicación, con lo que se permite al simulador Simpletron ejecutar el programa con más rapidez. En la figura 20.34 se muestra el LMS optimizado para el programa de la figura 20.31. Observe que hay cuatro instrucciones menos en el código optimizado; un ahorro de espacio en memoria del 25%.

```

1 04 +2098 (load)
2 05 +3097 (add)
3 06 +2196 (store)
4 07 +2096 (load)
5 08 +2198 (store)

```

Figura 20.33 | Código sin optimizar del programa de la figura 20.31.

Modifique el compilador de manera que proporcione una opción para optimizar el código de Lenguaje máquina Simpletron que produzca. Compare en forma manual el código sin optimizar con el código optimizado, y calcule el porcentaje de reducción.

Programa de Simple	Ubicación e instrucción de LMS	Descripción
5 rem sumar 1 a x	ninguna	rem se ignora
10 input x	00 +1099	leer x y colocar su valor en la ubicación 99
15 rem verificar si y == x	ninguna	rem se ignora
20 if y == x goto 60	01 +2098 02 +3199 03 +4211	cargar y (98) en el acumulador restar x (99) al acumulador ramificar a ubicación 11 si vale cero
25 rem incrementar y	ninguna	rem se ignora
30 let y = y + 1	04 +2098 05 +3097 06 +2198	cargar y en el acumulador sumar 1 (97) al acumulador almacenar acumulador en y (98)
35 rem sumar y al total	ninguna	rem se ignora
40 let t = t + y	07 +2096 08 +3098 09 +2196	cargar t de la ubicación (96) sumar y (98) al acumulador almacenar acumulador en t (96)
45 rem ciclo con y	ninguna	rem se ignora
50 goto 20	10 +4001	ramificar a ubicación 01
55 rem mostrar resultado	ninguna	rem se ignora
60 print t	11 +1196	mostrar t (96) en la pantalla
99 end	12 +4300	terminar la ejecución

Figura 20.34 | Código optimizado para el programa de la figura 20.31.

20.29 (*Modificaciones al compilador de Simple*) Realice las siguientes modificaciones al compilador de Simple. Algunas de estas modificaciones podrían requerir también que se modifique el programa simulador de Simpletron que se escribió en el ejercicio 8.19.

- Permitir el uso del operador módulo (%) en instrucciones `let`. El Lenguaje máquina Simpletron debe modificarse para incluir una instrucción módulo.
- Permitir la exponenciación en una instrucción `let` mediante el uso de ^ como operador de exponenciación. El Lenguaje máquina Simpletron debe modificarse para incluir una instrucción de exponenciación.
- Permitir al compilador que reconozca letras mayúsculas y minúsculas en instrucciones de Simple (por ejemplo, 'A' es equivalente a 'a'). No se requieren modificaciones al simulador de Simpletron.
- Permitir que las instrucciones `input` lean valores para múltiples variables, tal como `input x, y`. No se requieren modificaciones al simulador de Simpletron.
- Permitir que el compilador muestre múltiples valores en una sola instrucción `print` como `print a, b, c`. No se requieren modificaciones al simulador de Simpletron.
- Agregar al compilador la capacidad de comprobar la sintaxis, de manera que se muestren mensajes de error cuando se encuentren errores de sintaxis en un programa de Simple. No se requieren modificaciones al simulador de Simpletron.
- Permitir arreglos de enteros. No se requieren modificaciones al simulador de Simpletron.
- Permitir subrutinas especificadas por los comandos `gosub` y `return` de Simple. El comando `gosub` pasa el control del programa a una subrutina y el comando `return` pasa el control de vuelta a la instrucción que va después de `gosub`. Esto es similar a la llamada a una función en C++. La misma subrutina puede llamarse desde muchos comandos `gosub` distribuidos a lo largo de un programa. No se requieren modificaciones al simulador de Simpletron.
- Permitir instrucciones de repetición de la forma:

```
for x = 2 to 10 step 2
    instrucciones de Simple
next
```

Esta instrucción `for` realiza iteraciones del 2 al 10 con un incremento de 2. La línea `next` indica el final del cuerpo de la instrucción `for`. No se requieren modificaciones al simulador de Simpletron.

- Permitir instrucciones de repetición de la forma:

```
for x = 2 to 10
    instrucciones de Simple
next
```

Esta instrucción `for` realiza iteraciones del 2 al 10 con un incremento predeterminado de 1. No se requieren modificaciones al simulador de Simpletron.

- Permitir que el compilador procese operaciones de entrada y salida con cadenas. Para ello se requiere la modificación del simulador Simpletron para que procese y almacene valores de cadena. [Sugerencia: cada palabra de Simpletron puede dividirse en dos grupos, cada uno de los cuales almacena un entero de dos dígitos. Cada entero de dos dígitos representa el equivalente decimal ASCII de un carácter. Agregue una instrucción de lenguaje máquina que imprima una cadena, empezando en cierta ubicación de memoria Simpletron. La primera mitad de la palabra Simpletron en esa ubicación es un conteo del número de caracteres en la cadena (es decir, la longitud de la misma). Cada mitad de palabra subsiguiente contiene un carácter ASCII expresado mediante dos dígitos decimales. La instrucción de lenguaje máquina comprueba la longitud e imprime la cadena, traduciendo cada número de dos dígitos en su carácter equivalente].
- Permitir que el compilador procese valores de punto flotante, además de enteros. El simulador Simpletron también debe modificarse para procesar valores de punto flotante.

20.30 (*Un intérprete de Simple*) Un intérprete es un programa que lee la instrucción de un programa escrito en un lenguaje de alto nivel, determina la operación que va a realizar esa instrucción y la ejecuta inmediatamente. El programa en lenguaje de alto nivel no se convierte primero en lenguaje máquina. Los intérpretes se ejecutan con más lentitud que los compiladores, ya que cada instrucción que se encuentra en el programa que va a interpretarse debe primero descifrarse. Si las instrucciones están contenidas dentro de un ciclo, se descifran cada vez que se encuentran en éste. Las primeras versiones del lenguaje de programación BASIC se implementaron como intérpretes.

Escriba un intérprete para el lenguaje Simple descrito en el ejercicio 20.26. El programa debe utilizar el convertidor de expresiones infijo a postfix que se desarrolló en el ejercicio 20.12, junto con el evaluador de expresiones postfix que se desarrolló en el ejercicio 20.13, para evaluar las expresiones en una instrucción `let`. Las mismas restricciones impuestas sobre el lenguaje Simple en el ejercicio 20.26 se aplican también a este programa. Pruebe el intérprete con los programas de Simple que

se escribieron en el ejercicio 20.26. Compare los resultados de ejecutar estos programas en el intérprete con los resultados de compilar los programas de Simple y ejecutarlos en el simulador Simpletron que se construyó en el ejercicio 8.19.

20.31 (*Insertar/eliminar en cualquier parte de una lista enlazada*) Nuestra plantilla de clase de lista enlazada permite inserciones y eliminaciones sólo en la parte frontal y en la parte final de la lista enlazada. Estas capacidades eran convenientes para nosotros cuando utilizamos la herencia `private` y la composición para producir una plantilla de clase pila y una plantilla de clase cola con una mínima cantidad de código, con sólo reutilizar la clase lista. En realidad, las listas enlazadas son más generales que las que nosotros vimos. Modifique la plantilla de clase lista enlazada que desarrollamos en este capítulo para permitir inserciones y eliminaciones en cualquier parte de la lista.

20.32 (*Listas y colas sin apuntadores a la parte final*) En nuestra implementación de una lista enlazada (figuras 20.3 a 20.5) utilizamos un `primerPtr` y un `ultimoPtr`. El `ultimoPtr` era útil para las funciones miembro `insertarAlFinal` y `eliminarDelFinal` de la clase `Lista`. La función `insertarAlFinal` corresponde a la función miembro `enqueue` de la clase `Cola`. Vuelva a escribir la clase `Lista` de manera que no utilice un `ultimoPtr`. De esta forma, cualquier operación en la parte final de la lista deberá empezar buscando en la lista desde su parte inicial. ¿Afecta esto a nuestra implementación de la clase `Cola` (figura 20.16)?

20.33 Use la versión de composición del programa de la pila (figura 20.15) para formar un programa de pila funcional completo. Modifique este programa para poner en línea (`inline`) las funciones miembro. Compare los dos métodos. Sintetice las ventajas y desventajas de poner en línea las funciones miembro.

20.34 (*Rendimiento de los procesos de ordenamiento y búsqueda en árboles binarios*) Un problema con el ordenamiento de árboles binarios es que el orden en el que se insertan los datos afecta a la forma del árbol; para la misma colección de datos, distintos ordenamientos pueden producir árboles binarios de formas considerablemente distintas. El rendimiento de los algoritmos de ordenamiento y búsqueda en árboles binarios es susceptible a la forma del árbol binario. ¿Qué forma tendría un árbol binario si sus datos se insertaran en orden ascendente?, ¿en orden descendente?, ¿qué forma debería tener el árbol para lograr un máximo rendimiento en el proceso de búsqueda?

20.35 (*Listas indexadas*) Como se presentan en el texto, la búsqueda en las listas enlazadas debe llevarse a cabo en forma secuencial. Para las listas extensas, esto puede ocasionar un rendimiento pobre. Una técnica común para mejorar el rendimiento del proceso de búsqueda en las listas es crear y mantener un índice de la lista. Un índice es un conjunto de apuntadores a varios lugares clave en la lista. Por ejemplo, una aplicación que busca en una lista extensa de nombres podría mejorar el rendimiento al crear un índice con 26 entradas: una para cada letra del alfabeto. Una operación de búsqueda para un apellido que empieza con ‘Y’ buscaría entonces primero en el índice para determinar en dónde empiezan las entradas con “Y” y luego “saltaría” hasta ese punto en la lista para buscar linealmente hasta encontrar el nombre deseado. Esto sería mucho más rápido que buscar en la lista enlazada desde el principio. Utilice la clase `Lista` de las figuras 20.3 a 20.5 como la base para una clase `ListaIndizada`. Escriba un programa que demuestre la operación de las listas indexadas. Asegúrese de incluir las funciones miembro `insertarEnListaIndizada`, `buscarEnListaIndizada` y `eliminarDeListaIndizada`.



La misma vieja mentira caritativa que se repite a medida que pasan los años tiene un impacto perpetuo: “En realidad no has cambiado nada!”

—Margaret Fishback

El principal defecto del Rey Enrique era masticar pequeños trozos de cuerda.

—Hilaire Belloc

La escritura vigorosa es concisa. Una oración no debe contener palabras innecesarias, un párrafo no debe contener oraciones innecesarias.

—William Strunk, Jr.

Bits, caracteres, cadenas estilo C y estructuras

OBJETIVOS

En este capítulo aprenderá a:

- Crear y utilizar estructuras (`struct`).
- Pasar estructuras a las funciones, por valor y por referencia.
- Usar `typedef` para crear alias para los tipos de datos y estructuras previamente definidas.
- Manipular datos con los operadores a nivel de bits y crear campos de bits para almacenar los datos en forma compacta.
- Usar las funciones de la biblioteca de manejo de caracteres `<cctype>`.
- Usar las funciones de conversión de cadenas de la biblioteca de utilerías generales `<cstdlib>`.
- Usar las funciones de procesamiento de cadenas de la biblioteca de manejo de cadenas `<cstring>`.

- 21.1** Introducción
- 21.2** Definiciones de estructuras
- 21.3** Inicialización de estructuras
- 21.4** Uso de estructuras con funciones
- 21.5** `typedef`
- 21.6** Ejemplo: simulación para barajar y repartir cartas de alto rendimiento
- 21.7** Operadores a nivel de bits
- 21.8** Campos de bits
- 21.9** Biblioteca de manejo de caracteres
- 21.10** Funciones de conversión de cadenas basadas en apuntador
- 21.11** Funciones de búsqueda de la biblioteca de manejo de cadenas basadas en apuntador
- 21.12** Funciones de memoria de la biblioteca de manejo de cadenas basadas en apuntador
- 21.13** Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)

21.1 Introducción

Ahora veremos las estructuras y la manipulación de bits, caracteres y cadenas estilo C. Muchas de las técnicas que presentaremos aquí se incluyen para beneficio del programador de C++ que trabajará con código de C y C++ heredado.

Los diseñadores de C++ hicieron que las estructuras evolucionaran hacia la noción de una clase. Al igual que una clase, las estructuras de C++ pueden contener especificadores de acceso, funciones miembro, constructores y destructores. De hecho, las únicas diferencias entre las estructuras y las clases en C++ es que los miembros de las estructuras tienen el acceso predeterminado `public` y los miembros de las clases tienen el acceso predeterminado `private` cuando no se utilizan especificadores de acceso, y que las estructuras tienen herencia `public` predeterminada, mientras que las clases tienen herencia `private` predeterminada. A lo largo del libro hemos hablado detalladamente sobre las clases, por lo que en realidad no necesitamos describir las estructuras con detalle. Nuestra presentación de las estructuras en este capítulo se enfoca en cuanto a su uso en C, en donde las estructuras sólo contienen miembros de datos `public`. Este uso de las estructuras es común en el código de C heredado y en las primeras piezas de código de C++ que se pueden ver en la industria.

Vamos a ver cómo declarar estructuras, inicializarlas y pasárlas a funciones. Después presentaremos una simulación para barajar y repartir cartas de alto rendimiento, en la que utilizaremos objetos de estructuras y cadenas estilo C para representar las cartas. Hablaremos sobre los operadores a nivel de bits que permiten a los programadores utilizar y manipular los bits individuales en bytes de datos. También presentaremos los campos de bits: estructuras especiales que se pueden utilizar para especificar el número exacto de bits que ocupa una variable en memoria. Estas técnicas de manipulación de bits son comunes en los programas en C y C++ que interactúan directamente con dispositivos de hardware que tienen memoria limitada. El capítulo termina con ejemplos de muchas funciones de manipulación de caracteres y cadenas estilo C, algunas de las cuales están diseñadas para procesar bloques de memoria como arreglos de bytes.

21.2 Definiciones de estructuras

Las estructuras son **tipos de datos agregados**; es decir, pueden construirse mediante el uso de elementos de varios tipos, incluyendo otros tipos `struct`. Considere las siguientes definiciones de estructuras:

```
struct Carta
{
    char *cara;
    char *palo;
}; // fin de struct Carta
```

La palabra clave `struct` introduce la definición de la estructura `Carta`. El identificador `Carta` es el **nombre de la estructura** y se utiliza en C++ para declarar variables de **tipo estructura** (en C, el nombre del tipo de la estructura anterior es `struct Carta`). En este ejemplo, el tipo de estructura es `Carta`. Los datos (y posiblemente las funciones; al igual que con las clases) declarados dentro de las llaves de la definición de la estructura son los **miembros** de la estructura. Los miembros de la misma estructura deben tener nombres únicos, pero dos estructuras distintas pueden contener miembros del mismo nombre sin conflicto. Cada definición de estructura debe terminar con un signo de punto y coma.



Error común de programación 21.I

Olvidar el signo de punto y coma que termina la definición de una estructura es un error de sintaxis.

La definición de **Carta** contiene dos miembros de tipo **char *:** **cara** y **palo**. Los miembros de una estructura pueden ser variables de los tipos de datos fundamentales (por ejemplo, **int**, **double**, etc.) o agregados, como arreglos, otras estructuras y/o clases. Los miembros de datos en la definición de una estructura pueden ser de muchos tipos de datos. Por ejemplo, una estructura **Empleado** podría contener miembros de cadenas de caracteres para el primer nombre y el apellido paterno, un miembro **int** para la edad del empleado, un miembro **char** que contenga 'M' o 'F' para el género del empleado, un miembro **double** para el salario por hora del empleado, y así en lo sucesivo.

Una estructura no puede contener una instancia de sí misma. Por ejemplo, una variable de estructura **Carta** no se puede declarar en la definición para la estructura **Carta**. Sin embargo, un apuntador a una estructura **Carta** sí se puede incluir. Una estructura que contenga un miembro que sea un apuntador al mismo tipo de estructura se conoce como **estructura autorreferenciada**. En el capítulo 20, Estructuras de datos, utilizamos una construcción similar, para construir varios tipos de estructuras de datos enlazadas.

La definición de la estructura **Carta** no reserva espacio en memoria; en vez de ello, crea un nuevo tipo de datos que se utiliza para declarar variables de estructura. Estas variables se declaran de la misma forma que las variables de otros tipos. Las siguientes declaraciones

```
Carta unaCarta;
Carta mazo[ 52 ];
Carta *cartaPtr;
```

declaran a **unaCarta** como una variable de una estructura de tipo **Carta**, a **mazo** como un arreglo con 52 elementos de tipo **Carta** y a **cartaPtr** como un apuntador a una estructura **Carta**. También se pueden declarar variables de un tipo de estructura dado al colocar una lista separada por comas de los nombres de las variables entre la llave de cierre de la definición de la estructura y el punto y coma que termina la definición de la estructura. Por ejemplo, las declaraciones anteriores podrían haberse incorporado en la definición de la estructura **Carta** de la siguiente manera:

```
struct Carta
{
    char *cara;
    char *palo;
} unaCarta, mazo[ 52 ], *cartaPtr;
```

El nombre de la estructura es opcional. Si la definición de una estructura no contiene un nombre, las variables de ese tipo de estructura sólo se pueden declarar entre la llave derecha de cierre de la definición de la estructura y el punto y coma que termina esta definición.



Observación de Ingeniería de Software 21.I

Debe proporcionar el nombre de una estructura al crear un tipo de estructura. El nombre de la estructura es obligatorio para declarar nuevas variables del tipo de estructura más adelante en el programa, declarar parámetros del tipo de estructura y, si la estructura se va a utilizar como una clase de C++, para especificar el nombre del constructor y del destructor.

Las únicas operaciones integradas válidas que se pueden realizar en objetos de estructuras son: asignar un objeto de estructura a otro del mismo tipo, recibir la dirección (&) de un objeto de estructura, acceder a los miembros de un objeto de estructura (de la misma forma que se accede a los miembros de una clase) y utilizar el operador **sizeof** para determinar el tamaño de una estructura. Al igual que con las clases, la mayoría de los operadores se pueden sobrecargar para trabajar con objetos de un tipo de estructura.

Los miembros de las estructuras no se almacenan necesariamente en bytes consecutivos de memoria. Algunas veces hay "agujeros" en una estructura, debido a que ciertas computadoras almacenan tipos de datos específicos sólo en ciertos límites de memoria, como media palabra, palabra o doble palabra. Una palabra es una unidad estándar de memoria que se utiliza para almacenar datos en una computadora; por lo general dos o cuatro bytes, y comúnmente cuatro bytes en los sistemas populares de 32 bits de la actualidad. Considere la siguiente definición de una estructura, en la que se declaran los objetos estructura **ejemplo1** y **ejemplo2** de tipo **Ejemplo**:

```
struct Ejemplo
{
    char c;
    int i;
} ejemplo1, ejemplo2;
```

Una computadora con palabras de dos bytes podría requerir que cada uno de los miembros de `Ejemplo` se alineen en un límite de palabra (es decir, al principio de una palabra; esto es dependiente del equipo). En la figura 21.1 se muestra un ejemplo de alineación del almacenamiento para un objeto de tipo `Ejemplo` al que se le ha asignado el carácter 'a' y el entero 97 (se muestran las representaciones de bits de los valores). Si los miembros se almacenan empezando en límites de palabra, hay un agujero de un byte (`byte 1` en la figura) en el almacenamiento para los objetos de tipo `Ejemplo`. El valor en el agujero de un byte es indefinido. Si los valores de los miembros de `ejemplo1` y `ejemplo2` son de hecho iguales, los objetos estructura no son necesariamente iguales, ya que no es probable que los agujeros indefinidos de un byte contengan valores idénticos.

Error común de programación 21.2



Comparar estructuras es un error de compilación.



Figura 21.1 | Posible alineación del almacenamiento para una variable de tipo `Ejemplo`, en donde se muestra un área indefinida en la memoria.

Tip de portabilidad 21.1



Debido a que el tamaño de los elementos de datos de un tipo específico es dependiente del equipo, y como las consideraciones de alineación del almacenamiento son dependientes del equipo, también lo es la representación de una estructura.

21.3 Inicialización de estructuras

Las estructuras se pueden inicializar mediante listas inicializadoras, como con los arreglos. Por ejemplo, la declaración

```
Carta unaCarta = { "Tres", "Corazones" };
```

crea la variable `Carta` llamada `unaCarta`, inicializa el miembro `cara` con "Tres" y el miembro `palo` con "Corazones". Si hay menos inicializadores en la lista que miembros en la estructura, el resto de los miembros se inicializan con sus valores predeterminados. Las variables de estructura declaradas fuera de la definición de una función (es decir, en forma externa) se inicializan con sus valores predeterminados si no se inicializan explícitamente en la declaración externa. Las variables de estructura también se pueden establecer en expresiones de asignación, ya sea asignando una variable de estructura del mismo tipo o asignando valores a los miembros de datos individuales de la estructura.

21.4 Uso de estructuras con funciones

Hay dos formas de pasar la información en estructuras a las funciones. Podemos pasar toda la estructura, o pasar los miembros individuales de una estructura. De manera predeterminada, las estructuras se pasan por valor. Las estructuras y sus miembros también se pueden pasar por referencia, ya sea pasando referencias o apunadores.

Para pasar una estructura por referencia, se pasa la dirección del objeto estructura o una referencia al objeto estructura. Los arreglos de estructuras (al igual que los demás arreglos) se pasan por referencia.

En el capítulo 7 vimos que un arreglo se podía pasar por valor mediante el uso de una estructura. Para pasar un arreglo por valor, se crea una estructura (o clase) con el arreglo como miembro, y después se pasa un objeto de ese tipo de estructura (o clase) a una función por valor. Como los objetos estructura se pasan por valor, también el miembro arreglo se pasa por valor.

Tip de rendimiento 21.1



Pasar estructuras (en especial las estructuras grandes) por referencia es más eficiente que pasárlas por valor (para lo cual hay que copiar la estructura completa).

21.5 `typedef`

La palabra clave `typedef` proporciona un mecanismo para crear sinónimos (o alias) para los tipos de datos definidos con anterioridad. A menudo los nombres para los tipos de estructuras se definen con `typedef` para crear nombres de tipos más cortos, simples o legibles. Por ejemplo, la instrucción

```
typedef Carta *CartaPtr;
```

define el nuevo nombre de tipo `CartaPtr` como un sinónimo para el tipo `Carta`*



Buena práctica de programación 21.1

Escriba con mayúsculas los nombres de `typedef` para enfatizar que estos nombres son sinónimos para otros nombres de tipos.

Al crear un nuevo nombre con `typedef` no se crea un nuevo tipo; `typedef` simplemente crea un nuevo nombre de tipo, el cual después se puede utilizar en el programa como alias para un nombre de tipo existente.



Tip de portabilidad 21.2

Se pueden crear sinónimos para los tipos de datos integrados con `typedef` para que los programas sean más portables. Por ejemplo, un programa puede usar `typedef` para crear el alias `Integer` para los enteros de cuatro bytes. Así, `Integer` puede ser un alias para `int` en sistemas con enteros de cuatro bytes, y puede ser un alias para `long int` en sistemas con enteros de dos bytes, en donde los valores `long int` ocupan cuatro bytes. De esta forma, simplemente declaramos todas las variables enteras de cuatro bytes como de tipo `Integer`.

21.6 Ejemplo: simulación para barajar y repartir cartas de alto rendimiento

El programa de las figuras 21.2 a 21.4 se basa en la simulación para barajar y repartir cartas que vimos en el capítulo 8. El programa representa el mazo de cartas como un arreglo de estructuras y utiliza algoritmos para barajar y repartir de alto rendimiento.

```

1 // Fig. 21.2: MazoDeCartas.h
2 // Definición de la clase MazoDeCartas que
3 // representa un mazo de cartas de juego.
4
5 // definición de la estructura Carta
6 struct Carta
7 {
8     char *cara;
9     char *palo;
10}; // fin de la estructura Carta
11
12 // definición de la clase MazoDeCartas
13 class MazoDeCartas
14 {
15 public:
16     MazoDeCartas(); // el constructor inicializa el mazo
17     void barajar(); // baraja las cartas en el mazo
18     void repartir() const; // reparte las cartas en el mazo
19
20 private:
21     Carta mazo[ 52 ]; // representa el mazo de cartas
22 }; // fin de la clase MazoDeCartas

```

Figura 21.2 | Archivo de encabezado para la clase `MazoDeCartas`.

```

1 // Fig. 21.3: MazoDeCartas.cpp
2 // Definiciones de las funciones miembro para la clase MazoDeCartas que
3 // simula los procesos de barajar y repartir un mazo de cartas de juego.
4 #include <iostream>
5 using std::cout;
6 using std::left;
7 using std::right;
8

```

Figura 21.3 | Archivo de la clase `MazoDeCartas`. (Parte 1 de 2).

```

9 #include <iomanip>
10 using std::setw;
11
12 #include <cstdlib> // prototipos para rand y srand
13 using std::rand;
14 using std::srand;
15
16 #include <ctime> // prototipo para time
17 using std::time;
18
19 #include "MazoDeCartas.h" // definición de la clase MazoDeCartas
20
21 // el constructor de MazoDeCartas sin argumentos inicializa el mazo
22 MazoDeCartas::MazoDeCartas()
23 {
24     // inicializa el arreglo palo
25     static char *palo[ 4 ] =
26         { "Corazones", "Diamantes", "Treboles", "Espadas" };
27
28     // inicializa el arreglo cara
29     static char *cara[ 13 ] =
30         { "As", "Dos", "Tres", "Cuatro", "Cinco", "Seis", "Siete",
31           "Ocho", "Nueve", "Diez", "Sota", "Reina", "Rey" };
32
33     // establece los valores para el mazo de 52 objetos Carta
34     for ( int i = 0; i < 52; i++ )
35     {
36         mazo[ i ].cara = cara[ i % 13 ];
37         mazo[ i ].palo = palo[ i / 13 ];
38     } // fin de for
39
40     srand( time( 0 ) ); // siembra el generador de números aleatorios
41 } // fin del constructor de MazoDeCartas sin argumentos
42
43 // baraja las cartas en el mazo
44 void MazoDeCartas::barajar()
45 {
46     // baraja las cartas al azar
47     for ( int i = 0; i < 52; i++ )
48     {
49         int j = rand() % 52;
50         Carta temp = mazo[ i ];
51         mazo[ i ] = mazo[ j ];
52         mazo[ j ] = temp;
53     } // fin de for
54 } // fin de la función barajar
55
56 // reparte las cartas en el mazo
57 void MazoDeCartas::repartir() const
58 {
59     // muestra la cara y el palo de cada carta
60     for ( int i = 0; i < 52; i++ )
61         cout << right << setw( 5 ) << mazo[ i ].cara << " de "
62             << left << setw( 8 ) << mazo[ i ].palo
63             << (( i + 1 ) % 2 ? '\t' : '\n' );
64 } // fin de la función repartir

```

Figura 21.3 | Archivo de la clase MazoDeCartas. (Parte 2 de 2).

```

1 // Fig. 21.4: fig21_04.cpp
2 // Programa para barajar y repartir cartas.
3 #include "MazoDeCartas.h" // definición de la clase MazoDeCartas

```

Figura 21.4 | Simulación para barajar y repartir cartas de alto rendimiento. (Parte 1 de 2).

```

4
5 int main()
6 {
7     MazoDeCartas mazoDeCartas; // crea un objeto MazoDeCartas
8
9     mazoDeCartas.barajar(); // baraja las cartas en el mazo
10    mazoDeCartas.repartir(); // reparte las cartas en el mazo
11    return 0; // indica que terminó con éxito
12 } // fin de main

```

Sota de Corazones	Ocho de Diamantes
Dos de Treboles	Siete de Espadas
Reina de Treboles	As de Treboles
Dos de Diamantes	Dos de Espadas
Diez de Corazones	Nueve de Diamantes
Seis de Treboles	Tres de Corazones
Dos de Corazones	Diez de Diamantes
Siete de Treboles	Reina de Corazones
Diez de Treboles	Sota de Diamantes
As de Espadas	Reina de Espadas
Rey de Corazones	Cuatro de Treboles
Nueve de Corazones	Reina de Diamantes
Cinco de Espadas	Siete de Corazones
Nueve de Treboles	Nueve de Espadas
Cinco de Corazones	Cuatro de Diamantes
Cinco de Treboles	Sota de Treboles
Sota de Espadas	As de Corazones
Seis de Espadas	Rey de Diamantes
Siete de Diamantes	Ocho de Treboles
Diez de Espadas	Tres de Diamantes
Cuatro de Espadas	Ocho de Espadas
Rey de Treboles	Rey de Espadas
Cinco de Diamantes	Cuatro de Corazones
Tres de Treboles	Tres de Espadas
Seis de Corazones	As de Diamantes
Seis de Diamantes	Ocho de Corazones

Figura 21.4 | Simulación para barajar y repartir cartas de alto rendimiento. (Parte 2 de 2).

El constructor (líneas 22 a 41 de la figura 21.3) inicializa el arreglo `Carta` en orden, con cadenas de caracteres que representan del As hasta el Rey para cada palo. La función `barajar` implementa el algoritmo para barajar de alto rendimiento. La función itera a través de las 52 cartas (subíndices de arreglo 0 a 51). Para cada carta, se elige al azar un número entre 0 y 51. A continuación, la estructura `Carta` actual y la estructura `Carta` seleccionada al azar se intercambian en el arreglo. En una pasada de todo el arreglo se realiza un total de 52 intercambios, ¡y se baraja el arreglo de estructuras `Carta`! A diferencia del algoritmo para barajar presentado en el capítulo 8, este algoritmo no sufre de un aplazamiento indefinido. Como las estructuras `Carta` se intercambiaron en sus mismas posiciones en el arreglo, el algoritmo para repartir de alto rendimiento implementado en la función `repartir` sólo requiere una pasada del arreglo para repartir las cartas barajadas.

21.7 Operadores a nivel de bits

C++ cuenta con muchas herramientas de manipulación de bits para los programadores que necesitan adentrarse al denominado nivel de “bits y bytes”. Los sistemas operativos, el software de equipos de prueba, el software de red y muchos otros tipos de software requieren una comunicación “directa con el hardware”. En esta sección y en las siguientes, hablaremos sobre las herramientas de manipulación de bits de C++. Presentaremos cada uno de los muchos operadores a nivel de bits de C++ y veremos cómo ahorrar memoria al usar los campos de bits.

Las computadoras representan todos los datos internamente como secuencias de bits. Cada bit puede asumir el valor 0 o 1. En la mayoría de los sistemas, una secuencia de 8 bits forma un `byte`: la unidad de almacenamiento estándar para una variable de tipo `char`. Otros tipos de datos se almacenan en mayores números de bytes. Los operadores a nivel de bits se utilizan para manipular los bits de operandos integrales (`char`, `short`, `int` y `long`; tanto `signed` como `unsigned`). Por lo general se utilizan los enteros sin signo con los operadores a nivel de bits.



Tip de portabilidad 21.3

Las manipulaciones de datos a nivel de bits son dependientes del equipo.

Observe que las discusiones acerca de los operadores a nivel de bits en esta sección muestran las representaciones binarias de los operandos enteros. Para una explicación detallada del sistema numérico binario (también conocido como base 2), vea el apéndice D, Sistemas numéricos. Debido a la naturaleza dependiente del equipo de las manipulaciones de bits, tal vez algunos de estos programas no funcionen en su sistema a menos que los modifique.

Los operadores a nivel de bits son: AND (`&`) a nivel de bits, OR inclusivo (`|`) a nivel de bits, OR exclusivo (`^`) a nivel de bits, desplazamiento a la izquierda (`<<`), desplazamiento a la derecha (`>>`) y complemento a nivel de bit (`~`); a este último también se le conoce como complemento a uno. (Observe que hemos estado utilizando `&`, `<<` y `>>` para otros fines. Éste es un clásico ejemplo de la sobrecarga de operadores). Los operadores AND, OR inclusivo y OR exclusivo a nivel de bits comparan sus dos operandos bit por bit. El operador AND a nivel de bits establece cada bit en el resultado a 1 si el bit correspondiente en ambos operandos es 1. El operador OR inclusivo a nivel de bits establece cada bit en el resultado a 1 si el bit correspondiente en un operando (o en ambos) es 1. El operador OR exclusivo a nivel de bits establece cada bit en el resultado a 1 si el bit correspondiente en cada operando (pero no en ambos) es 1. El operador de desplazamiento a la izquierda desplaza los bits de su operando izquierdo a la izquierda, con base en el número de bits especificado en su operando derecho. El operador de desplazamiento a la derecha desplaza los bits en su operando izquierdo a la derecha, con base en el número de bits especificados en su operando derecho. El operador de complemento a nivel de bits establece todos los bits 0 en su operando a 1 en el resultado, y establece todos los bits 1 en su operando a 0 en el resultado. En los siguientes ejemplos aparecen discusiones detalladas de cada operador a nivel de bits. En la figura 21.5 se sintetizan los operadores a nivel de bits.

Operador	Nombre	Descripción
<code>&</code>	AND a nivel de bits	Los bits en el resultado se establecen en 1 si los bits correspondientes en los dos operandos son 1.
<code> </code>	OR inclusivo a nivel de bits	Los bits en el resultado se establecen en 1 si uno o ambos de los bits correspondientes en los dos operandos es 1.
<code>^</code>	OR exclusivo a nivel de bits	Los bits en el resultado se establecen en 1 si exactamente uno de los bits correspondientes en los dos operandos es 1.
<code><<</code>	desplazamiento a la izquierda	Desplaza los bits del primer operando a la izquierda, con base en el número de bits especificados por el segundo operando; rellena con bits 0 a partir de la derecha.
<code>>></code>	desplazamiento a la derecha con extensión de signo	Desplaza los bits del primer operando a la derecha, con base en el número de bits especificados por el segundo operando; el método para llenar a partir de la izquierda es dependiente del equipo.
<code>~</code>	complemento a nivel de bits	Todos los bits 0 se establecen en 1 y todos los bits 1 se establecen en 0.

Figura 21.5 | Operadores a nivel de bits.

Cómo imprimir una representación binaria de un valor entero

Cuando utilizamos los operadores a nivel de bits, es conveniente ilustrar sus efectos precisos mediante la impresión de valores en su representación binaria. El programa de la figura 21.6 imprime un entero `unsigned` en su representación binaria, en grupos de ocho bits cada uno.

```

1 // Fig. 21.6: fig21_06.cpp
2 // Cómo imprimir un entero sin signo en bits.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7

```

Figura 21.6 | Cómo imprimir un entero sin signo en bits. (Parte 1 de 2).

```

8 #include <iomanip>
9 using std::setw;
10
11 void mostrarBits( unsigned ); // prototipo
12
13 int main()
14 {
15     unsigned valorEntrada; // valor integral a imprimir en binario
16
17     cout << "Escriba un entero sin signo: ";
18     cin >> valorEntrada;
19     mostrarBits( valorEntrada );
20     return 0;
21 } // fin de main
22
23 // muestra los bits de un valor entero sin signo
24 void mostrarBits( unsigned valor )
25 {
26     const int DESPL = 8 * sizeof( unsigned ) - 1;
27     const unsigned MASCARA = 1 << DESPL;
28
29     cout << setw( 10 ) << valor << " = ";
30
31     // muestra los bits
32     for ( unsigned i = 1; i <= DESPL + 1; i++ )
33     {
34         cout << ( valor & MASCARA ? '1' : '0' );
35         valor <<= 1; // desplaza el valor a la izquierda por 1
36
37         if ( i % 8 == 0 ) // imprime un espacio después de 8 bits
38             cout << ' ';
39     } // fin de for
40
41     cout << endl;
42 } // fin de la función mostrarBits

```

Escriba un entero sin signo: **65000**
65000 = 00000000 00000000 11111101 11101000

Escriba un entero sin signo: **29**
29 = 00000000 00000000 00000000 00011101

Figura 21.6 | Cómo imprimir un entero sin signo en bits. (Parte 2 de 2).

La función `mostrarBits` (líneas 24 a 42) utiliza el operador AND a nivel de bits para combinar la variable `valor` con la constante `MASCARA`. A menudo, el operador AND a nivel de bits se utiliza con un operando conocido como **máscara**: un valor entero con bits específicos establecidos en 1. Las máscaras se utilizan para ocultar ciertos bits en un valor mientras se seleccionan otros. En `mostrarBits`, la línea 27 asigna a la constante `MASCARA` el valor `1 << DESPL`. El valor de la constante `DESPL` se calculó en la línea 26 con la expresión

`8 * sizeof(unsigned) - 1`

la cual multiplica el número de bytes que requiere un objeto `unsigned` en memoria por 8 (el número de bits en un byte) para obtener el número total de bits requeridos para almacenar un objeto `unsigned`, y después le resta 1. La representación de bits de `1 << DESPL` en una computadora que representa objetos `unsigned` en cuatro bytes de memoria es

`10000000 00000000 00000000 00000000`

El operador de desplazamiento a la izquierda desplaza el valor 1 del bit de menor orden (más a la derecha) hasta el bit de mayor orden (más a la izquierda) en `MASCARA`, y rellena con bits 0 partiendo de la derecha. En la línea 34 se determina si debe imprimirse un 1 o un 0 para el bit actual de más a la izquierda de la variable `valor`. Suponga que la variable `valor` contiene 65000 (00000000 00000000 11111101 11101000). Cuando se combinan `valor` y `MASCARA` usando `&`, todos

los bits excepto el de mayor orden en la variable `valor` se “enmascaran” (ocultan), ya que cualquier bit al que se le aplique la operación AND con 0 produce 0. Si el bit de más a la izquierda es 1, el valor & MASCARA se evalúa como

00000000	00000000	11111101	11101000	(valor)
10000000	00000000	00000000	00000000	(MASCARA)
<hr/>				
00000000	00000000	00000000	00000000	(valor & MASCARA)

que se interpreta como `false`, y se imprime 0. Después en la línea 35 se desplaza la variable `valor` a la izquierda por un bit mediante la expresión `valor <<=1` (es decir, `valor = valor << 1`). Estos pasos se repiten para cada bit de la variable `valor`. En un momento dado, un bit con un valor de 1 se desplaza a la posición de bit de más a la izquierda, y la manipulación de bits es la siguiente:

11111101	11101000	00000000	00000000	(valor)
10000000	00000000	00000000	00000000	(MASCARA)
<hr/>				
10000000	00000000	00000000	00000000	(valor & MASCARA)

Como ambos bits izquierdos son 1, el resultado de la expresión es distinto de cero (verdadero) y se imprime un valor de 1. En la figura 21.7 se sintetizan los resultados de combinar dos bits con el operador AND a nivel de bits.

Error común de programación 21.3



Utilizar el operador AND lógico (&&) en vez del operador AND a nivel de bits (&) y viceversa es un error lógico.

El programa de la figura 21.8 demuestra el operador AND a nivel de bits, el operador OR inclusivo a nivel de bits, el operador OR exclusivo a nivel de bits y el operador de complemento a nivel de bits. La función `mostrarBits` (líneas 57 a 75) imprime los valores enteros `unsigned`.

Operador AND a nivel de bits (&)

En la figura 21.8, la línea 21 asigna 2179876355 (10000001 11101110 01000110 00000011) a la variable `numero1`, y la línea 22 asigna 1 (00000000 00000000 00000000 00000001) a la variable `mascara`. Cuando se combinan `mascara` y `numero1` usando el operador AND a nivel de bits (`&`) en la expresión `numero1&mascara` (línea 27), el resultado es 00000000 00000000 00000000 00000001. Todos los bits excepto el bit de menor orden en la variable `numero1` se “enmascaran” (ocultan) mediante la operación AND con la constante `MASCARA`.

Bit 1	Bit 2	Bit 1 & Bit 2
0	0	0
1	0	0
0	1	0
1	1	1

Figura 21.7 | Resultados de combinar dos bits con el operador AND a nivel de bits (`&`).

```

1 // Fig. 21.8: fig21_08.cpp
2 // Uso de los operadores AND, OR inclusivo, OR exclusivo
3 // y complemento a nivel de bits.
4 #include <iostream>
5 using std::cout;
6
7 #include <iomanip>
8 using std::endl;
9 using std::setw;
10
11 void mostrarBits( unsigned ); // prototipo
12

```

Figura 21.8 | Operadores AND, OR inclusivo, OR exclusivo y complemento a nivel de bits. (Parte I de 3).

```

13 int main()
14 {
15     unsigned numero1;
16     unsigned numero2;
17     unsigned mascara;
18     unsigned establecerBits;
19
20     // demuestra & a nivel de bits
21     numero1 = 2179876355;
22     mascara = 1;
23     cout << "El resultado de combinar los siguientes valores\n";
24     mostrarBits( numero1 );
25     mostrarBits( mascara );
26     cout << "usando el operador AND a nivel de bits & es\n";
27     mostrarBits( numero1 & mascara );
28
29     // demuestra | a nivel de bits
30     numero1 = 15;
31     establecerBits = 241;
32     cout << "\nEl resultado de combinar los siguientes valores\n";
33     mostrarBits( numero1 );
34     mostrarBits( establecerBits );
35     cout << "usando el operador OR inclusivo a nivel de bits | es\n";
36     mostrarBits( numero1 | establecerBits );
37
38     // demuestra OR exclusivo a nivel de bits
39     numero1 = 139;
40     numero2 = 199;
41     cout << "\nEl resultado de combinar los siguientes valores\n";
42     mostrarBits( numero1 );
43     mostrarBits( numero2 );
44     cout << "usando el operador OR exclusivo a nivel de bits ^ es\n";
45     mostrarBits( numero1 ^ numero2 );
46
47     // demuestra complemento a nivel de bits
48     numero1 = 21845;
49     cout << "\nEl complemento a uno de\n";
50     mostrarBits( numero1 );
51     cout << "es" << endl;
52     mostrarBits( ~numero1 );
53
54 } // fin de main
55
56 // muestra los bits de un valor entero sin signo
57 void mostrarBits( unsigned valor )
58 {
59     const int DESPL = 8 * sizeof( unsigned ) - 1;
60     const unsigned MASCARA = 1 << DESPL;
61
62     cout << setw( 10 ) << valor << " = ";
63
64     // muestra los bits
65     for ( unsigned i = 1; i <= DESPL + 1; i++ )
66     {
67         cout << ( valor & MASCARA ? '1' : '0' );
68         valor <<= 1; // desplaza el valor a la izquierda por 1
69
70         if ( i % 8 == 0 ) // imprime un espacio después de 8 bits
71             cout << ' ';
72     } // fin de for
73
74     cout << endl;
75 } // fin de la función mostrarBits

```

Figura 21.8 | Operadores AND, OR inclusivo, OR exclusivo y complemento a nivel de bits. (Parte 2 de 3).

```

El resultado de combinar los siguientes valores
2179876355 = 10000001 1101110 01000110 00000011
    1 = 00000000 00000000 00000000 00000001
usando el operador AND a nivel de bits & es
    1 = 00000000 00000000 00000000 00000001

El resultado de combinar los siguientes valores
    15 = 00000000 00000000 00000000 00001111
    241 = 00000000 00000000 00000000 11110001
usando el operador OR inclusivo a nivel de bits | es
    255 = 00000000 00000000 00000000 11111111

El resultado de combinar los siguientes valores
    139 = 00000000 00000000 00000000 10001011
    199 = 00000000 00000000 00000000 11000111
usando el operador OR exclusivo a nivel de bits ^ es
    76 = 00000000 00000000 00000000 01001100

El complemento a uno de
    21845 = 00000000 00000000 01010101 01010101
es
4294945450 = 11111111 11111111 10101010 10101010

```

Figura 21.8 | Operadores AND, OR inclusivo, OR exclusivo y complemento a nivel de bits. (Parte 3 de 3).

Operador OR inclusivo a nivel de bits (|)

El operador OR inclusivo a nivel de bits se utiliza para establecer bits específicos en 1, en un operando. En la figura 21.8, la línea 30 asigna 15 (00000000 00000000 00000000 00001111) a la variable `numero1`, y la línea 31 asigna 241 (00000000 00000000 00000000 11110001) a la variable `establecerBits`. Cuando se combinan `numero1` y `establecerBits` usando el operador OR a nivel de bits en la expresión `numero1 | establecerBits` (línea 36), el resultado es 255 (00000000 00000000 00000000 11111111). En la figura 21.9 se sintetizan los resultados de combinar dos bits con el operador OR inclusivo a nivel de bits.



Error común de programación 21.4

Utilizar el operador OR lógico (||) en vez del operador OR inclusivo a nivel de bits (|) o viceversa es un error lógico.

OR exclusivo a nivel de bits (^)

El operador OR exclusivo a nivel de bits (^) establece cada bit en el resultado en 1, si *exactamente* uno de los bits correspondientes en sus dos operandos es 1. En la figura 21.8, en las líneas 39 a 40 se asignan a las variables `numero1` y `numero2` los valores 139 (00000000 00000000 00000000 10001011) y 199 (00000000 00000000 00000000 11000111), respectivamente. Cuando se combinan estas variables con el operador OR exclusivo en la expresión `numero1 ^ numero2` (línea 45), el resultado es 00000000 00000000 00000000 01001100. La figura 21.10 sintetiza los resultados de combinar dos bits con el operador OR exclusivo a nivel de bits.

Complemento a nivel de bits (~)

El operador de complemento a nivel de bits (~) establece todos los bits 1 en el operando a 0 en el resultado, y establece todos los bits 0 a 1 en el resultado; a esto también se le conoce como “obtener el complemento a uno del valor”. En la figura 21.8, línea 48, se asigna a la variable `numero1` el valor 21845 (00000000 00000000 01010101 01010101). Cuando se evalúa la expresión `~numero1`, el resultado es (11111111 11111111 10101010 10101010).

Bit 1	Bit 2	Bit 1 & Bit 2
0	0	0
1	0	1
0	1	1
1	1	1

Figura 21.9 | Combinación de dos bits con el operador OR inclusivo a nivel de bits (|).

Bit 1	Bit 2	Bit 1 & Bit 2
0	0	0
1	0	1
0	1	1
1	1	0

Figura 21.10 | Combinación de dos bits con el operador OR exclusivo a nivel de bits (\wedge).

En la figura 21.11 se demuestran el operador de desplazamiento a la izquierda ($<<$) y el operador de desplazamiento a la derecha ($>>$). La función `mostrarBits` (líneas 31 a 49) imprime los valores enteros `unsigned`.

Operador de desplazamiento a la izquierda

El operador de desplazamiento a la izquierda ($<<$) desplaza los bits de su operando izquierdo a la izquierda, con base en el número de bits especificados en su operando derecho. Los bits que se vacían a la derecha se reemplazan con 0s; los bits que se desplazan más allá de la última posición a la izquierda se pierden. En el programa de la figura 21.11, la línea 14 asigna a la variable `numero1` el valor 960 (00000000 00000000 00000011 11000000). El resultado de desplazar a la izquierda la variable `numero1` por 8 bits en la expresión `numero1 << 8` (línea 20) es 245760 (00000000 00000011 11000000 00000000).

```

1 // Fig. 21.11: fig21_11.cpp
2 // Uso de los operadores de desplazamiento a nivel de bits.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 void mostrarBits( unsigned ); // prototipo
11
12 int main()
13 {
14     unsigned numero1 = 960;
15
16     // demuestra el desplazamiento a la izquierda a nivel de bits
17     cout << "El resultado de desplazar a la izquierda\n";
18     mostrarBits( numero1 );
19     cout << "8 posiciones de bit mediante el operador de desplazamiento a la izquierda es\n";
20     mostrarBits( numero1 << 8 );
21
22     // demuestra el desplazamiento a la derecha a nivel de bits
23     cout << "\nEl resultado de desplazar a la derecha\n";
24     mostrarBits( numero1 );
25     cout << "8 posiciones de bit mediante el operador de desplazamiento a la derecha es\n";
26     mostrarBits( numero1 >> 8 );
27     return 0;
28 } // fin de main
29
30 // muestra los bits de un valor entero sin signo
31 void mostrarBits( unsigned valor )
32 {
33     const int DESPL = 8 * sizeof( unsigned ) - 1;
34     const unsigned MASCARA = 1 << DESPL;
35
36     cout << setw( 10 ) << valor << " = ";
37 }
```

Figura 21.11 | Operadores de desplazamiento a nivel de bits. (Parte I de 2).

```

38 // muestra los bits
39 for ( unsigned i = 1; i <= DESPL + 1; i++ )
40 {
41     cout << ( valor & MASCARA ? '1' : '0' );
42     valor <<= 1; // desplaza el valor a la izquierda por 1
43
44     if ( i % 8 == 0 ) // imprime un espacio después de 8 bits
45         cout << ' ';
46 } // fin de for
47
48 cout << endl;
49 } // fin de la función mostrarBits

```

El resultado de desplazar a la izquierda
960 = 00000000 00000000 00000011 11000000
8 posiciones de bit mediante el operador de desplazamiento a la izquierda es
245760 = 00000000 00000011 11000000 00000000

El resultado de desplazar a la derecha
960 = 00000000 00000000 00000011 11000000
8 posiciones de bit mediante el operador de desplazamiento a la derecha es
3 = 00000000 00000000 00000000 00000011

Figura 21.11 | Operadores de desplazamiento a nivel de bits. (Parte 2 de 2).

Operador de desplazamiento a la derecha

El operador de desplazamiento a la derecha (`>>`) desplaza los bits de su operando izquierdo a la derecha, con base en el número de bits especificados en su operando derecho. Al realizar un desplazamiento a la derecha en un entero `unsigned`, los bits que se vacían a la izquierda se reemplazan con 0s; los bits que se desplazan más allá de la última posición a la derecha se pierden. En el programa de la figura 21.11, el resultado de desplazar a la derecha `numero1` en la expresión `numero1 >> 8` (línea 26) es 3 (00000000 00000000 00000000 00000011).



Error común de programación 21.5

El resultado de desplazar un valor es indefinido si el operando derecho es negativo, o si es mayor o igual que el número de bits en donde se almacena el operando izquierdo.



Tip de portabilidad 21.4

El resultado de desplazar a la derecha un valor con signo es dependiente del equipo. Algunos equipos llenan los bits vacíos con ceros y otros usan el bit de signo.

Operadores de asignación a nivel de bits

Cada operador a nivel de bits (excepto el operador de complemento a nivel de bits) tiene su correspondiente operador de asignación. Estos *operadores de asignación a nivel de bits* se muestran en la figura 21.12; se utilizan de una manera similar a los operadores de asignación aritméticos que se presentaron en el capítulo 2.

Operadores de asignación a nivel de bits

<code>&=</code>	Operador de asignación AND a nivel de bits.
<code> =</code>	Operador de asignación OR inclusivo a nivel de bits.
<code>^=</code>	Operador de asignación OR exclusivo a nivel de bits.
<code><<=</code>	Operador de asignación de desplazamiento a la izquierda.
<code>>>=</code>	Operador de asignación de desplazamiento a la derecha con extensión de signo.

Figura 21.12 | Operadores de asignación a nivel de bits.

En la figura 21.13 se muestra la precedencia y asociatividad de los operadores introducidos hasta este punto en el libro. Se muestran de arriba hacia abajo en orden descendente de precedencia.

21.8 Campos de bits

C++ proporciona la habilidad de especificar el número de bits en los que se almacena un tipo integral o un miembro tipo enum de una clase o de una estructura. A dicho miembro se le conoce como un **campo de bits**. Los campos de bits permiten un mejor uso de la memoria, al almacenar los datos en el mínimo número de bits requeridos. Los miembros campos de bits se *deben* declarar como un tipo integral o enum.



Consideré la siguiente definición de una estructura:

```
struct bitCarta
{
    unsigned cara : 4;
    unsigned palo : 2;
    unsigned color : 1;
}; // fin de struct bitCarta
```

Operadores	Asociatividad	Tipo
:: (unario; derecha a izquierda) :: (binario; izquierda a derecha)	izquierda a derecha	más alto
[] . -> ++ -- static_cast< tipo >()	izquierda a derecha	unario
++ -- + - ! delete sizeof	derecha a izquierda	unario
* ~ & new		
* / %	izquierda a derecha	multiplicativo
+ -	izquierda a derecha	aditivo
<< >>	izquierda a derecha	desplazamiento
< <= > >=	izquierda a derecha	relacional
== !=	izquierda a derecha	igualdad
&	izquierda a derecha	AND a nivel de bits
^	izquierda a derecha	XOR a nivel de bits
	izquierda a derecha	OR a nivel de bits
&&	izquierda a derecha	AND lógico
	izquierda a derecha	OR lógico
:=	derecha a izquierda	condicional
= += -= *= /= %= &= = ^= <<= >>=	derecha a izquierda	asignación
,	izquierda a derecha	coma

Figura 21.13 | Precedencia y asociatividad de operadores.

La definición contiene tres campos de bits unsigned (*cara*, *palo* y *color*) que se utilizan para representar una carta de un mazo de 52 cartas. Para declarar un campo de bits, hay que colocar dos puntos (:) después de un tipo integral o enum, y una constante entera que represente la **anchura del campo de bits** (es decir, el número de bits en el que se almacena el miembro). La anchura debe ser una constante entera.

La definición de la estructura anterior indica que el miembro *cara* se almacena en 4 bits, el miembro *palo* en 2 bits y el miembro *color* en 1 bit. El número de bits se basa en el rango deseado de valores para cada miembro de la estructura.

El miembro `cara` almacena valores entre 0 (As) y 12 (Rey); 4 bits pueden almacenar un valor entre 0 y 15. El miembro `palo` almacena valores entre 0 y 3 (0 = Diamantes, 1 = Corazones, 2 = Tréboles, 3 = Espadas); 2 bits pueden almacenar un valor entre 0 y 3. Por último, el miembro `color` almacena 0 (Rojo) o 1 (Negro); 1 bit puede almacenar 0 o 1.

El programa de las figuras 21.14 a 21.16 crea el arreglo `mazo` que contiene 52 estructuras `bitCarta` (línea 21 de la figura 21.14). El constructor inserta las 52 cartas en el arreglo `mazo` y la función `repartir` imprime las 52 cartas. Obsérvese que los campos de bits se utilizan de la misma forma que cualquier otro miembro de la estructura (líneas 18 a 20 y 28 a 33 de la figura 21.15). El miembro `color` se incluye como medio para indicar el color de la carta en un sistema que permite mostrar los colores.

```

1 // Fig. 21.14: MazoDeCartas.h
2 // Definición de la clase MazoDeCartas que
3 // representa un mazo de cartas de juego.
4
5 // definición de la estructura BitCarta con campos de bits
6 struct BitCarta
7 {
8     unsigned cara : 4; // 4 bits; 0 a 15
9     unsigned palo : 2; // 2 bits; 0 a 3
10    unsigned color : 1; // 1 bit; 0 a 1
11}; // fin de struct BitCarta
12
13 // definición de la clase MazoDeCartas
14 class MazoDeCartas
15 {
16 public:
17     MazoDeCartas(); // el constructor inicializa el mazo
18     void repartir(); // reparte las cartas en el mazo
19
20 private:
21     BitCarta mazo[ 52 ]; // representa el mazo de cartas
22 }; // fin de la clase MazoDeCartas

```

Figura 21.14 | Archivo de encabezado para la clase `MazoDeCartas`.

```

1 // Fig. 21.15: MazoDeCartas.cpp
2 // Definiciones de las funciones miembro para la clase MazoDeCartas que
3 // simulan cómo barajar y repartir un mazo de cartas de juego.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 #include "MazoDeCartas.h" // definición de la clase MazoDeCartas
12
13 // el constructor de MazoDeCartas sin argumentos inicializa el mazo
14 MazoDeCartas::MazoDeCartas()
15 {
16     for ( int i = 0; i <= 51; i++ )
17     {
18         mazo[ i ].cara = i % 13; // caras en orden
19         mazo[ i ].palo = i / 13; // palos en orden
20         mazo[ i ].color = i / 26; // colores en orden
21     } // fin de for
22 } // fin del constructor de MazoDeCartas sin argumentos
23

```

Figura 21.15 | Archivo de la clase `MazoDeCartas`. (Parte I de 2).

```

24 // reparte las cartas en el mazo
25 void MazoDeCartas::repartir()
26 {
27     for ( int k1 = 0, k2 = k1 + 26; k1 <= 25; k1++, k2++ )
28         cout << "Carta:" << setw( 3 ) << mazo[ k1 ].cara
29             << " Palo:" << setw( 2 ) << mazo[ k1 ].palo
30             << " Color:" << setw( 2 ) << mazo[ k1 ].color
31             << " " << "Carta:" << setw( 3 ) << mazo[ k2 ].cara
32             << " Palo:" << setw( 2 ) << mazo[ k2 ].palo
33             << " Color:" << setw( 2 ) << mazo[ k2 ].color << endl;
34 } // fin de la función repartir

```

Figura 21.15 | Archivo de la clase `MazoDeCartas`. (Parte 2 de 2).

Es posible especificar un **campo de bits sin nombre**, en cuyo caso el campo se utiliza como **relleno** en la estructura. Por ejemplo, la definición de la estructura utiliza un campo sin nombre de 3 bits como relleno; nada se puede almacenar en esos 3 bits. El miembro `b` se almacena en otra unidad de almacenamiento.

```

struct Ejemplo
{
    unsigned a : 13;
    unsigned : 3; // se alinea al siguiente límite de unidad de almacenamiento
    unsigned b : 4;
}; // fin de struct Ejemplo

```

Un **campo de bits sin nombre con anchura cero** se utiliza para alinear el siguiente campo de bits en un nuevo límite de unidad de almacenamiento. Por ejemplo, la definición de la estructura

```

struct Ejemplo
{
    unsigned a : 13;
    unsigned : 0; // se alinea al siguiente límite de unidad de almacenamiento
    unsigned b : 4;
}; // fin de struct Ejemplo

```

utiliza un campo sin nombre de 0 bits para omitir los bits restantes (todos los que haya) de la unidad de almacenamiento en la que se almacena `a`, y alinea a `b` en el siguiente límite de unidad de almacenamiento.

```

1 // Fig. 21.16: fig21_16.cpp
2 // Programa para barajar y repartir cartas.
3 #include "MazoDeCartas.h" // definición de la clase MazoDeCartas
4
5 int main()
6 {
7     MazoDeCartas mazoDeCartas; // crea un objeto MazoDeCartas
8     mazoDeCartas.repartir(); // reparte las cartas en el mazo
9     return 0; // indica que terminó correctamente
10 } // fin de main

```

```

Carta: 0 Palo: 0 Color: 0 Carta: 0 Palo: 2 Color: 1
Carta: 1 Palo: 0 Color: 0 Carta: 1 Palo: 2 Color: 1
Carta: 2 Palo: 0 Color: 0 Carta: 2 Palo: 2 Color: 1
Carta: 3 Palo: 0 Color: 0 Carta: 3 Palo: 2 Color: 1
Carta: 4 Palo: 0 Color: 0 Carta: 4 Palo: 2 Color: 1
Carta: 5 Palo: 0 Color: 0 Carta: 5 Palo: 2 Color: 1
Carta: 6 Palo: 0 Color: 0 Carta: 6 Palo: 2 Color: 1
Carta: 7 Palo: 0 Color: 0 Carta: 7 Palo: 2 Color: 1
Carta: 8 Palo: 0 Color: 0 Carta: 8 Palo: 2 Color: 1
Carta: 9 Palo: 0 Color: 0 Carta: 9 Palo: 2 Color: 1
Carta: 10 Palo: 0 Color: 0 Carta: 10 Palo: 2 Color: 1
Carta: 11 Palo: 0 Color: 0 Carta: 11 Palo: 2 Color: 1
Carta: 12 Palo: 0 Color: 0 Carta: 12 Palo: 2 Color: 1

```

Figura 21.16 | Campos de bits utilizados para almacenar un mazo de cartas. (Parte 1 de 2).

```

Carta: 0 Palo: 1 Color: 0 Carta: 0 Palo: 3 Color: 1
Carta: 1 Palo: 1 Color: 0 Carta: 1 Palo: 3 Color: 1
Carta: 2 Palo: 1 Color: 0 Carta: 2 Palo: 3 Color: 1
Carta: 3 Palo: 1 Color: 0 Carta: 3 Palo: 3 Color: 1
Carta: 4 Palo: 1 Color: 0 Carta: 4 Palo: 3 Color: 1
Carta: 5 Palo: 1 Color: 0 Carta: 5 Palo: 3 Color: 1
Carta: 6 Palo: 1 Color: 0 Carta: 6 Palo: 3 Color: 1
Carta: 7 Palo: 1 Color: 0 Carta: 7 Palo: 3 Color: 1
Carta: 8 Palo: 1 Color: 0 Carta: 8 Palo: 3 Color: 1
Carta: 9 Palo: 1 Color: 0 Carta: 9 Palo: 3 Color: 1
Carta: 10 Palo: 1 Color: 0 Carta: 10 Palo: 3 Color: 1
Carta: 11 Palo: 1 Color: 0 Carta: 11 Palo: 3 Color: 1
Carta: 12 Palo: 1 Color: 0 Carta: 12 Palo: 3 Color: 1

```

Figura 21.16 | Campos de bits utilizados para almacenar un mazo de cartas. (Parte 2 de 2).



Tip de portabilidad 21.5

Las manipulaciones de los campos de bits son dependientes del equipo. Por ejemplo, algunas computadoras permiten que los campos de bits atraviesen límites de palabras, mientras que otras no.



Error común de programación 21.6

Tratar de acceder a los bits individuales de un campo de bits con el uso de subíndices como si fueran elementos de un arreglo es un error de compilación. Los campos de bits no son “arreglos de bits”.



Error común de programación 21.7

Tratar de tomar la dirección de un campo de bits (el operador & no se puede utilizar con los campos de bits, debido a que un apuntador sólo puede designar un byte específico en memoria, y los campos de bits pueden empezar en medio de un byte) es un error de compilación.



Tip de rendimiento 21.3

Aunque los campos de bits ahorran espacio, utilizarlos puede provocar que el compilador genere código en lenguaje máquina que se ejecute con más lentitud. Esto ocurre debido a que se requieren operaciones adicionales en lenguaje máquina para acceder sólo a ciertas porciones de una unidad de almacenamiento direccionable. Éste es uno de muchos ejemplos de las concesiones entre espacio y tiempo que ocurren en la ciencia computacional.

21.9 Biblioteca de manejo de caracteres

La mayoría de los datos se introducen en las computadoras en forma de caracteres; esto incluye a las letras, los dígitos y diversos símbolos especiales. En esta sección hablaremos sobre las herramientas de C++ para examinar y manipular caracteres individuales. En el resto del capítulo continuaremos con la discusión acerca de la manipulación de cadenas de caracteres que empezamos en el capítulo 8.

La biblioteca de manejo de caracteres incluye varias funciones que realizan pruebas y manipulaciones útiles de datos tipo carácter. Cada función recibe un carácter (representado como un `int`) o EOF como argumento. Los caracteres comúnmente se manipulan como enteros. Recuerde que EOF por lo general tiene el valor -1 y que ciertas arquitecturas de hardware no permiten almacenar valores negativos en variables `char`. Por lo tanto, las funciones de manejo de caracteres manipulan los caracteres como si fueran enteros. En la figura 21.17 se sintetizan las funciones de la biblioteca de manejo de caracteres. Al utilizar funciones de la biblioteca de manejo de caracteres, debemos incluir el archivo de encabezado `<cctype>`.

Prototipo	Descripción
<code>int isdigit(int c)</code>	Devuelve <code>true</code> si <code>c</code> es un dígito y <code>false</code> en caso contrario.
<code>int isalpha(int c)</code>	Devuelve <code>true</code> si <code>c</code> es una letra y <code>false</code> en caso contrario.
<code>int isalnum(int c)</code>	Devuelve <code>true</code> si <code>c</code> es un dígito o letra y <code>false</code> en caso contrario.

Figura 21.17 | Funciones de la biblioteca de manejo de caracteres. (Parte 1 de 2).

Prototipo	Descripción
<code>int isxdigit(int c)</code>	Devuelve <code>true</code> si <code>c</code> es un carácter de dígito hexadecimal y <code>false</code> en caso contrario. (En el apéndice D, Sistemas numéricos, encontrará una explicación detallada de los números binarios, octales, decimales y hexadecimales.)
<code>int islower(int c)</code>	Devuelve <code>true</code> si <code>c</code> es una letra minúscula y <code>false</code> en caso contrario.
<code>int isupper(int c)</code>	Devuelve <code>true</code> si <code>c</code> es una letra mayúscula; <code>false</code> en caso contrario
<code>int tolower(int c)</code>	Si <code>c</code> es una letra mayúscula, <code>tolower</code> devuelve <code>c</code> como una letra minúscula. En caso contrario, <code>tolower</code> devuelve el argumento sin modificación.
<code>int toupper(int c)</code>	Si <code>c</code> es una letra minúscula, <code>toupper</code> devuelve <code>c</code> como letra mayúscula. En caso contrario, <code>toupper</code> devuelve el argumento sin modificación.
<code>int isspace(int c)</code>	Devuelve <code>true</code> si <code>c</code> es un carácter de espacio en blanco: nueva línea (' <code>\n</code> '), espacio (' <code>'</code> '), avance de página (' <code>\f</code> '), retorno de carro (' <code>\r</code> '), tabulación horizontal (' <code>\t</code> ') o tabulación vertical (' <code>\v</code> '); y <code>false</code> en caso contrario.
<code>int iscntrl(int c)</code>	Devuelve <code>true</code> si <code>c</code> es un carácter de control, como nueva línea (' <code>\n</code> '), avance de página (' <code>\f</code> '), retorno de carro (' <code>\r</code> '), tabulación horizontal (' <code>\t</code> '), tabulación vertical (' <code>\v</code> '), alerta (' <code>\a</code> ') o retroceso (' <code>\b</code> '); y <code>false</code> en caso contrario.
<code>int ispunct(int c)</code>	Devuelve <code>true</code> si <code>c</code> es un carácter de impresión distinto de un espacio, dígito o letra, y <code>false</code> en caso contrario.
<code>int isprint(int c)</code>	Devuelve un valor <code>true</code> si <code>c</code> es un carácter de impresión, incluyendo el espacio (' <code>'</code> '), y <code>false</code> en caso contrario.
<code>int isgraph(int c)</code>	Devuelve <code>true</code> si <code>c</code> es un carácter de impresión distinto de espacio (' <code>'</code>) y <code>false</code> en caso contrario.

Figura 21.17 | Funciones de la biblioteca de manejo de caracteres. (Parte 2 de 2).

La figura 21.18 demuestra las funciones `isdigit`, `isalpha`, `isalnum` e `isxdigit`. La función `isdigit` determina si su argumento es un dígito (0 a 9). La función `isalpha` determina si su argumento es una letra mayúscula (A a Z) o minúscula (a a z). La función `isalnum` determina si su argumento es una letra mayúscula, una letra minúscula o un dígito. La función `isxdigit` determina si su argumento es un dígito hexadecimal (A a F, a a f, 0 a 9).

```

1 // Fig. 21.18: fig21_18.cpp
2 // Uso de las funciones isdigit, isalpha, isalnum e isxdigit.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cctype> // prototipos de las funciones de manejo de caracteres
8 using std::isalnum;
9 using std::isalpha;
10 using std::isdigit;
11 using std::isxdigit;
12
13 int main()
14 {
15     cout << "De acuerdo con isdigit:\n"
16     << ( isdigit( '8' ) ? "8 es un" : "8 no es un" ) << " digito\n"
17     << ( isdigit( '#' ) ? "# es un" : "# no es un" ) << " digito\n";
18 }
```

Figura 21.18 | Las funciones de manejo de caracteres `isdigit`, `isalpha`, `isalnum` e `isxdigit`. (Parte I de 2).

```

19     cout << "\nDe acuerdo con isalpha:\n"
20     << ( isalpha( 'A' ) ? "A es una" : "A no es una" ) << " letra\n"
21     << ( isalpha( 'b' ) ? "b es una" : "b no es una" ) << " letra\n"
22     << ( isalpha( '&'amp; ) ? "& es una" : "& no es una" ) << " letra\n"
23     << ( isalpha( '4' ) ? "4 es una" : "4 no es una" ) << " letra\n";
24
25     cout << "\nDe acuerdo con isalnum:\n"
26     << ( isalnum( 'A' ) ? "A es un" : "A no es un" )
27     << " digito o una letra\n"
28     << ( isalnum( '8' ) ? "8 es un" : "8 no es un" )
29     << " digito o una letra\n"
30     << ( isalnum( '#' ) ? "#" es un" : "#" no es un" )
31     << " digito o una letra\n";
32
33     cout << "\nDe acuerdo con isxdigit:\n"
34     << ( isxdigit( 'F' ) ? "F es un" : "F no es un" )
35     << " digito hexadecimal\n"
36     << ( isxdigit( 'J' ) ? "J es un" : "J no es un" )
37     << " digito hexadecimal\n"
38     << ( isxdigit( '7' ) ? "7 es un" : "7 no es un" )
39     << " digito hexadecimal\n"
40     << ( isxdigit( '$' ) ? "$ es un" : "$ no es una" )
41     << " digito hexadecimal\n"
42     << ( isxdigit( 'f' ) ? "f es un" : "f no es un" )
43     << " digito hexadecimal" << endl;
44
45     return 0;
46 } // fin de main

```

```

De acuerdo con isdigit:
8 es un digito
# no es un digito

De acuerdo con isalpha:
A es una letra
b es una letra
& no es una letra
4 no es una letra

De acuerdo con isalnum:
A es un digito o una letra
8 es un digito o una letra
# no es un digito o una letra

De acuerdo con isxdigit:
F es un digito hexadecimal
J no es un digito hexadecimal
7 es un digito hexadecimal
$ no es una digito hexadecimal
f es un digito hexadecimal

```

Figura 21.18 | Las funciones de manejo de caracteres `isdigit`, `isalpha`, `isalnum` e `isxdigit`. (Parte 2 de 2).

La figura 21.18 utiliza el operador condicional (`:?`) con cada función para determinar si se debe imprimir la cadena "`es un`" o la cadena "`no es un`" en la salida para cada carácter evaluado. Por ejemplo, la línea 16 indica que si '`8`' es un dígito (es decir, si `isdigit` devuelve un valor verdadero, distinto de cero) se imprime la cadena "`8 es un`". Si '`8`' no es un dígito (es decir, si `isdigit` devuelve 0), se imprime la cadena "`8 no es un`".

La figura 21.19 demuestra las funciones `islower`, `isupper`, `tolower` y `toupper`. La función `islower` determina si su argumento es una letra minúscula (a a z). La función `isupper` determina si su argumento es una letra mayúscula (A a Z). La función `tolower` convierte una letra mayúscula a minúscula y devuelve la letra minúscula (si el argumento no es una letra mayúscula, `tolower` devuelve el valor del argumento sin modificación). La función `toupper` convierte una letra minúscula a mayúscula y devuelve la letra mayúscula (si el argumento no es una letra minúscula, `toupper` devuelve el valor del argumento sin modificación).

```
1 // Fig. 21.19: fig21_19.cpp
2 // Uso de las funciones islower, isupper, tolower y toupper.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cctype> // prototipos de las funciones de manejo de caracteres
8 using std::islower;
9 using std::isupper;
10 using std::tolower;
11 using std::toupper;
12
13 int main()
14 {
15     cout << "De acuerdo con islower:\n"
16     << ( islower( 'p' ) ? "p es una" : "p no es una" )
17     << " letra minuscula\n"
18     << ( islower( 'P' ) ? "P es una" : "P no es una" )
19     << " letra minuscula\n"
20     << ( islower( '5' ) ? "5 es una" : "5 no es una" )
21     << " letra minuscula\n"
22     << ( islower( '!' ) ? "!" es una" : "!" no es una" )
23     << " letra minuscula\n";
24
25     cout << "\nDe acuerdo con isupper:\n"
26     << ( isupper( 'D' ) ? "D es una" : "D no es una" )
27     << " letra mayuscula\n"
28     << ( isupper( 'd' ) ? "d es una" : "d no es una" )
29     << " letra mayuscula\n"
30     << ( isupper( '8' ) ? "8 es una" : "8 no es una" )
31     << " letra mayuscula\n"
32     << ( isupper( '$' ) ? "$ es una" : "$ no es una" )
33     << " letra mayuscula\n";
34
35     cout << "\nu convertida a mayuscula es "
36     << static_cast< char >( toupper( 'u' ) )
37     << "\n7 convertido a mayuscula es "
38     << static_cast< char >( toupper( '7' ) )
39     << "\n$ convertido a mayuscula es "
40     << static_cast< char >( toupper( '$' ) )
41     << "\nL convertida a minuscula es "
42     << static_cast< char >( tolower( 'L' ) ) << endl;
43
44     return 0;
45 } // fin de main
```

De acuerdo con islower:
p es una letra minuscula
P no es una letra minuscula
5 no es una letra minuscula
! no es una letra minuscula

De acuerdo con isupper:
D es una letra mayuscula
d no es una letra mayuscula
8 no es una letra mayuscula
\$ no es una letra mayuscula

u convertida a mayuscula es U
7 convertido a mayuscula es 7
\$ convertido a mayuscula es \$
L convertida a minuscula es l

Figura 21.19 | Las funciones de manejo de caracteres islower, isupper, tolower y toupper.

La figura 21.20 demuestra las funciones `isspace`, `iscntrl`, `ispunct`, `isprint` e `isgraph`. La función `isspace` determina si su argumento es un carácter de espacio en blanco, como espacio (' '), avance de página ('\f'), nueva línea ('\n'), retorno de carro ('\r'), tabulador horizontal ('\t') o tabulador vertical ('\v'). La función `iscntrl` determina si su argumento es un carácter de control tal como tabulador horizontal ('\t'), tabulador vertical ('\v'), avance de página ('\f'), alerta ('\a'), retroceso ('\b'), retorno de carro ('\r') o nueva línea ('\n'). La función `ispunct` determina si su argumento es un carácter de impresión distinto de un espacio, dígito o letra, como \$, #, (,), [,], {, }, ;, : o %. La función `isprint` determina si su argumento es un carácter que puede mostrarse en la pantalla (incluyendo el carácter de espacio). La función `isgraph` evalúa los mismos caracteres que `isprint`, pero no se incluye el carácter de espacio.

```

1 // Fig. 21.20: fig21_20.cpp
2 // Uso de las funciones isspace, iscntrl, ispunct, isprint, isgraph.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cctype> // prototipos de las funciones de manejo de caracteres
8 using std::iscntrl;
9 using std::isgraph;
10 using std::isprint;
11 using std::ispunct;
12 using std::isspace;
13
14 int main()
15 {
16     cout << "De acuerdo con isspace:\nNueva linea "
17         << ( isspace( '\n' ) ? "es un" : "no es un" )
18         << " caracter de espacio en blanco\nTabulador horizontal "
19         << ( isspace( '\t' ) ? "es un" : "no es un" )
20         << " caracter de espacio en blanco\n"
21         << ( isspace( '%' ) ? "% es un" : "% no es un" )
22         << " caracter de espacio en blanco\n";
23
24     cout << "\nDe acuerdo con iscntrl:\nNueva linea "
25         << ( iscntrl( '\n' ) ? "es un" : "no es un" )
26         << " caracter de control\n"
27         << ( iscntrl( '$' ) ? "$ es un" : "$ no es un" )
28         << " caracter de control\n";
29
30     cout << "\nDe acuerdo con ispunct:\n"
31         << ( ispunct( ';' ) ? ";" es un" : "; no es un" )
32         << " caracter de puntuacion\n"
33         << ( ispunct( 'Y' ) ? "Y es un" : "Y no es un" )
34         << " caracter de puntuacion\n"
35         << ( ispunct('#') ? "# es un" : "# no es un" )
36         << " caracter de puntuacion\n";
37
38     cout << "\nDe acuerdo con isprint:\n"
39         << ( isprint( '$' ) ? "$ es un" : "$ no es un" )
40         << " caracter de impresion\nAlert "
41         << ( isprint( '\a' ) ? "es un" : "no es un" )
42         << " caracter de impresion\nSpace "
43         << ( isprint( ' ' ) ? "es un" : "no es un" )
44         << " caracter de impresion\n";
45
46     cout << "\nDe acuerdo con isgraph:\n"
47         << ( isgraph( 'Q' ) ? "Q es un" : "Q no es un" )
48         << " caracter de impresion distinto de un espacio\nSpace "
49         << ( isgraph(' ') ? "es un" : "no es un" )
50         << " caracter de impresion distinto de un espacio" << endl;
51
52     return 0;
53 } // fin de main

```

Figura 21.20 | Las funciones de manejo de caracteres `isspace`, `iscntrl`, `ispunct`, `isprint` y `isgraph`. (Parte I de 2).

```

De acuerdo con isspace:
Nueva linea es un caracter de espacio en blanco
Tabulador horizontal es un caracter de espacio en blanco
% no es un caracter de espacio en blanco

De acuerdo con iscntrl:
Nueva linea es un caracter de control
$ no es un caracter de control

De acuerdo con ispunct:
; es un caracter de puntuacion
Y no es un caracter de puntuacion
# es un caracter de puntuacion

De acuerdo con isprint:
$ es un caracter de impresion
Alert no es un caracter de impresion
Space es un caracter de impresion

De acuerdo con isgraph:
Q es un caracter de impresion distinto de un espacio
Space no es un caracter de impresion distinto de un espacio

```

Figura 21.20 | Las funciones de manejo de caracteres isspace, iscntrl, ispunct, isprint y isgraph. (Parte 2 de 2).

21.10 Funciones de conversión de cadenas basadas en apuntador

En el capítulo 8 hablamos sobre varias de las funciones de manipulación de cadenas basadas en apuntador más populares de C++. En las siguientes secciones veremos el resto de las funciones, incluyendo las funciones para convertir cadenas a valores numéricos, funciones para buscar cadenas y funciones para manipular, comparar y buscar en bloques de memoria.

En esta sección presentamos las **funciones de conversión de cadenas basadas en apuntador**, de la **biblioteca general de utilerías <cstdlib>**. Estas funciones convierten cadenas de caracteres basadas en apuntador a valores enteros y de punto flotante. En la figura 21.21 se sintetizan las funciones de conversión de cadenas basadas en apuntador. Observe el uso de **const** para declarar la variable **nPtr** en los encabezados de función (se lee de derecha a izquierda como "nPtr es un apuntador a una constante tipo carácter"). Al usar funciones de la biblioteca general de utilerías, hay que incluir el archivo de encabezado **<cstdlib>**.

Prototipo	Descripción
<code>double atof(const char *nPtr)</code>	Convierte la cadena nPtr a double. Si la cadena no se puede convertir, se devuelve 0.
<code>int atoi(const char *nPtr)</code>	Convierte la cadena nPtr a int. Si la cadena no se puede convertir, se devuelve 0.
<code>long atol(const char *nPtr)</code>	Convierte la cadena nPtr a long int. Si la cadena no se puede convertir, se devuelve 0.
<code>double strtod(const char *nPtr, char **endPtr)</code>	Convierte la cadena nPtr a double. endPtr es la dirección de un apuntador al resto de la cadena después del valor double. Si la cadena no se puede convertir, se devuelve 0.
<code>long strtol(const char *nPtr, char **endPtr, int base)</code>	Convierte la cadena nPtr a long. endPtr es la dirección de un apuntador al resto de la cadena después del valor long. Si la cadena no se puede convertir, se devuelve 0. El parámetro base indica la base del número a convertir (por ejemplo: 8 para octal, 10 para decimal o 16 para hexadecimal). El valor predeterminado es decimal.

Figura 21.21 | Funciones de conversión de cadenas basadas en apuntador de la biblioteca general de utilerías. (Parte 1 de 2).

Prototipo	Descripción
<code>unsigned long strtoul(const char *nPtr, char **endPtr, int base)</code>	Convierte la cadena nPtr a unsigned long. endPtr es la dirección de un apuntador al resto de la cadena después del valor unsigned long. Si la cadena no se puede convertir, se devuelve 0. El parámetro base indica la base del número a convertir (por ejemplo: 8 para octal, 10 para decimal o 16 para hexadecimal). El valor predeterminado es decimal.

Figura 21.21 | Funciones de conversión de cadenas basadas en apuntador de la biblioteca general de utilerías. (Parte 2 de 2).

La función `atof` (figura 21.22, línea 12) convierte su argumento (una cadena que representa un número de punto flotante) a un valor `double`. La función devuelve el valor `double`. Si la cadena no se puede convertir (por ejemplo, si el primer carácter de la cadena no es un dígito), la función `atof` devuelve cero.

La función `atoi` (figura 21.23, línea 12) convierte su argumento (una cadena de dígitos que representa un entero) a un valor `int`. La función devuelve el valor `int`. Si la cadena no se puede convertir, la función `atoi` devuelve cero.

La función `atol` (figura 21.24, línea 12) convierte su argumento (una cadena de dígitos que representa un entero largo) en un valor `long`. La función devuelve el valor `long`. Si la cadena no se puede convertir, la función `atol` devuelve cero. Si tanto `int` como `long` se almacenan en cuatro bytes, la función `atoi` y la función `atol` trabajan en forma idéntica.

```

1 // Fig. 21.22: fig21_21.cpp
2 // Uso de atof.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // prototipo de atof
8 using std::atof;
9
10 int main()
11 {
12     double d = atof( "99.0" ); // convierte cadena a double
13
14     cout << "La cadena \"99.0\" convertida a double es " << d
15         << "\nEl valor convertido dividido entre 2 es " << d / 2.0 << endl;
16     return 0;
17 } // fin de main

```

```

La cadena "99.0" convertida a double es 99
El valor convertido dividido entre 2 es 49.5

```

Figura 21.22 | La función `atof` de conversión de cadenas.

```

1 // Fig. 21.23: fig21_23.cpp
2 // Uso de atoi.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // prototipo de atoi
8 using std::atoi;
9
10 int main()
11 {
12     int i = atoi( "2593" ); // convierte cadena a int

```

Figura 21.23 | La función `atoi` de conversión de cadenas. (Parte 1 de 2).

```

13
14     cout << "La cadena \"2593\" convertida a int es " << i
15     << "\nEl valor convertido menos 593 es " << i - 593 << endl;
16
17 } // fin de main

```

```

La cadena "2593" convertida a int es 2593
El valor convertido menos 593 es 2000

```

Figura 21.23 | La función atoi de conversión de cadenas. (Parte 2 de 2).

```

1 // Fig. 21.24: fig21_24.cpp
2 // Uso de atol.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // prototipo de atol
8 using std::atol;
9
10 int main()
11 {
12     long x = atol( "1000000" ); // convierte cadena a long
13
14     cout << "La cadena \"1000000\" convertida a long es " << x
15     << "\nEl valor convertido dividido entre 2 es " << x / 2 << endl;
16
17 } // fin de main

```

```

La cadena "1000000" convertida a long es 1000000
El valor convertido dividido entre 2 es 500000

```

Figura 21.24 | La función atol de conversión de cadenas.

La función **strtod** (figura 21.25) convierte una secuencia de caracteres que representan un valor de punto flotante a **double**. La función **strtod** recibe dos argumentos: una cadena (**char ***) y la dirección de un apuntador **char *** (es decir, un **char ****). La cadena contiene la secuencia de caracteres que se va a convertir a **double**. El segundo argumento permite a **strtod** modificar un apuntador **char *** en la función que hizo la llamada, de tal forma que el apuntador apunte a la ubicación del primer carácter después de la porción convertida de la cadena. En la línea 16 se indica que a **d** se le asigna el valor **double** convertido de **cadena1**, y que a **cadenaPtr** se le asigna la ubicación del primer carácter después del valor convertido (51.2) en **cadena1**.

```

1 // Fig. 21.25: fig21_25.cpp
2 // Uso de strtod.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // prototipo de strtod
8 using std::strtod;
9
10 int main()
11 {
12     double d;
13     const char *cadena1 = "51.2% se admite";
14     char *cadenaPtr;
15

```

Figura 21.25 | La función strtod de conversión de cadenas. (Parte 1 de 2).

```

16     d = strtod( cadena1, &cadenaPtr ); // convierte caracteres a double
17
18     cout << "La cadena \\" << cadena1
19         << "\\ se convierte al\nvalor double " << d
20         << " y la cadena \\" << cadenaPtr << "\\" << endl;
21
22 } // fin de main

```

La cadena "51.2% se admite" se convierte al valor double 51.2 y la cadena "% se admite"

Figura 21.25 | La función `strtod` de conversión de cadenas. (Parte 2 de 2).

La función `strtol` (figura 21.26) convierte a `long` una secuencia de caracteres que representa un entero. La función recibe tres argumentos: una cadena (`char *`), la dirección de un apuntador `char *` y un entero. La cadena contiene la secuencia de caracteres a convertir. Al segundo argumento se le asigna la ubicación del primer argumento, después de la porción convertida de la cadena. El entero especifica la *base* del valor que se va a convertir. En la línea 16 se indica que a `x` se le asigna el valor `long` convertido de `cadena1`, y que a `restoPtr` se le asigna la ubicación del primer carácter después del valor convertido (-1234567) en `cadena1`. Si utilizamos un apuntador nulo para el segundo argumento, se ignorará el resto de la cadena. El tercer argumento (0) indica que el valor a convertir puede estar en octal (base 8), decimal (base 10) o hexadecimal (base 16). Esto se determina mediante los caracteres iniciales en la cadena: 0 indica un número octal, 0x indica hexadecimal y un número de 1-9 indica decimal.

En una llamada a la función `strtol`, la base se puede especificar como cero o cualquier valor entre 2 y 36. (En el apéndice D podrá obtener una explicación detallada sobre los sistemas numéricos octal, decimal, hexadecimal y binario). Las representaciones numéricas de los enteros de base 11 a base 36 utilizan los caracteres A a Z para representar los valores 10 a 35. Por ejemplo, los valores hexadecimales pueden consistir de los dígitos 0 a 9 y de los caracteres A a F. Un entero en base 11 puede consistir de los dígitos 0 a 9 y del carácter A. Un entero en base 24 puede consistir de los dígitos 0 a 9 y de los caracteres A a N. Un entero en base 36 puede consistir de los dígitos 0 a 9 y de los caracteres A a Z. [Nota: no importa si la letra utilizada está en mayúscula o minúscula].

La función `strtoul` (figura 21.27) convierte a `unsigned long` una secuencia de caracteres que representan un entero `unsigned long`. La función trabaja en forma idéntica a `strtol`. En la línea 17 se indica que a `x` se le asigna el valor `unsigned long` convertido de `cadena`, y que a `restoPtr` se le asigna la ubicación del primer carácter después del valor convertido (1234567) en `cadena1`. El tercer argumento (0) indica que el valor a convertir puede estar en formato octal, decimal o hexadecimal, dependiendo de los caracteres iniciales.

```

1 // Fig. 21.26: Fig21_26.cpp
2 // Uso de strtol.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // prototipo de strtol
8 using std::strtol;
9
10 int main()
11 {
12     long x;
13     const char *cadena1 = "-1234567abc";
14     char *restoPtr;
15
16     x = strtol( cadena1, &restoPtr, 0 ); // convierte caracteres a long
17
18     cout << "La cadena original es \\" << cadena1
19         << "\\nEl valor convertido es " << x
20         << "\\nEl resto de la cadena original es \\" << restoPtr
21         << "\\nEl valor convertido mas 567 es " << x + 567 << endl;

```

Figura 21.26 | La función `strtol` de conversión de cadenas. (Parte 1 de 2).

```

22     return 0;
23 } // fin de main

```

```

La cadena original es "-1234567abc"
El valor convertido es -1234567
El resto de la cadena original es "abc"
El valor convertido más 567 es -1234000

```

Figura 21.26 | La función `strtol` de conversión de cadenas. (Parte 2 de 2).

```

1 // Fig. 21.27: fig21_27.cpp
2 // Uso de strtoul.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // prototipo de strtoul
8 using std::strtoul;
9
10 int main()
11 {
12     unsigned long x;
13     const char *cadena1 = "1234567abc";
14     char *restoPtr;
15
16     // convierte una secuencia de caracteres a unsigned long
17     x = strtoul( cadena1, &restoPtr, 0 );
18
19     cout << "La cadena original es \""
20         << "\"\nEl valor convertido es " << x
21         << "\nEl resto de la cadena original es \""
22         << restoPtr
23         << "\nEl valor convertido menos 567 es " << x - 567 << endl;
24     return 0;
25 } // fin de main

```

```

La cadena original es "1234567abc"
El valor convertido es 1234567
El resto de la cadena original es "abc"
El valor convertido menos 567 es 1234000

```

Figura 21.27 | La función `strtoul` de conversión de cadenas.

21.11 Funciones de búsqueda de la biblioteca de manejo de cadenas basadas en apuntador

En esta sección presentamos las funciones de la biblioteca de manejo de cadenas que se utilizan para buscar caracteres y otras cadenas dentro de una cadena. Las funciones se sintetizan en la figura 21.28. Observe que las funciones `strcspn` y `strspn` especifican el tipo de valor de retorno `size_t`. Éste es un tipo definido por el estándar como el tipo integral del valor devuelto por el operador `sizeof`.

Prototipo	Descripción
<code>char *strchr(const char *s, int c)</code>	Localiza la primera ocurrencia del carácter <code>c</code> en la cadena <code>s</code> . Si se encuentra <code>c</code> , se devuelve un apuntador a <code>c</code> en <code>s</code> . En caso contrario se devuelve un apuntador nulo.
<code>char * strrchr(const char *s, int c)</code>	Busca desde el final de la cadena <code>s</code> y localiza la última ocurrencia del carácter <code>c</code> en la cadena <code>s</code> . Si se encuentra <code>c</code> , se devuelve un apuntador a <code>c</code> en la cadena <code>s</code> . En caso contrario se devuelve un apuntador nulo.

Figura 21.28 | Funciones de búsqueda de la biblioteca de manejo de cadenas basada en apuntador. (Parte 1 de 2).

Prototipo	Descripción
<code>size_t strspn(const char *s1, const char *s2)"</code>	Determina y devuelve la longitud del segmento inicial de la cadena <code>s1</code> , que consiste sólo de los caracteres contenidos en la cadena <code>s2</code> .
<code>char *strupbrk(const char *s1, const char *s2)</code>	Localiza la primera ocurrencia en la cadena <code>s1</code> de cualquier carácter en la cadena <code>s2</code> . Si se encuentra un carácter de la cadena <code>s2</code> , se devuelve un apuntador al carácter en la cadena <code>s1</code> . En caso contrario se devuelve un apuntador nulo.
<code>size_t strcspn(const char *s1, const char *s2)</code>	Determina y devuelve la longitud del segmento inicial de la cadena <code>s1</code> , que consiste de los caracteres que no están contenidos en la cadena <code>s2</code> .
<code>char *strstr(const char *s1, const char *s2)</code>	Localiza la primera ocurrencia en la cadena <code>s1</code> de la cadena <code>s2</code> . Si se encuentra la cadena, se devuelve un apuntador a la cadena en <code>s1</code> . En caso contrario se devuelve un apuntador nulo.

Figura 21.28 | Funciones de búsqueda de la biblioteca de manejo de cadenas basada en apuntador. (Parte 2 de 2).

La función `strchr` busca la primera ocurrencia de un carácter en una cadena. Si se encuentra el carácter, `strchr` devuelve un apuntador al carácter en la cadena; en caso contrario, `strchr` devuelve un apuntador nulo. El programa de la figura 21.29 utiliza `strchr` (líneas 17 y 25) para buscar las primeras ocurrencias de '`a`' y '`z`' en la cadena "`Esta es una prueba`".

```

1 // Fig. 21.29: fig21_29.cpp
2 // Uso de strchr.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // prototipo de strchr
8 using std::strchr;
9
10 int main()
11 {
12     const char *cadena1 = "Esta es una prueba";
13     char caracter1 = 'a';
14     char caracter2 = 'z';
15
16     // busca el caracter1 en cadena1
17     if ( strchr( cadena1, caracter1 ) != NULL )
18         cout << '\'' << caracter1 << "' se encontró en \""
19         << cadena1 << "\">\n";
20     else
21         cout << '\'' << caracter1 << "' no se encontró en \""
22         << cadena1 << "\">\n";
23
24     // busca el caracter2 en cadena1
25     if ( strchr( cadena1, caracter2 ) != NULL )
26         cout << '\'' << caracter2 << "' se encontró en \""
27         << cadena1 << "\">\n";
28     else
29         cout << '\'' << caracter2 << "' no se encontró en \""
30         << cadena1 << "\">\n" << endl;
31
32     return 0;
33 } // fin de main

```

Figura 21.29 | La función `strchr` para realizar búsquedas en cadenas. (Parte 1 de 2).

```
'a' se encontro en "Esta es una prueba".
'z' no se encontro en "Esta es una prueba".
```

Figura 21.29 | La función `strchr` para realizar búsquedas en cadenas. (Parte 2 de 2).

La función `strcspn` (figura 21.30, línea 18) determina la longitud de la parte inicial de la cadena en su primer argumento que no contiene caracteres de la cadena en su segundo argumento. La función devuelve la longitud del segmento.

La función `strpbrk` busca la primera ocurrencia en el primer argumento de cadena de cualquier carácter en su segundo argumento de cadena. Si se encuentra un carácter del segundo argumento, `strpbrk` devuelve un apuntador al carácter en el primer argumento; en caso contrario, `strpbrk` devuelve un apuntador nulo. En la línea 16 de la figura 21.31 se localiza la primera ocurrencia en `cadena1` de cualquier carácter de `cadena2`.

```
1 // Fig. 21.30: fig21_30.cpp
2 // Uso de strcspn.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // prototipo de strcspn
8 using std::strcspn;
9
10 int main()
11 {
12     const char *cadena1 = "El valor es 3.14159";
13     const char *cadena2 = "1234567890";
14
15     cout << "cadena1 = " << cadena1 << "\ncadena2 = " << cadena2
16         << "\n\nLa longitud del segmento inicial de cadena1"
17         << "\nque no contiene caracteres de cadena2 = "
18         << strcspn( cadena1, cadena2 ) << endl;
19
20 } // fin de main
```

```
cadena1 = El valor es 3.14159
cadena2 = 1234567890
```

```
La longitud del segmento inicial de cadena1
que no contiene caracteres de cadena2 = 12
```

Figura 21.30 | La función `strcspn` para realizar búsquedas en cadenas.

```
1 // Fig. 21.31: fig21_31.cpp
2 // Uso de strpbrk.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // prototipo de strpbrk
8 using std::strpbrk;
9
10 int main()
11 {
12     const char *cadena1 = "Esta es una prueba";
13     const char *cadena2 = "cuidado";
14
15     cout << "De los caracteres en \" " << cadena2 << "\"\n"
16         << *strpbrk( cadena1, cadena2 ) << "' es el primer caracter "
17         << "que aparece en\n\" " << cadena1 << '\"' << endl;
```

Figura 21.31 | La función `strpbrk` para realizar búsquedas en cadenas. (Parte 1 de 2).

```

18     return 0;
19 } // fin de main

```

De los caracteres en "cuidado"
 'a' es el primer carácter que aparece en
 "Esta es una prueba"

Figura 21.31 | La función `strpbrk` para realizar búsquedas en cadenas. (Parte 2 de 2).

La función `strrchr` busca la última ocurrencia del carácter especificado en una cadena. Si se encuentra el carácter, `strrchr` devuelve un apuntador al carácter en la cadena; en caso contrario, `strrchr` devuelve 0. La línea 18 de la figura 21.32 busca la última ocurrencia del carácter 'z' en la cadena "Un zoologico tiene muchos animales incluyendo zopilotes".

La función `strspn` (figura 21.33, línea 18) determina la longitud de la parte inicial de la cadena en su primer argumento que sólo contiene caracteres de la cadena en su segundo argumento. La función devuelve la longitud del segmento.

La función `strstr` busca la primera ocurrencia de su segundo argumento de cadena en su primer argumento de cadena. Si se encuentra la segunda cadena en la primera, se devuelve un apuntador a la ubicación de la cadena en el primer argumento; en caso contrario devuelve 0. En la línea 18 de la figura 21.34 se utiliza `strstr` para buscar la cadena "def" en la cadena "abcdefabcdef".

```

1 // Fig. 21.32: fig21_32.cpp
2 // Uso de strrchr.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // prototipo de strrchr
8 using std::strrchr;
9
10 int main()
11 {
12     const char *cadena1 = "Un zoologico tiene muchos animales incluyendo zopilotes";
13     char c = 'z';
14
15     cout << "cadena1 = " << cadena1 << "\n" << endl;
16     cout << "El resto de cadena1 empezando con la\n"
17         << "última ocurrencia del carácter '\n"
18         << c << "' es: \"<< strrchr( cadena1, c ) << '\" << endl;
19     return 0;
20 } // fin de main

```

cadena1 = Un zoologico tiene muchos animales incluyendo zopilotes
 El resto de cadena1 empezando con la
 última ocurrencia del carácter 'z' es: "zopilotes"

Figura 21.32 | La función `strrchr` para realizar búsquedas en cadenas.

```

1 // Fig. 21.33: fig21_33.cpp
2 // Uso de strspn.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // prototipo de strspn

```

Figura 21.33 | La función `strspn` para realizar búsquedas en cadenas. (Parte 1 de 2).

```

8  using std::strspn;
9
10 int main()
11 {
12     const char *cadena1 = "El valor es 3.14159";
13     const char *cadena2 = "aerls Eov";
14
15     cout << "cadena1 = " << cadena1 << "\ncadena2 = " << cadena2
16         << "\n\nLa longitud del segmento inicial de cadena1\n"
17         << "que solo contiene caracteres de cadena 2 cadena2 = "
18         << strspn( cadena1, cadena2 ) << endl;
19     return 0;
20 } // fin de main

```

```

cadena1 = El valor es 3.14159
cadena2 = aerls Eov

La longitud del segmento inicial de cadena1
que solo contiene caracteres de cadena 2 = 12

```

Figura 21.33 | La función `strspn` para realizar búsquedas en cadenas. (Parte 2 de 2).

```

1  // Fig. 21.34: fig21_34.cpp
2  // Uso de strstr.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <cstring> // prototipo de strstr
8  using std::strstr;
9
10 int main()
11 {
12     const char *cadena1 = "abcdefabcdef";
13     const char *cadena2 = "def";
14
15     cout << "cadena1 = " << cadena1 << "\ncadena2 = " << cadena2
16         << "\n\nEl resto de cadena1 que empieza con la\n"
17         << "primera ocurrencia de cadena2 es: "
18         << strstr( cadena1, cadena2 ) << endl;
19     return 0;
20 } // fin de main

```

```

cadena1 = abcdefabcdef
cadena2 = def

El resto de cadena1 que empieza con la
primera ocurrencia de cadena2 es: defabcdef

```

Figura 21.34 | La función `strstr` para realizar búsquedas en cadenas.

21.12 Funciones de memoria de la biblioteca de manejo de cadenas basadas en apuntador

Las funciones de manejo de cadenas que presentamos en esta sección facilitan la manipulación, comparación y realización de búsquedas en bloques de memoria. Las funciones tratan a los bloques de memoria como arreglos de bytes. Estas funciones pueden manipular cualquier bloque de datos. La figura 21.35 sintetiza las funciones de memoria de la biblioteca de manejo de cadenas. En las discusiones acerca de las funciones, “objeto” se refiere a un bloque de datos. [Nota: las funciones de procesamiento de cadenas en las secciones anteriores operan sobre cadenas de caracteres con terminación nula. Las funciones en esta sección operan sobre arreglos de bytes. El valor de carácter nulo (es decir, un byte que contiene 0) no tiene aplicación con las funciones en esta sección].

Prototipo	Descripción
void *memcpy(void *s1, const void *s2, size_t n)	Copia n caracteres del objeto al que apunta s2 y los coloca en el objeto al que apunta s1. Se devuelve un apuntador al objeto resultante. El área de la que se copian los caracteres no puede traslapar el área a la que se copian los caracteres.
void *memmove(void *s1, const void *s2, size_t n)	Copia n caracteres del objeto al que apunta s2 y los coloca en el objeto al que apunta s1. La copia se realiza como si los caracteres se copiaran primero del objeto al que apunta s2 hacia un arreglo temporal, y después se copian del arreglo temporal al objeto al que apunta s1. Se devuelve un apuntador al objeto resultante. El área a partir de la cual se copian los caracteres puede traslapar el área a la que se copian los caracteres.
int memcmp(const void *s1, const void *s2, size_t n)	Compara los primeros n caracteres de los objetos a los que apuntan s1 y s2. La función devuelve 0, un valor menor que 0 o mayor que 0 si s1 es igual, menor o mayor que s2, respectivamente.
void *memchr(const void *s, int c, size_t n)	Localiza la primera ocurrencia de c (se convierte a <code>unsigned char</code>) en los primeros n caracteres del objeto al que apunta s. Si se encuentra c, se devuelve un apuntador a c en el objeto. En caso contrario, se devuelve 0.
void *memset(void *s, int c, size_t n)	Copia c (se convierte a <code>unsigned char</code>) a los primeros n caracteres del objeto al que apunta s. Se devuelve un apuntador al resultado.

Figura 21.35 | Funciones de memoria de la biblioteca de manejo de caracteres.

Los parámetros tipo apuntador para estas funciones se declaran `void *`. En el capítulo 8 vimos que un apuntador a cualquier tipo de datos se puede asignar directamente a un apuntador de tipo `void *`. Por esta razón, estas funciones pueden recibir apuntadores a cualquier tipo de datos. Recuerde que un apuntador de tipo `void *` no se puede asignar directamente a un apuntador de cualquier otro tipo de datos. Como un apuntador `void *` no se puede desreferenciar, cada función recibe un argumento de tamaño que especifica el número de caracteres (bytes) que procesará la función. Para simplificar, los ejemplos en esta sección manipulan arreglos de caracteres (bloques de caracteres).

La función `memcpy` copia un número especificado de caracteres (bytes) del objeto al que apunta su segundo argumento, al objeto al que apunta su primer argumento. La función puede recibir un apuntador a cualquier tipo de objeto. El resultado de esta función es indefinido si los dos objetos se traslanan en memoria (es decir, si son parte del mismo objeto). El programa de la figura 21.36 utiliza `memcpy` (línea 17) para copiar la cadena del arreglo s2 al arreglo s1.

```

1 // Fig. 21.36: fig21_36.cpp
2 // Uso de memcpy.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // prototipo de memcpy
8 using std::memcpy;
9
10 int main()
11 {
12     char s1[ 18 ];
13
14     // 18 caracteres en total (incluye la terminación nula)
15     char s2[] = "Copia esta cadena";
16
17     memcpy( s1, s2, 18 ); // copia 18 caracteres de s2 a s1
18

```

Figura 21.36 | La función `memcpy` para manejo de memoria. (Parte I de 2).

```

19     cout << "Una vez que se copia s2 a s1 mediante memcpy,\n"
20         << "s1 contiene \"\" << s1 << '\"' << endl;
21     return 0;
22 } // fin de main

```

Una vez que se copia s2 a s1 mediante memcpy,
s1 contiene "Copia esta cadena"

Figura 21.36 | La función `memcpy` para manejo de memoria. (Parte 2 de 2).

Al igual que `memcpy`, la función `memmove` copia un número especificado de bytes del objeto al que apunta su segundo argumento, al objeto al que apunta su primer argumento. La copia se realiza como si los bytes se copiaran del segundo argumento a un arreglo temporal de caracteres, y después se copian del arreglo temporal al primer argumento. Esto permite copiar caracteres de una parte de una cadena, a otra parte de la misma cadena.

Error común de programación 21.8



Las funciones de manipulación de cadenas distintas de `memmove` que copian caracteres tienen resultados indefinidos cuando la copia se realiza entre partes de la misma cadena.

El programa de la figura 21.37 utiliza `memmove` (línea 16) para copiar los últimos 11 bytes del arreglo `x` a los primeros 11 bytes del arreglo `x`.

```

1 // Fig. 21.37: fig21_37.cpp
2 // Uso de memmove.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // prototipo de memmove
8 using std::memmove;
9
10 int main()
11 {
12     char x[] = "Hogar dulce hogar";
13
14     cout << "La cadena en el arreglo x antes de memmove es: " << x;
15     cout << "\nLa cadena en el arreglo x despues de memmove es: "
16         << static_cast< char * >( memmove( x, &x[ 6 ], 11 ) ) << endl;
17     return 0;
18 } // fin de main

```

La cadena en el arreglo x antes de memmove es: Hogar dulce hogar
La cadena en el arreglo x despues de memmove es: dulce hogar hogar

Figura 21.37 | La función `memmove` para manejo de memoria.

La función `memcmp` (figura 21.38, líneas 19, 20 y 21) compara el número especificado de caracteres de su primer argumento con los caracteres correspondientes de su segundo argumento. La función devuelve un valor mayor que cero si el primer argumento es mayor que el segundo argumento, cero si los argumentos son iguales, y un valor menor que cero si el primer argumento es menor que el segundo. [Nota: con algunos compiladores, la función `memcmp` devuelve -1, 0 o 1, como en la salida de ejemplo de la figura 21.38. Con otros compiladores, esta función devuelve 0 o la diferencia entre los códigos numéricos de los primeros caracteres que difieren en las cadenas que se van a comparar. Por ejemplo, cuando se comparan `s1` y `s2`, el primer carácter que difiere entre estas dos cadenas es el quinto carácter de cada cadena: E (código numérico 69) para `s1` y X (código numérico 72) para `s2`. En este caso, el valor de retorno será 19 (o -19 cuando se compara `s2` con `s1`)].

La función `memchr` busca la primera ocurrencia de un byte, representado como un valor `unsigned char`, en el número especificado de bytes de un objeto. Si el byte se encuentra en el objeto, se devuelve un apuntador al byte; en caso contrario, la función devuelve un apuntador nulo. En la línea 16 de la figura 21.39 se busca el carácter (byte) 'r' en la cadena "Este es un corazon".

```

1 // Fig. 21.38: fig21_38.cpp
2 // Uso de memcmp.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 #include <cstring> // prototipo de memcmp
11 using std::memcmp;
12
13 int main()
14 {
15     char s1[] = "ABCDEFG";
16     char s2[] = "ABCDXYZ";
17
18     cout << "s1 = " << s1 << "\ns2 = " << s2 << endl
19         << "\nmemcmp(s1, s2, 4) = " << setw( 3 ) << memcmp( s1, s2, 4 )
20         << "\nmemcmp(s1, s2, 7) = " << setw( 3 ) << memcmp( s1, s2, 7 )
21         << "\nmemcmp(s2, s1, 7) = " << setw( 3 ) << memcmp( s2, s1, 7 )
22         << endl;
23
24     return 0;
25 } // fin de main

```

```

s1 = ABCDEFG
s2 = ABCDXYZ

memcmp(s1, s2, 4) =  0
memcmp(s1, s2, 7) = -1
memcmp(s2, s1, 7) =  1

```

Figura 21.38 | La función `memcmp` para manejo de memoria.

```

1 // Fig. 21.39: fig21_39.cpp
2 // Uso de memchr.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // prototipo de memchr
8 using std::memchr;
9
10 int main()
11 {
12     char s[] = "Este es un corazon";
13
14     cout << "s = " << s << "\n" << endl;
15     cout << "El resto de s despues de encontrar el caracter 'r' es \""
16         << static_cast< char * >( memchr( s, 'r', 18 ) ) << "\"" << endl;
17
18     return 0;
19 } // fin de main

```

```

s = Este es un corazon
El resto de s despues de encontrar el caracter 'r' es "razon"

```

Figura 21.39 | La función `memchr` para manejo de memoria.

La función `memset` copia el valor del byte en su segundo argumento a un número especificado de byte del objeto al que apunta su primer argumento. En la línea 16 de la figura 21.40 se utiliza `memset` para copiar 'b' a los primeros 7 bytes de `cadena1`.

```

1 // Fig. 21.40: fig21_40.cpp
2 // Uso de memset.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // prototipo de memset
8 using std::memset;
9
10 int main()
11 {
12     char cadena1[ 15 ] = "BBBBBBBBBBBBBB";
13
14     cout << "cadena1 = " << cadena1 << endl;
15     cout << "cadena1 despues de memset = "
16         << static_cast< char * >( memset( cadena1, 'b', 7 ) ) << endl;
17     return 0;
18 } // fin de main

```

```

cadena1 = BBBB BBBB BBBB BB
cadena1 despues de memset = bbbb bbb BBBB BBBB

```

Figura 21.40 | La función `memset` para manejo de memoria.

21.13 Repaso

En este capítulo presentamos las definiciones `struct`, cómo inicializarlas y utilizarlas con funciones. Hablamos sobre el uso de `typedef` para crear alias y ayudar a promover la portabilidad. También presentamos los operadores a nivel de bits para manipular datos y campos de bits, para almacenar datos en forma compacta. También aprendió acerca de las funciones de conversión de cadenas en `<cstdlib>` y las funciones de procesamiento de cadenas en `<cstring>`. En el siguiente capítulo continuaremos nuestra discusión acerca de las estructuras de datos, al hablar sobre los contenedores: estructuras de datos definidas en la Biblioteca de plantillas estándar de C++. También presentaremos los diversos algoritmos definidos en la STL.

Resumen

Sección 21.2 Definiciones de estructuras

- Las estructuras son colecciones de variables relacionadas (o agregados) bajo un solo nombre.
- Las estructuras pueden contener variables de distintos tipos de datos.
- La palabra clave `struct` empieza toda definición de estructura. Entre las llaves de la definición de la estructura se encuentran las declaraciones de sus miembros.
- Los miembros de la misma estructura deben tener nombres únicos.
- La definición de una estructura crea un nuevo tipo de datos que se puede utilizar para declarar variables.

Sección 21.3 Inicialización de estructuras

- Una estructura se puede inicializar mediante una lista inicializadora, colocando después de la variable en la declaración un signo de igual y una lista separada por comas de inicializadores encerrados entre llaves. Si hay menos inicializadores en la lista que miembros en la estructura, el resto de los miembros se inicializan en cero (o apuntador nulo en los miembros apuntadores).
- Se pueden asignar variables de estructuras completas a variables de estructuras del mismo tipo.
- Una variable de estructura se puede inicializar con una variable de estructura del mismo tipo.

Sección 21.4 Uso de estructuras con funciones

- Las variables de estructuras y los miembros de estructuras individuales se pasan a las funciones por valor.
- Para pasar una estructura por referencia, hay que pasar la dirección de la variable de la estructura o una referencia a esa variable. Un arreglo de estructuras se pasa por referencia. Para pasar un arreglo por valor, hay que crear una estructura con un arreglo como miembro.

Sección 21.5 `typedef`

- Al crear un nuevo nombre de tipo con `typedef` no se crea un nuevo tipo; se crea un nombre que es sinónimo de un tipo definido anteriormente.

Sección 21.7 Operadores a nivel de bits

- El operador AND a nivel de bits (`&`) recibe dos operandos enteros. Un bit en el resultado se establece si uno de los bits correspondientes en cada uno de los operandos es uno.
- Las máscaras se utilizan con el operador AND a nivel de bits para ocultar ciertos bits, mientras se preservan otros.
- El operador OR inclusivo a nivel de bits (`|`) recibe dos operandos. Un bit en el resultado se establece a uno si el bit correspondiente en cualquier operando es uno.
- Cada uno de los operadores a nivel de bits (excepto el complemento) tiene su correspondiente operador de asignación.
- El operador OR exclusivo a nivel de bits (`^`) recibe dos operandos. Un bit en el resultado se establece a uno si exactamente uno de los bits correspondientes en los dos operandos es uno.
- El operador de desplazamiento a la izquierda (`<<`) desplaza los bits de su operando a la izquierda, con base en el número de bits especificados por su operando derecho. Los bits que se vacían a la derecha se sustituyen con ceros.
- El operador de desplazamiento a la derecha (`>>`) desplaza los bits de su operando a la derecha, con base en el número de bits especificados en su operando derecho. Al desplazar a la derecha un entero sin signo, los bits que se vacían a la izquierda se reemplazan con ceros. Los bits que se vacían en los enteros con signo se pueden reemplazar con ceros o unos.
- El operador de complemento a nivel de bits (`~`) recibe un operando e invierte sus bits; esto produce el complemento a uno del operando.

Sección 21.8 Campos de bits

- Los campos de bits reducen el uso del almacenamiento al guardar los datos en el mínimo número de bits requeridos. Los miembros de los campos de bits se pueden declarar como `int` o `unsigned`.
- Para declarar un campo de bits, se coloca un signo de dos puntos y la anchura del campo de bits después del nombre de miembro `unsigned` o `int`.
- La anchura del campo de bits debe ser una constante entera.
- Si se especifica un campo de bits sin un nombre, se utiliza como relleno en la estructura.
- Un campo de bits sin nombre con anchura 0 alinea el siguiente campo de bits en un nuevo límite de palabra de máquina.

Sección 21.9 Biblioteca de manejo de caracteres

- La función `islower` determina si su argumento es una letra minúscula (a a z). La función `isupper` determina si su argumento es una letra mayúscula (A a Z).
- La función `isdigit` determina si su argumento es un dígito (0 a 9).
- La función `isalpha` determina si su argumento es una letra mayúscula (A a Z) o minúscula (a a z).
- La función `isalnum` determina si su argumento es una letra mayúscula (A a Z), minúscula (a a z) o un dígito (0 a 9).
- La función `isxdigit` determina si su argumento es un dígito hexadecimal (A a F, a a f, 0 a 9).
- La función `toupper` convierte una letra minúscula en una letra mayúscula. La función `tolower` convierte una letra mayúscula en minúscula.
- La función `isspace` determina si su argumento es uno de los siguientes caracteres de espacio en blanco: ' ' (espacio), '\f', '\n', '\r', '\t' o '\v'.
- La función `iscntrl` determina si su argumento es un carácter de control tal como '\t', '\v', '\f', '\a', '\b', '\r' o '\n'.
- La función `ispunct` determina si su argumento es un carácter de impresión distinto de un espacio, dígito o letra.
- La función `isprint` determina si su argumento es un carácter que puede mostrarse en la pantalla (incluyendo el carácter de espacio).
- La función `isgraph` determina si su argumento es un carácter que puede mostrarse en la pantalla, distinto del espacio.

Sección 21.10 Funciones de conversión de cadenas basadas en apuntador

- La función `atof` convierte su argumento (una cadena que empieza con una serie de dígitos que representa un número de punto flotante) a un valor `double`.
- La función `atoi` convierte su argumento (una cadena que empieza con una serie de dígitos que representa un entero) a un valor `int`.
- La función `atol` convierte su argumento (una cadena que empieza con una serie de dígitos que representa un entero largo) a un valor `long`.
- La función `strtod` convierte una secuencia de caracteres que representan un valor de punto flotante a `double`. La función recibe dos argumentos: una cadena (`char *`) y la dirección de un apuntador `char *`. La cadena contiene la secuencia de caracteres que se va a convertir, y al apuntador a `char *` se le asigna el resto de la cadena después de la conversión.

- La función `strtol` convierte a `long` una secuencia de caracteres que representa un entero. La función recibe tres argumentos: una cadena (`char *`), la dirección de un apuntador `char *` y un entero. La cadena contiene la secuencia de caracteres a convertir, al apuntador a `char *` se le asigna la ubicación del primer carácter después del valor convertido, y el entero especifica la base del valor que se va a convertir.
- La función `strtoul` convierte a `unsigned long` una secuencia de caracteres que representan un entero. La función recibe tres argumentos: una cadena (`char *`), la dirección de un apuntador `char *` y un entero. La cadena contiene la secuencia de caracteres a convertir, al apuntador a `char *` se le asigna la ubicación del primer carácter después del valor convertido, y el entero especifica la base del valor que se va a convertir.

Sección 21.11 Funciones de búsqueda de la biblioteca de manejo de cadenas basadas en apuntador

- La función `strchr` busca la primera ocurrencia de un carácter en una cadena. Si se encuentra el carácter, `strchr` devuelve un apuntador al carácter en la cadena; en caso contrario, `strchr` devuelve un apuntador nulo.
- La función `strcspn` determina la longitud de la parte inicial de la cadena en su primer argumento que no contiene caracteres de la cadena en su segundo argumento. La función devuelve la longitud del segmento.
- La función `strupr` busca la primera ocurrencia en su primer argumento de cualquier carácter que aparezca en su segundo argumento. Si se encuentra un carácter del segundo argumento, `strupr` devuelve un apuntador al carácter en el primer argumento; en caso contrario, `strupr` devuelve un apuntador nulo.
- La función `strrchr` busca la última ocurrencia de un carácter en una cadena. Si se encuentra el carácter, `strrchr` devuelve un apuntador al carácter en la cadena; en caso contrario, devuelve un apuntador nulo.
- La función `strspn` determina la longitud de la parte inicial de su primer argumento que sólo contiene caracteres de la cadena en su segundo argumento y devuelve la longitud del segmento.
- La función `strstr` busca la primera ocurrencia de su segundo argumento de cadena en su primer argumento de cadena. Si se encuentra la segunda cadena en la primera, se devuelve un apuntador a la ubicación de la cadena en el primer argumento; en caso contrario devuelve 0.

Sección 21.12 Funciones de memoria de la biblioteca de manejo de cadenas basadas en apuntador

- La función `memcpy` copia un número especificado de caracteres del objeto al que apunta su segundo argumento, al objeto al que apunta su primer argumento. La función puede recibir un apuntador a cualquier tipo de objeto. Los apuntadores se reciben como apuntadores `void` y se convierten en apuntadores `char` para utilizarlos en la función. La función `memcpy` manipula los bytes de su argumento como caracteres.
- La función `memmove` copia un número especificado de bytes del objeto al que apunta su segundo argumento, al objeto al que apunta su primer argumento. La copia se realiza como si los bytes se copiaran del segundo argumento a un arreglo de caracteres temporal, y después se copiaran del arreglo temporal al primer argumento.
- La función `memcmp` compara el número especificado de caracteres de su primero y segundo argumentos.
- La función `memchr` busca la primera ocurrencia de un byte (representada como `unsigned char`) en el número de bytes especificado de un objeto. Si se encuentra el byte, se devuelve un apuntador a éste; en caso contrario, se devuelve un apuntador nulo.
- La función `memset` copia su segundo argumento (que se trata como `unsigned char`) a un número especificado de bytes del objeto al que apunta el primer argumento.

Terminología

<code>&</code> , operador AND a nivel de bits	campo de bits con anchura cero
<code>&=</code> , operador AND de asignación a nivel de bits	campo de bits sin nombre
<code><<</code> , operador de desplazamiento a la izquierda	complemento a uno
<code><<=</code> , operador de asignación de desplazamiento a la izquierda	<code><cstdlib></code>
<code>>></code> , operador de desplazamiento a la derecha	estructura autorreferenciada
<code>>>=</code> , operador de asignación de desplazamiento a la derecha	funciones de conversión de cadenas
<code>^</code> , operador OR exclusivo a nivel de bits	<code>isalnum</code>
<code>^=</code> , operador OR exclusivo de asignación a nivel de bits	<code>isalpha</code>
<code> </code> , operador OR inclusivo a nivel de bits	<code>iscntrl</code>
<code> =</code> , operador OR inclusivo de asignación a nivel de bits	<code>isdigit</code>
<code>~, operador de complemento a nivel de bits</code>	<code>isgraph</code>
<code>anchura de un campo de bits</code>	<code>islower</code>
<code>atof</code>	<code>isprint</code>
<code>atoi</code>	<code>ispunct</code>
<code>atol</code>	<code>isspace</code>
biblioteca general de utilerías	<code>isupper</code>
campo de bits	<code>isxdigit</code>

máscara	strrchr
memcmp	strspn
memcpy	strstr
memchr	strtod
memmove	strtol
memset	strtoul
nombre de estructura	struct
operadores a nivel de bits	tipo de datos agregado
operadores de asignación a nivel de bits	tipo de estructura
relleno	tolower
strcspn	toupper
strchr	typedef
strpbrk	

Ejercicios de autoevaluación

21.1 Complete los siguientes enunciados:

- a) Una _____ es una colección de variables relacionadas bajo un solo nombre.
- b) Los bits en el resultado de una expresión que utilice el operador _____ se establecen en uno si los bits correspondientes en cada operando se establecen en uno. En caso contrario, los bits se establecen en cero.
- c) Las variables declaradas en la definición de una estructura se conoce como sus _____.
- d) Los bits en el resultado de una expresión que utilice el operador _____ se establecen en uno si por lo menos uno de los bits correspondientes en cualquiera de sus operandos se establece en uno. En caso contrario, los bits se establecen en cero.
- e) La palabra clave _____ introduce la declaración de una estructura.
- f) La palabra clave _____ se utiliza para crear un sinónimo para un tipo de datos definido con anterioridad.
- g) Cada bit en el resultado de una expresión que utilice el operador _____ se establece en uno si exactamente uno de los bits correspondientes en cualquiera de sus operandos se establece en uno.
- h) El operador AND a nivel de bits & se utiliza comúnmente para _____ bits (es decir, seleccionar ciertos bits de una cadena de bits, mientras se establecen en cero los demás).
- i) Para acceder al miembro de una estructura, se utilizan los operadores _____ o _____.
- j) Los operadores _____ y _____ se utilizan para desplazar los bits de un valor a la izquierda y a la derecha, respectivamente.

21.2 Conteste con *verdadero* o *falso* cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.

- a) Las estructuras sólo pueden contener un tipo de datos.
- b) Los miembros de distintas estructuras deben tener nombres únicos.
- c) La palabra clave `typedef` se utiliza para definir nuevos tipos de datos.
- d) Las estructuras siempre se pasan a las funciones por referencia.

21.3 Escriba una sola instrucción o un conjunto de instrucciones para realizar cada una de las siguientes acciones:

- a) Defina una estructura llamada `Pieza`, que contenga la variable `int numeroPieza` y el arreglo `char nombrePieza`, cuyos valores pueden tener una longitud de hasta 25 caracteres.
- b) Defina a `PiezaPtr` como sinónimo para el tipo `Pieza *`.
- c) Utilice instrucciones separadas para declarar la variable `a` de tipo `Pieza`, el arreglo `b[10]` de tipo `Pieza` y la variable `ptr` de tipo apuntador a `Pieza`.
- d) Lea el número y nombre de una pieza mediante el teclado, y coloque estos datos en los miembros de la variable `a`.
- e) Asigne los valores de los miembros de la variable `a` al elemento tres del arreglo `b`.
- f) Asigne la dirección del arreglo `b` a la variable apuntador `ptr`.
- g) Imprima los valores de los miembros del elemento tres del arreglo `b`, usando la variable `ptr` y el operador apuntador a estructura para hacer referencia a los nombres.

21.4 Encuentre el error en cada uno de los siguientes enunciados:

- a) Suponga que se ha definido la `struct Carta` y que contiene dos apuntadores al tipo `char`; `a_saber`, `cara` y `palo`. Además, se ha declarado la variable `c` de tipo `Carta`, y la variable `cPtr` de tipo apuntador a `Carta`. A la variable `cPtr` se le ha asignado la dirección de `c`.

```
cout << *cPtr.cara << endl;
```

- b) Suponga que se ha definido la `struct Carta` y que contiene dos apuntadores al tipo `char`; a `saber`, `cara` y `palo`. Además, se ha declarado el arreglo `corazones[13]` de tipo `Carta`. La siguiente instrucción debe imprimir el miembro `cara` del elemento 10 del arreglo.

```
cout << corazones.cara << endl;
```

- c) struct Persona

```
{
    char apellidoPaterno[ 15 ];
    char primerNombre[ 15 ];
    int edad;
}
```

- d) Suponga que se ha declarado la variable `p` de tipo `Persona`, y que se ha declarado la variable `c` de tipo `Carta`.

```
p = c;
```

- 21.5** Escriba una sola instrucción para realizar cada una de las siguientes acciones. Suponga que las variables `c` (que almacena un carácter), `x`, `y` y `z` son de tipo `int`; las variables `d`, `e` y `f` son de tipo `double`; la variable `ptr` es de tipo `char *` y los arreglos `s1[100]` y `s2[100]` son de tipo `char`.

- Convierta el carácter almacenado en la variable `c` a una letra mayúscula. Asigne el resultado a la variable `c`.
- Determine si el valor de la variable `c` es un dígito. Use el operador condicional como se muestra en las figuras 21.18 a 21.20 para imprimir " es un " o " no es un " cuando se muestre el resultado.
- Convierta la cadena "1234567" a `long` e imprima el valor.
- Determine si el valor de la variable `c` es un carácter de control. Use el operador condicional para imprimir " es un " o " no es un " cuando se muestre el resultado.
- Asigne a `ptr` la ubicación de la última ocurrencia de `c` en `s1`.
- Convierta la cadena "8.63582" a `double` e imprima el valor.
- Determine si el valor de `c` es una letra. Utilice el operador condicional para imprimir " es una " o " no es una " cuando se muestre el resultado.
- Asigne a `ptr` la ubicación de la primera ocurrencia de `s2` en `s1`.
- Determine si el valor de la variable `c` es un carácter de impresión. Use el operador condicional para imprimir " es un " o " no es un " cuando se muestre el resultado.
- Asigne a `ptr` la ubicación de la primera ocurrencia en `s1` de cualquier carácter de `s2`.
- Asigne a `ptr` la ubicación de la primera ocurrencia de `c` en `s1`.
- Convierta la cadena "-21" a `int`, e imprima el valor.

Respuestas a los ejercicios de autoevaluación

- 21.1** a) estructura. b) AND a nivel de bits (&). c) miembros. d) OR inclusivo a nivel de bits (|). e) `struct`. f) `typedef`. g) OR exclusivo a nivel de bits (^). h) máscara. i) miembro de estructura (.), apuntador de estructura (->). j) operador de desplazamiento a la izquierda (<<), operador de desplazamiento a la derecha (>>).

- 21.2** a) Falso. Una estructura puede contener muchos tipos de datos.
 b) Falso. Los miembros de estructuras separadas pueden tener los mismos nombres, pero los miembros de la misma estructura deben tener nombres únicos.
 c) Falso. `typedef` se utiliza para definir alias para los tipos de datos definidos con anterioridad.
 d) Falso. Las estructuras se pasan a las funciones por valor de manera predeterminada, y se pueden pasar por referencia.

- 21.3** a) struct Pieza

```
{
    int numeroPieza;
    char numeroPieza[ 26 ];
};
```

- b) `typedef Pieza * PiezaPtr;`

- c) `Pieza a;`

```
Pieza b[ 10 ];
```

```
Pieza *ptr;
```

- d) `cin >> a.numeroPieza >> a.nombrePieza;`

- e) `b[3] = a;`

- f) `ptr = b;`

- g) `cout << (ptr + 3)->numeroPieza << ''
 << (ptr + 3)->numeroPieza << endl;`
- 21.4 a) *Error:* se han omitido los paréntesis que deben encerrar a `*cPtr`, lo cual hace que el orden de evaluación de la expresión sea incorrecto.
 b) *Error:* se ha omitido el subíndice del arreglo. La expresión debe ser `corazones[10].cara.`
 c) *Error:* se requiere un punto y coma para terminar la definición de una estructura.
 d) *Error:* las variables de tipos distintos de estructuras no se pueden asignar unas a otras.
- 21.5 a) `c = toupper(c);`
 b) `cout << '\' << c << '\''
 << (isdigit(c) ? "es un" : "no es un")
 << " digito" << endl;`
 c) `cout << atol("1234567") << endl;`
 d) `cout << '\' << c << '\''
 << (iscntrl(c) ? "es un" : "no es un")
 << " caracter de control" << endl;`
 e) `ptr = strrchr(s1, c);`
 f) `cout << atof("8.63582") << endl;`
 g) `cout << '\' << c << '\''
 << (isalpha(c) ? "es una" : "no es una")
 << " letra" << endl;`
 h) `ptr = strstr(s1, s2);`
 i) `cout << '\' << c << '\''
 << (isprint(c) ? "es un" : "no es un")
 << " caracter de impresión" << endl;`
 j) `ptr = strpbrk(s1, s2);`
 k) `ptr = strchr(s1, c);`
 l) `cout << atoi("-21") << endl;`

Ejercicios

- 21.6 Proporcione la definición para cada una de las siguientes estructuras:
 a) La estructura `Inventario`, que contiene el arreglo de caracteres `nombrePieza[30]`, el entero `numeroPieza`, el valor de punto flotante `precio`, el entero `existencia` y el entero `resurtir`.
 b) Una estructura llamada `Direccion` que contiene los arreglos de caracteres `direccionCalle[25]`, `ciudad[20]`, `estado[3]` y `codigoPostal[6]`.
 c) La estructura `Estudiante`, que contiene los arreglos `primerNombre[15]` y `apellidoPaterno[15]`, y la variable `direccionHogar` del tipo `struct Direccion` de la parte (b).
 d) La estructura `Prueba`, que contiene 16 campos de bits con anchuras de 1 bit. Los nombres de los campos de bits son las letras `a` a `p`.

- 21.7 Consideré las siguientes definiciones de estructuras y declaraciones de variables:

```
struct Cliente {  

    char apellidoPaterno[ 15 ]  

    char primerNombre[ 15 ]  

    int numeroCliente;  

    struct {  

        char numeroTelefonico[ 11 ];  

        char direccion[ 50 ];  

        char ciudad[ 15 ];  

        char estado[ 3 ];  

        char codigoPostal[ 6 ];  

    } personal;  

}  

registroCliente, *clientePtr;  

clientePtr = & registroCliente;
```

Escriba una expresión separada que acceda a los miembros de la estructura en cada una de las siguientes partes:

- a) El miembro `apellidoPaterno` de la estructura `registroCliente`.
- b) El miembro `apellidoPaterno` de la estructura a la que apunta `clientePtr`.
- c) El miembro `primerNombre` de la estructura `registroCliente`.
- d) El miembro `primerNombre` de la estructura a la que apunta `clientePtr`.
- e) El miembro `numeroCliente` de la estructura `registroCliente`.
- f) El miembro `numeroCliente` de la estructura a la que apunta `clientePtr`.
- g) El miembro `numeroTelefonico` del miembro `personal` de la estructura `registroCliente`.
- h) El miembro `numeroTelefonico` del miembro `personal` de la estructura a la que apunta `clientePtr`.
- i) El miembro `direccion` del miembro `personal` de la estructura `registroCliente`.
- j) El miembro `direccion` del miembro `personal` de la estructura a la que apunta `clientePtr`.
- k) El miembro `ciudad` del miembro `personal` de la estructura `registroCliente`.
- l) El miembro `ciudad` del miembro `personal` de la estructura a la que apunta `clientePtr`.
- m) El miembro `estado` del miembro `personal` de la estructura `registroCliente`.
- n) El miembro `estado` del miembro `personal` de la estructura a la que apunta `clientePtr`.
- o) El miembro `codigoPostal` del miembro `personal` de la estructura `registroCliente`.
- p) El miembro `codigoPostal` del miembro `personal` de la estructura a la que apunta `clientePtr`.

21.8 Modifique el programa de la figura 21.14 para barajar las cartas usando un algoritmo de alto rendimiento, como se muestra en la figura 21.3. Imprima el mazo resultante en formato de dos columnas, como en la figura 21.4. Anteponga a cada carta su color.

21.9 Escriba un programa que desplace a la derecha una variable entera por 4 bits. El programa debe imprimir el entero en bits antes y después de la operación de desplazamiento. ¿Su sistema coloca ceros o unos en los bits vacíos?

21.10 Desplazar a la izquierda un entero `unsigned` por 1 bit equivale a multiplicar el valor por 2. Escriba la función `potencia2` que reciba dos argumentos enteros llamados `numero` y `pot`, y que calcule

```
numero * 2^pot
```

Utilice un operador de desplazamiento para calcular el resultado. El programa debe imprimir los valores como enteros y como bits.

21.11 El operador de desplazamiento a la izquierda se puede utilizar para empaquetar dos valores tipo carácter en una variable entera sin signo de dos bytes. Escriba un programa que reciba como entrada dos caracteres mediante el teclado, y los pase a la función `empaquetarCaracteres`. Para empaquetar dos caracteres en una variable entera `unsigned`, asigne el primer carácter a la variable `unsigned`, desplace la variable `unsigned` 8 posiciones de bit a la izquierda y combine la variable `unsigned` con el segundo carácter, usando el operador OR inclusivo a nivel de bits. El programa debe imprimir los caracteres en su formato de bits antes y después de empaquetarlos en el entero `unsigned` para demostrar que, de hecho, se empaquetaron correctamente en la variable `unsigned`.

21.12 Use el operador de desplazamiento a la derecha, el operador AND a nivel de bits y una máscara para escribir la función `desempaquetarCaracteres` que reciba el entero `unsigned` del ejercicio 21.11 y lo desempaque en dos caracteres. Para desempaquetar dos caracteres de un entero `unsigned` de dos bytes, combine el entero sin signo con la máscara 65280 (11111111 00000000) y desplace a la derecha el resultado por 8 bits. Asigne el valor resultante a una variable `char`. Después combine el entero `unsigned` con la máscara 255 (00000000 11111111). Asigne el resultado a otra variable `char`. El programa debe imprimir el entero `unsigned` en bits antes de desempaquetarlo, y después debe imprimir los caracteres en bits para confirmar que se hayan desempaquetado en forma correcta.

21.13 Si su sistema utiliza enteros de cuatro bytes, vuelva a escribir el programa del ejercicio 21.11 para empaquetar cuatro caracteres.

21.14 Si su sistema utiliza enteros de cuatro bytes, vuelva a escribir la función `desempaquetarCaracteres` del ejercicio 21.12 para desempaquetar cuatro caracteres. Cree las máscaras que necesita para desempaquetar los cuatro caracteres, desplazando el valor 255 a la izquierda en la máscara por 8 bits 0, 1, 2 o 3 veces (dependiendo del byte que esté desempaquetando).

21.15 Escriba un programa que invierta el orden de los bits en un valor entero `unsigned`. El programa debe introducir el valor del usuario y llamar a la función `invertirBits` para imprimir los bits en orden inverso. Imprima el valor en bits, antes y después de que éstos se inviertan para confirmar que se hayan invertido en forma apropiada.

21.16 Escriba un programa que demuestre cómo pasar un arreglo por valor. [Sugerencia: use una `struct`]. Para demostrar que se haya pasado una copia, modifique la copia del arreglo en la función a la que se llamó.

21.17 Escriba un programa que reciba como entrada un carácter mediante el teclado y que lo evalúe con cada función en la biblioteca de manejo de caracteres. Imprima el valor devuelto por cada función.

21.18 El siguiente programa utiliza la función `multiplo` para determinar si el entero que se introdujo mediante el teclado es un múltiplo de algún entero X. Examine la función `multiplo` y después determine el valor de X.

```

1 // Ejercicio 21.18: ej21_18.cpp
2 // Este programa determina si un valor es un múltiplo de X.
3 // #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 bool multiplo( int );
10
11 int main()
12 {
13     int y;
14
15     cout << "Escriba un entero entre 1 y 32000: ";
16     cin >> y;
17
18     if ( multiplo( y ) )
19         cout << y << " es un múltiplo de X" << endl;
20     else
21         cout << y << " no es un múltiplo de X" << endl;
22
23     return 0;
24
25 } // fin de main
26
27 // determina si num es múltiplo de X
28 bool múltiplo( int num )
29 {
30     bool mult = true;
31
32     for ( int i = 0, mascara = 1; i < 10; i++, mascara <= 1 )
33
34         if ( ( num & mascara ) != 0 ) {
35             mult = false;
36             break;
37
38         } // fin de if
39
40     return mult;
41
42 } // fin de la función multiplo

```

21.19 ¿Qué es lo que hace el siguiente programa?

```

1 // Ejercicio 21.19: ej21_19.cpp
2 #include <iostream>
3
4 using std::cout;
5 using std::cin;
6 using std::endl;
7 using std::boolalpha;
8
9 bool misterio( unsigned );
10
11 int main()
12 {
13     unsigned x;
14
15     cout << "Escriba un entero: ";
16     cin >> x;
17     cout << boolalpha
18         << "El resultado es " << misterio( x ) << endl;

```

```

19
20     return 0;
21
22 } // fin de main
23
24 // ¿Qué hace esta función?
25 bool misterio( unsigned bits )
26 {
27     const int DESPL = 8 * sizeof( unsigned ) - 1;
28     const unsigned MASCARA = 1 << DESPL;
29     unsigned total = 0;
30
31     for ( int i = 0; i < DESPL + 1; i++, bits <= 1 )
32
33         if ( ( bits & MASCARA ) == MASCARA )
34             ++total;
35
36     return !( total % 2 );
37
38 } // fin de la función misterio

```

21.20 Escriba un programa que reciba una línea de texto con la función miembro `getline` de `istream` (como en el capítulo 15) y la coloque en el arreglo de caracteres `s[100]`. Imprima la línea en letras mayúsculas y minúsculas.

21.21 Escriba un programa que reciba como entrada cuatro cadenas que representen enteros, convierta las cadenas a enteros, sume los valores e imprima el total de los cuatro valores. Use sólo las técnicas de procesamiento de cadenas estilo C que mostramos en este capítulo.

21.22 Escriba un programa que reciba como entrada cuatro cadenas que representen valores de punto flotante, convierta las cadenas a valores `double`, sume los valores e imprima el total de los cuatro valores. Use sólo las técnicas de procesamiento de cadenas estilo C que mostramos en este capítulo.

21.23 Escriba un programa que reciba una línea de texto y una cadena de búsqueda mediante el teclado. Utilizando la función `strstr`, localice la primera ocurrencia de la cadena de búsqueda en la línea de texto y asigne la ubicación a la variable `busquedaPtr` de tipo `char *`. Si se encuentra la cadena de búsqueda, imprima el resto de la línea de texto, empezando con la cadena de búsqueda. Después utilice `strstr` de nuevo para localizar la siguiente ocurrencia de la cadena de búsqueda en la línea de texto. Si se encuentra una segunda ocurrencia, imprima el resto de la línea, empezando con la segunda ocurrencia. [Sugerencia: la segunda llamada a `strstr` debe contener la expresión `busquedaPtr + 1` como su primer argumento].

21.24 Escriba un programa con base en el programa del ejercicio 21.23 que reciba varias líneas de texto y una cadena de búsqueda, y que después utilice la función `strstr` para determinar el número total de ocurrencias de la cadena en las líneas de texto. Imprima el resultado.

21.25 Escriba un programa que reciba varias líneas de texto y un carácter de búsqueda, y que utilice la función `strchr` para determinar el número total de ocurrencias del carácter en las líneas de texto.

21.26 Escriba un programa con base en el programa del ejercicio 21.25, que reciba como entrada varias líneas de texto y utilice la función `strchr` para determinar el número total de ocurrencias de cada letra del alfabeto en el texto. Las letras mayúsculas y minúsculas deben contarse en conjunto. Almacene los totales para cada letra en un arreglo e imprima los valores en formato tabular, después de haber determinado los totales.

21.27 La tabla en el apéndice B muestra las representaciones de código numérico para los caracteres en el conjunto de caracteres ASCII. Estudie esta tabla y después indique si cada uno de los siguientes enunciados es *verdadero* o *falso*.

- La letra "A" está antes que la letra "B".
 - El dígito "9" está antes que el dígito "0".
 - Los símbolos de uso común para la suma, resta, multiplicación y división están antes que cualquiera de los dígitos.
 - Los dígitos están antes que las letras.
 - Si un programa ordena cadenas en orden ascendente, entonces debe colocar el símbolo de un paréntesis derecho antes del símbolo de un paréntesis izquierdo.
- 21.28** Escriba un programa que lea una serie de cadenas e imprima sólo las que empiecen con la letra "b".
- 21.29** Escriba un programa que lea una serie de cadenas e imprima sólo las que terminen con las letras "ED".
- 21.30** Escriba un programa que reciba como entrada un código ASCII e imprima el carácter correspondiente. Modifique este programa, de manera que genere todos los códigos posibles de tres dígitos en el rango de 000 a 255, y que trate de imprimir los caracteres correspondientes. ¿Qué ocurre al ejecutar este programa?

- 21.31** Utilice la tabla de caracteres ASCII en el apéndice B como guía para escribir sus propias versiones de las funciones de manejo de caracteres de la figura 21.17.
- 21.32** Escriba sus propias versiones de las funciones de la figura 21.21 para convertir cadenas a números.
- 21.33** Escriba sus propias versiones de las funciones de la figura 21.28 para realizar búsquedas en cadenas.
- 21.34** Escriba sus propias versiones de las funciones de la figura 21.35 para manipular bloques de memoria.
- 21.35** (*Proyecto: un corrector ortográfico*) Muchos paquetes populares de software de procesamiento de palabras tienen correctores ortográficos integrados. Utilizamos las herramientas de corrección ortográfica para preparar este libro y descubrimos que, sin importar qué tan cuidadosos fuéramos al escribir un capítulo, el software siempre podía encontrar unos cuantos errores ortográficos más de los que podíamos detectar en forma manual.

En este proyecto le pedimos que desarrolle su propia herramienta de corrección ortográfica. Haremos unas sugerencias para ayudarlo a empezar. Después será conveniente que agregue más herramientas. Tal vez sea bueno que utilice un diccionario computarizado como fuente de palabras.

¿Por qué escribimos tantas palabras en forma incorrecta? En algunos casos es porque simplemente no conocemos la manera correcta de escribirlas, por lo que tratamos de adivinar lo mejor que podemos. En otros casos, es porque transponemos dos letras (por ejemplo, “perdeterminado” en lugar de “predeterminado”). Algunas veces escribimos una letra doble por accidente (por ejemplo, “útil” en vez de “ útil”). Otras veces escribimos una tecla que está cerca de la que pretendíamos escribir (por ejemplo, “cumpleaños” en vez de “cumpleaños”), y así sucesivamente.

Diseñe e implemente un programa de corrección ortográfica. Su programa debe mantener un arreglo de cadenas de caracteres llamado `listaDePalabras`. Puede introducir estas cadenas u obtenerlas de un diccionario computarizado.

Su programa le pide al usuario que escriba una palabra. A continuación el programa busca en el arreglo `listaDePalabras`. Si la palabra se encuentra en el arreglo, su programa deberá imprimir “La palabra está escrita correctamente”.

Si la palabra no se encuentra en el arreglo, su programa debe imprimir “La palabra no está escrita correctamente”. Después su programa debe tratar de localizar otras palabras en la `listaDePalabras` que puedan ser la palabra que el usuario trataba de escribir. Por ejemplo, puede probar con todas las transposiciones simples posibles de letras adyacentes para descubrir que la palabra “predeterminado” concuerda directamente con una palabra en `listaDePalabras`. Desde luego que esto implica que su programa comprobará todas las otras transposiciones posibles, tales como “rpedeterminado”, “perdeterminado”, “predetremiando”, “predetemrinado” y “predetermniado”. Cuando encuentre una nueva palabra que concuerde con una en la `listaDePalabras`, imprima esa palabra en un mensaje tal como “¿Quiso usted decir “predeterminado”?”.

Implemente otras pruebas, tales como reemplazar cada letra doble con una sola letra y cualquier otra prueba que pueda desarrollar para aumentar el valor de su corrector ortográfico.



*¡Las figuras que puede
contener un contenedor
brillante!*

—Theodore Roethke

*Un viaje a través de todo
el universo en un mapa.*

—Miguel de Cervantes

*Ob, maldita iteración,
que eres capaz de corromper
basta un santo.*

—William Shakespeare

*Ese gran montón de polvo
llamado "historia".*

—Augustine Birrell

*El historiador es un profeta
en reversa.*

—Friedrich von Schlegel

*Intenta hasta el final,
y nunca te detengas
ante la duda;
Nada es tan difícil que no
se pueda resolver mediante
investigación.*

—Robert Herrick

Biblioteca de plantillas estándar (STL)

OBJETIVOS

En este capítulo aprenderá a:

- Utilizar los contenedores de la STL, los adaptadores de contenedores y los “casi contenedores”.
- Programar con las docenas de algoritmos de la STL.
- Entender cómo los algoritmos utilizan los iteradores para acceder a los elementos de los contenedores de la STL.
- Familiarizarse con los recursos de la STL disponibles en Internet y en World Wide Web.

- 22.1** Introducción a la Biblioteca de plantillas estándar (STL)
 - 22.1.1** Introducción a los contenedores
 - 22.1.2** Introducción a los iteradores
 - 22.1.3** Introducción a los algoritmos
- 22.2** Contenedores de secuencia
 - 22.2.1** Contenedor de secuencia `vector`
 - 22.2.2** Contenedor de secuencia `list`
 - 22.2.3** Contenedor de secuencia `deque`
- 22.3** Contenedores asociativos
 - 22.3.1** Contenedor asociativo `multiset`
 - 22.3.2** Contenedor asociativo `set`
 - 22.3.3** Contenedor asociativo `multimap`
 - 22.3.4** Contenedor asociativo `map`
- 22.4** Adaptadores de contenedores
 - 22.4.1** Adaptador `stack`
 - 22.4.2** Adaptador `queue`
 - 22.4.3** Adaptador `priority_queue`
- 22.5** Algoritmos
 - 22.5.1** `fill`, `fill_n`, `generate` y `generate_n`
 - 22.5.2** `equal`, `mismatch` y `lexicographical_compare`
 - 22.5.3** `remove`, `remove_if`, `remove_copy` y `remove_copy_if`
 - 22.5.4** `replace`, `replace_if`, `replace_copy` y `replace_copy_if`
 - 22.5.5** Algoritmos matemáticos
 - 22.5.6** Algoritmos básicos de búsqueda y ordenamiento
 - 22.5.7** `swap`, `iter_swap` y `swap_ranges`
 - 22.5.8** `copy_backward`, `merge`, `unique` y `reverse`
 - 22.5.9** `inplace_merge`, `unique_copy` y `reverse_copy`
 - 22.5.10** Operaciones set
 - 22.5.11** `lower_bound`, `upper_bound` y `equal_range`
 - 22.5.12** Ordenamiento de montón (heapsort)
 - 22.5.13** `min` y `max`
 - 22.5.14** Algoritmos de la STL que no se cubren en este capítulo
- 22.6** La clase `bitset`
- 22.7** Objetos de funciones
- 22.8** Repaso
- 22.9** Recursos Web de la STL

22.1 Introducción a la Biblioteca de plantillas estándar (STL)

En repetidas ocasiones hemos enfatizado la importancia de reutilizar software. Al reconocer que los programadores de C++ utilizan comúnmente muchas estructuras de datos y algoritmos, el comité del estándar de C++ agregó la **Biblioteca de plantillas estándar (STL)** a la Biblioteca estándar de C++. La STL define componentes poderosos, basados en plantillas y reutilizables, que implementan muchas estructuras de datos comunes y algoritmos que se utilizan para procesar esas estructuras de datos. La STL ofrece la prueba del concepto para la programación genérica con plantillas, que se introdujo en el capítulo 14, Plantillas, y se utilizó extensivamente en el capítulo 20, Estructuras de datos. [Nota: en la industria, las características que se presentan en este capítulo se conocen comúnmente como la Biblioteca de plantillas estándar de la STL. Sin embargo, estos términos no se utilizan en el documento del estándar de C++, debido a que estas características se consideran simplemente como parte de la Biblioteca estándar de C++].

La STL fue desarrollada por Alexander Stepanov y Meng Lee en Hewlett-Packard, y se basa en su investigación en el campo de la programación genérica, con contribuciones considerables por parte de David Musser. Como veremos, la STL fue concebida y diseñada teniendo en cuenta el rendimiento y la flexibilidad.

En este capítulo presentamos la STL y hablamos sobre sus tres componentes clave: **contenedores** (plantillas de estructuras de datos populares), **iteradores** y **algoritmos**. Los contenedores de la STL son estructuras de datos capaces de almacenar objetos de casi cualquier tipo de datos (hay ciertas restricciones). Más adelante veremos que hay tres estilos de clases contenedoras: **contenedores de primera clase, adaptadores y casi contenedores**.



Tip de rendimiento 22.1

Para cualquier aplicación específica, pueden ser apropiados varios contenedores distintos de la STL. Seleccione el contenedor más apropiado que logre el mejor rendimiento (es decir, balance entre velocidad y tamaño) para esa aplicación. La eficiencia fue una consideración crucial en el diseño de la STL.



Tip de rendimiento 22.2

Las herramientas de la Biblioteca estándar se implementan para operar con eficiencia en muchas aplicaciones. Para ciertas aplicaciones con requerimientos de rendimiento únicos, tal vez sea necesario que el programador escriba sus propias implementaciones personalizadas.

Cada contenedor de la STL tiene funciones miembro asociadas. En todos los contenedores de la STL hay definido un conjunto de estas funciones miembro. Presentamos la mayor parte de estas funcionalidades comunes en nuestros ejemplos de los contenedores de la STL **vector** (un arreglo que puede cambiar su tamaño en forma dinámica, y que presentamos en el capítulo 7, Arreglos y vectores), **list** (una lista con enlace doble) y **deque** (una cola con doble terminación, y se pronuncia como “deck”). Presentaremos la funcionalidad específica para cada contenedor en ejemplos para cada uno de los otros contenedores de la STL.

Los iteradores de la STL, que tienen propiedades similares a las de los apuntadores, son utilizados por los programas para manipular los elementos de los contenedores de la STL. De hecho, los arreglos estándar se pueden manipular mediante algoritmos de la STL, utilizando apuntadores estándar como iteradores. Más adelante veremos que es conveniente manipular los contenedores mediante iteradores, ya que nos proporcionan un tremendo poder de expresión al combinarlos con los algoritmos de la STL; en ciertos casos, se reducen muchas líneas de código a una sola instrucción. Hay cinco categorías de iteradores, las cuales describiremos en la sección 22.1.2 y utilizaremos en el capítulo.

Los algoritmos de la STL son funciones que realizan manipulaciones de datos comunes, tales como búsqueda, ordenamiento y comparación de elementos (o contenedores completos). La STL proporciona aproximadamente 70 algoritmos. La mayoría utilizan iteradores para acceder a los elementos de un contenedor. Cada algoritmo tiene requerimientos mínimos para los tipos de iteradores que se pueden utilizar con éste. Más adelante veremos que cada contenedor de primera clase soporta tipos específicos de iteradores, algunos más poderosos que otros. El tipo de iterador soportado por un contenedor determina si éste se puede utilizar con un algoritmo específico. Los iteradores encapsulan el mecanismo utilizado para acceder a los elementos del contenedor. Este encapsulamiento permite aplicar muchos de los algoritmos a varios contenedores, sin tener que preocuparse por la implementación subyacente del contenedor. Mientras que los iteradores de un contenedor soporten los requerimientos mínimos del algoritmo, éste puede procesar los elementos de ese contenedor. Esto también permite a los programadores crear nuevos algoritmos que puedan procesar los elementos de varios tipos de contenedores.



Observación de Ingeniería de Software 22.1

La metodología de la STL permite escribir los programas generales de tal manera que el código no dependa del contenedor subyacente. Dicho estilo de programación se conoce como programación genérica.

En el capítulo 20 estudiamos las estructuras de datos. Creamos listas enlazadas, colas, pilas y árboles. Entrelazamos con cuidado los objetos con apuntadores. El código basado en apuntador es complejo, y la más ligera omisión o descuido puede producir graves violaciones al acceso de memoria y errores de fuga de memoria, sin que el compilador se queje. Para implementar estructuras de datos adicionales, como deques, colas de prioridad, conjuntos y mapas, se requiere un trabajo adicional considerable. Además, si muchos programadores en un proyecto extenso implementan contenedores y algoritmos similares para distintas tareas, el código se vuelve difícil de modificar, mantener y depurar. Una ventaja de la STL es que los programadores pueden reutilizar los contenedores, iteradores y algoritmos de la STL para implementar representaciones de datos y manipulaciones comunes. Esta reutilización puede ahorrar una cantidad considerable de tiempo de desarrollo, dinero y esfuerzo.



Observación de Ingeniería de Software 22.2

Evite reinventar la rueda; programe con los componentes reutilizables de la Biblioteca estándar de C++. La STL incluye muchas de las estructuras de datos más populares como contenedores, y proporciona varios algoritmos populares para procesar datos en estos contenedores.



Tip para prevenir errores 22.1

Al programar estructuras de datos basadas en apuntadores y algoritmos, debemos realizar nuestro propio proceso de prueba y depuración para estar seguros de que nuestras estructuras de datos, clases y algoritmos funcionen de manera apropiada. Es fácil cometer errores al manipular apuntadores a bajo nivel. Las fugas de memoria y las violaciones de acceso a la memoria son comunes en dicho código personalizado. Para la mayoría de los programadores y para la mayoría de las aplicaciones que tendrán que escribir, los contenedores tipo plantilla pre-empaquetados de la STL son suficientes. El uso de la STL ayuda a los programadores a reducir el tiempo de prueba y depuración. Hay que tener en cuenta que, para los proyectos extensos, el tiempo de compilación de las plantillas puede ser considerable.

En este capítulo se introduce la STL. Esto por ningún motivo significa que esté completo o sea exhaustivo. Sin embargo, es un capítulo amigable y accesible, que convencerá al lector acerca del valor de la STL en la reutilización de software, y le servirá como base para un estudio más detallado.

22.1.1 Introducción a los contenedores

Los tipos de contenedores de la STL se muestran en la figura 22.1. Los contenedores se dividen en tres categorías principales: **contenedores de secuencia**, **contenedores asociativos** y **adaptadores de contenedores**.

Generalidades de los contenedores de la STL

Los contenedores de secuencia representan estructuras de datos lineales, tales como vectores y listas enlazadas. Los contenedores asociativos son contenedores no lineales que por lo general pueden localizar elementos almacenados en ellos rápidamente. Dichos contenedores pueden almacenar conjuntos de valores, o pares clave/valor. Los contenedores de secuencia y los asociativos se conocen colectivamente como contenedores de primera clase. Como vimos en el capítulo 20, las pilas y las colas en realidad son versiones restringidas de los contenedores de secuencia. Por esta razón, la STL implementa a las pilas y colas como adaptadores de contenedores que permiten a un programa ver a un contenedor de secuencia en una manera restringida. Existen otros tipos de contenedores que se consideran “casi contenedores”: los

Clase contenedora de la Biblioteca estándar	Descripción
<i>Contenedores de secuencia</i>	
<code>vector</code>	Inserciones y eliminaciones rápidas en la parte final. Acceso directo a cualquier elemento.
<code>deque</code>	Inserciones y eliminaciones rápidas en la parte inicial o final. Acceso directo a cualquier elemento.
<code>list</code>	Lista con enlace doble, inserción y eliminación rápida en cualquier parte.
<i>Contenedores asociativos</i>	
<code>set</code>	Búsqueda rápida, no se permiten duplicados.
<code>multiset</code>	Búsqueda rápida, se permiten duplicados
<code>map</code>	Asignación de uno a uno, no se permiten duplicados, búsqueda rápida basada en claves.
<code>multimap</code>	Asignación de uno a varios, se permiten duplicados, búsqueda rápida basada en claves.
<i>Adaptadores de contenedores</i>	
<code>stack</code>	Último en entrar, primero en salir (UEPS).
<code>queue</code>	Primero en entrar, primero en salir (PEPS).
<code>priority_queue</code>	El elemento de mayor prioridad siempre es el primero en salir.

Figura 22.1 | Clases contenedoras de la Biblioteca estándar.

arreglos basados en apuntador tipo C (descritos en el capítulo 7), los contenedores `bitset` para mantener conjuntos de valores de bandera y los contenedores `valarray` para llevar a cabo operaciones vectoriales matemáticas de alta velocidad (esta última clase está optimizada para un buen rendimiento del compilador y no es tan flexible como los contenedores de primera clase). Estos tipos se consideran como “casi contenedores”, ya que exhiben capacidades similares a las de los contenedores de primera clase, pero no soportan todas sus capacidades. El tipo `string` (descrito en el capítulo 18) soporta la misma funcionalidad que un contenedor de secuencia, pero sólo almacena datos tipo carácter.

Funciones comunes de los contenedores de la STL

La mayoría de los contenedores de la STL proporcionan una funcionalidad similar. Hay muchas operaciones genéricas, como la función miembro `size`, que se aplican a todos los contenedores, y otras operaciones que se aplican a subconjuntos de contenedores similares. Esto promueve la extensibilidad de la STL con nuevas clases. La figura 22.2 describe las funciones comunes para todos los contenedores de la Biblioteca estándar. [Nota: Los operadores sobrecargados `operator<`, `operator<=`, `operator>`, `operator>=`, `operator==` y `operator!=` no se proporcionan para contenedores `priority_queue`].

Funciones miembro comunes para todos los contenedores de la STL	Descripción
constructor predeterminado	Un constructor para crear un contenedor vacío. Por lo general, cada contenedor cuenta con varios constructores que proporcionan distintos métodos de inicialización.
constructor de copia	Un constructor que inicializa al contenedor para que sea una copia de un contenedor existente del mismo tipo.
destructor	La función destructora para encargarse de la limpieza, una vez que el contenedor ya no es necesario.
<code>empty</code>	Devuelve <code>true</code> si no hay elementos en el contenedor; en caso contrario devuelve <code>false</code> .
<code>insert</code>	Inserta un elemento en el contenedor.
<code>size</code>	Devuelve el número de elementos que hay actualmente en el contenedor.
<code>operator=</code>	Asigna un contenedor a otro.
<code>operator<</code>	Devuelve <code>true</code> si el primer contenedor es menor que el segundo; en caso contrario devuelve <code>false</code> .
<code>operator<=</code>	Devuelve <code>true</code> si el primer contenedor es menor o igual que el segundo; en caso contrario devuelve <code>false</code> .
<code>operator></code>	Devuelve <code>true</code> si el primer contenedor es mayor que el segundo; en caso contrario devuelve <code>false</code> .
<code>operator>=</code>	Devuelve <code>true</code> si el primer contenedor es mayor o igual que el segundo; en caso contrario devuelve <code>false</code> .
<code>operator==</code>	Devuelve <code>true</code> si el primer contenedor es igual que el segundo; en caso contrario devuelve <code>false</code> .
<code>operator!=</code>	Devuelve <code>true</code> si el primer contenedor es distinto que el segundo; en caso contrario devuelve <code>false</code> .
<code>swap</code>	Intercambia los elementos de dos contenedores.
<i>Funciones que sólo se encuentran en contenedores de primera clase</i>	
<code>max_size</code>	Devuelve el número máximo de elementos para un contenedor.
<code>begin</code>	Las dos versiones de esta función devuelven ya sea un <code>iterator</code> o un <code>const_iterator</code> que hace referencia al primer elemento del contenedor.
<code>end</code>	Las dos versiones de esta función devuelven ya sea un <code>iterator</code> o un <code>const_iterator</code> que hace referencia a la siguiente posición después del final del contenedor.

Figura 22.2 | Funciones comunes de los contenedores de la STL. (Parte I de 2).

Funciones miembro comunes para todos los contenedores de la STL	Descripción
<code>rbegin</code>	Las dos versiones de esta función devuelven ya sea un <code>reverse_iterator</code> o un <code>const_reverse_iterator</code> que hace referencia al último elemento del contenedor.
<code>rend</code>	Las dos versiones de esta función devuelven ya sea un <code>reverse_iterator</code> o un <code>const_reverse_iterator</code> que hace referencia a la posición que está antes del primer elemento del contenedor.
<code>erase</code>	Elimina uno o más elementos del contenedor.
<code>clear</code>	Elimina todos los elementos del contenedor.

Figura 22.2 | Funciones comunes de los contenedores de la STL. (Parte 2 de 2).

Archivos de encabezado de los contenedores de la STL

Los archivos de encabezado para cada uno de los contenedores de la Biblioteca estándar se muestran en la figura 22.3. Todo el contenido de estos archivos de encabezado está en el namespace `std`.

Definiciones `typedef` comunes de los contenedores de primera clase

La figura 22.4 muestra los elementos `typedef` comunes (para crear sinónimos o alias para nombres de tipo extensos) que se encuentran en los contenedores de primera clase. Estos elementos `typedef` se utilizan en declaraciones genéricas de variables, parámetros a funciones y valores de retorno de las funciones. Por ejemplo, `value_type` en cada contenedor es siempre un `typedef` que representa el tipo de valor almacenado en el contenedor.

Archivos de encabezado de los contenedores de la Biblioteca estándar	
<code><vector></code>	
<code><list></code>	
<code><deque></code>	
<code><queue></code>	Contiene tanto a <code>queue</code> como a <code>priority_queue</code> .
<code><stack></code>	
<code><map></code>	Contiene tanto a <code>map</code> como a <code>multimap</code> .
<code><set></code>	Contiene tanto a <code>set</code> como a <code>multiset</code> .
<code><valarray></code>	
<code><bitset></code>	

Figura 22.3 | Archivos de encabezado de los contenedores de la Biblioteca estándar.

typedef	Descripción
<code>allocator_type</code>	El tipo del objeto utilizado para asignar la memoria del contenedor.
<code>value_type</code>	El tipo de elemento almacenado en el contenedor.
<code>reference</code>	Una referencia al tipo de elemento almacenado en el contenedor.
<code>const_reference</code>	Una referencia constante al tipo de elemento almacenado en el contenedor. Dicha referencia sólo puede ser utilizada para <i>leer</i> elementos y colocarlos en el contenedor, y para realizar operaciones <code>const</code> .
<code>pointer</code>	Un apuntador al tipo de elemento almacenado en el contenedor.
<code>const_pointer</code>	Un apuntador al tipo de elemento constante almacenado en el contenedor
<code>iterator</code>	Un iterador que apunta al tipo de elemento almacenado en el contenedor.

Figura 22.4 | Elementos `typedef` encontrados en los contenedores de primera clase. (Parte 1 de 2).

typedef	Descripción
<code>const_iterator</code>	Un iterador constante que apunta al tipo de elemento almacenado en el contenedor y que puede utilizarse sólo para <i>leer</i> elementos.
<code>reverse_iterator</code>	Un iterador inverso que apunta al tipo de elemento almacenado en el contenedor. Este tipo de iterador es para iterar a través de un contenedor en sentido inverso.
<code>const_reverse_iterator</code>	Un iterador inverso constante que apunta al tipo de elemento almacenado en el contenedor y que puede utilizarse sólo para <i>leer</i> elementos. Este tipo de iterador es para iterar a través de un contenedor en sentido inverso.
<code>difference_type</code>	El tipo del resultado obtenido al restar dos iteradores que hacen referencia al mismo contenedor (<code>operator -</code> no está definido para iteradores de contenedores <code>list</code> y contenedores asociativos).
<code>size_type</code>	El tipo utilizado para contar elementos en un contenedor e indizar a través de un contenedor de secuencia (no se puede indizar a través de un contenedor <code>list</code>).

Figura 22.4 | Elementos `typedef` encontrados en los contenedores de primera clase. (Parte 2 de 2).



Tip de rendimiento 22.3

La STL generalmente evita utilizar funciones *virtual* y de herencia a favor de usar programación genérica con plantillas para alcanzar un mejor rendimiento en tiempo de ejecución.



Tip de portabilidad 22.1

Programar con la STL mejora la portabilidad de su código.

Al prepararse para utilizar un contenedor de la STL, es importante asegurar que el tipo de elemento que vaya a almacenarse en el contenedor soporte un conjunto mínimo de funcionalidad. Al insertar un elemento en un contenedor, se crea una copia de ese elemento. Por esta razón, el tipo de elemento debe proporcionar su propio constructor de copia y operador de asignación. [Nota: esto se requiere solamente si la copia y la asignación predeterminadas a nivel de miembro no realizan operaciones apropiadas de copia y asignación para el tipo del elemento]. Además, los contenedores asociativos y muchos algoritmos requieren de la comparación de elementos. Por esta razón, el tipo del elemento debe proporcionar un operador de igualdad (`==`) y un operador menor que (`<`).



Observación de Ingeniería de Software 22.3

Técnicamente, los contenedores de la STL no requieren comparar sus elementos con los operadores de igualdad y menor que, a menos que un programa utilice una función miembro de contenedor que deba comparar sus elementos (por ejemplo, la función `sort` en la clase `list`). Por desgracia, algunos compiladores de C++ previos al estándar no son capaces de ignorar las partes de una plantilla que no se utilizan en un programa específico. En los compiladores con este problema, tal vez no se puedan utilizar los contenedores de la STL con objetos de clases que no definen operadores de igualdad y menor que sobrecargados.

22.1.2 Introducción a los iteradores

Los iteradores tienen muchas características en común con los apuntadores y se utilizan para apuntar a los elementos de los contenedores de primera clase (y para unos cuantos propósitos más, como veremos más adelante). Los iteradores almacenan información de estado susceptible para los contenedores específicos en los que operan; por lo tanto, los iteradores se implementan apropiadamente para cada tipo de contenedor. Ciertas operaciones de los iteradores son uniformes para todos los contenedores. Por ejemplo, el operador de desreferencia (*) actúa sobre un iterador para que el programador pueda utilizar el elemento al que apunta. La operación `++` en un iterador lo desplaza al siguiente elemento del contenedor (algo muy parecido a incrementar un apuntador en un arreglo para que apunte a su siguiente elemento).

Los contenedores de primera clase de la STL cuentan con las funciones miembro `begin` y `end`. La función `begin` devuelve un iterador que apunta al primer elemento del contenedor. La función `end` devuelve un iterador que apunta al primer elemento más allá del final del contenedor (un elemento que no existe). Si el iterador `i` apunta a un elemento

específico, entonces la operación `++i` apunta al “siguiente” elemento y `*i` se refiere al elemento al que apunta `i`. El iterador que resulta de `end` se utiliza comúnmente en una comparación de igualdad o desigualdad, para determinar si el “iterador móvil” (`i` en este caso) ha llegado al final del contenedor.

Utilizamos un objeto de tipo `iterator` para hacer referencia al elemento de un contenedor que puede modificarse. Utilizamos un objeto de tipo `const_iterator` para hacer referencia al elemento de un contenedor que no puede modificarse.

Uso de `istream_iterator` para entrada y de `ostream_iterator` para salida

Utilizamos los iteradores con **flujos** (también conocidos como **rangos**). Estos flujos pueden estar en contenedores, o pueden ser **flujos de entrada** o de **salida**. El programa de la figura 22.5 demuestra cómo se introducen datos desde la entrada estándar (un flujo de datos para introducir en un programa) utilizando un `istream_iterator`, y cómo se envían datos a la salida estándar (un flujo de datos como salida de un programa) utilizando un `ostream_iterator`. El programa recibe como entrada dos enteros por parte del usuario desde el teclado y muestra la suma de los mismos.¹

En la línea 15 se crea un `istream_iterator` capaz de extraer (introducir) valores `int` del objeto de entrada estándar `cin`, en una manera que ofrece seguridad de tipos. En la línea 17 se desreferencia el iterador `entradaInt` para leer el primer entero de `cin` y asigna ese entero a `numero1`. Observe que el operador de desreferencia `*` que se aplica a `entradaInt` obtiene el valor del flujo asociado con `entradaInt`; esto es similar al desreferenciamiento de un apuntador. En la línea 18 se coloca el iterador `entradaInt` en el siguiente valor del flujo de entrada. En la línea 19 se introduce el siguiente entero desde `entradaInt` y lo asigna a `numero2`.

En la línea 22 se crea un `ostream_iterator` que es capaz de insertar (enviar) valores `int` en el objeto de salida estándar `cout`. En la línea 25 se envía un entero a `cout` asignando a `*salidaInt` la suma de `numero1` y `numero2`. Observe el uso del operador de desreferencia `*` para utilizar a `*salidaInt` como *lvalue* en la instrucción de asignación. Si desea enviar otro valor utilizando `salidaInt`, el iterador debe incrementarse con `++` (puede utilizarse el preincremento y el postincremento, pero la forma prefijo debe tener prioridad por cuestiones de rendimiento).



Tip para prevenir errores 22.2

El operador `` (desreferencia) de cualquier iterador `const` devuelve una referencia `const` al elemento del contenedor, deshabilitando así el uso de las funciones miembro que no son `const`.*

```

1 // Fig. 22.5: Fig22_05.cpp
2 // Demostración de las operaciones de entrada y salida con iteradores.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include <iterator> // ostream_iterator e istream_iterator
9
10 int main()
11 {
12     cout << "Escriba dos enteros: ";
13
14     // crea istream_iterator para leer valores int de cin
15     std::istream_iterator< int > entradaInt( cin );
16
17     int numero1 = *entradaInt; // lee int de la entrada estándar
18     ++entradaInt; // desplaza el iterador al siguiente valor de entrada
19     int numero2 = *entradaInt; // lee int de la entrada estándar
20

```

Figura 22.5 | Iteradores de flujos de entrada y salida. (Parte 1 de 2).

1. En los ejemplos en este capítulo se coloca el prefijo “`std::`” antes de una función de la STL y de cada definición de un objeto contenedor de la STL en vez de colocar las declaraciones o directivas `using` al principio del programa, como se muestra en ejemplos anteriores. Las diferencias en los compiladores y en el complejo código que se genera al utilizar la STL hacen que sea difícil construir un conjunto apropiado de declaraciones o directivas `using` que permitan que el programa se compile sin errores. Para que estos programas puedan compilarse en la más amplia variedad de plataformas, optamos por el método del prefijo “`std::`”.

```

21 // crea ostream_iterator para escribir valores int a cout
22 std::ostream_iterator< int > salidaInt( cout );
23
24 cout << "La suma es: ";
25 *salidaInt = numero1 + numero2; // envía el resultado a cout
26 cout << endl;
27 return 0;
28 } // fin de main

```

Entran dos enteros: 12 25
La suma es: 37

Figura 22.5 | Iteradores de flujos de entrada y salida. (Parte 2 de 2).



Error común de programación 22.1

Intentar desreferenciar un iterador colocado fuera de su contenedor es un error lógico en tiempo de ejecución. En especial, el iterador devuelto por end no puede ser desreferenciado ni incrementado.



Error común de programación 22.2

Al intentar crear un iterador que no es const para un contenedor const produce un error de compilación.

Categorías de iteradores y la jerarquía de categorías de iteradores

La figura 22.6 muestra las categorías de iteradores de la STL. Cada categoría ofrece un conjunto específico de funcionalidad. La figura 22.7 muestra la jerarquía de categorías de iteradores. A medida que vaya siguiendo la jerarquía desde la parte superior hasta la inferior, notará que cada categoría de iteradores soporta toda la funcionalidad de las categorías

Categoría	Descripción
<i>entrada</i>	Utilizada para leer un elemento de un contenedor. Un iterador de entrada puede desplazarse sólo hacia adelante (es decir, desde el inicio del contenedor hasta su final), un elemento a la vez. Los iteradores de entrada soportan sólo algoritmos de una pasada: no puede utilizarse el mismo iterador de entrada para pasar a través de una secuencia dos veces.
<i>salida</i>	Se utiliza para escribir un elemento en un contenedor. Un iterador de salida puede desplazarse sólo hacia adelante, un elemento a la vez. Los iteradores de salida soportan sólo algoritmos de una pasada: el mismo iterador de salida no puede utilizarse para pasar a través de una secuencia dos veces.
<i>de avance</i>	Combina las capacidades de los iteradores de entrada y de salida, y retiene su posición en el contenedor (como la información sobre el estado).
<i>bidireccional</i>	Combina las capacidades de un iterador de avance con la habilidad para desplazarse en dirección hacia atrás (es decir, desde el final del contenedor hasta su inicio). Los iteradores bidireccionales soportan algoritmos de varias pasadas.
<i>de acceso aleatorio</i>	Combina las capacidades de un iterador bidireccional con la habilidad para tener acceso directo a cualquier elemento del contenedor (es decir, para saltar hacia adelante o hacia atrás un número arbitrario de elementos).

Figura 22.6 | Categorías de iteradores.



Figura 22.7 | Jerarquía de categorías de iteradores.

que están por encima de ella en la figura. Por lo tanto, los tipos de iteradores “más débiles” se encuentran en la parte superior y el tipo de iterador más poderoso se encuentra en la parte inferior. Observe que ésta no es una jerarquía de herencia.

La categoría de iteradores que soporta cada contenedor determina si éste puede utilizarse con ciertos algoritmos en la STL. Los contenedores que soportan a los iteradores de acceso aleatorio pueden utilizarse con todos los algoritmos de la STL. Como veremos, los apuntadores a los arreglos pueden utilizarse en vez de los iteradores en la mayoría de los algoritmos de la STL, incluyendo aquellos que requieren de los iteradores de acceso aleatorio. La figura 22.8 muestra la categoría de iteradores de cada uno de los contenedores de la STL. Hay que tener en cuenta que sólo los contenedores `vector`, `deque`, `list`, `set`, `multiset`, `map`, `multimap` (es decir, los contenedores de primera clase), `string` y `array` pueden recorrerse con iteradores.



Observación de Ingeniería de Software 22.4

El uso del “iterador más débil” que genera un rendimiento aceptable ayuda a producir componentes con una máxima capacidad de reutilización. Por ejemplo, si un algoritmo sólo requiere iteradores de avance, se puede utilizar con cualquier contenedor que soporte iteradores de avance, bidireccionales o de acceso aleatorio. Sin embargo, un algoritmo que requiere iteradores de acceso aleatorio sólo se puede utilizar con contenedores que tengan iteradores de acceso aleatorio.

Elementos `typedef` de iteradores predefinidos

La figura 22.9 muestra los elementos `typedef` de los iteradores predefinidos que se encuentran en las definiciones de clase de los contenedores de la STL. No todos los `typedef` están definidos para cada contenedor. Nosotros utilizamos versiones `const` de los iteradores para recorrer contenedores de sólo lectura. Utilizamos iteradores inversos para recorrer los contenedores en dirección inversa.

Contenedor	Tipo de iterador soportado
<i>Contenedores de secuencia (primera clase)</i>	
<code>vector</code>	acceso aleatorio
<code>deque</code>	acceso aleatorio
<code>list</code>	bidireccional
<i>Contenedores asociativos (primera clase)</i>	
<code>set</code>	bidireccional
<code>multiset</code>	bidireccional
<code>map</code>	bidireccional
<code>multimap</code>	bidireccional
<i>Adaptadores de contenedores</i>	
<code>stack</code>	no se soportan iteradores
<code>queue</code>	no se soportan iteradores
<code>priority_queue</code>	no se soportan iteradores

Figura 22.8 | Tipos de iteradores soportados por cada contenedor de la Biblioteca estándar.

Elementos <code>typedef</code> predefinidos para los tipos de iteradores	Dirección de ++	Capacidad
<code>iterator</code>	avance	leer/escribir
<code>const_iterator</code>	avance	leer
<code>reverse_iterator</code>	retroceso	leer/escribir
<code>const_reverse_iterator</code>	retroceso	leer

Figura 22.9 | Elementos `typedef` de los iteradores.



Tip para prevenir errores 22.3

Las operaciones realizadas sobre un `const_iterator` devuelven referencias `const` para evitar que se modifiquen los elementos del contenedor que se está manipulando. Utilizar de preferencia iteradores del tipo `const_iterator` en vez de `iterator` siempre que sea apropiado es otro ejemplo del principio del menor privilegio.

Operaciones con iteradores

La figura 22.10 muestra ciertas operaciones que pueden realizarse con cada tipo de iterador. Observe que las operaciones para cada tipo de operador incluyen a todas las operaciones anteriores a ese tipo en la figura. Observe también que, para los iteradores de entrada y de salida, no es posible guardar el iterador y utilizar después el valor guardado.

Operación de iterador	Descripción
<i>Todos los iteradores</i>	
<code>++p</code>	Preincrementar un iterador.
<code>p++</code>	Postincrementar un iterador.
<i>Iteradores de entrada</i>	
<code>*p</code>	Desreferenciar un iterador.
<code>p = p1</code>	Asignar un iterador a otro
<code>p == p1</code>	Comparar igualdad de iteradores
<code>p != p1</code>	Comparar desigualdad de iteradores
<i>Iteradores de salida</i>	
<code>*p</code>	Desreferenciar un iterador.
<code>p = p1</code>	Asignar un iterador a otro.
<i>Iteradores de avance</i>	Los iteradores de avance proporcionan toda la funcionalidad de los iteradores de entrada y los de salida.
<i>Iteradores bidireccionales</i>	
<code>--p</code>	Predecrementar un iterador.
<code>p--</code>	Postdecrementar un iterador.
<i>Iteradores de acceso aleatorio</i>	
<code>p += i</code>	Incrementar el iterador <code>p</code> en <code>i</code> posiciones.
<code>p -= i</code>	Decrementar el iterador <code>p</code> en <code>i</code> posiciones.
<code>p + i</code> o <code>i + p</code>	El valor de la expresión es un iterador posicionado en <code>p</code> , incrementado en <code>i</code> posiciones.
<code>p - i</code>	El valor de la expresión es un iterador posicionado en <code>p</code> , decrementado en <code>i</code> posiciones.
<code>p - p1</code>	El valor de la expresión es un entero que representa la distancia entre dos elementos en el mismo contenedor.
<code>p[i]</code>	Devuelve una referencia al desplazamiento del elemento de <code>p</code> , por <code>i</code> posiciones.
<code>p < p1</code>	Devuelve <code>true</code> si el iterador <code>p</code> es menor que el iterador <code>p1</code> (es decir, si el iterador <code>p</code> se encuentra antes que el iterador <code>p1</code> en el contenedor); en cualquier otro caso devuelve <code>false</code> .
<code>p <= p1</code>	Devuelve <code>true</code> si el iterador <code>p</code> es menor o igual que el iterador <code>p1</code> (es decir, si el iterador <code>p</code> se encuentra antes o en la misma ubicación que el iterador <code>p1</code> en el contenedor); en cualquier otro caso devuelve <code>false</code> .
<code>p > p1</code>	Devuelve <code>true</code> si el iterador <code>p</code> es mayor que el iterador <code>p1</code> (es decir, si el iterador <code>p</code> se encuentra después que el iterador <code>p1</code> en el contenedor); en cualquier otro caso devuelve <code>false</code> .
<code>p >= p1</code>	Devuelve <code>true</code> si el iterador <code>p</code> es mayor o igual que el iterador <code>p1</code> (es decir, si el iterador <code>p</code> se encuentra después o en la misma ubicación que el iterador <code>p1</code> en el contenedor); en cualquier otro caso devuelve <code>false</code> .

Figura 22.10 | Operaciones con iteradores para cada tipo de iterador.

22.1.3 Introducción a los algoritmos

Los algoritmos de la STL se pueden utilizar genéricamente a través de una variedad de contenedores. La STL provee muchos de los algoritmos que usted utilizará frecuentemente para manipular contenedores. Los algoritmos para insertar, eliminar, buscar, ordenar y demás son apropiados para algunos o todos los contenedores de la STL.

Esta biblioteca incluye aproximadamente 70 algoritmos estándar. Nosotros le ofrecemos ejemplos de código activo de la mayoría de estos algoritmos y sintetizamos a los demás en tablas. Los algoritmos operan sobre los elementos de los contenedores solamente en forma indirecta, a través de los iteradores. Muchos algoritmos operan en secuencias de elementos definidas por pares de iteradores: un primer iterador que apunta al primer elemento de la secuencia y un segundo iterador que apunta a un elemento más allá del último elemento de la secuencia. Además, es posible crear sus propios nuevos algoritmos que operen en una forma similar, de manera que puedan utilizarse con los contenedores e iteradores de la STL.

Los algoritmos comúnmente devuelven iteradores que indican sus resultados. Por ejemplo, el algoritmo `find` localiza un elemento y devuelve un iterador para ese elemento. Si el elemento no se encuentra, `find` devuelve el iterador que está “una posición más allá de `end`”, que se pasó para definir el final del rango en el que se va a buscar, el cual se puede evaluar para determinar si se encontró o no un elemento. El algoritmo `find` puede utilizarse con cualquier contenedor de primera clase de la STL. Los algoritmos de la STL crean otra oportunidad más para la reutilización: el uso de la vasta colección de algoritmos populares puede ahorrar a los programadores bastante tiempo y esfuerzo.

Si un algoritmo utiliza iteradores menos poderosos, éste puede utilizarse también con contenedores que soporten iteradores más poderosos. Algunos algoritmos demandan el uso de iteradores poderosos; por ejemplo, `sort` demanda el uso de iteradores de acceso aleatorio.



Observación de Ingeniería de Software 22.5

La STL está implementada en forma concisa. Hasta ahora, los diseñadores de clases hubieran asociado a los algoritmos con los contenedores, convirtiendo a los algoritmos en funciones miembro de los contenedores. La STL toma un enfoque distinto. Los algoritmos se separan de los contenedores y operan sobre los elementos de éstos sólo indirectamente, a través de los iteradores. Esta separación facilita la escritura de algoritmos genéricos que se apliquen a muchas clases de contenedores.



Observación de Ingeniería de Software 22.6

La STL es extensible. Es bastante simple agregar nuevos algoritmos sin necesidad de modificar los contenedores de la STL.



Observación de Ingeniería de Software 22.7

Los algoritmos de la STL pueden operar sobre los contenedores de la STL y sobre arreglos basados en apuntadores estilo C.



Tip de portabilidad 22.2

Como los algoritmos de la STL procesan contenedores sólo en forma indirecta a través de los iteradores, un algoritmo puede a menudo utilizarse con muchos contenedores distintos.

La figura 22.11 muestra muchos de los **algoritmos de secuencia cambiantes**; es decir, los algoritmos que producen modificaciones en los contenedores a los que se aplican.

Algoritmos de secuencia cambiantes

<code>copy</code>	<code>remove</code>	<code>reverse_copy</code>
<code>copy_backward</code>	<code>remove_copy</code>	<code>rotate</code>
<code>fill</code>	<code>remove_copy_if</code>	<code>rotate_copy</code>
<code>fill_n</code>	<code>remove_if</code>	<code>stable_partition</code>
<code>generate</code>	<code>replace</code>	<code>swap</code>
<code>generate_n</code>	<code>replace_copy</code>	<code>swap_ranges</code>
<code>iter_swap</code>	<code>replace_copy_if</code>	<code>transform</code>
<code>partition</code>	<code>replace_if</code>	<code>unique</code>
<code>random_shuffle</code>	<code>reverse</code>	<code>unique_copy</code>

Figura 22.11 | Algoritmos de secuencia cambiantes.

La figura 22.12 muestra muchos de los algoritmos de secuencia no cambiantes; es decir, los algoritmos que no producen modificaciones en los contenedores a los que se aplican. La figura 22.13 muestra los algoritmos numéricos del archivo de encabezado `<numeric>`.

Algoritmos de secuencia no cambiantes		
adjacent_find	find	find_if
count	find_each	mismatch
count_if	find_end	search
equal	find_first_of	search_n

Figura 22.12 | Algoritmos de secuencia no cambiantes.

Algoritmos numéricos del archivo de encabezado <code><numeric></code>		
accumulate		partial_sum
innerproduct		adjacent_difference

Figura 22.13 | Algoritmos numéricos del archivo de encabezado `<numeric>`.

22.2 Contenedores de secuencia

La Biblioteca de plantillas estándar de C++ cuenta con tres contenedores de secuencia: `vector`, `list` y `deque`. Las plantillas de clases `vector` y `deque` se basan en arreglos. La plantilla de clase `list` implementa una estructura de datos de lista enlazada, similar a nuestra clase `Lista` presentada en el capítulo 20, pero más robusta.

Uno de los contenedores más populares en la STL es `vector`. En el capítulo 7 presentamos la plantilla de clase `vector` como un tipo más robusto de arreglo. Un `vector` puede cambiar su tamaño en forma dinámica. A diferencia de los arreglos “puros” de C y C++ (vea el capítulo 7), los contenedores `vector` pueden asignarse uno a otro. Esto no es posible con los arreglos estilo C basados en apuntadores, ya que esos nombres de arreglos son apuntadores constantes y no pueden ser destinos de asignaciones. Al igual que con los arreglos de C, el uso de subíndices en contenedores `vector` no realiza una comprobación de rango automática, pero la clase `vector` sí ofrece esta capacidad mediante la función miembro `at` (que también vimos en el capítulo 7).



Tip de rendimiento 22.4

La inserción en la parte final de un `vector` es un proceso eficiente. El `vector` simplemente crece, si es necesario, para dar cabida al nuevo elemento. Es costoso insertar (o eliminar) un elemento a la mitad de un `vector`; toda la porción completa del `vector` después del punto de inserción (o eliminación) debe desplazarse, ya que los elementos de un `vector` ocupan celdas contiguas en memoria, al igual que los arreglos “puros” de C o C++.

La figura 22.2 presenta las operaciones comunes para todos los contenedores de la STL. Aparte de esas operaciones, cada contenedor generalmente cuenta con una variedad de capacidades adicionales. Muchas de estas capacidades son comunes para varios contenedores. Sin embargo, estas operaciones no siempre son igual de eficientes para cada contenedor. El programador debe elegir el contenedor más apropiado para la aplicación.



Tip de rendimiento 22.5

Las aplicaciones que requieren de inserciones y eliminaciones frecuentes en ambos extremos de un contenedor por lo general utilizan un contenedor `deque`, en vez de un `vector`. Aunque podemos insertar y eliminar elementos en la parte inicial y final tanto de un `vector` como de un `deque`, la clase `deque` es más eficiente que `vector` para llevar a cabo inserciones y eliminaciones en la parte inicial.



Tip de rendimiento 22.6

Las aplicaciones con inserciones y eliminaciones frecuentes a la mitad y/o a los extremos de un contenedor por lo general utilizan un contenedor `list`, debido a su eficiente implementación de los procesos de inserción y eliminación en cualquier parte de la estructura de datos.

Además de las operaciones comunes descritas en la figura 22.2, los contenedores de secuencia tienen otras operaciones comunes más: `front` para devolver una referencia al primer elemento en un contenedor que no esté vacío, `back` para devolver una referencia al último elemento en un contenedor que no esté vacío, `push_back` para insertar un nuevo elemento al final del contenedor y `pop_back` para quitar el último elemento del contenedor.

22.2.1 Contenedor de secuencia vector

La plantilla de clase `vector` ofrece una estructura de datos con ubicaciones contiguas en memoria. Esto permite un acceso eficiente y directo a cualquier elemento de un vector mediante el operador de subíndice `[]`, en forma idéntica a un arreglo “puro” de C o C++. La plantilla de clase `vector` se utiliza más comúnmente cuando los datos en el contenedor deben ser fácilmente accesibles mediante un subíndice, o deben ordenarse. Cuando la memoria de un `vector` se agota, éste asigna un área contigua de memoria más extensa, copia los elementos originales en la nueva área de memoria y desasigna la memoria anterior.



Tip de rendimiento 22.7

Seleccione el contenedor `vector` con el mejor rendimiento de acceso aleatorio.



Tip de rendimiento 22.8

Los objetos de la plantilla de clase `vector` ofrecen un acceso indizado rápido mediante el operador de subíndice `[]` sobre cargado, ya que se almacenan en espacios contiguos al igual que un arreglo puro de C o C++.



Tip de rendimiento 22.9

Es más rápido insertar muchos elementos al mismo tiempo que uno a la vez.

Una parte importante de todo contenedor es el tipo de iterador que soporta. Esto determina qué algoritmos pueden aplicarse a ese contenedor. Un `vector` soporta los iteradores de acceso aleatorio; es decir, todas las operaciones que se muestran en la figura 22.10 pueden aplicarse al iterador de un `vector`. Todos los algoritmos de la STL pueden operar sobre un `vector`. Los iteradores para un `vector` se implementan normalmente como apuntadores a los elementos del `vector`. Cada uno de los algoritmos de la STL que toman iteradores como argumentos requieren que éstos cuenten con un mínimo nivel de funcionalidad. Por ejemplo, si un algoritmo requiere de un iterador de avance, ese algoritmo puede operar en cualquier contenedor que cuente con iteradores de avance, bidireccionales o de acceso aleatorio. Mientras que el contenedor soporte la funcionalidad mínima del iterador de ese algoritmo, éste puede operar sobre el contenedor.

Uso de `vector` y de los iteradores

La figura 22.14 muestra varias funciones de la plantilla de clase `vector`. Muchas de estas funciones están disponibles en todos los contenedores de primera clase. Debe incluir el archivo de encabezado `<vector>` para poder utilizar la plantilla de clase `vector`.

```

1 // Fig. 22.14: Fig22_14.cpp
2 // Demostración de la plantilla de clase vector de la Biblioteca estándar.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <vector> // definición de la plantilla de clase vector
8 using std::vector;
9
10 // prototipo para la plantilla de función imprimirVector
11 template < typename T > void imprimirVector( const vector< T > &enteros2 );
12
13 int main()
14 {
15     const int TAMANIO = 6; // define el tamaño del arreglo
16     int arreglo[ TAMANIO ] = { 1, 2, 3, 4, 5, 6 }; // inicializa el arreglo

```

Figura 22.14 | Plantilla de clase `vector` de la Biblioteca estándar. (Parte I de 2).

```

17     vector< int > enteros; // crea un vector de valores int
18
19     cout << "El tamaño inicial de enteros es: " << enteros.size()
20     << "\nLa capacidad inicial de enteros es: " << enteros.capacity();
21
22     // la función push_back se encuentra en toda colección de secuencia
23     enteros.push_back( 2 );
24     enteros.push_back( 3 );
25     enteros.push_back( 4 );
26
27     cout << "\nEl tamaño de enteros es: " << enteros.size()
28     << "\nLa capacidad de enteros es: " << enteros.capacity();
29     cout << "\n\nImpresion de un arreglo usando notacion de apuntador: ";
30
31     // muestra el arreglo usando notación de apuntador
32     for ( int *ptr = arreglo; ptr != arreglo + TAMANIO; ptr++ )
33         cout << *ptr << ' ';
34
35     cout << "\nImpresion de vector usando notacion de iterador: "
36     imprimirVector( enteros );
37     cout << "\nContenido invertido del vector enteros: ";
38
39     // dos iteradores const_reverse
40     vector< int >::const_reverse_iterator iteradorInverso;
41     vector< int >::const_reverse_iterator iteradorTemp = enteros.rend();
42
43     // muestra el vector en orden inverso usando reverse_iterator
44     for ( iteradorInverso = enteros.rbegin();
45         iteradorInverso!= iteradorTemp; ++iteradorInverso )
46         cout << *iteradorInverso << ' ';
47
48     cout << endl;
49     return 0;
50 } // fin de main
51
52 // plantilla de función para mostar los elementos de un vector
53 template < typename T > void imprimirVector( const vector< T > &enteros2 )
54 {
55     typename vector< T >::const_iterator iteradorConst; // const_iterator
56
57     // muestra los elementos del vector usando const_iterator
58     for ( iteradorConst = enteros2.begin();
59         iteradorConst != enteros2.end(); ++iteradorConst )
60         cout << *iteradorConst << ' ';
61 } // fin de la función imprimirVector

```

```

El tamaniode enteros es: 0
La capacidad de enteros es: 0
El tamaniode enteros es: 3
La capacidad de enteros es: 4

Impresion de un arreglo usando notacion de apuntador: 1 2 3 4 5 6
Impresion de vector usando notacion de iterador: 2 3 4
Contenido invertido del vector enteros: 4 3 2

```

Figura 22.14 | Plantilla de clase `vector` de la Biblioteca estándar. (Parte 2 de 2).

La línea 17 define una instancia llamada `enteros` de la plantilla de clase `vector` que almacena valores `int`. Al crear la instancia de este objeto, se crea un `vector` vacío con tamaño 0 (es decir, el número de elementos almacenados en el `vector`) y capacidad 0 (es decir, el número de elementos que pueden almacenarse sin asignar más memoria al `vector`).

Las líneas 19 y 20 demuestran el uso de las funciones `size` y `capacity`; cada una devuelve inicialmente 0 para el `vector` `enteros` en este ejemplo. La función `size` (disponible en todos los contenedores) devuelve el número de elementos

actualmente almacenados en el contenedor. La función `capacity` devuelve el número de elementos que pueden almacenarse en el `vector` antes de que éste cambie de tamaño dinámicamente para dar cabida a más elementos.

Las líneas de la 23 a la 25 utilizan la función `push_back` (disponible en todos los contenedores de secuencia) para agregar un elemento al final del `vector`. Si se agrega un elemento a un `vector` lleno, éste aumenta su tamaño; algunas implementaciones de la STL hacen que el `vector` aumente su tamaño al doble.



Tip de rendimiento 22.10

El proceso de duplicar el valor de un vector cuando se necesita más espacio puede provocar un desperdicio considerable del mismo. Por ejemplo, un vector lleno con 1,000,000 elementos cambia su tamaño para dar cabida a 2,000,000 de elementos cuando se agregue uno nuevo. Esto deja a 999,999 elementos sin usar. Los programadores pueden utilizar a `resize` y `reserve` para controlar mejor el uso del espacio.

Las líneas 27 y 28 utilizan `size` y `capacity` para ilustrar el nuevo tamaño y capacidad del `vector` después de las tres operaciones con `push_back`. La función `size` devuelve 3: el número de elementos agregados al vector. La función `capacity` devuelve 4, indicando que podemos agregar un elemento adicional sin necesidad de asignar más memoria para el `vector`. Cuando agregamos el primer elemento, el tamaño de `enteros` se hizo 1 y la capacidad también se hizo 1. Cuando agregamos el segundo elemento, el tamaño de `enteros` se hizo 2 y la capacidad también se hizo 2. Cuando agregamos el tercer elemento, la capacidad se volvió a duplicar para quedar en 4. Por lo tanto, podemos agregar otro elemento antes de que el `vector` necesite asignar más espacio. Cuando el `vector` se llene en un momento dado y el programa intente agregar un elemento más, duplicara su capacidad a 8 elementos.

La forma en que crece un `vector` para dar cabida a más elementos (una operación que consume mucho tiempo) no está especificada por el Documento del estándar de C++. Los implementadores de la biblioteca de C++ utilizan varios esquemas astutos para minimizar la sobrecarga de cambiar el tamaño de un `vector`. Por ende, la salida de este programa puede variar, dependiendo de la versión de `vector` que incluya su compilador. Algunos implementadores de bibliotecas asignan una capacidad inicial extensa. Si un `vector` almacena un pequeño número de elementos, dicha capacidad puede ser un desperdicio de espacio. Sin embargo, puede mejorar considerablemente el rendimiento si un programa agrega muchos elementos a un `vector` y no tiene que reasignar memoria para dar cabida a esos elementos. Ésta es una clásica concesión entre espacio y tiempo. Los implementadores de bibliotecas deben balancear la cantidad de memoria utilizada y la cantidad de tiempo requerido para realizar varias operaciones con objetos `vector`.

Las líneas 32 y 33 demostraron cómo mostrar el contenido de un arreglo mediante el uso de apuntadores y aritmética de apuntadores. La línea 36 llama a la función `imprimirVector` (definida en las líneas 53 a 61) para mostrar el contenido de un `vector`, utilizando iteradores. La plantilla de función `imprimirVector` recibe una referencia `const` a un `vector` (`enteros2`) como su argumento. La línea 55 define a un `const_iterator` llamado `iteradorConst` que itera a través del `vector` y muestra su contenido. Observe que a la declaración de la línea 55 le antecede la palabra clave `typename`. Debido a que `imprimirVector` es una plantilla de función, y a que `vector<T>` se especializará de manera distinta para cada especialización de la plantilla de función, el compilador no puede saber en tiempo de compilación si `vector<T>::const_iterator` es un tipo. En una especialización específica, `const_iterator` podría ser una variable `static`. El compilador necesita esta información para compilar el programa correctamente. Por lo tanto, el programador debe indicar al compilador que debe esperar un nombre calificado (cuando el calificador es un tipo dependiente) en cada especialización.

Un `const_iterator` permite al programa leer los elementos del `vector`, pero no permite que el programa modifique a estos elementos. La estructura `for` que aparece en las líneas de la 58 a la 60 inicializa a `iteradorConst` mediante el uso de la función miembro `begin` de `vector`, que devuelve un `const_iterator` al primer elemento en el `vector`; hay otra versión de `begin` que devuelve un `iterator` que puede utilizarse para los contenedores que no son `const`. Observe que se devuelve un `const_iterator` debido a que el identificador `enteros2` se declaró como `const` en la lista de parámetros de la función `imprimirVector`. El ciclo continúa mientras que `iteradorConst` no llegue al final del `vector`. Esto se determina mediante la comparación de `iteradorConst` con el resultado de `enteros2.end()`, que devuelve un iterador que indica la posición que está después del último elemento del `vector`. Si `iteradorConst` es igual a este valor, se ha llegado al final del `vector`. Las funciones `begin` y `end` están disponibles para todos los contenedores de primera clase. El cuerpo del ciclo desreferencia al iterador `iteradorConst` para obtener el valor en el elemento actual del `vector`. Recuerde que el iterador actúa como un apuntador al elemento y que el operador `*` está sobrecargado para devolver una referencia al elemento. La expresión `iteradorConst++` (línea 59) coloca al iterador en el siguiente elemento del `vector`.



Tip de rendimiento 22.11

Use el preincremento cuando se aplique a los iteradores de la STL, ya que el operador de preincremento no devuelve un valor que se deba almacenar en un objeto temporal.



Tip para prevenir errores 22.4

Sólo los iteradores de acceso aleatorio soportan al operador <. Es mejor utilizar != y end para evaluar el final del contenedor.

En la línea 40 se declara un `const_reverse_iterator` que puede utilizarse para iterar a través de un `vector` en forma inversa. En la línea 41 se declara la variable `const_reverse_iterator` llamada `iteradorTemp` y se inicializa con el iterador devuelto por la función `rend` (es decir, el iterador para el punto final cuando se itera a través del contenedor en forma inversa). Todos los contenedores de primera clase soportan este tipo de iterador. En las líneas de la 44 a la 46 se utiliza una estructura `for` similar a la de la función `imprimirVector` para iterar a través del `vector`. En este ciclo, las funciones `rbegin` (es decir, el iterador del punto inicial para iterar en sentido inverso a través del contenedor) e `iteradorTemp` delinean el rango de elementos a mostrar. Al igual que con las funciones `begin` y `end`, `rbegin` y `rend` pueden devolver un `const_reverse_iterator` o un `reverse_iterator`, dependiendo de si el contenedor es o no constante.



Tip de rendimiento 22.12

Por cuestiones de rendimiento, debe capturar el valor de finalización del ciclo antes del ciclo y compararlo con ese valor, en vez de hacer una llamada a una función (que posiblemente consuma muchos recursos) por cada iteración.

Funciones de manipulación de elementos de vectores

La figura 22.15 muestra las funciones que permiten la recuperación y manipulación de los elementos de un `vector`. La línea 17 utiliza un constructor sobrecargado de `vector` que toma dos iteradores como argumentos para inicializar a `enteros`. Recuerde que los apuntadores a un arreglo pueden utilizarse como iteradores. En la línea 17 se inicializa `enteros` con el contenido de `arreglo`, desde la posición `arreglo` hasta (pero sin incluir) la posición `arreglo + TAMANIO`.

```

1 // Fig. 22.15: Fig22_15.cpp
2 // Prueba de las funciones de manipulación de elementos de la
3 // plantilla de clase vector de la Biblioteca estándar.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <vector> // definición de la plantilla de clase vector
9 #include <algorithm> // algoritmo de copia
10 #include <iterator> // iterador ostream_iterator
11 #include <stdexcept> // excepción out_of_range
12
13 int main()
14 {
15     const int TAMANIO = 6;
16     int arreglo[ TAMANIO ] = { 1, 2, 3, 4, 5, 6 };
17     std::vector< int > enteros( arreglo, arreglo + TAMANIO );
18     std::ostream_iterator< int > salida( cout, " " );
19
20     cout << "El vector enteros contiene: ";
21     std::copy( enteros.begin(), enteros.end(), salida );
22
23     cout << "\nPrimer elemento de enteros: " << enteros.front()
24         << "\nÚltimo elemento de enteros: " << enteros.back();
25
26     enteros[ 0 ] = 7; // establece el primer elemento a 7
27     enteros.at( 2 ) = 10; // establece el elemento en la posición 2 a 10
28
29     // inserta 22 como 2do elemento
30     enteros.insert( enteros.begin() + 1, 22 );
31
32     cout << "\n\nContenido del vector enteros después de los cambios: ";
33     std::copy( enteros.begin(), enteros.end(), salida );
34

```

Figura 22.15 | Funciones de manipulación de elementos de la plantilla de clase `vector`. (Parte I de 2).

```

35 // acceso a un elemento fuera de rango
36 try
37 {
38     enteros.at( 100 ) = 777;
39 } // fin de try
40 catch ( std::out_of_range &fueraDeRango ) // excepción out_of_range
41 {
42     cout << "\n\nExcepcion: " << fueraDeRango.what();
43 } // fin de catch
44
45 // borra el primer elemento
46 enteros.erase( enteros.begin() );
47 cout << "\n\nVector enteros despues de borrar el primer elemento: ";
48 std::copy( enteros.begin(), enteros.end(), salida );
49
50 // borra los elementos restantes
51 enteros.erase( enteros.begin(), enteros.end() );
52 cout << "\nDespues de borrar todos los elementos, el vector enteros "
53     << ( enteros.empty() ? "esta" : "no esta" ) << " vacio";
54
55 // inserta los elementos del arreglo
56 enteros.insert( enteros.begin(), arreglo, arreglo + TAMANIO );
57 cout << "\n\nContenido del vector enteros antes de clear: ";
58 std::copy( enteros.begin(), enteros.end(), salida );
59
60 // vacia enteros; clear llama a erase para vaciar una colección
61 enteros.clear();
62 cout << "\nDespues de clear, el vector enteros "
63     << ( enteros.empty() ? "esta" : "no esta" ) << " vacio" << endl;
64 return 0;
65 } // fin de main

```

El vector enteros contiene: 1 2 3 4 5 6
Primer elemento de enteros: 1
Último elemento de enteros: 6

Contenido del vector enteros despues de los cambios: 7 22 2 10 4 5 6

Excepcion: invalid vector<T> subscript

Vector enteros despues de borrar el primer elemento: 22 2 10 4 5 6
Despues de borrar todos los elementos, el vector enteros esta vacio

Contenido del vector enteros antes de clear: 1 2 3 4 5 6
Despues de clear, el vector enteros esta vacio

Figura 22.15 | Funciones de manipulación de elementos de la plantilla de clase `vector`. (Parte 2 de 2).

La línea 18 define un `ostream_iterator` llamado `salida` que puede utilizarse para mostrar los enteros separados por espacios sencillos mediante `cout`. Un `ostream_iterator< int >` es un mecanismo de salida que ofrece seguridad de tipos y que muestra sólo valores de tipo `int` o de un tipo compatible. El primer argumento para el constructor especifica el flujo de salida y el segundo argumento es una cadena que especifica los caracteres de separación para mostrar los valores; en este caso, la cadena contiene un carácter de espacio. Utilizamos el `ostream_iterator` (definido en el encabezado `<iostream>`) para mostrar el contenido del `vector` en este ejemplo.

La línea 21 utiliza el algoritmo `copy` de la Biblioteca estándar para mostrar todo el contenido completo del `vector` `enteros` en la salida estándar. El algoritmo `copy` copia cada elemento en el contenedor, empezando con la posición especificada por el iterador en su primer argumento y hasta (pero sin incluir) la posición especificada por el iterador en su segundo argumento. El primer y segundo argumentos deben satisfacer los requerimientos del iterador de entrada: deben ser iteradores a través de los cuales puedan leerse valores de un contenedor. Además, al aplicar el operador `++` al primer iterador se debe occasionar eventualmente que éste llegue al segundo argumento iterador en el contenedor. Los elementos se copian en la ubicación especificada por el iterador de salida (es decir, un iterador a través del cual se puede almacenar o mostrar un valor) que se especifica como el último argumento. En este caso, el iterador de salida es un `ostream_iterator`

(salida) que se anexa a `cout`, de manera que los elementos se copian a la salida estándar. Para utilizar los algoritmos de la Biblioteca estándar, debe incluir el archivo de encabezado `<algorithm>`.

Las líneas 23 y 24 utilizan las funciones `front` y `back` (disponibles para todos los contenedores de secuencia) para determinar el primer y último elementos del `vector`, respectivamente. Observe la diferencia entre las funciones `front` y `begin`. La función `front` devuelve una referencia al primer elemento en el vector, mientras que la función `begin` devuelve un iterador de acceso aleatorio que apunta al primer elemento en el vector. Observe además la diferencia entre las funciones `back` y `end`. La función `back` devuelve una referencia al último elemento en el vector, mientras que la función `end` devuelve un iterador de acceso aleatorio que apunta al final del vector (la ubicación después del último elemento).



Error común de programación 22.3

El vector no debe estar vacío; de no ser así, los resultados de las funciones front y back están indefinidos.

Las líneas 26 y 27 muestran dos maneras de utilizar subíndices a través de un `vector` (esto también puede utilizarse con los contenedores `deque`). La línea 26 utiliza el operador de subíndice que se sobrecarga para devolver ya sea una referencia al valor en la ubicación especificada, o una referencia constante a ese valor, dependiendo de si el contenedor es constante o no. La función `at` (línea 27) realiza la misma operación, pero con comprobación de límites. La función `at` primero comprueba el valor suministrado como argumento y determina si se encuentra dentro de los límites del `vector`. De no ser así, la función `at` lanza una excepción `out_of_bounds` definida en el encabezado `<stdexcept>` (como se demuestra en las líneas de la 36 a la 43). La figura 22.16 muestra algunos de los tipos de excepciones de la STL. (Los tipos de excepciones de la Biblioteca estándar se describen en el capítulo 16, Manejo de excepciones).

La línea 30 utiliza una de las tres funciones `insert` sobrecargadas que proporciona cada contenedor de secuencia. En la línea 30 se inserta el valor 22 antes del elemento que se encuentra en la posición especificada por el iterador en el primer argumento. En este ejemplo, el iterador apunta al segundo elemento del vector, por lo que se inserta 22 en el segundo elemento y el segundo elemento original se convierte en el tercer elemento del vector. Otras versiones de `insert` permiten insertar varias copias del mismo valor, empezando desde una posición específica en el contenedor, o insertar un rango de valores desde otro contenedor (o arreglo), empezando desde una posición específica en el contenedor original.

Las líneas 46 y 51 utilizan las dos funciones `erase` que están disponibles en todos los contenedores de primera clase. La línea 46 indica que el elemento en la ubicación especificada por el argumento iterador debe eliminarse del contenedor (en este ejemplo, el elemento que está al inicio del `vector`). La línea 51 especifica que todos los elementos en el rango que empieza con la posición del primer argumento hasta (pero sin incluir) la posición del segundo argumento deben eliminarse del contenedor. En este ejemplo, todos los elementos se eliminan del `vector`. En la línea 53 se utiliza la función `empty` (disponible para todos los contenedores y adaptadores) para confirmar que el `vector` está vacío.



Error común de programación 22.4

Eliminar un elemento que contiene un apuntador a un objeto asignado en forma dinámica no elimina al objeto; esto puede provocar una fuga de memoria.

En la línea 56 se demuestra la versión de la función `insert` que utiliza al segundo y tercer argumentos para especificar las posiciones inicial y final en una secuencia de valores (posiblemente de otro contenedor; en este caso, del arreglo de enteros `arreglo`) que deben insertarse en el `vector`. Recuerde que la posición final especifica la posición en la secuencia que se encuentra después del último elemento a insertar; la copia se lleva a cabo hasta (pero sin incluir) esta posición.

Tipos de excepciones de la STL	Descripción
<code>out_of_range</code>	Indica cuando el subíndice está fuera de rango; por ejemplo, cuando se especifica un subíndice inválido en la función miembro <code>at</code> de vector.
<code>invalid_argument</code>	Indica que se pasó un argumento inválido a una función.
<code>length_error</code>	Indica un intento de crear un contenedor demasiado largo, un objeto <code>string</code> , etc.
<code>bad_alloc</code>	Indica que un intento de asignar memoria con <code>new</code> (o con un asignador) falló debido a que no había suficiente memoria disponible.

Figura 22.16 | Algunos tipos de excepciones de la STL.

Por último, en la línea 61 se utiliza la función `clear` (disponible en todos los contenedores de primera clase) para vaciar el `vector`. Esta función llama a la versión de `erase` utilizada en la línea 51 para vaciar el `vector`.

[Nota: hay otras funciones que son comunes para todos los contenedores y para todos los contenedores de secuencia que aún no hemos cubierto. Cubriremos la mayoría de estas funciones en las siguientes secciones. También cubriremos muchas funciones específicas para cada contenedor].

22.2.2 Contenedor de secuencia `list`

El contenedor de secuencia `list` cuenta con una eficiente implementación para las operaciones de inserción y eliminación en cualquier posición del contenedor. Si la mayoría de las inserciones y eliminaciones ocurren en los extremos del contenedor, la estructura de datos `deque` (sección 22.2.3) ofrece una implementación más eficiente. La plantilla de clase `list` se implementa como una lista con enlace doble: cada nodo en la lista contiene un apuntador al nodo anterior y al siguiente nodo de esta lista. Esto permite a la plantilla de clase `list` soportar iteradores bidireccionales que permitan que el contenedor se recorra tanto hacia adelante como hacia atrás. Cualquier algoritmo que requiera iteradores de entrada, salida, de avance o bidireccionales puede operar sobre un contenedor `list`. Muchas de las funciones miembro de este contenedor manipulan a sus elementos como un conjunto ordenado.

Además de las funciones miembro de todos los contenedores de la STL en la figura 22.2 y de las funciones miembro comunes de todos los contenedores de secuencia descritas en la sección 22.2, la plantilla de clase `list` cuenta con otras nueve funciones miembro: `splice`, `push_front`, `pop_front`, `remove`, `remove_if`, `unique`, `merge`, `reverse` y `sort`. Varias de estas funciones miembro son implementaciones optimizadas en el contenedor `list` de los algoritmos de la STL que se presentan en la sección 22.5. La figura 22.17 demuestra varias características de la clase `list`. Recuerde que muchas de las funciones presentadas en las figuras de la 22.14 a la 22.15 pueden utilizarse con la clase `list`. Debe incluirse el archivo de encabezado `<list>` para poder utilizar la clase `list`.

```

1 // Fig. 22.17: Fig22_17.cpp
2 // Programa de prueba de la plantilla de clase list de la Biblioteca estándar.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <list> // definición de la plantilla de clase list
8 #include <algorithm> // algoritmo de copia
9 #include <iterator> // ostream_iterator
10
11 // prototipo para la plantilla de función imprimirLista
12 template < typename T > void imprimirLista( const std::list< T > &refLista );
13
14 int main()
15 {
16     const int TAMANIO = 4;
17     int arreglo[ TAMANIO ] = { 2, 6, 4, 8 };
18     std::list< int > valores; // crea una lista de valores int
19     std::list< int > otrosValores; // crea una lista de valores int
20
21     // inserta elementos en valores
22     valores.push_front( 1 );
23     valores.push_front( 2 );
24     valores.push_back( 4 );
25     valores.push_back( 3 );
26
27     cout << "valores contiene: ";
28     imprimirLista( valores );
29
30     valores.sort(); // ordena los valores
31     cout << "\nvalores despues de ordenar contiene: ";
32     imprimirLista( valores );
33
34     // inserta elementos de arreglo en otrosValores
35     otrosValores.insert( otrosValores.begin(), arreglo, arreglo + TAMANIO );

```

Figura 22.17 | Plantilla de clase `list` de la Biblioteca estándar. (Parte I de 3).

```
36     cout << "\nDespues de insert, otrosValores contiene: ";
37     imprimirLista( otrosValores );
38
39     // elimina los elementos de otrosValores e inserta al final de valores
40     valores.splice( valores.end(), otrosValores );
41     cout << "\nDespues de splice, valores contiene: ";
42     imprimirLista( valores );
43
44     valores.sort(); // ordena valores
45     cout << "\nDespues de sort, valores contiene: ";
46     imprimirLista( valores );
47
48     // inserta elementos de arreglo en otrosValores
49     otrosValores.insert( otrosValores.begin(), arreglo, arreglo + TAMANIO );
50     otrosValores.sort();
51     cout << "\nDespues de insert y sort, otrosValores contiene: ";
52     imprimirLista( otrosValores );
53
54     // elimina los elementos de otrosValores y los inserta en valores por orden
55     valores.merge( otrosValores );
56     cout << "\nDespues de merge:\n    valores contiene: ";
57     imprimirLista( valores );
58     cout << "\n    otrosValores contiene: ";
59     imprimirLista( otrosValores );
60
61     valores.pop_front(); // elimina elemento de la parte inicial
62     valores.pop_back(); // elimina elemento de la parte final
63     cout << "\nDespues de pop_front y pop_back:\n    valores contiene: ";
64     imprimirLista( valores );
65
66     valores.unique(); // elimina elementos duplicados
67     cout << "\nDespues de unique, valores contiene: ";
68     imprimirLista( valores );
69
70     // intercambia los elementos de valores y otrosValores
71     valores.swap( otrosValores );
72     cout << "\nDespues de swap:\n    valores contiene: ";
73     imprimirLista( valores );
74     cout << "\n    otrosValores contiene: ";
75     imprimirLista( otrosValores );
76
77     // reemplaza el contenido de valores con elementos de otrosValores
78     valores.assign( otrosValores.begin(), otrosValores.end() );
79     cout << "\nDespues de assign, valores contiene: ";
80     imprimirLista( valores );
81
82     // elimina los elementos de otrosValores y los inserta en valores por orden
83     valores.merge( otrosValores );
84     cout << "\nDespues de merge, valores contiene: ";
85     imprimirLista( valores );
86
87     valores.remove( 4 ); // elimina todos los 4s
88     cout << "\nDespues de remove( 4 ), valores contiene: ";
89     imprimirLista( valores );
90     cout << endl;
91     return 0;
92 } // fin de main
93
94 // definición de la plantilla de función imprimirLista; usa
95 // ostream_iterator y copy algorithm para mostrar los elementos de la lista
96 template < typename T > void imprimirLista( const std::list< T > &refLista )
97 {
98     if ( refLista.empty() ) // lista está vacía
```

Figura 22.17 | Plantilla de clase `list` de la Biblioteca estándar. (Parte 2 de 3).

```

99         cout << "Lista esta vacia";
100    else
101    {
102        std::ostream_iterator< T > output( cout, " " );
103        std::copy( reflista.begin(), reflista.end(), output );
104    } // fin de else
105 } // fin de la función imprimirLista

valores contiene: 2 1 4 3
valores despues de ordenar contiene: 1 2 3 4
Despues de insert, otrosValores contiene: 2 6 4 8
Despues de splice, valores contiene: 1 2 3 4 2 6 4 8
Despues de sort, valores contiene: 1 2 2 3 4 4 6 8
Despues de insert y sort, otrosValores contiene: 2 4 6 8
Despues de merge:
    valores contiene: 1 2 2 2 3 4 4 4 6 6 8 8
    otrosValores contiene: Lista esta vacia
Despues de pop_front y pop_back:
    valores contiene: 2 2 2 3 4 4 4 6 6 8
Despues de unique, valores contiene: 2 3 4 6 8
Despues de swap:
    valores contiene: Lista esta vacia
    otrosValores contiene: 2 3 4 6 8
Despues de assign, valores contiene: 2 3 4 6 8
Despues de merge, valores contiene: 2 2 3 3 4 4 6 6 8 8
Despues de remove( 4 ), valores contiene: 2 2 3 3 6 6 8 8

```

Figura 22.17 | Plantilla de clase `list` de la Biblioteca estándar. (Parte 3 de 3).

En las líneas 18 y 19 se crean instancias de dos objetos `list` capaces de almacenar enteros. Las líneas 22 y 23 usan la función `push_front` para insertar enteros al inicio de `valores`. La función `push_front` es específica para las clases `list` y `deque` (no para `vector`). Las líneas 24 y 25 utilizan la función `push_back` para insertar enteros al final de `valores`. Recuerde que la función `push_back` es común para todos los contenedores de secuencia.

La línea 30 utiliza la función miembro `sort` de `list` para ordenar los elementos en forma ascendente. [Nota: esta función es distinta a la función `sort` en los algoritmos de la STL]. Hay una segunda versión de `sort` que permite al programador suministrar una función predicado binaria que toma dos argumentos (valores en la lista), realiza una comparación y devuelve un valor `bool` indicando el resultado. Esta función determina el orden en el que se ordenan los elementos del contenedor `list`. Esta versión podría ser especialmente útil para un contenedor `list` que almacene apunadores en vez de valores. [Nota: en la figura 22.28 demostramos el uso de una función predicado unaria. Este tipo de función toma un solo argumento, realiza una comparación utilizando ese argumento y devuelve un valor `bool` indicando el resultado].

La línea 40 utiliza la función `splice` de `list` para eliminar los elementos en `otrosValores` e insertarlos en `valores` antes de la posición del iterador especificado como el primer argumento. Hay otras dos versiones de esta función. La función `splice` con tres argumentos permite que se elimine un elemento del contenedor especificado como el segundo argumento y desde la posición especificada por el iterador en el tercer argumento. La función `splice` con cuatro argumentos utiliza los últimos dos para especificar un rango de posiciones que deben eliminarse del contenedor en el segundo argumento, y colocarse en la ubicación especificada en el primer argumento.

Después de insertar más elementos en la lista `otrosValores` y de ordenar tanto a `valores` como a `otrosValores`, en la línea 55 se utiliza la función miembro `merge` de `list` para eliminar a todos los elementos de `otrosValores` e insertarlos en forma ordenada en `valores`. Ambas listas deben ordenarse en el mismo orden antes de realizar esta operación. Una segunda versión de `merge` permite al programador suministrar una función predicado que toma dos argumentos (valores en la lista) y devuelve un valor `bool`. La función predicado especifica el orden utilizado por `merge`.

En la línea 61 se utiliza la función `pop_front` de `list` para eliminar el primer elemento en la lista. En la línea 62 se utiliza la función `pop_back` (disponible para todos los contenedores de secuencia) para eliminar el último elemento en la lista.

En la línea 66 se utiliza la función `unique` de `list` para eliminar los elementos duplicados en la lista. Ésta debe estar ordenada (de manera que todos los duplicados estén uno al lado del otro) antes de realizar esta operación, para garantizar que se eliminan todos los duplicados. Una segunda versión de `unique` permite al programador suministrar una función predicado que toma dos argumentos (valores en la lista) y devuelve un valor `bool` especificando si dos elementos son iguales.

En la línea 71 se utiliza la función `swap` (disponible para todos los contenedores) para intercambiar el contenido de `valores` con el contenido de `otrosValores`.

En la línea 78 se utiliza la función `assign` de `list` para reemplazar el contenido de `valores` con el contenido de `otrosValores`, en el rango especificado por los dos argumentos iteradores. Una segunda versión de `assign` reemplaza el contenido original con copias del valor especificado en el segundo argumento. El primer argumento de la función especifica el número de copias. En la línea 87 se utiliza la función `remove` de `list` para eliminar todas las copias del valor 4 de la lista.

22.2.3 Contenedor de secuencia deque

La clase `deque` ofrece muchos de los beneficios de `vector` y `list` en un contenedor. El término en inglés `deque` es la abreviación de “cola con dos partes finales”. La clase `deque` está implementada para ofrecer un acceso indizado eficiente (mediante los subíndices) para leer y modificar sus elementos, en forma muy parecida a un `vector`. La clase `deque` también está implementada para producir operaciones eficientes de inserción y eliminación en su parte inicial y final, en forma muy parecida a un contenedor `list` (aunque `list` es también capaz de realizar inserciones y eliminaciones eficientemente a la mitad de un contenedor `list`). La clase `deque` ofrece soporte para los iteradores de acceso aleatorio, por lo que puede utilizarse con todos los algoritmos de la STL. Uno de los usos más comunes de un contenedor `deque` es el de mantener una cola de elementos del tipo “primero en entrar, primero en salir”. De hecho, `deque` es la implementación subyacente predeterminada para el adaptador `queue` (sección 22.4.2).

Puede asignarse más espacio de almacenamiento para un contenedor `deque` en cualquiera de sus extremos, en bloques de memoria que se mantienen generalmente como un arreglo de apuntadores a esos bloques.² Debido a la distribución de un contenedor `deque` en áreas de memoria no contiguas, un iterador para este tipo de contenedores debe ser más inteligente que los apuntadores utilizados para iterar a través de contenedores `vector` o arreglos basados en apuntadores.

Tip de rendimiento 22.13



En general, deque tiene mayor sobrecarga que vector.

Tip de rendimiento 22.14



Las inserciones y eliminaciones a la mitad de un contenedor deque se optimizan para minimizar el número de elementos copiados, por lo que es más eficiente que un contenedor vector pero menos eficiente que un contenedor list para este tipo de modificación.

La clase `deque` cuenta con las mismas operaciones básicas que la clase `vector`, pero agrega las funciones miembro `push_front` y `push_back` para permitir la inserción y eliminación al principio del contenedor `deque`, respectivamente.

La figura 22.18 demuestra las características de la clase `deque`. Recuerde que muchas de las funciones presentadas en las figuras 22.14, 22.15 y 22.17 también pueden utilizarse con la clase `deque`. Debe incluirse el archivo de encabezado `<deque>` para poder utilizar esta clase.

En la línea 13 se crea una instancia de un contenedor `deque` que puede almacenar valores `double`. En las líneas 17 a 19 se utilizan las funciones `push_front` y `push_back` para insertar elementos en la parte inicial y final del contenedor `deque`. Recuerde que `push_back` está disponible para todos los contenedores de secuencia, pero `push_front` está disponible sólo para las clases `list` y `deque`.

```

1 // Fig. 22.18: Fig22_18.cpp
2 // Programa de prueba de la clase deque de la Biblioteca estándar.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <deque> // definición de la plantilla de clase deque
8 #include <algorithm> // algoritmo de copia
9 #include <iterator> // ostream_iterator
10
11 int main()
12 {
13     std::deque< double > valores; // crea deque de valores double
14     std::ostream_iterator< double > output( cout, " " );

```

Figura 22.18 | Plantilla de clase `deque` de la Biblioteca estándar. (Parte I de 2).

2. Este detalle es específico de cada implementación, no un requerimiento del estándar de C++.

```

15 // inserta elementos en valores
16 valores.push_front( 2.2 );
17 valores.push_front( 3.5 );
18 valores.push_back( 1.1 );
19
20 cout << "valores contiene: ";
21
22 // usa el operador subíndice para obtener elementos de valores
23 for ( unsigned int i = 0; i < valores.size(); i++ )
24     cout << valores[ i ] << ' ';
25
26 valores.pop_front(); // elimina el primer elemento
27 cout << "\nDespues de pop_front, valores contiene: ";
28 std::copy( valores.begin(), valores.end(), output );
29
30 // usa el operador subíndice para modificar el elemento en la ubicación 1
31 valores[ 1 ] = 5.4;
32 cout << "\nDespues de valores[ 1 ] = 5.4, valores contiene: ";
33 std::copy( valores.begin(), valores.end(), output );
34 cout << endl;
35 return 0;
36
37 } // fin de main

```

valores contiene: 3.5 2.2 1.1
 Despues de pop_front, valores contiene: 2.2 1.1
 Despues de valores[1] = 5.4, valores contiene: 2.2 5.4

Figura 22.18 | Plantilla de clase `deque` de la Biblioteca estándar. (Parte 2 de 2).

La instrucción `for` en las líneas 24 y 25 utiliza el operador subíndice para recuperar el valor en cada elemento del contenedor `deque` y así poder mostrarlo. Observe que la condición utiliza la función `size` para asegurar que no tratemos de utilizar un elemento fuera de los límites del contenedor `deque`.

En la línea 27 se utiliza la función `pop_front` para demostrar cómo eliminar el primer elemento del contenedor `deque`. Recuerde que esta función está disponible sólo para las clases `list` y `deque` (no para la clase `vector`).

En la línea 32 se utiliza el operador de subíndice para crear un *lvalue*. Esto permite que los valores se asignen directamente a cualquier elemento del contenedor `deque`.

22.3 Contenedores asociativos

Los contenedores asociativos de la STL ofrecen un acceso directo para almacenar y recuperar elementos mediante **claves** (comúnmente conocidas como **claves de búsqueda**). Los cuatro contenedores asociativos son `multiset`, `set`, `multimap` y `map`. Cada contenedor asociativo mantiene sus claves en orden. Al iterar a través de un contenedor asociativo, éste se recorre en el orden que tenga asignado. Las clases `multiset` y `set` cuentan con operaciones para manipular conjuntos de valores en donde éstos son las claves; no hay un valor separado asociado con cada clave. La principal diferencia entre `multiset` y `set` es que `multiset` permite claves duplicadas y `set` no. Las clases `multimap` y `map` ofrecen operaciones para manipular valores asociados con claves (estos valores algunas veces se conocen como **valores asignados**). La principal diferencia entre `multimap` y `map` es que `multimap` permite claves duplicadas con los valores asociados que van a almacenarse y `map` solamente permite claves únicas con los valores asociados. Además de las funciones miembro comunes de todos los contenedores presentadas en la figura 22.2, todos los contenedores asociativos soportan otras funciones miembro, incluyendo `find`, `lower_bound`, `upper_bound` y `count`. En las siguientes subsecciones se presentan ejemplos de cada uno de los contenedores asociativos y las funciones miembro comunes para ellos.

22.3.1 Contenedor asociativo `multiset`

El contenedor asociativo `multiset` ofrece rapidez en el almacenamiento y la recuperación de las claves, además de que permite el uso de claves duplicadas. El ordenamiento de los elementos se determina mediante un **objeto función de comparación**. Por ejemplo, en un `multiset` entero, los elementos pueden ordenarse en forma ascendente si se ordenan las claves con el **objeto función de comparación** `less< int >`. En la sección 22.7 hablaremos con detalle sobre los objetos función.

El tipo de datos de las claves en todos los contenedores asociativos debe soportar la comparación, basándose apropiadamente en el objeto función de comparación especificado; las claves ordenadas con `less< T >` deben soportar la comparación con `operator<`. Si las claves utilizadas en los contenedores asociativos son de tipos de datos definidos por el usuario, esos tipos deben proporcionar los operadores de comparación apropiados. Un contenedor `multiset` soporta los iteradores bidireccionales (pero no los iteradores de acceso aleatorio).

La figura 22.19 demuestra el uso del contenedor asociativo `multiset` mediante un `multiset` de enteros ordenados en forma ascendente. Debe incluirse el archivo de encabezado `<set>` para poder utilizar la clase `multiset`. Los contenedores `multiset` y `set` cuentan con la misma funcionalidad básica.

La línea 10 utiliza un `typedef` para crear un nuevo nombre de tipo (alias) para un `multiset` de enteros ordenados en forma ascendente, utilizando el objeto función `less< int >`. El orden ascendente es el valor predeterminado para un conjunto `multiset`, por lo que se puede omitir `std::less< int >` en la línea 10. Este nuevo tipo (`Ims`) se utiliza entonces para crear una instancia de un objeto entero `multiset` de nombre `intMultiset` (línea 19).



Buena práctica de programación 22.1

Use instrucciones `typedef` para que el código con nombres de tipo largos (como `multiset`) sea más fácil de leer.

En la instrucción de salida de la línea 22 se utiliza la función `count` (disponible para todos los contenedores asociativos) para contar el número de ocurrencias del valor 15 que haya actualmente en el `multiset`.

```
1 // Fig. 22.19: Fig22_19.cpp
2 // Prueba de la clase multiset de la Biblioteca estándar
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <set> // definición de la plantilla de clase multiset
8
9 // define un nombre corto para el tipo multiset utilizado en este programa
10 typedef std::multiset< int, std::less< int > > Ims;
11
12 #include <algorithm> // algoritmo de copia
13 #include <iostream> // ostream_iterator
14
15 int main()
16 {
17     const int TAMANIO = 10;
18     int a[ TAMANIO ] = { 7, 22, 9, 1, 18, 30, 100, 22, 85, 13 };
19     Ims multisetInt; // Ims es el typedef para "multiset entero"
20     std::ostream_iterator< int > salida( cout, " " );
21
22     cout << "Hay actualmente " << multisetInt.count( 15 )
23         << " valores de 15 en el multiset\n";
24
25     multisetInt.insert( 15 ); // inserta 15 en multisetInt
26     multisetInt.insert( 15 ); // inserta 15 en multisetInt
27     cout << "Despues de insert, hay " << multisetInt.count( 15 )
28         << " valores de 15 en el multiset\n\n";
29
30     // iterador que no se puede utilizar para modificar los valores de los elementos
31     Ims::const_iterator resultado;
32
33     // busca 15 en multisetInt; find devuelve el iterador
34     resultado = multisetInt.find( 15 );
35
36     if ( resultado != multisetInt.end() ) // si el iterador no está al final
37         cout << "Se encontro el valor 15\n"; // encontró el valor de búsqueda 15
38 }
```

```

39 // busca 20 en multisetInt; find devuelve el iterador
40 resultado = multisetInt.find( 20 );
41
42 if ( resultado == multisetInt.end() ) // será verdadero, ya que
43     cout << "No se encontró el valor 20\n"; // no encontró el 20
44
45 // inserta los elementos del arreglo a en multisetInt
46 multisetInt.insert( a, a + TAMANIO );
47 cout << "\nDespués de insert, multisetInt contiene:\n";
48 std::copy( multisetInt.begin(), multisetInt.end(), salida );
49
50 // determina los límites inferior y superior de 22 en multisetInt
51 cout << "\n\nLímite inferior de 22: "
52     << *( multisetInt.lower_bound( 22 ) );
53 cout << "\nLímite superior de 22: " << *( multisetInt.upper_bound( 22 ) );
54
55 // p representa un par de iteradores const_iterator
56 std::pair< Ims::const_iterator, Ims::const_iterator > p;
57
58 // usa equal_range para determinar los límites inferior y superior
59 // de 22 en multisetInt
60 p = multisetInt.equal_range( 22 );
61
62 cout << "\n\nequal_range de 22:" << "\n Límite inferior: "
63     << *( p.first ) << "\n Límite superior: " << *( p.second );
64 cout << endl;
65 return 0;
66 } // fin de main

```

```

Hay actualmente 0 valores de 15 en el multiset
Después de insert, hay 2 valores de 15 en el multiset

Se encontró el valor 15
No se encontró el valor 20

Después de insert, multisetInt contiene:
1 7 9 13 15 15 18 22 22 30 85 100

Límite inferior de 22: 22
Límite superior de 22: 30

equal_range de 22:
 Límite inferior: 22
 Límite superior: 30

```

Figura 22.19 | Plantilla de clase `multiset` de la Biblioteca estándar. (Parte 2 de 2).

En las líneas 25 y 26 se utiliza una de las tres versiones de la función `insert` para agregar el valor 15 al `multiset` dos veces. Una segunda versión de `insert` toma un iterador y un valor como argumentos y empieza la búsqueda del punto de inserción para la posición especificada del iterador. Una tercera versión de `insert` toma dos iteradores como argumentos, que especifican un rango de valores a agregar al `multiset` provenientes de otro contenedor.

En la línea 34 se utiliza a la función `find` (disponible para todos los contenedores asociativos) para localizar el valor 15 en el `multiset`. La función `find` devuelve un `iterator` o un `const_iterator` que apunta a la primera ubicación en la que se encontró el valor. Si éste no se encuentra, `find` devuelve un `iterator` o un `const_iterator` igual al valor devuelto por una llamada a `end`. La línea 42 demuestra este caso.

En la línea 46 se utiliza la función `insert` para insertar los elementos del arreglo `a` en el `multiset`. En la línea 48, el algoritmo `copy` copia los elementos del `multiset` a la salida estándar. Observe que los elementos se muestran en orden ascendente.

En las líneas 52 y 53 se utilizan las funciones `lower_bound` y `upper_bound` (disponibles en todos los contenedores asociativos) para localizar la primera ocurrencia del valor 22 en el `multiset` y el elemento que está *después* de la primera ocurrencia del valor 22 en el `multiset`. Ambas funciones devuelven iteradores tipo `iterator` o `const_iterator` que apuntan a la posición apropiada del iterador devuelto por `end`, en caso de que el valor no se encuentre en el contenedor `multiset`.

En la línea 56 se crea una instancia de la clase `pair` de nombre `p`. Los objetos de la clase `pair` se utilizan para asociar pares de valores. En este ejemplo, el contenido de un objeto `pair` está compuesto de dos iteradores del tipo `const_iterator` para nuestro `multiset` basado en enteros. El propósito de `p` es almacenar el valor de retorno de la función `equal_range` de `multiset`, que devuelve un objeto `pair` que contiene los resultados de las operaciones con `lower_bound` y `upper_bound`. El tipo `pair` contiene dos miembros de datos `public` llamados `first` y `second`.

En la línea 60 se utiliza la función `equal_range` para determinar el límite inferior (`lower_bound`) y el límite superior (`upper_bound`) de 22 en el contenedor `multiset`. En la línea 63 se utiliza `p.first` y `p.second` respectivamente, para tener acceso a los límites menor (`lower_bound`) e inferior (`upper_bound`). Desreferenciamos a los iteradores para mostrar los valores en las posiciones devueltas por `equal_range`.

22.3.2 Contenedor asociativo set

El contenedor asociativo `set` se utiliza para almacenar y recuperar rápidamente claves únicas. La implementación de `set` es idéntica a la de `multiset`, sólo que un contenedor `set` debe tener claves únicas. Por lo tanto, si se trata de insertar una clave duplicada en un contenedor `set` se ignora ese duplicado; como éste es el comportamiento matemático deseado para un conjunto, no lo identificamos como un error de programación común. Un contenedor `set` soporta iteradores bidireccionales (pero no los iteradores de acceso aleatorio). La figura 22.20 demuestra el uso de un contenedor `set` con valores `double`. Debe incluirse el archivo de encabezado `<set>` para poder utilizar la clase `set`.

```

1 // Fig. 22.20: Fig22_20.cpp
2 // Programa de prueba de la clase set de la Biblioteca estándar.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <set>
8
9 // define un nombre corto para el tipo set utilizado en este programa
10 typedef std::set< double, std::less< double > > SetDouble;
11
12 #include <algorithm>
13 #include <iterator> // ostream_iterator
14
15 int main()
16 {
17     const int TAMANIO = 5;
18     double a[ TAMANIO ] = { 2.1, 4.2, 9.5, 2.1, 3.7 };
19     SetDouble setDouble( a, a + TAMANIO );
20     std::ostream_iterator< double > salida( cout, " " );
21
22     cout << "setDouble contiene: ";
23     std::copy( setDouble.begin(), setDouble.end(), salida );
24
25     // p representa un par que contiene elementos const_iterator y bool
26     std::pair< SetDouble::const_iterator, bool > p;
27
28     // insert< 13.8 en setDouble; insert devuelve un par en el cual
29     // p.first representa la ubicación de 13.8 en setDouble y
30     // p.second representa si se insertó o no el valor 13.8
31     p = setDouble.insert( 13.8 ); // el valor no está en el conjunto
32     cout << "\n\n" << *( p.first )
33     << ( p.second ? " se" : " no se" ) << " inserto";
34     cout << "\ndoubleSet contiene: ";
35     std::copy( setDouble.begin(), setDouble.end(), salida );
36
37     // inserta 9.5 en setDouble
38     p = setDouble.insert( 9.5 ); // el valor ya está en el conjunto
39     cout << "\n\n" << *( p.first )
40     << ( p.second ? " se" : " no se" ) << " inserto";

```

Figura 22.20 | Plantilla de clase `set` de la Biblioteca estándar. (Parte I de 2).

```

41     cout << "\ndoubleSet contiene: ";
42     std::copy( setDouble.begin(), setDouble.end(), salida );
43     cout << endl;
44     return 0;
45 } // fin de main

```

```

setDouble contiene: 2.1 3.7 4.2 9.5
13.8 se inserto
doubleSet contiene: 2.1 3.7 4.2 9.5 13.8
9.5 no se inserto
doubleSet contiene: 2.1 3.7 4.2 9.5 13.8

```

Figura 22.20 | Plantilla de clase `set` de la Biblioteca estándar. (Parte 2 de 2).

En la línea 10 se utiliza a `typedef` para crear un nuevo nombre de tipo (`setDouble`) para un conjunto de valores `double` ordenados en forma ascendente, utilizando el objeto función `less< double >`.

En la línea 19 se utiliza el nuevo tipo `SetDouble` para crear una instancia del objeto `setDouble`. La llamada al constructor toma los elementos en el arreglo `a` que se encuentran entre las posiciones `a` y `a + TAMANIO` (es decir, todo el arreglo) y los inserta en el contenedor `set`. En la línea 23 se utiliza el algoritmo `copy` para mostrar el contenido del conjunto. Observe que el valor `2.1`, que aparece dos veces en el arreglo `a`, aparece sólo una vez en `setDouble`. Esto se debe a que el contenedor `set` no permite duplicados.

En la línea 26 se define un objeto `pair` que consiste de un `const_iterator` para un `SetDouble` y de un valor `bool`. Este objeto almacena el resultado de una llamada a la función `insert` de `set`.

En la línea 31 se utiliza la función `insert` para colocar el valor `13.8` en el contenedor `set`. El objeto `pair` devuelto, llamado `p`, contiene un iterador `p.first` que apunta al valor `13.8` en el contenedor `set` y un valor `bool` que es `true` si el valor se insertó, y `false` si el valor no se insertó (porque ya se encontraba en el contenedor `set`). En este caso `13.8` no se encontraba en el conjunto, por lo que se insertó. En la línea 38 se hace un intento por insertar `9.5`, que ya se encuentra en el conjunto. La salida de las líneas 39 y 40 muestra que `9.5` no se insertó.

22.3.3 Contenedor asociativo `multimap`

El contenedor asociativo `multimap` se utiliza para almacenar y recuperar rápidamente las claves y los valores asociados (a menudo conocidos como pares clave/valor). Muchas de las funciones utilizadas con los contenedores `multiset` y `set` también se utilizan con `multimap` y `map`. Los elementos de los contenedores `multimap` y `map` son pares (objetos de la clase `pair`) de claves y valores, en vez de valores individuales. Al insertar datos en un `multimap` o `map` se utiliza un objeto `pair` que contiene la clave y el valor. El orden de las claves se determina mediante un objeto función de comparación. Por ejemplo, en un contenedor `multimap` que utilice enteros como el tipo para las claves, éstas pueden ordenarse en forma ascendente mediante el objeto función de comparación `less< int >`. En un contenedor `multimap` se permiten claves duplicadas, por lo que pueden asociarse varios valores con una sola clave. Esto a menudo se conoce como una relación de uno a varios. Por ejemplo, en un sistema para procesar transacciones de tarjetas de crédito, la cuenta de una tarjeta de crédito puede tener muchas transacciones asociadas; en una universidad, un estudiante puede tomar muchas clases y un profesor puede enseñar a muchos estudiantes; en la milicia, un rango (como "cabo") tiene muchas personas. Un contenedor `multimap` soporta el uso de iteradores bidireccionales, pero no los iteradores de acceso aleatorio. En la figura 22.21 se demuestra el uso del contenedor asociativo `multimap`. Debe incluirse el archivo de encabezado `<map>` para poder utilizar la clase `multimap`.



Tip de rendimiento 22.15

Un contenedor `multimap` se implementa para localizar eficientemente todos los valores que forman pares con una clave dada.

En la línea 10 se utiliza a `typedef` para definir el alias `Mmid` para un tipo `multimap` en el que el tipo de la clave es `int`, el tipo del valor asociado con una clave es `double` y los elementos se ordenan en forma ascendente. En la línea 14 se utiliza el nuevo tipo para crear una instancia de un contenedor `multimap` llamada `pares`. En la línea 16 se utiliza la función `count` para determinar el número de pares clave/valor con una clave de 15.

```

1 // Fig. 22.21: Fig22_21.cpp
2 // Programa de prueba de la clase multimap de la Biblioteca estándar.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <map> // definición de la plantilla de clase map
8
9 // define un nombre corto para el tipo multimap utilizado en este programa
10 typedef std::multimap< int, double, std::less< int > > Mmid;
11
12 int main()
13 {
14     Mmid pares; // declara los pares de multimap
15
16     cout << "Hay actualmente " << pares.count( 15 )
17         << " pares con la clave 15 en el multimap\n";
18
19     // inserta dos objetos value_type en pares
20     pares.insert( Mmid::value_type( 15, 2.7 ) );
21     pares.insert( Mmid::value_type( 15, 99.3 ) );
22
23     cout << "Despues de las inserciones, hay " << pares.count( 15 )
24         << " pares con la clave 15\n\n";
25
26     // inserta cinco objetos value_type en pares
27     pares.insert( Mmid::value_type( 30, 111.11 ) );
28     pares.insert( Mmid::value_type( 10, 22.22 ) );
29     pares.insert( Mmid::value_type( 25, 33.333 ) );
30     pares.insert( Mmid::value_type( 20, 9.345 ) );
31     pares.insert( Mmid::value_type( 5, 77.54 ) );
32
33     cout << "Los pares de multimap contienen:\nClave\tValor\n";
34
35     // usa const_iterator para recorrer los elementos de pares
36     for ( Mmid::const_iterator iter = pares.begin();
37           iter != pares.end(); ++iter )
38         cout << iter->first << '\t' << iter->second << '\n';
39
40     cout << endl;
41     return 0;
42 } // fin de main

```

Hay actualmente 0 pares con la clave 15 en el multimap
 Despues de las inserciones, hay 2 pares con la clave 15

Los pares de multimap contienen:

Clave	Valor
5	77.54
10	22.22
15	2.7
15	99.3
20	9.345
25	33.333
30	111.11

Figura 22.21 | Plantilla de clase `multimap` de la Biblioteca estándar.

En la línea 20 se utiliza la función `insert` para agregar un nuevo par clave/valor al contenedor `multimap`. La expresión `Mmid::value_type(15, 2.7)` crea un objeto `pair` en el que `first` es la clave (15) de tipo `int` y `second` es el valor (2.7) de tipo `double`. El tipo `Mmid::value_type` está definido como parte del `typedef` para el contenedor `multimap`. En la línea 21 se inserta otro objeto `pair` con la clave 15 y el valor 99.3. Después, en las líneas 23 y 24 se muestra el número de pares con la clave 15.

En las líneas 27 a 31 se insertan cinco pares adicionales en el contenedor `multimap`. La instrucción `for` en las líneas 36 a 38 muestra el contenido del `multimap`, incluyendo claves y valores. En la línea 38 se utiliza el `const_iterator` llamado `iter` para tener acceso a los miembros del par en cada elemento del contenedor `multimap`. Observe en la salida que las claves aparecen en orden ascendente.

22.3.4 Contenedor asociativo map

El contenedor asociativo `map` se utiliza para almacenar y recuperar rápidamente claves únicas y los valores asociados. No se permiten claves duplicadas en un `map`, por lo que sólo puede asociarse un valor con cada clave. Esto se conoce como **asignación de uno a uno**. Por ejemplo, una compañía que utiliza números únicos para los empleados tales como 100, 200 y 300 podría tener un contenedor `map` que asocie los números de los empleados con sus extensiones telefónicas: 4321, 4114 y 5217, respectivamente. Con un contenedor `map` el programador especifica la clave y recibe rápidamente de vuelta los datos asociados. Un contenedor `map` se conoce comúnmente como **arreglo asociativo**. Al proporcionar la clave en el operador subíndice `[]` de un contenedor `map` se localiza el valor asociado con esa clave en el contenedor `map`. Pueden realizarse inserciones y eliminaciones en cualquier parte de un `map`.

En la figura 22.22 se demuestra el uso del contenedor asociativo `map` y se utilizan las mismas características que la figura 22.21 para demostrar el uso del operador subíndice. Debe incluirse el archivo de encabezado `<map>` para poder utilizar la clase `map`. En las líneas 33 y 34 se utiliza el operador subíndice de la clase `map`. Cuando el subíndice es una clave que ya se encuentra en el contenedor `map` (línea 33), el operador devuelve una referencia a su valor asociado. Cuando el subíndice es una clave que no se encuentra en el contenedor `map` (línea 34), el operador inserta la clave en el contenedor y devuelve una referencia que puede utilizarse para asociar un valor con esa clave. En la línea 33 se sustituye el valor de la clave 25 (anteriormente 33.333, según lo especificado en la línea 21) con un nuevo valor: 9999.99. En la línea 34 se inserta un nuevo par clave/valor (lo que se conoce como **crear una asociación**) en el contenedor `map`.

```

1 // Fig. 22.22: Fig22_22.cpp
2 // Programa de prueba de la clase map de la Biblioteca estándar.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <map> // definición de la plantilla de clase map
8
9 // define un nombre corto para el tipo map utilizado en este programa
10 typedef std::map< int, double, std::less< int > > Mid;
11
12 int main()
13 {
14     Mid pares;
15
16     // inserta ocho objetos value_type en pares
17     pares.insert( Mid::value_type( 15, 2.7 ) );
18     pares.insert( Mid::value_type( 30, 111.11 ) );
19     pares.insert( Mid::value_type( 5, 1010.1 ) );
20     pares.insert( Mid::value_type( 10, 22.22 ) );
21     pares.insert( Mid::value_type( 25, 33.333 ) );
22     pares.insert( Mid::value_type( 5, 77.54 ) ); // se ignora el valor duplicado
23     pares.insert( Mid::value_type( 20, 9.345 ) );
24     pares.insert( Mid::value_type( 15, 99.3 ) ); // se ignora el valor duplicado
25
26     cout << "pares contiene:\nClave\tValor\n";
27
28     // usa const_iterator para recorrer los elementos de pares
29     for ( Mid::const_iterator iter = pares.begin();
30           iter != pares.end(); ++iter )
31         cout << iter->first << '\t' << iter->second << '\n';
32
33     pares[ 25 ] = 9999.99; // usa subíndices para modificar el valor de la clave 25
34     pares[ 40 ] = 8765.43; // usa subíndices para modificar el valor de la clave 40

```

Figura 22.22 | Plantilla de clase `map` de la Biblioteca estándar. (Parte I de 2).

```

35     cout << "\nDespues de las operaciones de subindices, pares contiene:\nClave\tValor\n";
36
37
38 // usa const_iterator para recorrer los elementos de pares
39 for ( Mid::const_iterator iter2 = pares.begin();
40       iter2 != pares.end(); ++iter2 )
41     cout << iter2->first << '\t' << iter2->second << '\n';
42
43     cout << endl;
44     return 0;
45 } // fin de main

```

pares contiene:

Clave	Valor
5	1010.1
10	22.22
15	2.7
20	9.345
25	33.333
30	111.11

Despues de las operaciones de subindices, pares contiene:

Clave	Valor
5	1010.1
10	22.22
15	2.7
20	9.345
25	9999.99
30	111.11
40	8765.43

Figura 22.22 | Plantilla de clase `map` de la Biblioteca estándar. (Parte 2 de 2).

22.4 Adaptadores de contenedores

La STL cuenta con tres adaptadores de contenedores: `stack`, `queue` y `priority_queue`. Los adaptadores no son contenedores de primera clase, ya que no cuentan con la implementación de la estructura de datos actual en la que los elementos pueden almacenarse, y debido a que los adaptadores no soportan el uso de iteradores. El beneficio de una clase de adaptador es que el programador puede elegir una estructura de datos subyacente apropiada. Las tres clases de adaptadores cuentan con las funciones miembro `push` y `pop` que insertan apropiadamente un elemento en cada estructura de datos de adaptador y eliminan apropiadamente un elemento de cada estructura de datos de adaptador. Las siguientes subsecciones ofrecen ejemplos de las clases de adaptadores.

22.4.1 Adaptador `stack`

La clase `stack` permite realizar inserciones y eliminaciones en un extremo de la estructura de datos subyacente (lo que comúnmente se conoce como una estructura de datos del tipo “último en entrar, primero en salir”). Una pila puede implementarse con cualquiera de los contenedores de secuencia: `vector`, `list` y `deque`. Este ejemplo crea tres pilas de enteros utilizando cada uno de los contenedores de secuencia de la Biblioteca estándar como estructura de datos subyacente para representar al adaptador `stack`. De manera predeterminada, una pila se implementa con un `deque`. Las operaciones de la pila son `push` para insertar un elemento en su parte superior (lo que se implementa mediante una llamada a la función `push_back` del contenedor subyacente), `pop` para eliminar el elemento superior de la pila (lo que se implementa mediante una llamada a la función `pop_back` del contenedor subyacente), `top` para obtener una referencia al elemento superior de la pila (lo que se implementa mediante una llamada a la función `back` del contenedor subyacente), `empty` para determinar si la pila está vacía o no (lo que se implementa mediante una llamada a la función `empty` del contenedor subyacente) y `size` para obtener el número de elementos en la pila (lo que se implementa mediante una llamada a la función `size` del contenedor subyacente).

Tip de rendimiento 22.16



Cada una de las operaciones comunes de un adaptador `stack` se implementa como una función `inline` que llama a la función apropiada del contenedor subyacente. Esto evita la sobrecarga de una segunda llamada a una función.



Tip de rendimiento 22.17

Para el mejor rendimiento, use la clase vector como contenedor subyacente para un adaptador stack.

En la figura 22.23 se demuestra el uso de la clase de adaptador **stack**. El archivo de encabezado **<stack>** debe incluirse para poder utilizar la clase **stack**.

En las líneas 20, 23 y 26 se crean instancias de tres pilas de enteros. En la línea 20 se especifica una pila de enteros que utiliza el contenedor **deque** predeterminado como su estructura de datos subyacente. En la línea 23 se especifica una pila de enteros que utiliza a un **vector** de enteros como su estructura de datos subyacente. En la línea 26 se especifica una pila de enteros que utiliza un contenedor **list** de enteros como su estructura de datos subyacente.

La función **meterElementos** (líneas 49 a 56) mete los elementos en cada pila. En la línea 53 se utiliza la función **push** (disponible en cada clase de adaptador) para colocar un entero en la parte superior de la pila. En la línea 54 se utiliza la función **top** de **pila** para obtener el elemento superior de la pila para mostrarlo en pantalla. La función **top** no elimina el elemento superior.

La función **sacarElementos** (líneas 59 a 66) saca los elementos de cada pila. En la línea 63 se utiliza la función **top** de **stack** para obtener el elemento superior de la pila y mostrarlo en pantalla. En la línea 64 se utiliza la función **pop** (disponible en cada clase de adaptador) para eliminar el elemento superior de la pila. La función **pop** no devuelve un valor.

```

1 // Fig. 22.23: Fig22_23.cpp
2 // Programa de prueba del adaptador stack de la Biblioteca estándar.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <stack> // definición del adaptador stack
8 #include <vector> // definición de la plantilla de clase vector
9 #include <list> // definición de la plantilla de clase list
10
11 // prototipo de la plantilla de función meterElementos
12 template< typename T > void meterElementos( T &refStack );
13
14 // prototipo de la plantilla de función sacarElementos
15 template< typename T > void sacarElementos( T &refStack );
16
17 int main()
18 {
19     // pila con el contenedor deque subyacente predeterminado
20     std::stack< int > pilaDequeInt;
21
22     // pila con el contenedor vector subyacente
23     std::stack< int, std::vector< int > > pilaVectorInt;
24
25     // pila con el contenedor list subyacente
26     std::stack< int, std::list< int > > pilaListInt;
27
28     // mete los valores 0 a 9 en cada pila
29     cout << "Metiendo datos en pilaDequeInt: ";
30     meterElementos( pilaDequeInt );
31     cout << "\nMetiendo datos en pilaVectorInt: ";
32     meterElementos( pilaVectorInt );
33     cout << "\nMetiendo datos en pilaListInt: ";
34     meterElementos( pilaListInt );
35     cout << endl << endl;
36
37     // muestra y elimina los elementos de cada pila
38     cout << "Sacando datos de pilaDequeInt: ";
39     sacarElementos( pilaDequeInt );
40     cout << "\nSacando datos de pilaVectorInt: ";
41     sacarElementos( pilaVectorInt );

```

Figura 22.23 | Clase de adaptador **stack** de la Biblioteca estándar. (Parte I de 2).

```

42     cout << "\nSacando datos de pilaListInt: ";
43     sacarElementos( pilaListInt );
44     cout << endl;
45     return 0;
46 } // fin de main
47
48 // mete elementos al objeto pila al que refStack hace referencia
49 template< typename T > void meterElementos( T &refStack )
50 {
51     for ( int i = 0; i < 10; i++ )
52     {
53         refStack.push( i ); // mete elemento en la pila
54         cout << refStack.top() << ' '; // ve (y muestra) el elemento superior
55     } // fin de for
56 } // fin de la función meterElementos
57
58 // saca elementos del objeto pila al que refStack hace referencia
59 template< typename T > void sacarElementos( T &refStack )
60 {
61     while ( !refStack.empty() )
62     {
63         cout << refStack.top() << ' '; // ve (y muestra) el elemento superior
64         refStack.pop(); // elimina el elemento superior
65     } // fin de while
66 } // fin de la función sacarElementos

```

```

Metiendo datos en pilaDequeInt: 0 1 2 3 4 5 6 7 8 9
Metiendo datos en pilaVectorInt: 0 1 2 3 4 5 6 7 8 9
Metiendo datos en pilaListInt: 0 1 2 3 4 5 6 7 8 9

Sacando datos de pilaDequeInt: 9 8 7 6 5 4 3 2 1 0
Sacando datos de pilaVectorInt: 9 8 7 6 5 4 3 2 1 0
Sacando datos de pilaListInt: 9 8 7 6 5 4 3 2 1 0

```

Figura 22.23 | Clase de adaptador `stack` de la Biblioteca estándar. (Parte 2 de 2).

22.4.2 Adaptador `queue`

La clase `queue` permite realizar inserciones en la parte final de la estructura de datos subyacente y eliminaciones en la parte inicial de la misma (lo que comúnmente se conoce como una estructura de datos del tipo “primero en entrar, primero en salir”). Una cola puede implementarse con la estructura de datos `list` o `deque` de la STL. De manera predefinida, una cola se implementa con `deque`. Las operaciones comunes de un adaptador `queue` son `push` para insertar un elemento en la parte final (lo que se implementa mediante una llamada a la función `push_back` del contenedor subyacente), `pop` para eliminar el elemento en la parte inicial de la cola (lo que se implementa mediante una llamada a la función `pop_front` del contenedor subyacente), `front` para obtener una referencia al primer elemento en la cola (lo que se implementa mediante una llamada a la función `front` del contenedor subyacente), `back` para obtener una referencia al último elemento en la cola (lo que se implementa mediante una llamada a la función `back` del contenedor subyacente), `empty` para determinar si la cola está vacía o no (lo que se implementa mediante una llamada a la función `empty` del contenedor subyacente) y `size` para obtener el número de elementos en la cola (lo que se implementa mediante una llamada a la función `size` del contenedor subyacente).



Tip de rendimiento 22.18

Cada una de las operaciones comunes de un adaptador `queue` se implementan como una función `inline` que llama a la función apropiada del contenedor subyacente. Esto evita la sobrecarga de una segunda llamada a una función.



Tip de rendimiento 22.19

Para el mejor rendimiento, use la clase `deque` como el contenedor subyacente para un adaptador `queue`.

La figura 22.24 demuestra el uso de la clase de adaptador `queue`. El archivo de encabezado `<queue>` debe incluirse para poder utilizar esta clase.

En la línea 11 se crea una instancia de `queue` para almacenar valores `double`. En las líneas 14 a 16 se utiliza la función `push` para agregar elementos a la cola. La instrucción `while` en las líneas 21 a 25 usa la función `empty` (disponible en todos los contenedores) para determinar si la cola está vacía o no (línea 21). Mientras haya mas elementos en `queue`, la línea 23 utiliza la función `front` de `queue` para leer (pero no eliminar) el primer elemento en `queue` para salir. La línea 24 elimina el primer elemento en `queue` mediante la función `pop` (disponible en todas las clases de adaptadores).

```

1 // Fig. 22.24: Fig22_24.cpp
2 // Programa de prueba del adaptador queue de la Biblioteca estándar.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <queue> // definición del adaptador queue
8
9 int main()
10 {
11     std::queue< double > valores; // cola con valores double
12
13     // mete los elementos en la cola valores
14     valores.push( 3.2 );
15     valores.push( 9.8 );
16     valores.push( 5.4 );
17
18     cout << "Sacando datos de valores: ";
19
20     // saca los elementos de la cola
21     while ( !valores.empty() )
22     {
23         cout << valores.front() << ' '; // ve el elemento que está al frente
24         valores.pop(); // elimina el elemento
25     } // fin de while
26
27     cout << endl;
28     return 0;
29 } // fin de main

```

```
Sacando datos de valores: 3.2 9.8 5.4
```

Figura 22.24 | Plantillas de clase del adaptador `queue` de la Biblioteca estándar.

22.4.3 Adaptador `priority_queue`

La clase `priority_queue` ofrece una funcionalidad que permite inserciones ordenadas en la estructura de datos subyacente, y eliminaciones de su parte inicial. Un adaptador `priority_queue` puede implementarse con los contenedores de secuencia `vector` o `deque` de la STL. De manera predeterminada, un adaptador `priority_queue` se implementa con un vector como estructura de datos subyacente. Al agregar elementos a un adaptador `priority_queue`, éstos se insertan en orden de prioridad de forma que el elemento con la prioridad más alta (es decir, el valor más grande) sea el primer elemento removido del adaptador. Esto se logra generalmente mediante el uso de una técnica de ordenamiento llamada `heapsort`, que siempre mantiene el valor más grande (es decir, el de mayor prioridad) en la parte inicial de la estructura de datos; dicha estructura de datos se conoce como **montón (heap)**. La comparación de elementos se lleva a cabo con el objeto función de comparación `less< T >` de manera predeterminada, pero el programador puede suministrar un comparador distinto.

Hay varias operaciones comunes de un adaptador `priority_queue`. `push` inserta un elemento en la posición apropiada, con base en el orden de prioridad del adaptador `priority_queue` (lo que se implementa mediante una llamada a la función `push_back` del contenedor subyacente y luego se vuelven a ordenar los elementos mediante la técnica `heapsort`). `pop` elimina el elemento de mayor prioridad en el adaptador `priority_queue` (lo que se implementa mediante una llamada a la función `pop_back` del contenedor subyacente, después de eliminar el elemento superior del montón). `top` obtiene una referencia al elemento superior del adaptador `priority_queue` (lo que se implementa mediante una llamada a la función `front` del contenedor subyacente). `empty` determina si el adaptador `priority_queue` está vacío o no (lo que se implementa mediante una llamada a la función `empty` del contenedor subyacente). `size` obtiene el número de elementos en el adaptador (lo que se implementa mediante una llamada a la función `size` del contenedor subyacente).



Tip de rendimiento 22.20

Cada una de las operaciones comunes de un adaptador `priority_queue` se implementa mediante una función `inline` que llama a la función apropiada del contenedor subyacente. Esto evita la sobrecarga de una segunda llamada a una función.



Tip de rendimiento 22.21

Para el mejor rendimiento, use la clase `vector` como contenedor subyacente para un adaptador `priority_queue`.

La figura 22.25 demuestra el uso de la clase de adaptador `priority_queue`. Debe incluirse el archivo de encabezado `<queue>` para poder utilizar esta clase.

```

1 // Fig. 22.25: Fig22_25.cpp
2 // Programa de prueba del adaptador priority_queue de la Biblioteca estándar.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <queue> // definición del adaptador priority_queue
8
9 int main()
10 {
11     std::priority_queue< double > prioridades; // crea un priority_queue
12
13     // mete elementos en prioridades
14     prioridades.push( 3.2 );
15     prioridades.push( 9.8 );
16     prioridades.push( 5.4 );
17
18     cout << "Sacando datos de prioridades: ";
19
20     // saca elemento de priority_queue
21     while ( !prioridades.empty() )
22     {
23         cout << prioridades.top() << ' '; // ve el elemento superior
24         prioridades.pop(); // elimina el elemento superior
25     } // fin de while
26
27     cout << endl;
28     return 0;
29 } // fin de main

```

Sacando datos de prioridades: 9.8 5.4 3.2

Figura 22.25 | Clase de adaptador `priority_queue` de la Biblioteca estándar.

En la línea 11 se crea una instancia de `priority_queue` que almacena valores `double` y utiliza un `vector` como la estructura de datos subyacente. En las líneas 14 a 16 se utiliza la función `push` para agregar elementos al adaptador `priority_queue`. La estructura `while` en las líneas 21 a 25 utiliza la función `empty` (disponible en todos los contenedores) para determinar si el adaptador está vacío o no (línea 21). Mientras haya más elementos, en la línea 23 se utiliza la función `top` de `priority_queue` para obtener el elemento de mayor prioridad en el adaptador y mostrarlo en pantalla. En la línea 24 se elimina el elemento de mayor prioridad del adaptador mediante la función `pop` (disponible en todas las clases de adaptadores).

22.5 Algoritmos

Hasta la llegada de la STL, las bibliotecas de clases de contenedores y algoritmos eran esencialmente incompatibles entre los distribuidores. Las primeras bibliotecas de contenedores generalmente utilizaban herencia y polimorfismo, con la sobrecarga asociada a las llamadas a funciones `virtual`. Además, estas bibliotecas creaban los algoritmos y los integraban a las clases contenedoras como comportamientos. La STL separa a los algoritmos de los contenedores. Esto facilita

considerablemente el agregar nuevos algoritmos. Con la STL, los elementos de los contenedores se utilizan a través de iteradores. Las siguientes subsecciones demuestran muchos de los algoritmos de la STL.



Tip de rendimiento 22.22

La STL está implementada tomando en cuenta la eficiencia. Evita la sobrecarga de las llamadas a funciones virtuales.



Observación de Ingeniería de Software 22.8

Los algoritmos de la STL no dependen de los detalles de implementación de los contenedores sobre los que operan. Mientras que los iteradores del contenedor (o arreglo) cumplan con los requerimientos del algoritmo, los algoritmos de la STL pueden trabajar con arreglos estilo C basados en apuntadores, contenedores de la STL y estructuras de datos definidas por el usuario.



Observación de Ingeniería de Software 22.9

Pueden agregarse algoritmos a la STL fácilmente, sin necesidad de modificar las clases contenedoras.

22.5.1 `fill`, `fill_n`, `generate` y `generate_n`

La figura 22.6 demuestra el uso de los algoritmos `fill`, `fill_n`, `generate` y `generate_n`. Las funciones `fill` y `fill_n` establecen todos los elementos en un rango de elementos del contenedor a un valor específico. Las funciones `generate` y `generate_n` usan una función generadora para crear valores para cada elemento en un rango de elementos del contenedor. La función generadora no toma argumentos y devuelve un valor que puede colocarse en un elemento del contenedor.

```

1 // Fig. 22.26: Fig22_26.cpp
2 // Los algoritmos fill, fill_n, generate y generate_n de la Biblioteca estándar.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <algorithm> // definiciones de los algoritmos
8 #include <vector> // definición de la plantilla de clase vector
9 #include <iterator> // ostream_iterator
10
11 char siguienteLetra(); // prototipo de la función generadora
12
13 int main()
14 {
15     std::vector< char > chars( 10 );
16     std::ostream_iterator< char > salida( cout, " " );
17     std::fill( chars.begin(), chars.end(), '5' ); // llena chars con 5s
18
19     cout << "Vector chars despues de llenarlo con 5s:\n";
20     std::copy( chars.begin(), chars.end(), salida );
21
22     // llena los primeros cinco elementos de chars con As
23     std::fill_n( chars.begin(), 5, 'A' );
24
25     cout << "\n\nVector chars despues de llenar cinco elementos con As:\n";
26     std::copy( chars.begin(), chars.end(), salida );
27
28     // genera los valores para todos los elementos de chars con siguienteLetra
29     std::generate( chars.begin(), chars.end(), siguienteLetra );
30
31     cout << "\n\nVector chars despues de generar las letras A-J:\n";
32     std::copy( chars.begin(), chars.end(), salida );
33
34     // genera valores para los primeros cinco elementos de chars con siguienteLetra
35     std::generate_n( chars.begin(), 5, siguienteLetra );
36

```

Figura 22.26 | Los algoritmos `fill`, `fill_n`, `generate` y `generate_n`. (Parte I de 2).

```

37     cout << "\n\nVector chars despues de generar K-O para los"
38     << " primeros cinco elementos:\n";
39     std::copy( chars.begin(), chars.end(), salida );
40     cout << endl;
41     return 0;
42 } // fin de main
43
44 // función generadora que devuelve la siguiente letra (empieza con A)
45 char siguienteLetra()
46 {
47     static char letra = 'A';
48     return letra++;
49 } // fin de la función siguienteLetra

```

Vector chars despues de llenarlo con 5s:

5 5 5 5 5 5 5 5 5 5

Vector chars despues de llenar cinco elementos con As:

A A A A A 5 5 5 5 5

Vector chars despues de generar las letras A-J:

A B C D E F G H I J

Vector chars despues de generar K-O para los primeros cinco elementos:

K L M N O F G H I J

Figura 22.26 | Los algoritmos `fill`, `fill_n`, `generate` y `generate_n`. (Parte 2 de 2).

En la línea 15 se define un vector de 10 elementos que almacena valores `char`. En la línea 17 se utiliza la función `fill` para colocar el carácter '5' en cada elemento del vector `chars`, desde `chars.begin()` hasta, pero sin incluir a, `chars.end()`. Observe que los iteradores suministrados como primer y segundo argumentos deben ser por lo menos iteradores de avance (es decir, que puedan utilizarse tanto como entrada desde un contenedor como de salida hacia un contenedor, en dirección hacia adelante).

En la línea 23 se utiliza a la función `fill_n` para colocar el carácter 'A' en los primeros cinco elementos del vector `chars`. El iterador suministrado como primer argumento debe ser por lo menos un iterador de salida (es decir, que pueda utilizarse para enviar datos a un contenedor en dirección hacia adelante). El segundo argumento especifica el número de elementos a llenar. El tercer argumento especifica el valor a colocar en cada elemento.

En la línea 29 se utiliza a la función `generate` para colocar el resultado de una llamada a la función generadora `siguienteLetra` en cada elemento del vector `chars`, desde `chars.begin()` hasta, pero sin incluir a, `chars.end()`. Los iteradores suministrados como primer y segundo argumentos deben ser por lo menos iteradores de avance. La función `siguienteLetra` (líneas 45 a 49) empieza con el carácter 'A' que se mantiene en una variable local `static`. La instrucción en la línea 48 post-incrementa el valor de `letra` y devuelve su valor anterior cada vez que se hace una llamada a `siguienteLetra`.

En la línea 35 se utiliza la función `generate_n` para colocar el resultado de una llamada a la función generadora `siguienteLetra` en cinco elementos del vector `chars`, empezando desde `chars.begin()`. El iterador que se suministra como el primer argumento debe ser por lo menos un iterador de salida.

22.5.2 `equal`, `mismatch` y `lexicographical_compare`

En la figura 22.27 se demuestra la comparación de secuencias de valores para ver si son iguales mediante el uso de los algoritmos `equal`, `mismatch` y `lexicographical_compare`.

```

1 // Fig. 22.27: Fig22_27.cpp
2 // Las funciones equal, mismatch y lexicographical_compare de la Biblioteca estándar.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6

```

Figura 22.27 | Los algoritmos `equal`, `mismatch` y `lexicographical_compare`. (Parte 1 de 2).

```

7 #include <algorithm> // definiciones de los algoritmos
8 #include <vector> // definición de la plantilla de clase vector
9 #include <iostream> // ostream_iterator
10
11 int main()
12 {
13     const int TAMANIO = 10;
14     int a1[ TAMANIO ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
15     int a2[ TAMANIO ] = { 1, 2, 3, 4, 1000, 6, 7, 8, 9, 10 };
16     std::vector< int > v1( a1, a1 + TAMANIO ); // copia de a1
17     std::vector< int > v2( a1, a1 + TAMANIO ); // copia de a1
18     std::vector< int > v3( a2, a2 + TAMANIO ); // copia de a2
19     std::ostream_iterator< int > salida( cout, " " );
20
21     cout << "El vector v1 contiene: ";
22     std::copy( v1.begin(), v1.end(), salida );
23     cout << "\nEl vector v2 contiene: ";
24     std::copy( v2.begin(), v2.end(), salida );
25     cout << "\nEl vector v3 contiene: ";
26     std::copy( v3.begin(), v3.end(), salida );
27
28     // compara los vectores v1 y v2 para ver si son iguales
29     bool resultado = std::equal( v1.begin(), v1.end(), v2.begin() );
30     cout << "\n\nEl vector v1 " << ( resultado ? "es" : "no es" )
31         << " igual al vector v2.\n";
32
33     // compara los vectores v1 y v3 para ver si son iguales
34     resultado = std::equal( v1.begin(), v1.end(), v3.begin() );
35     cout << "El vector v1 " << ( resultado ? "es" : "no es" )
36         << " igual al vector v3.\n";
37
38     // ubicacion representa el par de iteradores del vector
39     std::pair< std::vector< int >::iterator,
40               std::vector< int >::iterator > ubicacion;
41
42     // comprueba que no haya inconsistencia entre v1 y v3
43     ubicacion = std::mismatch( v1.begin(), v1.end(), v3.begin() );
44     cout << "\nHay una inconsistencia entre v1 y v3 en la ubicacion "
45         << ( ubicacion.first - v1.begin() ) << "\nen donde v1 contiene "
46         << *ubicacion.first << " y v3 contiene " << *ubicacion.second
47         << "\n\n";
48
49     char c1[ TAMANIO ] = "HOLA";
50     char c2[ TAMANIO ] = "BYE BYE";
51
52     // realiza una comparación lexicográfica de c1 y c2
53     resultado = std::lexicographical_compare( c1, c1 + TAMANIO, c2, c2 + TAMANIO );
54     cout << c1 << ( resultado ? " es menor que " :
55                         " es mayor o igual a " ) << c2 << endl;
56     return 0;
57 } // fin de main

```

```

El vector v1 contiene: 1 2 3 4 5 6 7 8 9 10
El vector v2 contiene: 1 2 3 4 5 6 7 8 9 10
El vector v3 contiene: 1 2 3 4 1000 6 7 8 9 10

El vector v1 es igual al vector v2.
El vector v1 no es igual al vector v3.

Hay una inconsistencia entre v1 y v3 en la ubicacion 4
en donde v1 contiene 5 y v3 contiene 1000

HOLA es mayor o igual a BYE BYE

```

Figura 22.27 | Los algoritmos `equal`, `mismatch` y `lexicographical_compare`. (Parte 2 de 2).

La línea 29 utiliza la función `equal` para comparar dos secuencias de valores y ver si son iguales. Cada secuencia no necesariamente contiene el mismo número de elementos: `equal` devuelve `false` si las secuencias no son de la misma longitud. La función `operator==` (ya sea integrada o sobrecargada) realiza la comparación de los elementos. En este ejemplo se comparan los elementos del vector `v1` desde `v1.begin()` hasta (pero sin incluir a) `v1.end()` con los elementos del vector `v2` empezando desde `v2.begin()`. En este ejemplo, `v1` y `v2` son iguales. Los tres iteradores que se toman como argumentos deben ser por lo menos iteradores de entrada (es decir, que puedan utilizarse para introducir valores desde una secuencia en dirección hacia adelante). En la línea 34 se utiliza a la función `equal` para comparar los vectores `v1` y `v3`, que no son iguales.

Hay otra versión de la función `equal` que toma una función predicado binaria como cuarto parámetro. Esta función recibe los dos elementos que se van a comparar y devuelve un valor `bool` que indica si los elementos son iguales o no. Esto puede ser útil en secuencias que almacenan objetos o apuntadores a valores, en vez de los valores actuales, ya que se pueden definir una o más comparaciones. Por ejemplo, puede comparar objetos `Empleado` con base en la edad, número de seguro social o ubicación en vez de comparar objetos completos. Puede comparar a qué hacen referencia los apuntadores, en vez de comparar su contenido (es decir, las direcciones almacenadas en los apuntadores).

Las líneas 39 a 43 empiezan por crear la instancia de un objeto `pair` de iteradores llamado `ubicacion`, para un vector de enteros. Este objeto almacena el resultado de la llamada a `mismatch` (línea 43). La función `mismatch` compara dos secuencias de valores y devuelve un objeto `pair` de iteradores que indican la posición en cada secuencia de los elementos inconsistentes. Si todos los elementos concuerdan, los dos iteradores en el objeto `pair` son iguales al último iterador para cada secuencia. Los tres iteradores que se toman como argumentos deben ser por lo menos iteradores de entrada. En la línea 45 se determina la ubicación actual de la inconsistencia en los vectores mediante la expresión `ubicacion.first - v1.begin()`. El resultado de este cálculo es el número de elementos entre los iteradores (esto es análogo a la aritmética de apuntadores que estudiamos en el capítulo 8). Esto corresponde al número de elementos en este ejemplo, ya que la comparación se realiza desde el inicio de cada vector. Al igual que con la función `equal`, hay otra versión de la función `mismatch` que recibe una función predicado binaria como cuarto parámetro.

En la línea 53 se utiliza la función `lexicographical_compare` para comparar el contenido de dos arreglos de caracteres. Los cuatro iteradores que se toman como argumentos en esta función deben ser por lo menos iteradores de entrada. Como usted sabe, los apuntadores en arreglos son iteradores de acceso aleatorio. Los primeros dos argumentos de iteradores especifican el rango de posiciones en la primera secuencia. Los últimos dos especifican el rango de las ubicaciones en la segunda secuencia. Al iterar a través de las secuencias, `lexicographical_compare` comprueba si el elemento en la primera secuencia es menor que el elemento correspondiente en la segunda secuencia. De ser así, la función devuelve `true`. Si el elemento en la primera secuencia es mayor o igual que el elemento en la segunda, la función devuelve `false`. Esta función se puede utilizar para ordenar las secuencias en forma lexicográfica. Por lo general, dichas secuencias contienen cadenas.

22.5.3 `remove`, `remove_if`, `remove_copy` y `remove_copy_if`

La figura 22.28 demuestra cómo eliminar valores de una secuencia mediante los algoritmos `remove`, `remove_if`, `remove_copy` y `remove_copy_if`.

En la línea 26 se utiliza la función `remove` para eliminar todos los elementos con el valor 10 en el rango desde `v.begin()` hasta, pero sin incluir a, `v.end()` del vector `v`. Los primeros dos iteradores que se reciben como argumentos deben ser iteradores de avance, para que el algoritmo pueda modificar los elementos en la secuencia. Esta función no modifica el número de elementos en el vector ni destruye los elementos eliminados, pero sí desplaza a todos los elementos que no se eliminan hacia el inicio del vector. La función devuelve un iterador colocado después del último elemento del vector que no se haya eliminado. Los elementos que se encuentran desde la posición del iterador hasta el final del vector tienen valores indefinidos (en este ejemplo, cada posición “indefinida” tiene el valor 0).

En la línea 36 se utiliza la función `remove_copy` para copiar todos los elementos que no tengan el valor 10 que se encuentren en el rango desde `v2.begin()` hasta, pero sin incluir a, `v2.end()` del vector `v2`. Los elementos se colocan en `c`, empezando en la posición `c.begin()`. Los iteradores que se suministran como los primeros dos argumentos deben ser iteradores de entrada. El iterador que se suministra como tercer argumento debe ser un iterador de salida, para que el elemento que se va a copiar pueda insertarse en la ubicación de copia. Esta función devuelve un iterador que está colocado después del último elemento copiado al vector `c`. Observe en la línea 31 el uso del constructor del vector que recibe el número de elementos en el mismo y los valores iniciales de esos elementos.

```

1 // Fig. 22.28: Fig22_28.cpp
2 // Las funciones remove, remove_if, remove_copy y
3 // remove_copy_if de la Biblioteca estándar.

```

Figura 22.28 | Los algoritmos `remove`, `remove_if`, `remove_copy` y `remove_copy_if`. (Parte I de 3).

```

4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm> // definiciones de los algoritmos
9 #include <vector> // definición de la plantilla de clase vector
10 #include <iterator> // ostream_iterator
11
12 bool mayor9( int ); // prototipo
13
14 int main()
15 {
16     const int TAMANIO = 10;
17     int a[ TAMANIO ] = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
18     std::ostream_iterator< int > salida( cout, " " );
19     std::vector< int > v( a, a + TAMANIO ); // copia de a
20     std::vector< int >::iterator nuevoUltimoElemento;
21
22     cout << "Vector v antes de eliminar todos los 10s:\n    ";
23     std::copy( v.begin(), v.end(), salida );
24
25     // elimina todos los 10s de v
26     nuevoUltimoElemento = std::remove( v.begin(), v.end(), 10 );
27     cout << "\nVector v despues de eliminar todos los 10s:\n    ";
28     std::copy( v.begin(), nuevoUltimoElemento, salida );
29
30     std::vector< int > v2( a, a + TAMANIO ); // copia de a
31     std::vector< int > c( TAMANIO, 0 ); // crea instancia del vector c
32     cout << "\n\nVector v2 antes de eliminar todos los 10s y copiar:\n    ";
33     std::copy( v2.begin(), v2.end(), salida );
34
35     // copia de v2 a c, eliminando los 10s en el proceso
36     std::remove_copy( v2.begin(), v2.end(), c.begin(), 10 );
37     cout << "\nVector c despues de eliminar todos los 10s de v2:\n    ";
38     std::copy( c.begin(), c.end(), salida );
39
40     std::vector< int > v3( a, a + TAMANIO ); // copia de a
41     cout << "\n\nVector v3 antes de eliminar todos los elementos"
42         << "\nmayores que 9:\n    ";
43     std::copy( v3.begin(), v3.end(), salida );
44
45     // elimina los elementos mayores que 9 de v3
46     nuevoUltimoElemento = std::remove_if( v3.begin(), v3.end(), mayor9 );
47     cout << "\nVector v3 despues de eliminar todos los elementos"
48         << "\nmayores que 9:\n    ";
49     std::copy( v3.begin(), nuevoUltimoElemento, salida );
50
51     std::vector< int > v4( a, a + TAMANIO ); // copia de a
52     std::vector< int > c2( TAMANIO, 0 ); // crea instancia del vector c2
53     cout << "\n\nVector v4 antes de eliminar todos los elementos"
54         << "\nmayores que 9 y copiar:\n    ";
55     std::copy( v4.begin(), v4.end(), salida );
56
57     // copia elementos de v4 a c2, eliminando los elementos
58     // mayores que 9 en el proceso
59     std::remove_copy_if( v4.begin(), v4.end(), c2.begin(), mayor9 );
60     cout << "\nVector c2 despues de eliminar todos los elementos"
61         << "\nmayores que 9 de v4:\n    ";
62     std::copy( c2.begin(), c2.end(), salida );
63     cout << endl;
64     return 0;
65 } // fin de main

```

Figura 22.28 | Los algoritmos remove, remove_if, remove_copy y remove_copy_if. (Parte 2 de 3).

```

66 // determina si el argumento es mayor que 9
67 bool mayor9( int x )
68 {
69     return x > 9;
70 } // fin de la función mayor9

Vector v antes de eliminar todos los 10s:
    10 2 10 4 16 6 14 8 12 10
Vector v despues de eliminar todos los 10s:
    2 4 16 6 14 8 12

Vector v2 antes de eliminar todos los 10s y copiar:
    10 2 10 4 16 6 14 8 12 10
Vector c despues de eliminar todos los 10s de v2:
    2 4 16 6 14 8 12 0 0 0

Vector v3 antes de eliminar todos los elementos
mayores que 9:
    10 2 10 4 16 6 14 8 12 10
Vector v3 despues de eliminar todos los elementos
mayores que 9:
    2 4 6 8

Vector v4 antes de eliminar todos los elementos
mayores que 9 y copiar:
    10 2 10 4 16 6 14 8 12 10
Vector c2 despues de eliminar todos los elementos
mayores que 9 de v4:
    2 4 6 8 0 0 0 0 0 0

```

Figura 22.28 | Los algoritmos `remove`, `remove_if`, `remove_copy` y `remove_copy_if`. (Parte 3 de 3).

En la línea 46 se utiliza la función `remove_if` para eliminar todos aquellos elementos en el rango de `v3.begin()` hasta, pero sin incluir `a`, `v3.end()` de `v3`, para lo cual nuestra función predicado unaria `mayor9` definida por el usuario devuelve `true`. Esta función (definida en las líneas 68 a 71) devuelve `true` si el valor que recibe es mayor que 9; en caso contrario devuelve `false`. Los iteradores suministrados como los primeros dos argumentos deben ser iteradores de avance, para que el algoritmo pueda modificar los elementos en la secuencia. Esta función no modifica el número de elementos en el vector, pero sí desplaza al inicio del vector todos los elementos que no se eliminan. Esta función devuelve un iterador que se coloca después del último elemento en el vector que no se haya eliminado. Todos los elementos a partir de la posición del iterador y hasta el final del vector tienen valores indefinidos.

En la línea 59 se utiliza la función `remove_copy_if` para copiar todos aquellos elementos en el rango desde `v4.begin()` hasta, pero sin incluir `a`, `v4.end()` del vector `v4` para los que la función predicado unaria `mayor9` devuelva `true`. Los elementos se colocan en `c2`, empezando en la posición `c2.begin()`. Los iteradores que se suministran como los primeros dos argumentos deben ser iteradores de entrada. El iterador que se suministra como tercer argumento debe ser un iterador de salida, para que el elemento que se va a copiar pueda insertarse en la ubicación de copia. Esta función devuelve un iterador que se coloca después del último elemento copiado en `c2`.

22.5.4 `replace`, `replace_if`, `replace_copy` y `replace_copy_if`

La figura 22.29 demuestra cómo reemplazar valores de una secuencia mediante los algoritmos `replace`, `replace_if`, `replace_copy` y `replace_copy_if`.

La línea 25 utiliza la función `replace` para reemplazar todos los elementos con el valor 10 en el rango de `v1.begin()` hasta, pero sin incluir `a`, `v1.end()` en el vector `v1` con el nuevo valor 100. Los iteradores que se suministran como los primeros dos argumentos deben ser iteradores de avance, de manera que el algoritmo pueda modificar los elementos en la secuencia.

En la línea 35 se utiliza la función `replace_copy` para copiar todos los elementos en el rango de `v2.begin()` hasta, pero sin incluir `a`, `v2.end()` del vector `v2`, reemplazando a todos los elementos que tienen el valor 10 con el nuevo valor 100. Los elementos se copian en `c1`, empezando en la posición `c1.begin()`. Los iteradores que se suministran como los primeros dos argumentos deben ser iteradores de entrada. El iterador que se suministra como el tercer argumento debe

```

1 // Fig. 22.29: Fig22_29.cpp
2 // Las funciones replace, replace_if, replace_copy
3 // y replace_copy_if de la Biblioteca estándar.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm>
9 #include <vector>
10 #include <iterator> // ostream_iterator
11
12 bool mayor9( int ); // prototipo de la función predicado
13
14 int main()
15 {
16     const int TAMANIO = 10;
17     int a[ TAMANIO ] = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
18     std::ostream_iterator< int > salida( cout, " " );
19
20     std::vector< int > v1( a, a + TAMANIO ); // copia de a
21     cout << "Vector v1 antes de reemplazar todos los 10s:\n    ";
22     std::copy( v1.begin(), v1.end(), salida );
23
24     // reemplaza todos los 10s en v1 con 100
25     std::replace( v1.begin(), v1.end(), 10, 100 );
26     cout << "\nVector v1 después de reemplazar todos los 10s con 100s:\n    ";
27     std::copy( v1.begin(), v1.end(), salida );
28
29     std::vector< int > v2( a, a + TAMANIO ); // copia de a
30     std::vector< int > c1( TAMANIO ); // crea una instancia del vector c1
31     cout << "\n\nVector v2 antes de reemplazar todos los 10s y copiar:\n    ";
32     std::copy( v2.begin(), v2.end(), salida );
33
34     // copia de v2 a c1, reemplazando los 10s con 100s
35     std::replace_copy( v2.begin(), v2.end(), c1.begin(), 10, 100 );
36     cout << "\nVector c1 después de reemplazar todos los 10s en v2:\n    ";
37     std::copy( c1.begin(), c1.end(), salida );
38
39     std::vector< int > v3( a, a + TAMANIO ); // copia de a
40     cout << "\n\nVector v3 antes de reemplazar valores mayores que 9:\n    ";
41     std::copy( v3.begin(), v3.end(), salida );
42
43     // reemplaza los valores mayores que 9 en v3 con 100
44     std::replace_if( v3.begin(), v3.end(), mayor9, 100 );
45     cout << "\nVector v3 después de reemplazar todos los valores "
46         << "\nmayores que 9 con 100s:\n    ";
47     std::copy( v3.begin(), v3.end(), salida );
48
49     std::vector< int > v4( a, a + TAMANIO ); // copia de a
50     std::vector< int > c2( TAMANIO ); // crea instancia del vector c2
51     cout << "\n\nVector v4 antes de reemplazar todos los valores "
52         << "que 9 y copiar:\n    ";
53     std::copy( v4.begin(), v4.end(), salida );
54
55     // copia v4 a c2, reemplazando los elementos mayores que 9 con 100
56     std::replace_copy_if(
57         v4.begin(), v4.end(), c2.begin(), mayor9, 100 );
58     cout << "\nVector c2 después de reemplazar todos los valores mayores "
59         << "que 9 en v4:\n    ";
60     std::copy( c2.begin(), c2.end(), salida );
61     cout << endl;
62     return 0;

```

Figura 22.29 | Algoritmos replace, replace_if, replace_copy y replace_copy_if. (Parte I de 2).

```

63 } // fin de main
64
65 // determina si el argumento es mayor que 9
66 bool mayor9( int x )
67 {
68     return x > 9;
69 } // fin de la función mayor9

Vector v1 antes de reemplazar todos los 10s:
10 2 10 4 16 6 14 8 12 10
Vector v1 despues de reemplazar los 10s con 100s:
100 2 100 4 16 6 14 8 12 100

Vector v2 antes de reemplazar todos los 10s y copiar:
10 2 10 4 16 6 14 8 12 10
Vector c1 despues de reemplazar todos los 10s en v2:
100 2 100 4 16 6 14 8 12 100

Vector v3 antes de reemplazar valores mayores que 9:
10 2 10 4 16 6 14 8 12 10
Vector v3 despues de reemplazar todos los valores
mayores que 9 con 100s:
100 2 100 4 100 6 100 8 100 100

Vector v4 antes de reemplazar todos los valores mayores que 9 y copiar:
10 2 10 4 16 6 14 8 12 10
Vector c2 despues de reemplazar todos los valores mayores que 9 en v4:
100 2 100 4 100 6 100 8 100 100

```

Figura 22.29 | Algoritmos replace, replace_if, replace_copy y replace_copy_if. (Parte 2 de 2).

ser un iterador de salida, para que el elemento que va a copiarse pueda insertarse en la ubicación de copia. Esta función devuelve un iterador que se coloca después del último elemento copiado en `c1`.

En la línea 44 se utiliza la función `replace_if` para reemplazar a todos aquellos elementos en el rango de `v3.begin()` hasta, pero sin incluir `a, v3.end()` en el vector `v3` para el que la función predicado unaria `mayor9` devuelva `true`. Esta función (definida en las líneas 66 a 69) devuelve `true` si el valor que recibe es mayor que 9; en cualquier otro caso devuelve `false`. El valor 100 sustituye a cualquier valor mayor de 9. Los iteradores que se suministran como los primeros dos argumentos deben ser iteradores de avance, de manera que el algoritmo pueda modificar los elementos en la secuencia.

En las líneas 56 y 57 se utiliza la función `replace_copy_if` para copiar todos los elementos en el rango que empieza desde `v4.begin()` hasta, pero sin incluir a `v4.end()` del vector `v4`. Los elementos para los que la función predicado unaria `mayor9` devuelva `true` se reemplazan con el valor 100. Los elementos se colocan en `c2`, empezando en la posición `c2.begin()`. Los iteradores que se suministran como los primeros dos argumentos deben ser iteradores de entrada. El iterador que se suministra como el tercer argumento debe ser un iterador de salida, para que el elemento que va a copiarse pueda insertarse en la ubicación de copia. Esta función devuelve un iterador que se coloca después del último elemento copiado en `c2`.

22.5.5 Algoritmos matemáticos

La figura 22.30 demuestra varios algoritmos matemáticos comunes de la STL, incluyendo a `random_shuffle`, `count`, `count_if`, `min_element`, `max_element`, `accumulate`, `for_each` y `transform`.

```

1 // Fig. 22.30: Fig22_30.cpp
2 // Algoritmos matemáticos de la Biblioteca estándar.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <algorithm> // definiciones de algoritmos

```

Figura 22.30 | Algoritmos matemáticos de la Biblioteca estándar. (Parte 1 de 2).

```

8 #include <numeric> // aquí se define accumulate
9 #include <vector>
10 #include <iostream>
11
12 bool mayor9( int ); // prototipo de la función predicado
13 void imprimirCuadrado( int ); // imprime el cuadrado de un valor
14 int calcularCubo( int ); // calcula el cubo de un valor
15
16 int main()
17 {
18     const int TAMANIO = 10;
19     int a1[ TAMANIO ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
20     std::vector< int > v( a1, a1 + TAMANIO ); // copia de a1
21     std::ostream_iterator< int > salida( cout, " " );
22
23     cout << "Vector v antes de random_shuffle: ";
24     std::copy( v.begin(), v.end(), salida );
25
26     std::random_shuffle( v.begin(), v.end() ); // revuelve los elementos de v
27     cout << "\nVector v después de random_shuffle: ";
28     std::copy( v.begin(), v.end(), salida );
29
30     int a2[ TAMANIO ] = { 100, 2, 8, 1, 50, 3, 8, 8, 9, 10 };
31     std::vector< int > v2( a2, a2 + TAMANIO ); // copia de a2
32     cout << "\n\nVector v2 contiene: ";
33     std::copy( v2.begin(), v2.end(), salida );
34
35     // cuenta el número de elementos en v2 con el valor 8
36     int resultado = std::count( v2.begin(), v2.end(), 8 );
37     cout << "\nNúmero de elementos que concuerdan con 8: " << resultado;
38
39     // cuenta el número de elementos en v2 mayores que 9
40     resultado = std::count_if( v2.begin(), v2.end(), mayor9 );
41     cout << "\nNúmero de elementos mayores que 9: " << resultado;
42
43     // localiza el elemento mínimo en v2
44     cout << "\n\nEl elemento mínimo en el vector v2 es: "
45     << *( std::min_element( v2.begin(), v2.end() ) );
46
47     // localiza el elemento máximo en v2
48     cout << "\nEl elemento máximo en vector v2 es: "
49     << *( std::max_element( v2.begin(), v2.end() ) );
50
51     // calcula la suma de los elementos en v
52     cout << "\n\nEl total de los elementos en el vector v es: "
53     << std::accumulate( v.begin(), v.end(), 0 );
54
55     // imprime el cuadrado de cada elemento en v
56     cout << "\n\nEl cuadrado de cada entero en el vector v es:\n";
57     std::for_each( v.begin(), v.end(), imprimirCuadrado );
58
59     std::vector< int > cubos( TAMANIO ); // crea instancia del vector cubos
60
61     // calcula el cubo de cada elemento en v; coloca los resultados en cubos
62     std::transform( v.begin(), v.end(), cubos.begin(), calcularCubo );
63     cout << "\n\nEl cubo de cada entero en el vector v es:\n";
64     std::copy( cubos.begin(), cubos.end(), salida );
65     cout << endl;
66     return 0;
67 } // fin de main
68

```

Figura 22.30 | Algoritmos matemáticos de la Biblioteca estándar. (Parte I de 2).

```

69 // determina si el argumento es mayor que 9
70 bool mayor9( int valor )
71 {
72     return valor > 9;
73 } // fin de la función mayor9
74
75 // imprime el cuadrado del argumento
76 void imprimirCuadrado( int valor )
77 {
78     cout << valor * valor << ' ';
79 } // fin de la función imprimirCuadrado
80
81 // devuelve el cubo del argumento
82 int calcularCubo( int valor )
83 {
84     return valor * valor * valor;
85 } // fin de la función calcularCubo

```

```

Vector v antes de random_shuffle: 1 2 3 4 5 6 7 8 9 10
Vector v despues de random_shuffle: 9 2 10 3 1 6 8 4 5 7

Vector v2 contiene: 100 2 8 1 50 3 8 8 9 10
Número de elementos que concuerdan con 8: 3
Número de elementos mayores que 9: 3

El elemento minimo en el vector v2 es: 1
El elemento maximo en vector v2 es: 100
El total de los elementos en el vector v es: 55
El cuadrado de cada entero en el vector v es:
81 4 100 9 1 36 64 16 25 49

El cubo de cada entero en el vector v es:
729 8 1000 27 1 216 512 64 125 343

```

Figura 22.30 | Algoritmos matemáticos de la Biblioteca estándar. (Parte 2 de 2).

En la línea 26 se utiliza la función `random_shuffle` para reordenar en forma aleatoria los elementos en el rango empezando desde `v.begin()` hasta, pero sin incluir `a, v.end()` en el vector `v`. Esta función toma dos iteradores de acceso aleatorio como argumentos.

En la línea 36 se utiliza a la función `count` para contar los elementos que tengan el valor de 8 en el rango que empieza desde `v2.begin()` hasta, pero sin incluir `a, v2.end()` en el vector `v2`. Esta función requiere que sus dos argumentos iteradores sean por lo menos iteradores de entrada.

En la línea 40 se utiliza la función `count_if` para contar los elementos en el rango desde `v2.begin()` hasta, pero sin incluir `a, v2.end()` en el vector `v2` para los que la función predicado `mayor9` devuelva `true`. La función `count_if` requiere que sus dos argumentos iteradores sean por lo menos iteradores de entrada.

En la línea 45 se utiliza la función `min_element` para localizar el menor elemento en el rango que empieza desde `v2.begin()` hasta, pero sin incluir `a, v2.end()` en el vector `v2`. La función devuelve un iterador de entrada localizado en el menor elemento o, si el rango está vacío, devuelve al iterador en sí. La función requiere que sus dos argumentos iteradores sean por lo menos iteradores de entrada. Una segunda versión de esta función toma como tercer argumento una función binaria que compara los elementos en la secuencia. Esta función binaria devuelve el valor `bool true` si el primer argumento es menor que el segundo.



Buena práctica de programación 22.2

Es una buena práctica comprobar que el rango especificado en una llamada a `min_element` no esté vacío y comprobar también que el valor de retorno no sea el iterador que está “más allá del final”.

En la línea 49 se utiliza la función `max_element` para localizar el mayor elemento en el rango que empieza desde `v2.begin()` hasta, pero sin incluir `a, v2.end()` del vector `v2`. La función devuelve un iterador de entrada que se coloca en el elemento más grande. Esta función requiere que sus dos argumentos iteradores sean por lo menos iteradores de entrada. Una segunda versión de esta función toma como su tercer argumento a una función predicado binaria que

compara los elementos en la secuencia. La función binaria toma dos argumentos y devuelve el valor `bool true` si el primer argumento es menor que el segundo.

En la línea 53 se utiliza la función `accumulate` (cuya plantilla se encuentra en el archivo de encabezado `<numeric>`) para sumar los valores en el rango que empieza desde `v.begin()` hasta, pero sin incluir `a, v.end()` en el vector `v`. Los dos argumentos iteradores de la función deben ser por lo menos iteradores de entrada y su tercer argumento representa el valor inicial del total. Una segunda versión de esta función toma como cuarto argumento una función general que determina cómo se acumulan los elementos. Esta función general debe tomar dos argumentos y devolver un resultado. El primer argumento de esta función es el valor actual de la acumulación. El segundo argumento es el valor del elemento actual en la secuencia que se va a acumular.

En la línea 57 se utiliza la función `for_each` para aplicar una función general a cada elemento en el rango que empieza desde `v.begin()` hasta, pero sin incluir `a, v.end()` en el vector `v`. La función general toma el elemento actual como argumento y puede modificarlo (si se recibe por referencia). La función `for_each` requiere que sus dos argumentos iteradores sean por lo menos iteradores de entrada.

En la línea 62 se utiliza la función `transform` para aplicar una función general a cada elemento en el rango que empieza desde `v.begin()` hasta, pero sin incluir `a, v.end()` del vector `v`. La función general (el cuarto argumento) debe tomar el elemento actual como argumento, no debe modificarlo y debe regresar el valor transformado (mediante `transform`). La función `transform` requiere que sus primeros dos argumentos iteradores sean por lo menos iteradores de entrada y que su tercer argumento sea por lo menos un iterador de salida. El tercer argumento especifica en dónde deben colocarse los valores transformados. Observe que el tercer argumento puede ser igual al primero. Otra versión de `transform` acepta cinco argumentos: los primeros dos argumentos son iteradores de entrada que especifican un rango de elementos de un contenedor de origen, el tercer argumento es un iterador de entrada que especifica el primer elemento en otro contenedor de origen, el cuarto argumento es un iterador de salida que especifica en dónde se deben colocar los valores transformados, y el último argumento es una función general que recibe dos argumentos. Esta versión de `transform` toma un elemento de cada uno de los dos orígenes y aplica la función general a ese par de elementos, y después coloca el valor transformado en la ubicación especificada por el cuarto argumento.

22.5.6 Algoritmos básicos de búsqueda y ordenamiento

La figura 22.31 demuestra algunas de las herramientas básicas de búsqueda y ordenamiento de la Biblioteca estándar, incluyendo a `find`, `find_if`, `sort` y `binary_search`.

```

1 // Fig. 22.31: Fig22_31.cpp
2 // Algoritmos de búsqueda y ordenamiento de la Biblioteca estándar.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <algorithm> // definiciones de los algoritmos
8 #include <vector> // definición de la plantilla de clase vector
9 #include <iterator>
10
11 bool mayor10( int valor ); // prototipo de la función predicado
12
13 int main()
14 {
15     const int TAMANIO = 10;
16     int a[ TAMANIO ] = { 10, 2, 17, 5, 16, 8, 13, 11, 20, 7 };
17     std::vector< int > v( a, a + TAMANIO ); // copia de a
18     std::ostream_iterator< int > salida( cout, " " );
19
20     cout << "El vector v contiene: ";
21     std::copy( v.begin(), v.end(), salida ); // muestra el vector de salida
22
23     // localiza la primera ocurrencia de 16 en v
24     std::vector< int >::iterator ubicacion;
25     ubicacion = std::find( v.begin(), v.end(), 16 );

```

Figura 22.31 | Algoritmos básicos de búsqueda y ordenamiento de la Biblioteca estándar. (Parte I de 2).

```

26     if ( ubicacion != v.end() ) // found 16
27         cout << "\n\nSe encontro el 16 en la ubicacion " << ( ubicacion - v.begin() );
28     else // no se encontro el 16
29         cout << "\n\nNo se encontro el 16";
30
31
32     // localiza la primera ocurrencia de 100 en v
33     ubicacion = std::find( v.begin(), v.end(), 100 );
34
35     if ( ubicacion != v.end() ) // encontró el 100
36         cout << "\nSe encontro el 100 en la ubicacion " << ( ubicacion - v.begin() );
37     else // no se encontró el 100
38         cout << "\nNo se encontro el 100";
39
40     // localiza la primera ocurrencia del valor que sea mayor que 10 en v
41     ubicacion = std::find_if( v.begin(), v.end(), mayor10 );
42
43     if ( ubicacion != v.end() ) // encontro un valor mayor que 10
44         cout << "\n\nEl primer valor mayor que 10 es " << *ubicacion
45         << "\nse encontro en la ubicacion " << ( ubicacion - v.begin() );
46     else // no se encontró un valor mayor que 10
47         cout << "\n\nNo se encontraron valores mayores que 10";
48
49     // ordena los elementos de v
50     std::sort( v.begin(), v.end() );
51     cout << "\n\nVector v despues de sort: ";
52     std::copy( v.begin(), v.end(), salida );
53
54     // usa binary_search para localizar el 13 en v
55     if ( std::binary_search( v.begin(), v.end(), 13 ) )
56         cout << "\n\nSe encontro el 13 en v";
57     else
58         cout << "\n\nNo se encontro el 13 en v";
59
60     // usa binary_search para localizar el 100 en v
61     if ( std::binary_search( v.begin(), v.end(), 100 ) )
62         cout << "\nSe encontro el 100 en v";
63     else
64         cout << "\nNo se encontro el 100 en v";
65
66     cout << endl;
67     return 0;
68 } // fin de main
69
70 // determina si el argumento es mayor que 10
71 bool mayor10( int valor )
72 {
73     return valor > 10;
74 } // fin de la función mayor10

```

```

El vector v contiene: 10 2 17 5 16 8 13 11 20 7

Se encontro el 16 en la ubicacion 4
No se encontro el 100

El primer valor mayor que 10 es 17
se encontro en la ubicacion 2

Vector v despues de sort: 2 5 7 8 10 11 13 16 17 20
Se encontro el 13 en v

No se encontro el 100 en v

```

Figura 22.31 | Algoritmos básicos de búsqueda y ordenamiento de la Biblioteca estándar. (Parte 2 de 2).

En la línea 25 se utiliza la función `find` para localizar el valor 16 en el rango que empieza desde `v.begin()` hasta, pero sin incluir `a, v.end()` del vector `v`. La función requiere que sus dos argumentos iteradores sean por lo menos iteradores de entrada, y devuelve un iterador de entrada que, o se coloca en el primer elemento que contiene el valor, o indica el final de la secuencia (como es el caso en la línea 33).

En la línea 41 se utiliza la función `find_if` para localizar el primer valor en el rango empezando desde `v.begin()` hasta, pero sin incluir `a, v.end()` en el vector `v` para el que la función predicado unaria `mayor10` devuelva `true`. Esta función `mayor10` (definida en las líneas 71 a 74) toma un entero y devuelve un valor `bool` que indica si el argumento entero es mayor que 10. La función `find_if` requiere que sus dos argumentos iteradores sean por lo menos iteradores de entrada. La función devuelve un iterador de entrada que, o se coloca en el primer elemento que contiene un valor para el que la función predicado devuelva `true`, o indica el final de la secuencia.

En la línea 50 se utiliza la función `sort` para ordenar los elementos en el rango que empieza desde `v.begin()` hasta, pero sin incluir `a, v.end()` en el vector `v`, en orden ascendente. La función requiere que sus dos argumentos iteradores sean iteradores de acceso aleatorio. Una segunda versión de esta función toma un tercer argumento, el cual es una función predicado binaria que toma dos argumentos que son valores en la secuencia y devuelve un `bool` que indica el orden de los elementos; si el valor de retorno es `true`, los dos elementos que se están comparando se encuentran en orden.



Error común de programación 22.5

Tratar de ordenar un contenedor (con sort) mediante el uso de un iterador que no sea de acceso aleatorio es un error de sintaxis. La función sort requiere un iterador de acceso aleatorio.

En la línea 55 se utiliza la función `binary_search` para determinar si el valor 13 se encuentra en el rango empezando desde `v.begin()` hasta, pero sin incluir `a, v.end()` en el vector `v`. La secuencia de valores debe ordenarse primero en forma ascendente. La función `binary_search` requiere que sus dos argumentos iteradores sean por lo menos iteradores de avance. La función devuelve un valor `bool` que indica si se encontró el valor en la secuencia. En la línea 61 se demuestra una llamada a la función `binary_search`, en donde el valor no se encontró. Una segunda versión de esta función toma un cuarto argumento, el cual es una función predicado binaria que toma dos argumentos que son valores en la secuencia y devuelve un `bool`. La función predicado devuelve `true` si los dos elementos que se están comparando se encuentran en orden. Para obtener la ubicación de la clave de búsqueda en el contenedor, use los algoritmos `lower_bound` o `find`.

22.5.7 swap, iter_swap y swap_ranges

La figura 22.32 demuestra el uso de los algoritmos `swap`, `iter_swap` y `swap_ranges` para intercambiar elementos. En la línea 20 se utiliza la función `swap` para intercambiar dos valores. En este ejemplo se intercambian el primer y segundo elementos del arreglo `a`. La función toma como argumentos las referencias a los dos valores que se van a intercambiar.

```

1 // Fig. 22.32: Fig22_32.cpp
2 // Los algoritmos iter_swap, swap and swap_ranges de la Biblioteca estándar.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <algorithm> // definiciones de los algoritmos
8 #include <iterator>
9
10 int main()
11 {
12     const int TAMANIO = 10;
13     int a[ TAMANIO ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
14     std::ostream_iterator< int > salida( cout, " " );
15
16     cout << "El arreglo a contiene:\n    ";
17     std::copy( a, a + TAMANIO, salida ); // muestra el arreglo a

```

Figura 22.32 | Demostración de `swap`, `iter_swap` y `swap_ranges`. (Parte I de 2).

```

18 // intercambia los elementos en las ubicaciones 0 y 1 del arreglo a
19 std::swap( a[ 0 ], a[ 1 ] );
20
21 cout << "\nEl arreglo a despues de intercambiar a[0] y a[1] mediante swap:\n    ";
22 std::copy( a, a + TAMANIO, salida ); // muestra el arreglo a
23
24 // usa iteradores para intercambiar los elementos en las ubicaciones 0 y 1 del arreglo a
25 std::iter_swap( &a[ 0 ], &a[ 1 ] ); // intercambia con iteradores
26 cout << "\nEl arreglo a despues de intercambiar a[0] y a[1] mediante iter_swap:\n    ";
27 std::copy( a, a + TAMANIO, salida );
28
29 // intercambia los primeros cinco elementos del arreglo a con
30 // los últimos cinco elementos del arreglo a
31 std::swap_ranges( a, a + 5, a + 5 );
32
33 cout << "\nEl arreglo a despues de intercambiar los primeros cinco elementos\n"
34     << "con los ultimos cinco:\n    ";
35 std::copy( a, a + TAMANIO, salida );
36 cout << endl;
37 return 0;
38 } // fin de main

```

```

El arreglo a contiene:
1 2 3 4 5 6 7 8 9 10
El arreglo a despues de intercambiar a[0] y a[1] mediante swap:
2 1 3 4 5 6 7 8 9 10
El arreglo a despues de intercambiar a[0] y a[1] mediante iter_swap:
1 2 3 4 5 6 7 8 9 10
El arreglo a despues de intercambiar los primeros cinco elementos
con los ultimos cinco:
6 7 8 9 10 1 2 3 4 5

```

Figura 22.32 | Demostración de `swap`, `iter_swap` y `swap_ranges`. (Parte 2 de 2).

En la línea 26 se utiliza la función `iter_swap` para intercambiar los dos elementos. La función toma dos iteradores de avance como argumentos (en este caso, apuntadores a elementos de un arreglo) e intercambia los valores en los elementos a los que hacen referencia los iteradores.

En la línea 32 se utiliza la función `swap_ranges` para intercambiar los elementos en el rango que empieza desde `a` hasta, pero sin incluir `a, a + 5` con los elementos que empiezan desde la posición `a + 5`. La función requiere tres iteradores de avance como argumentos. Los primeros dos argumentos especifican el rango de elementos en la primera secuencia que va a intercambiarse con los elementos en la segunda secuencia, empezando a partir del iterador en el tercer argumento. En este ejemplo, las dos secuencias de valores se encuentran en el mismo arreglo, pero las secuencias pueden provenir de distintos arreglos o contenedores.

22.5.8 `copy_backward`, `merge`, `unique` y `reverse`

La figura 22.33 demuestra el uso de los algoritmos `copy_backward`, `merge`, `unique` y `reverse`. En la línea 28 se utiliza la función `copy_backward` para copiar elementos en el rango que empieza desde `v1.begin()` hasta, pero sin incluir `a, v1.end()` del vector `v1`, colocar los elementos en `resultados` empezando desde elemento que está antes de `resultados.end()` y avanzando hacia el inicio del vector. La función devuelve un iterador colocado en el último elemento que se copia a `resultados` (es decir, el inicio de `resultados`, ya que vamos avanzando al revés). Los elementos se colocan en `resultados` en el mismo orden que `v1`. Esta función requiere de tres iteradores bidireccionales como argumentos (iteradores que pueden incrementarse y decrementarse para iterar hacia delante y hacia atrás a través de una secuencia, respectivamente). La principal diferencia entre `copy` y `copy_backward` es que el iterador que se devuelve de `copy` se coloca *después* del último elemento copiado, y el iterador que se devuelve de `copy_backward` se coloca *en* el último elemento copiado (es decir, el primer elemento en la secuencia). Además, `copy_backward` puede manipular rangos traslapados de elementos en un contenedor, siempre y cuando el primer elemento a copiar no se encuentre en el rango de destino de los elementos.

```

1 // Fig. 22.33: Fig22_33.cpp
2 // Las funciones copy_backward, merge, unique y reverse de la Biblioteca estándar.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <algorithm> // definiciones de los algoritmos
8 #include <vector> // definición de la plantilla de clase vector
9 #include <iterator> // ostream_iterator
10
11 int main()
12 {
13     const int TAMANIO = 5;
14     int a1[ TAMANIO ] = { 1, 3, 5, 7, 9 };
15     int a2[ TAMANIO ] = { 2, 4, 5, 7, 9 };
16     std::vector< int > v1( a1, a1 + TAMANIO ); // copia de a1
17     std::vector< int > v2( a2, a2 + TAMANIO ); // copia de a2
18     std::ostream_iterator< int > salida( cout, " " );
19
20     cout << "El vector v1 contiene: ";
21     std::copy( v1.begin(), v1.end(), salida ); // muestra la salida del vector
22     cout << "\nEl vector v2 contiene: ";
23     std::copy( v2.begin(), v2.end(), salida ); // muestra la salida del vector
24
25     std::vector< int > resultados( v1.size() );
26
27     // coloca los elementos de v1 en resultados, en orden inverso
28     std::copy_backward( v1.begin(), v1.end(), resultados.end() );
29     cout << "\n\nDespues de copy_backward, resultados contiene: ";
30     std::copy( resultados.begin(), resultados.end(), salida );
31
32     std::vector< int > resultados2( v1.size() + v1.size() );
33
34     // combina los elementos de v1 y v2 en resultados2, en orden
35     std::merge( v1.begin(), v1.end(), v2.begin(), v2.end(),
36                 resultados2.begin() );
37
38     cout << "\n\nDespues de combinar v1 y v2, resultados2 contiene:\n";
39     std::copy( resultados2.begin(), resultados2.end(), salida );
40
41     // elimina valores duplicados de resultados2
42     std::vector< int >::iterator ubicacionFinal;
43     ubicacionFinal = std::unique( resultados2.begin(), resultados2.end() );
44
45     cout << "\n\nDespues de unique, resultados2 contiene:\n";
46     std::copy( resultados2.begin(), ubicacionFinal, salida );
47
48     cout << "\n\nVector v1 despues de reverse: ";
49     std::reverse( v1.begin(), v1.end() ); // invierte los elementos de v1
50     std::copy( v1.begin(), v1.end(), salida );
51     cout << endl;
52     return 0;
53 } // fin de main

```

```

El vector v1 contiene: 1 3 5 7 9
El vector v2 contiene: 2 4 5 7 9
Despues de copy_backward, resultados contiene: 1 3 5 7 9
Despues de combinar v1 y v2, resultados2 contiene:
1 2 3 4 5 5 7 7 9 9
Despues de unique, resultados2 contiene:
1 2 3 4 5 7 9
Vector v1 despues de reverse: 9 7 5 3 1

```

Figura 22.33 | Demostración de copy_backward, merge, unique y reverse.

En las líneas 35 y 36 se utiliza la función `merge` para combinar dos secuencias de valores ordenados en forma ascendente en una tercera secuencia ordenada también en forma ascendente. La función requiere de cinco iteradores como argumentos. Los primeros cuatro argumentos deben ser por lo menos iteradores de entrada y el último argumento debe ser por lo menos un iterador de salida. Los primeros dos argumentos especifican el rango de elementos en la primera secuencia ordenada (`v1`), los siguientes dos argumentos especifican el rango de elementos en la segunda secuencia ordenada (`v2`) y el último argumento especifica la posición inicial en la tercera secuencia (`resultados2`) en donde se van a mezclar los elementos. Una segunda versión de esta función toma como su sexto argumento a una función predicado binaria que especifica la forma en que se van a ordenar los elementos.

Observe que en la línea 32 se crea el vector `resultados2` con el número de elementos definido por `v1.size() + v2.size()`. Para utilizar la función `merge` como se muestra aquí, se requiere que la secuencia en donde van a almacenarse los resultados sea por lo menos de un tamaño igual al de las dos secuencias que van a mezclarse. Si no desea asignar el número de elementos para la secuencia resultante antes de la operación `merge`, puede utilizar las siguientes instrucciones:

```
std::vector< int > resultados2();
std::merge (v1.begin(), v1.end(), v2.begin(), v2.end(),
            std::back_inserter( resultados2 ) );
```

El argumento `std::back_inserter(resultados2)` utiliza la plantilla de función `back_inserter` (en el archivo de encabezado `<iostream>`) para el contenedor `resultados2`. Esta función llama a la función `push_back` predeterminada del contenedor para insertar un elemento al final del mismo. Lo que es más importante, si se inserta un elemento en un contenedor que no tenga más espacio disponible, el contenedor aumenta su tamaño. Por lo tanto, el número de elementos en el contenedor no tiene que conocerse de antemano. Hay otros dos insertadores: `front_inserter` (para insertar un elemento al inicio de un contenedor especificado como su argumento) e `inserter` (para insertar un elemento antes del iterador que se proporciona como su segundo argumento en el contenedor que se proporciona como su primer argumento).

En la línea 43 se utiliza la función `unique` en la secuencia ordenada de elementos en el rango que empieza desde `resultados2.begin()` hasta, pero sin incluir a, `resultados2.end()` del vector `resultados2`. Una vez que se aplica esta función a una secuencia ordenada con valores duplicados, sólo se retiene una copia de cada valor en la secuencia. La función toma dos argumentos que deben ser por lo menos iteradores de avance. La función devuelve un iterador que se coloca después del último elemento en la secuencia de valores únicos. Los valores de todos los elementos en el contenedor después del último valor único están indefinidos. Una segunda versión de esta función toma como tercer argumento a una función predicado binaria que especifica cómo comparar dos elementos para ver si son iguales.

En la línea 49 se utiliza la función `reverse` para invertir todos los elementos en el rango que empieza desde `v1.begin()` hasta, pero sin incluir a, `v1.end()` del vector `v1`. La función toma dos argumentos que deben ser por lo menos iteradores bidireccionales.

22.5.9 `inplace_merge`, `unique_copy` y `reverse_copy`

La figura 22.34 demuestra el uso de los algoritmos `inplace_merge`, `unique_copy` y `reverse_copy`. La línea 24 utiliza la función `inplace_merge` para mezclar dos secuencias ordenadas de elementos en el mismo contenedor. En este ejemplo, los elementos de `v1.begin()` hasta, pero sin incluir a, `v1.begin() + 5` se mezclan con los elementos de `v1.begin() + 5` hasta, pero sin incluir a, `v1.end()`. Esta función requiere que sus tres argumentos iteradores sean por lo menos iteradores bidireccionales. Una segunda versión de esta función toma como cuarto argumento a una función predicado binaria para comparar elementos en las dos secuencias.

```
1 // Fig. 22.34: Fig22_34.cpp
2 // Los algoritmos inplace_merge, reverse_copy
3 // y unique_copy de la Biblioteca estándar.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm> // definiciones de los algoritmos
9 #include <vector> // definición de la plantilla de clase vector
10 #include <iiterator> // definición de back_inserter
```

Figura 22.34 | Demostración de `inplace_merge`, `unique_copy` y `reverse_copy`. (Parte I de 2).

```

11
12 int main()
13 {
14     const int TAMANIO = 10;
15     int a1[ TAMANIO ] = { 1, 3, 5, 7, 9, 1, 3, 5, 7, 9 };
16     std::vector< int > v1( a1, a1 + TAMANIO ); // copia de a
17     std::ostream_iterator< int > salida( cout, " " );
18
19     cout << "El vector v1 contiene: ";
20     std::copy( v1.begin(), v1.end(), salida );
21
22     // combina la primera mitad de v1 con la segunda mitad de v1, de tal forma
23     // que v1 contiene un conjunto ordenado de elementos después de la combinación
24     std::inplace_merge( v1.begin(), v1.begin() + 5, v1.end() );
25
26     cout << "\nDespues de inplace_merge, v1 contiene: ";
27     std::copy( v1.begin(), v1.end(), salida );
28
29     std::vector< int > resultados1;
30
31     // copia sólo los elementos únicos de v1 a resultados1
32     std::unique_copy(
33         v1.begin(), v1.end(), std::back_inserter( resultados1 ) );
34     cout << "\nDespues de unique_copy, resultados1 contiene: ";
35     std::copy( resultados1.begin(), resultados1.end(), salida );
36
37     std::vector< int > resultados2;
38
39     // copia los elementos de v1 a resultados2 en orden inverso
40     std::reverse_copy(
41         v1.begin(), v1.end(), std::back_inserter( resultados2 ) );
42     cout << "\nDespues de reverse_copy, resultados2 contiene: ";
43     std::copy( resultados2.begin(), resultados2.end(), salida );
44     cout << endl;
45
46 } // fin de main

```

```

El vector v1 contiene: 1 3 5 7 9 1 3 5 7 9
Despues de inplace_merge, v1 contiene: 1 1 3 3 5 5 7 7 9 9
Despues de unique_copy, resultados1 contiene: 1 3 5 7 9
Despues de reverse_copy, resultados2 contiene: 9 9 7 7 5 5 3 3 1 1

```

Figura 22.34 | Demostración de `inplace_merge`, `unique_copy` y `reverse_copy`. (Parte 2 de 2).

En las líneas 32 y 33 se utiliza la función `unique_copy` para crear una copia de todos los elementos únicos en la secuencia ordenada de valores, empezando desde `v1.begin()` hasta, pero sin incluir a, `v1.end()`. Los elementos copiados se colocan en el vector `resultados1`. Los primeros dos argumentos deben ser por lo menos iteradores de entrada y el último argumento debe ser por lo menos un iterador de salida. En este ejemplo no asignamos previamente suficientes elementos en `resultados1` como para almacenar todos los elementos copiados de `v1`. Lo que hicimos fue utilizar la función `back_inserter` (definida en el archivo de encabezado `<iostream>`) para agregar elementos al final de `v1`. Esta función utiliza la capacidad de la clase `vector` de insertar elementos al final del `vector`. Como `back_inserter` inserta un elemento en vez de reemplazar el valor de un elemento existente, el `vector` puede crecer para dar cabida a más elementos. Una segunda versión de la función `unique_copy` toma como cuarto argumento a una función predicado binaria para comparar la igualdad entre los elementos.

En las líneas 40 y 41 se utiliza la función `reverse_copy` para crear una copia inversa de los elementos en el rango que empieza desde `v1.begin()` hasta, pero sin incluir a, `v1.end()`. Los elementos copiados se insertan en `resultados2` mediante el uso de un objeto `back_inserter` para asegurar que el `vector` pueda crecer para dar cabida al número apropiado de elementos que se copien. La función `reverse_copy` requiere que sus primeros dos argumentos iteradores sean por lo menos iteradores bidireccionales y que su tercer argumento iterador sea por lo menos un iterador de salida.

22.5.10 Operaciones set

En la figura 22.35 se demuestra el uso de las funciones `includes`, `set_difference`, `set_intersection`, `set_symmetric_difference` y `set_union` de la Biblioteca estándar para manipular conjuntos de valores ordenados. Para demostrar que las funciones de la Biblioteca estándar pueden aplicarse a los arreglos y contenedores, este ejemplo utiliza sólo arreglos (recuerde, un apuntador a un arreglo es un iterador de acceso aleatorio).

En las líneas 27 y 33 se llama a la función `includes` en las condiciones de las estructuras `if`. Esta función compara dos conjuntos de valores ordenados para determinar si cada elemento del segundo conjunto se encuentra en el primero. De ser así, `includes` devuelve `true`; en caso contrario, `includes` devuelve `false`. Los primeros dos argumentos iteradores deben ser por lo menos iteradores de entrada y deben describir el primer conjunto de valores. En la línea 27, el primer conjunto consiste de los elementos desde `a1` hasta, pero sin incluir `a1 + TAMANIO1`. Los últimos dos argumentos iteradores deben ser por lo menos iteradores de entrada y deben describir el segundo conjunto de valores. En este ejemplo, el segundo conjunto consiste de los elementos desde `a2` hasta, pero sin incluir `a2 + TAMANIO2`. Una segunda versión de la función `includes` toma un quinto argumento, el cual es una función predicado binaria para comparar la igualdad entre elementos.

```

1 // Fig. 22.35: Fig22_35.cpp
2 // Los algoritmos includes, set_difference, set_intersection,
3 // set_symmetric_difference y set_union.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm> // definiciones de los algoritmos
9 #include <iterator> // ostream_iterator
10
11 int main()
12 {
13     const int TAMANIO1 = 10, TAMANIO2 = 5, TAMANIO3 = 20;
14     int a1[ TAMANIO1 ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
15     int a2[ TAMANIO2 ] = { 4, 5, 6, 7, 8 };
16     int a3[ TAMANIO2 ] = { 4, 5, 6, 11, 15 };
17     std::ostream_iterator< int > salida( cout, " " );
18
19     cout << "a1 contiene: ";
20     std::copy( a1, a1 + TAMANIO1, salida ); // muestra el arreglo a1
21     cout << "\na2 contiene: ";
22     std::copy( a2, a2 + TAMANIO2, salida ); // muestra el arreglo a2
23     cout << "\na3 contiene: ";
24     std::copy( a3, a3 + TAMANIO2, salida ); // muestra el arreglo a3
25
26     // determina si el conjunto a2 está completamente contenido en a1
27     if ( std::includes( a1, a1 + TAMANIO1, a2, a2 + TAMANIO2 ) )
28         cout << "\n\na1 incluye a a2";
29     else
30         cout << "\n\na1 no incluye a a2";
31
32     // determina si el conjunto a3 está completamente contenido en a1
33     if ( std::includes( a1, a1 + TAMANIO1, a3, a3 + TAMANIO2 ) )
34         cout << "\n\na1 incluye a a3";
35     else
36         cout << "\n\na1 no incluye a a3";
37
38     int diferencia[ TAMANIO1 ];
39
40     // determina los elementos de a1 que no están en a2
41     int *ptr = std::set_difference( a1, a1 + TAMANIO1,
42                                     a2, a2 + TAMANIO2, diferencia );
43     cout << "\n\nset_difference de a1 y a2 es: ";
44     std::copy( diferencia, ptr, salida );

```

Figura 22.35 | Operaciones set de la Biblioteca estándar. (Parte I de 2).

```

45 int interseccion[ TAMANIO1 ];
46
47 // determina los elementos que están tanto en a1 como en a2
48 ptr = std::set_intersection( a1, a1 + TAMANIO1,
49     a2, a2 + TAMANIO2, interseccion );
50 cout << "\n\nset_intersection de a1 y a2 es: ";
51 std::copy( interseccion, ptr, salida );
52
53 int symmetric_difference[ TAMANIO1 + TAMANIO2 ];
54
55 // determina los elementos de a1 que no están en a2 y
56 // los elementos de a2 que no están en a1
57 ptr = std::set_symmetric_difference( a1, a1 + TAMANIO1,
58     a3, a3 + TAMANIO2, symmetric_difference );
59 cout << "\n\nset_symmetric_difference de a1 y a3 es: ";
60 std::copy( symmetric_difference, ptr, salida );
61
62 int conjuntoUnion[ TAMANIO3 ];
63
64 // determina los elementos que están en uno o ambos conjuntos
65 ptr = std::set_union( a1, a1 + TAMANIO1, a3, a3 + TAMANIO2, conjuntoUnion );
66 cout << "\n\nset_union de a1 y a3 es: ";
67 std::copy( conjuntoUnion, ptr, salida );
68 cout << endl;
69 return 0;
70
71 } // fin de main

```

```

a1 contiene: 1 2 3 4 5 6 7 8 9 10
a2 contiene: 4 5 6 7 8
a3 contiene: 4 5 6 11 15

a1 incluye a a2
a1 no incluye a a3

set_difference de a1 y a2 es: 1 2 3 9 10
set_intersection de a1 y a2 es: 4 5 6 7 8
set_symmetric_difference de a1 y a3 es: 1 2 3 7 8 9 10 11 15
set_union de a1 y a3 es: 1 2 3 4 5 6 7 8 9 10 11 15

```

Figura 22.35 | Operaciones set de la Biblioteca estándar. (Parte 2 de 2).

En las líneas 41 y 42 se utiliza la función **set_difference** para buscar los elementos del primer conjunto de valores ordenados que no se encuentren en el segundo conjunto de valores ordenados (ambos conjuntos de valores deben estar en orden ascendente). Los elementos que son diferentes se copian en el quinto argumento (en este caso, en el arreglo **diferencia**). Los primeros dos argumentos iteradores deben ser por lo menos iteradores de entrada para el primer conjunto de valores. Los siguientes dos argumentos iteradores deben ser por lo menos iteradores de entrada para el segundo conjunto de valores. El quinto argumento debe ser por lo menos un iterador de salida que indique en dónde se debe almacenar una copia de los valores que sean distintos. La función devuelve un iterador de salida que se coloca inmediatamente después del último valor copiado en el conjunto al que apunta el quinto argumento. Una segunda versión de la función **set_difference** toma un sexto argumento que viene siendo una función predicado binaria, la cual indica el orden en el que se encontraban originalmente los elementos. Las dos secuencias deben ordenarse mediante la misma función de comparación.

En las líneas 49 y 50 se utiliza la función **set_intersection** para determinar qué elementos del primer conjunto de valores ordenados se encuentran en el segundo conjunto de valores ordenados (ambos conjuntos de valores deben estar en orden ascendente). Los elementos comunes en ambos conjuntos se copian al quinto argumento (en este caso, en el arreglo **interseccion**). Los primeros dos argumentos iteradores deben ser por lo menos iteradores de entrada para el primer conjunto de valores. Los siguientes dos argumentos iteradores deben ser por lo menos iteradores de entrada para el segundo conjunto de valores. El quinto argumento debe ser por lo menos un iterador de salida que indique en dónde

se debe almacenar una copia de los valores que sean iguales. La función devuelve un iterador de salida que se coloca inmediatamente después del último valor copiado en el conjunto al que apunta el quinto argumento. Una segunda versión de la función `set_intersection` toma un sexto argumento que viene siendo una función predicado binaria, la cual indica el orden en el que se encontraban originalmente los elementos. Las dos secuencias deben ordenarse mediante la misma función de comparación.

En las líneas 58 y 59 se utiliza la función `set_symmetric_difference` para determinar qué elementos en el primer conjunto no se encuentran en el segundo, y qué elementos en el segundo conjunto no se encuentran en el primero (ambos conjuntos de valores deben estar en orden ascendente). Los elementos que sean diferentes se copian de ambos conjuntos hacia el quinto argumento (en este caso, en el arreglo `symmetric_difference`). Los primeros dos argumentos iteradores deben ser por lo menos iteradores de entrada para el primer conjunto de valores. Los siguientes dos argumentos iteradores deben ser por lo menos iteradores de entrada para el segundo conjunto de valores. El quinto argumento debe ser por lo menos un iterador de salida que indique en dónde debe almacenarse una copia de los valores que sean diferentes. La función devuelve un iterador de salida que se coloca inmediatamente después del último valor copiado en el conjunto al que apunta el quinto argumento. Una segunda versión de la función `set_symmetric_difference` toma un sexto argumento que viene siendo una función predicado binaria, la cual indica el orden en el que se encontraban originalmente los elementos. Las dos secuencias deben ordenarse mediante la misma función de comparación.

En la línea 66 se utiliza la función `set_union` para crear un conjunto de todos los elementos que se encuentran en cada uno de los dos conjuntos ordenados, o en ambos (los dos conjuntos de valores deben estar en orden ascendente). Los elementos se copian de ambos conjuntos hacia el quinto argumento (en este caso el arreglo `conjuntoUnion`). Los elementos que aparecen en ambos conjuntos sólo se copian del primer conjunto. Los primeros dos argumentos iteradores deben ser por lo menos iteradores de entrada para el primer conjunto de valores. Los siguientes dos argumentos iteradores deben ser por lo menos iteradores de entrada para el segundo conjunto de valores. El quinto argumento debe ser por lo menos un iterador de salida que indique en dónde deben almacenarse los elementos copiados. La función devuelve un iterador de salida que se coloca inmediatamente después del último valor copiado en el conjunto al que apunta el quinto argumento. Una segunda versión de la función `set_union` toma un sexto argumento que viene siendo una función predicado binaria, la cual indica el orden en el que se encontraban originalmente los elementos. Las dos secuencias deben ordenarse mediante la misma función de comparación.

22.5.11 `lower_bound`, `upper_bound` y `equal_range`

En la figura 22.36 se muestra el uso de las funciones `lower_bound`, `upper_bound` y `equal_range`. En la línea 24 se utiliza la función `lower_bound` para buscar la primera posición en una secuencia ordenada de valores en donde pueda insertarse el tercer argumento, de manera que la secuencia permanezca en orden ascendente. Los primeros dos argumentos iteradores deben ser por lo menos iteradores de avance. El tercer argumento es el valor para el que se debe determinar el límite inferior. La función devuelve un iterador de avance que apunta a la posición en la que puede realizarse la inserción. Una segunda versión de la función `lower_bound` toma como cuarto argumento una función predicado binaria, la cual indica el orden en el que se encontraban originalmente los elementos.

En la línea 30 se utiliza la función `upper_bound` para buscar la última posición en una secuencia ordenada de valores en donde pueda insertarse el tercer argumento, de manera que la secuencia permanezca en orden ascendente. Los primeros dos argumentos iteradores deben ser por lo menos iteradores de avance. El tercer argumento es el valor para el que se va a determinar el límite superior. La función devuelve un iterador de avance que apunta a la posición en la que puede realizarse la inserción. Una segunda versión de la función `upper_bound` toma como cuarto argumento una función predicado binaria, la cual indica el orden en el que se encontraban originalmente los elementos.

```

1 // Fig. 22.36: Fig22_36.cpp
2 // Las funciones lower_bound, upper_bound y equal_range
3 // de la Biblioteca estándar para una secuencia ordenada de valores.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm> // definiciones de los algoritmos
9 #include <vector> // definición de la plantilla de clase vector
10 #include <iterator> // ostream_iterator
11
12 int main()

```

Figura 22.36 | Los algoritmos `lower_bound`, `upper_bound` y `equal_range`. (Parte 1 de 3).

```

13  {
14      const int TAMANIO = 10;
15      int a1[ TAMANIO ] = { 2, 2, 4, 4, 4, 6, 6, 6, 6, 8 };
16      std::vector< int > v( a1, a1 + TAMANIO ); // copia de a1
17      std::ostream_iterator< int > salida( cout, " " );
18
19      cout << "El vector v contiene:\n";
20      std::copy( v.begin(), v.end(), salida );
21
22      // determina el punto de inserción del límite inferior para 6 en v
23      std::vector< int >::iterator inferior;
24      inferior = std::lower_bound( v.begin(), v.end(), 6 );
25      cout << "\n\nEl límite inferior de 6 es el elemento "
26          << ( inferior - v.begin() ) << " del vector v";
27
28      // determina el punto de inserción del límite superior para 6 en v
29      std::vector< int >::iterator superior;
30      superior = std::upper_bound( v.begin(), v.end(), 6 );
31      cout << "\n\nEl límite superior de 6 es el elemento "
32          << ( superior - v.begin() ) << " del vector v";
33
34      // usa equal_range para determinar los puntos de inserción
35      // inferior y superior para 6
36      std::pair< std::vector< int >::iterator,
37                  std::vector< int >::iterator > eq;
38      eq = std::equal_range( v.begin(), v.end(), 6 );
39      cout << "\nUsando equal_range:\n    El límite inferior de 6 es el elemento "
40          << ( eq.first - v.begin() ) << " del vector v";
41      cout << "\n    El límite superior de 6 es el elemento "
42          << ( eq.second - v.begin() ) << " del vector v";
43      cout << "\n\nUsa lower_bound para localizar el primer punto\n"
44          << "en el que se puede insertar el 5 en orden";
45
46      // determine inferior-bound insertion point for 5 in v
47      inferior = std::lower_bound( v.begin(), v.end(), 5 );
48      cout << "\n    El límite inferior de 5 es el elemento "
49          << ( inferior - v.begin() ) << " del vector v";
50      cout << "\n\nUsa upper_bound para localizar el ultimo punto\n"
51          << "en el que se puede insertar el 7 en orden";
52
53      // determina el punto de inserción del límite superior para 7 en v
54      superior = std::upper_bound( v.begin(), v.end(), 7 );
55      cout << "\n    El límite superior de 7 es el elemento "
56          << ( superior - v.begin() ) << " del vector v";
57      cout << "\n\nUsa equal_range para localizar el primer y\n"
58          << "último punto en el que se puede insertar el 5 en orden";
59
60      // usa equal_range para determinar los puntos de inserción
61      // inferior y superior para el 5
62      eq = std::equal_range( v.begin(), v.end(), 5 );
63      cout << "\n    El límite inferior de 5 es el elemento "
64          << ( eq.first - v.begin() ) << " del vector v";
65      cout << "\n    El límite superior de 5 es el elemento "
66          << ( eq.second - v.begin() ) << " del vector v" << endl;
67
68  } // fin de main

```

El vector v contiene:
2 2 4 4 4 6 6 6 6 8

El límite inferior de 6 es el elemento 5 del vector v
El límite superior de 6 es el elemento 9 del vector v

Figura 22.36 | Los algoritmos lower_bound, upper_bound y equal_range. (Parte 2 de 3).

```

Usando equal_range:
  El límite inferior de 6 es el elemento 5 del vector v
  El límite superior de 6 es el elemento 9 del vector v

Usa lower_bound para localizar el primer punto
en el que se puede insertar el 5 en orden
  El límite inferior de 5 es el elemento 5 del vector v

Usa upper_bound para localizar el último punto
en el que se puede insertar el 7 en orden
  El límite superior de 7 es el elemento 9 del vector v

Usa equal_range para localizar el primer y
último punto en el que se puede insertar el 5 en orden
  El límite inferior de 5 es el elemento 5 del vector v
  El límite superior de 5 es el elemento 5 del vector v

```

Figura 22.36 | Los algoritmos `lower_bound`, `upper_bound` y `equal_range`. (Parte 3 de 3).

En la línea 38 se utiliza la función `equal_range` para devolver un par (un objeto `pair`) de iteradores de avance que contienen los resultados combinados de llevar a cabo las operaciones `lower_bound` y `upper_bound`. Los primeros dos argumentos iteradores deben ser por lo menos iteradores de avance. El tercer argumento es el valor para el que se va a localizar el rango equivalente. La función devuelve un par de iteradores de avance para los límites inferior (`eq.first`) y superior (`eq.second`), respectivamente.

Las funciones `lower_bound`, `upper_bound` e `equal_range` se utilizan a menudo para localizar puntos de inserción en secuencias ordenadas. En la línea 47 se utiliza `lower_bound` para localizar el primer punto en el que puede insertarse un 5 en orden, en el vector `v`. En la línea 54 se utiliza `upper_bound` para localizar el último punto en el que puede insertarse un 7 en orden, en el vector `v`. En la línea 62 se utiliza `equal_range` para localizar el primer y último puntos en los que puede insertarse un 5 en orden, en el vector `v`.

22.5.12 Ordenamiento de montón (heapsort)

En la figura 22.37 se demuestra el uso de las funciones de la Biblioteca estándar para llevar a cabo el **algoritmo de ordenamiento heapsort**. Heapsort es un algoritmo mediante el cual se ordena un arreglo de elementos en un árbol binario especial conocido como **montón** (**heap**). Las características clave de un montón son que el elemento más grande siempre se encuentra en su parte superior y los valores de los hijos de cualquier nodo en el árbol binario son siempre menores o iguales que el valor de ese nodo. Un montón ordenado de esta manera se conoce comúnmente como **maxheap**. El algoritmo heapsort se describe con detalle en los cursos de ciencias computacionales llamados “Estructuras de datos” y “Algoritmos”.

```

1 // Fig. 22.37: Fig22_37.cpp
2 // Los algoritmos push_heap, pop_heap, make_heap
3 // y sort_heap de la Biblioteca estándar.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm>
9 #include <vector>
10 #include <iterator>
11
12 int main()
13 {
14     const int TAMANIO = 10;
15     int a[ TAMANIO ] = { 3, 100, 52, 77, 22, 31, 1, 98, 13, 40 };
16     std::vector< int > v( a, a + TAMANIO ); // copia de a
17     std::vector< int > v2;

```

Figura 22.37 | Uso de las funciones de la Biblioteca estándar para realizar un algoritmo heapsort. (Parte 1 de 3).

```

18     std::ostream_iterator< int > salida( cout, " " );
19
20     cout << "El vector v antes de make_heap:\n";
21     std::copy( v.begin(), v.end(), salida );
22
23     std::make_heap( v.begin(), v.end() ); // crea un montón en base al vector v
24     cout << "\nEl vector v después de make_heap:\n";
25     std::copy( v.begin(), v.end(), salida );
26
27     std::sort_heap( v.begin(), v.end() ); // ordena los elementos con sort_heap
28     cout << "\nEl vector v después de sort_heap:\n";
29     std::copy( v.begin(), v.end(), salida );
30
31     // realiza el algoritmo heapsort con push_heap y pop_heap
32     cout << "\n\nEl arreglo a contiene: ";
33     std::copy( a, a + TAMANIO, salida ); // muestra el arreglo a
34     cout << endl;
35
36     // coloca los elementos del arreglo a en v2 y
37     // mantiene los elementos de v2 en el montón
38     for ( int i = 0; i < TAMANIO; i++ )
39     {
40         v2.push_back( a[ i ] );
41         std::push_heap( v2.begin(), v2.end() );
42         cout << "\nv2 después de push_heap(a[" << i << "]): ";
43         std::copy( v2.begin(), v2.end(), salida );
44     } // fin de for
45
46     cout << endl;
47
48     // elimina los elementos del montón en orden
49     for ( unsigned int j = 0; j < v2.size(); j++ )
50     {
51         cout << "\nv2 después de sacar " << v2[ 0 ] << " del montón\n";
52         std::pop_heap( v2.begin(), v2.end() - j );
53         std::copy( v2.begin(), v2.end(), salida );
54     } // fin de for
55
56     cout << endl;
57     return 0;
58 } // fin de main

```

```

El vector v antes de make_heap:
3 100 52 77 22 31 1 98 13 40
El vector v después de make_heap:
100 98 52 77 40 31 1 3 13 22
El vector v después de sort_heap:
1 3 13 22 31 40 52 77 98 100

El arreglo a contiene: 3 100 52 77 22 31 1 98 13 40
v2 después de push_heap(a[0]): 3
v2 después de push_heap(a[1]): 100 3
v2 después de push_heap(a[2]): 100 3 52
v2 después de push_heap(a[3]): 100 77 52 3
v2 después de push_heap(a[4]): 100 77 52 3 22
v2 después de push_heap(a[5]): 100 77 52 3 22 31
v2 después de push_heap(a[6]): 100 77 52 3 22 31 1
v2 después de push_heap(a[7]): 100 98 52 77 22 31 1 3
v2 después de push_heap(a[8]): 100 98 52 77 22 31 1 3 13
v2 después de push_heap(a[9]): 100 98 52 77 40 31 1 3 13 22
v2 después de sacar 100 del montón
98 77 52 22 40 31 1 3 13 100

```

Figura 22.37 | Uso de las funciones de la Biblioteca estándar para realizar un algoritmo heapsort. (Parte 2 de 3).

```
v2 despues de sacar 98 del monton
77 40 52 22 13 31 1 3 98 100
v2 despues de sacar 77 del monton
52 40 31 22 13 3 1 77 98 100
v2 despues de sacar 52 del monton
40 22 31 1 13 3 52 77 98 100
v2 despues de sacar 40 del monton
31 22 3 1 13 40 52 77 98 100
v2 despues de sacar 31 del monton
22 13 3 1 31 40 52 77 98 100
v2 despues de sacar 22 del monton
13 1 3 22 31 40 52 77 98 100
v2 despues de sacar 13 del monton
3 1 13 22 31 40 52 77 98 100
v2 despues de sacar 3 del monton
1 3 13 22 31 40 52 77 98 100
v2 despues de sacar 1 del monton
1 3 13 22 31 40 52 77 98 100
```

Figura 22.37 | Uso de las funciones de la Biblioteca estándar para realizar un algoritmo heapsort. (Parte 3 de 3).

En la línea 23 se utiliza la función `make_heap` para tomar una secuencia de valores en el rango que empieza desde `v.begin()` hasta, pero sin incluir `a`, `v.end()`, y crear un montón que pueda utilizarse para producir una secuencia ordenada. Los dos iteradores que se toman como argumentos deben ser de acceso aleatorio, por lo que esta función sólo trabaja con arreglos, vectores y contenedores `deque`. Una segunda versión de esta función toma como argumento a una función predicado binaria para comparar valores.

En la línea 27 se utiliza la función `sort_heap` para ordenar una secuencia de valores en el rango que empieza desde `v.begin()` hasta, pero sin incluir `a`, `v.end()`, que ya se encuentran ordenados en un montón. Los dos iteradores que se toman como argumentos deben ser iteradores de acceso aleatorio. Una segunda versión de esta función toma como tercer argumento a una función predicado binaria para comparar valores.

En la línea 41 se utiliza la función `push_heap` para agregar un nuevo valor a un montón. Tomamos un elemento del arreglo `a` a la vez, anexando ese elemento al final del vector `v2` y realizamos la operación `push_heap`. Si el elemento anexado es el único en el vector, éste es ya un montón. De no ser así, la función `push_heap` reordena los elementos del vector en un montón. Cada vez que se llama a `push_heap`, se asume que el último elemento actualmente en el vector (es decir, el que se anexa antes de la llamada a la función `push_heap`) es el que se va a agregar al montón, y que todos los demás elementos en el vector ya se encuentran ordenados como un montón. Los dos argumentos iteradores para `push_heap` deben ser iteradores de acceso aleatorio. Una segunda versión de esta función toma como tercer argumento a una función predicado binaria para comparar valores.

En la línea 52 se utiliza `pop_heap` para eliminar el elemento de la parte superior del montón. Esta función asume que los elementos en el rango especificado por sus dos argumentos iteradores de acceso aleatorio ya son un montón. El proceso de eliminar en forma repetida el elemento superior del montón produce una secuencia ordenada de valores. La función `pop_heap` intercambia el primer elemento del montón (`v2.begin()` en este ejemplo) con su último elemento (el elemento antes de `v2.end() - i`) y luego se asegura que los elementos hasta (pero sin incluir) el último elemento sigan formando un montón. Observe en la salida que, después de las operaciones con `pop_heap`, el vector está ordenado en forma ascendente. Una segunda versión de esta función toma como tercer argumento a una función predicado binaria para comparar valores.

22.5.13 min y max

Los algoritmos `min` y `max` determinan los valores mínimo y máximo de dos elementos, respectivamente. En la figura 22.38 se demuestra el uso de `min` y `max` con valores `int` y `char`.

```
1 // Fig. 22.38: Fig22_38.cpp
2 // Los algoritmos min y max de la Biblioteca estándar.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
```

Figura 22.38 | Los algoritmos `min` y `max`. (Parte 1 de 2).

```

6
7 #include <algorithm>
8
9 int main()
10 {
11     cout << "El minimo de 12 y 7 es: " << std::min( 12, 7 );
12     cout << "\nEl maximo de 12 y 7 es: " << std::max( 12, 7 );
13     cout << "\nEl minimo de 'G' y 'Z' es: " << std::min( 'G', 'Z' );
14     cout << "\nEl maximo de 'G' y 'Z' es: " << std::max( 'G', 'Z' );
15     cout << endl;
16     return 0;
17 } // fin de main

```

```

El minimo de 12 y 7 es: 7
El maximo de 12 y 7 es: 12
El minimo de 'G' y 'Z' es: G
El maximo de 'G' y 'Z' es: Z

```

Figura 22.38 | Los algoritmos `min` y `max`. (Parte 2 de 2).

22.5.14 Algoritmos de la STL que no se cubren en este capítulo

En la figura 22.39 se muestran los algoritmos que no se cubren en este capítulo.

Algoritmo	Descripción
<code>inner_product</code>	Calcular la suma de los productos de dos secuencias, tomando los elementos correspondientes en cada secuencia, multiplicándolos y agregando el resultado a un total.
<code>adjacent_difference</code>	Comenzando con el segundo elemento en una secuencia, calcular la diferencia (utilizando el operador <code>-</code>) entre el elemento actual y el anterior, y guardar el resultado. Los primeros dos argumentos, que deben ser iteradores de entrada, indican el rango de elementos en el contenedor y el tercer argumento, que debe ser un iterador de salida, indica en dónde deben almacenarse los resultados. Una segunda versión de este algoritmo toma como cuarto argumento a una función binaria, la cual realiza un cálculo entre el elemento actual y el anterior.
<code>partial_sum</code>	Calcular el total actual (utilizando el operador <code>+</code>) de los valores en una secuencia. Los primeros dos argumentos, que deben ser iteradores de entrada, indican el rango de elementos en el contenedor y el tercer argumento, que debe ser un iterador de salida, indica en dónde deben almacenarse los resultados. Una segunda versión de este algoritmo toma como cuarto argumento a una función binaria que realiza un cálculo entre el valor actual en la secuencia y el total actual.
<code>nth_element</code>	Utiliza tres iteradores de acceso aleatorio para particionar un rango de elementos. Los argumentos primero y tercero representan el rango de elementos. El segundo argumento indica la posición del elemento particionador. Después de la ejecución de este algoritmo, todos los elementos a la izquierda del elemento particionador son menores que ese elemento, y todos los elementos a la derecha del elemento particionador son mayores o iguales que ese elemento. Una segunda versión de este algoritmo toma como cuarto argumento a una función binaria de comparación.
<code>partition</code>	Este algoritmo es similar a <code>nth_element</code> , pero requiere de iteradores bidireccionales menos poderosos, lo que lo hace más flexible que <code>nth_element</code> . El algoritmo <code>partition</code> requiere de dos iteradores bidireccionales que indican el rango de elementos a partitionar. El tercer elemento es una función predicado unaria, la cual ayuda a partitionar los elementos de manera que todos los que estén en la secuencia para la que la función predicado sea <code>true</code> se encuentren a la izquierda (hacia el inicio de la secuencia) de todos los elementos para los que la función predicado sea <code>false</code> . Se devuelve un iterador bidireccional, el cual indica el primer elemento en la secuencia para el que la función predicado devuelve <code>false</code> .
<code>stable_partition</code>	Este algoritmo es similar a <code>partition</code> , excepto que garantiza que los elementos equivalentes se mantendrán en su orden original.
<code>next_permutation</code>	La siguiente permutación lexicográfica de una secuencia.

Figura 22.39 | Los algoritmos que no se cubren en este capítulo. (Parte 1 de 2).

Algoritmo	Descripción
<code>prev_permutation</code>	La anterior permutación lexicográfica de una secuencia.
<code>rotate</code>	Utiliza tres iteradores de avance como argumentos para rotar la secuencia indicada por el primer y último argumentos, según el número de posiciones indicadas al restar el primer argumento del segundo. Por ejemplo, la secuencia 1, 2, 3, 4, 5 rotada en dos posiciones sería 4, 5, 1, 2, 3.
<code>rotate_copy</code>	Este algoritmo es idéntico a <code>rotate</code> , excepto que los resultados se almacenan en una secuencia separada, indicada por el cuarto argumento: un iterador de salida. Las dos secuencias deben tener el mismo número de elementos.
<code>adjacent_find</code>	Este algoritmo devuelve un iterador de entrada que indica el primero de dos elementos adyacentes idénticos en una secuencia. Si no hay elementos adyacentes idénticos, el iterador se coloca al final (<code>end</code>) de la secuencia.
<code>search</code>	Este algoritmo busca una subsecuencia de elementos dentro de una secuencia de elementos <code>y</code> , si se encuentra dicha subsecuencia, devuelve un iterador de avance que indica el primer elemento de esa subsecuencia. Si no hay coincidencias, el iterador se coloca al final de la secuencia en la que se va a realizar la búsqueda.
<code>search_n</code>	Este algoritmo busca en una secuencia de elementos una subsecuencia en la que los valores de un número especificado de elementos tienen un valor específico <code>y</code> , si se encuentra dicha subsecuencia, devuelve un iterador de avance que indica el primer elemento de esa subsecuencia. Si no hay coincidencias, el iterador se posiciona al final de la secuencia en la que se va a realizar la búsqueda.
<code>partial_sort</code>	Usa tres iteradores de acceso aleatorio como argumentos para ordenar parte de una secuencia. El primer y último argumentos indican la secuencia de elementos. El segundo argumento indica la posición final para la parte ordenada de la secuencia. De manera predeterminada, los elementos se ordenan utilizando el operador <code><</code> (también puede suministrarse una función predicado binaria). Los elementos a partir del segundo argumento iterador hasta el final de la secuencia se encuentran en un orden indefinido.
<code>partial_sort_copy</code>	Usa dos iteradores de entrada y dos iteradores de acceso aleatorio para ordenar parte de la secuencia indicada por los dos iteradores de entrada que sirven de argumentos. Los resultados se almacenan en la secuencia indicada por los dos argumentos iteradores de acceso aleatorio. Los elementos se ordenan de manera predeterminada mediante el operador <code><</code> (también puede suministrarse una función predicado binaria). El número de elementos ordenados es el que sea menor del número de elementos en el resultado y el número de elementos en la secuencia original.
<code>stable_sort</code>	El algoritmo es similar a <code>sort</code> , excepto que todos los elementos equivalentes se mantienen en su orden original. Este orden es $O(n \log n)$ si hay suficiente memoria disponible; en caso contrario, es $O(n(\log n)^2)$.

Figura 22.39 | Los algoritmos que no se cubren en este capítulo. (Parte 2 de 2).

22.6 La clase bitset

La clase `bitset` facilita la creación y manipulación de **conjuntos de bits**, los cuales son útiles para representar un conjunto de banderas de bits. Los objetos `bitset` tienen un tamaño fijo en tiempo de compilación. La clase `bitset` es una herramienta alternativa para la manipulación de bits, que vimos en el capítulo 21. La declaración

```
bitset< tamaño > b;
```

crea el objeto `bitset` `b`, en el que todos los bits son inicialmente 0. La instrucción

```
b.set( numeroBit );
```

hace que el bit `numeroBit` del objeto `bitset` `b` se “encienda”. La expresión `b.set()` hace que todos los bits en `b` se “enciendan”.

La instrucción

```
b.reset( numeroBit );
```

hace que el bit `numeroBit` del objeto `bitset` `b` se “apague”. La expresión `b.reset()` hace que todos los bits en `b` se “apaguen”. La instrucción

```
b.flip( numeroBit );
```

“voltea” el bit `numeroBit` del objeto `bitset` `b` (es decir, si el bit está encendido, `flip` lo apaga y viceversa). La expresión `b.flip()` voltea todos los bits en `b`. La instrucción

```
b[ numeroBit ];
```

devuelve una referencia al bit `numeroBit` del objeto `bitset` `b`. De manera similar,

```
b.at( numeroBit );
```

realiza primero una comprobación de rango en `numeroBit`. Después, si `numeroBit` se encuentra dentro del rango, `at` devuelve una referencia a ese bit. En caso contrario, `at` lanza una excepción `out_of_range`. La instrucción

```
b.test( numeroBit );
```

realiza primero una comprobación de rango en `numeroBit`. Después, si `numeroBit` se encuentra dentro del rango, `test` devuelve `true` si el bit está encendido y `false` si está apagado. En caso contrario, `test` lanza una excepción `out_of_range`. La expresión

```
b.size()
```

devuelve el número de bits en el objeto `bitset` `b`. La expresión

```
b.count()
```

devuelve el número de bits encendidos en el objeto `bitset` `b`. La expresión

```
b.any()
```

devuelve `true` si alguno de los bits en el objeto `bitset` `b` está encendido. La expresión

```
b.none()
```

devuelve `true` si ninguno de los bits en el objeto `bitset` `b` está encendido. Las expresiones

```
b == b1  
b != b1
```

comparan los dos objetos `bitset` para ver si son iguales y desiguales, respectivamente.

Cada uno de los operadores de asignación a nivel de bits `&=`, `|=` y `^=` puede utilizarse para combinar objetos `bitset`. Por ejemplo,

```
b &= b1;
```

realiza una operación AND lógica bit por bit, entre los objetos `bitset` `b` y `b1`. El resultado se almacena en `b`. Las operaciones OR y XOR lógicas a nivel de bits se realizan mediante

```
b |= b1;  
b ^= b2;
```

La expresión

```
b >>= n;
```

desplaza los bits en el objeto `bitset` `b` a la derecha, `n` posiciones. La expresión

```
b <<= n;
```

desplaza los bits en el objeto `bitset` `b` a la izquierda, `n` posiciones. Las expresiones

```
b.to_string()  
b.to_ulong()
```

convierten el objeto `bitset` `b` en un objeto `string` y en un `unsigned long`, respectivamente.

La criba de Eratóstenes con `bitset`

La figura 22.40 muestra otra vez el método de la Criba de Eratóstenes para buscar números primos que vimos en el ejercicio 7.29. Se utiliza un objeto `bitset` en lugar de un arreglo para implementar el algoritmo. El programa muestra todos los números primos desde el 2 hasta el 1023 y después permite que el usuario escriba un número para determinar si es o no primo.

En la línea 20 se crea un objeto `bitset` con un número de bits especificado por la variable `TAMANIO` (1024 en este ejemplo). De manera predeterminada, todos los bits en el objeto `bitset` están “apagados”. En la línea 21 se hace una llamada a la función `flip` para “encender” todos los bits. Los números 0 y 1 no son números primos, por lo que en las líneas 22 y 23 se hace una llamada a la función `reset` para “apagar” los bits 0 y 1. En las líneas 29 a 36 se determinan

todos los números primos que haya desde el 2 hasta el 1023. El entero `bitFinal` (línea 26) se utiliza para determinar cuándo se completa el algoritmo. El algoritmo básico es que un número es primo si no tiene divisores aparte del 1 y de sí mismo. Empezando con el número 2, una vez que sabemos que un número es primo, podemos eliminar todos los múltiplos de ese número. El número 2 es divisible sólo por 1 y por sí mismo, entonces es primo. Por lo tanto, podemos eliminar los números 4, 6, 8 y así sucesivamente. El número 3 es divisible sólo por 1 y por sí mismo. Por lo tanto, podemos eliminar todos los múltiplos de 3 (tenga en cuenta que todos los números pares ya han sido eliminados).

```

1 // Fig. 22.40: Fig22_40.cpp
2 // Uso de bitset para demostrar la Criba de Eratóstenes.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 #include <cmath>
12 using std::sqrt; // prototipo de sqrt
13
14 #include <bitset> // definición de la clase bitset
15
16 int main()
17 {
18     const int TAMANIO = 1024;
19     int valor;
20     std::bitset< TAMANIO > criba; // crea bitset de 1024 bits
21     criba.flip(); // cambia todos los bits en el objeto bitset criba
22     criba.reset( 0 ); // restablece el primer bit (número 0)
23     criba.reset( 1 ); // restablece el segundo bit (número 1)

24
25     // realiza la Criba de Eratóstenes
26     int bitFinal = sqrt( static_cast< double >( criba.size() ) ) + 1;
27
28     // determina todos los números primos del 2 al 1024
29     for ( int i = 2; i < bitFinal; i++ )
30     {
31         if ( criba.test( i ) ) // el bit i está encendido
32         {
33             for ( int j = 2 * i; j < TAMANIO; j += i )
34                 criba.reset( j ); // apaga el bit j
35         } // fin de if
36     } // fin de for

37
38     cout << "Los numeros primos en el rango de 2 a 1023 son:\n";
39
40     // muestra los números primos en el rango de 2 a 1023
41     for ( int k = 2, contador = 1; k < TAMANIO; k++ )
42     {
43         if ( criba.test( k ) ) // el bit k está encendido
44         {
45             cout << setw( 5 ) << k;
46
47             if ( contador++ % 12 == 0 ) // contador es un múltiplo de 12
48                 cout << '\n';
49         } // fin de if
50     } // fin de for

51
52     cout << endl;

```

Figura 22.40 | La clase `bitset` y la Criba de Eratóstenes. (Parte I de 2).

```

53
54 // obtiene el valor del usuario
55 cout << "\nEscriba un valor de 2 a 1023 (-1 para terminar): ";
56 cin >> valor;
57
58 // determina si la entrada del usuario es un número primo
59 while ( valor != -1 )
60 {
61     if ( criba[ valor ] ) // número primo
62         cout << valor << " es un numero primo\n";
63     else // no es un número primo
64         cout << valor << " no es un numero primo\n";
65
66     cout << "\nEscriba un valor de 2 a 1023 (-1 para terminar): ";
67     cin >> valor;
68 } // fin de while
69
70 return 0;
71 } // fin de main

```

Los números primos en el rango de 2 a 1023 son:

2	3	5	7	11	13	17	19	23	29	31	37
41	43	47	53	59	61	67	71	73	79	83	89
97	101	103	107	109	113	127	131	137	139	149	151
157	163	167	173	179	181	191	193	197	199	211	223
227	229	233	239	241	251	257	263	269	271	277	281
283	293	307	311	313	317	331	337	347	349	353	359
367	373	379	383	389	397	401	409	419	421	431	433
439	443	449	457	461	463	467	479	487	491	499	503
509	521	523	541	547	557	563	569	571	577	587	593
599	601	607	613	617	619	631	641	643	647	653	659
661	673	677	683	691	701	709	719	727	733	739	743
751	757	761	769	773	787	797	809	811	821	823	827
829	839	853	857	859	863	877	881	883	887	907	911
919	929	937	941	947	953	967	971	977	983	991	997
1009	1013	1019	1021								

Escriba un valor de 2 a 1023 (-1 para terminar): 389

389 es un numero primo

Escriba un valor de 2 a 1023 (-1 para terminar): 88

88 no es un numero primo

Escriba un valor de 2 a 1023 (-1 para terminar): -1

Figura 22.40 | La clase `bitset` y la Criba de Eratóstenes. (Parte 2 de 2).

22.7 Objetos de funciones

Muchos algoritmos de la STL no permiten pasar un apuntador a una función al algoritmo, para ayudarlo a que lleve a cabo su tarea. Por ejemplo, el algoritmo `binary_search` que vimos en la sección 22.5.6 está sobre cargado con una versión que requiere como su cuarto parámetro un apuntador a una función que recibe dos argumentos y devuelve un valor `bool`. El algoritmo `binary_search` utiliza esta función para comparar la clave de búsqueda con un elemento en la colección. La función devuelve `true` si la clave de búsqueda y el elemento que se van a comparar son iguales; en caso contrario, la función devuelve `false`. Esto permite a `binary_search` buscar en una colección de elementos para la cual el tipo del elemento no proporcione un operador de igualdad `==` sobre cargado.

Los diseñadores de la STL hicieron los algoritmos más flexibles al permitir que cualquier argumento que pueda recibir un apuntador a una función reciba un objeto de una clase que sobre cargue al operador paréntesis con una función llamada `operator()`, siempre y cuando el operador sobre cargado cumpla con los requerimientos del algoritmo; en el caso de `binary_search`, debe recibir dos argumentos y devolver un valor `bool`. Un objeto de dicha clase se conoce como **objeto de función**, y se puede usar en forma sintáctica y semántica como una función o un apuntador a una función; el operador paréntesis sobre cargado se invoca mediante el uso del nombre de un objeto de función, seguido por paréntesis

que contienen los argumentos para la función. En conjunto, los objetos de función y las funciones utilizadas se conocen como **functores**. La mayoría de los algoritmos pueden utilizar objetos de función y funciones de igual forma.

Los objetos de función ofrecen varias ventajas en comparación con los apuntadores a funciones. Como los objetos de función se implementan comúnmente como plantillas de clases que se incluyen en cada archivo de código fuente que las utiliza, el compilador puede poner en línea un operador `operator()` sobrecargado para mejorar el rendimiento. Además, como son objetos de clases, los objetos de función pueden tener miembros de datos que `operator()` puede utilizar para realizar su tarea.

Objetos función predefinidos de la Biblioteca de plantillas estándar

Muchos objetos de función predefinidos se pueden encontrar en el encabezado `<functional>`. En la figura 22.41 se enlistan varios de los objetos de función de la STL, todos los cuales se implementan como plantillas de clases. Utilizamos el objeto de función `less< T >` en los ejemplos con `set`, `multiset` y `priority_queue`, para especificar el orden de los elementos en un contenedor.

Uso del algoritmo accumulate de la STL

En la figura 22.42 se demuestra el uso del algoritmo numérico `accumulate` (descrito en la figura 22.30) para calcular la suma de los cuadrados de los elementos en un `vector`. El cuarto argumento para `accumulate` es un **objeto de función binaria** (es decir, un objeto de función para el que `operator()` recibe dos argumentos) o un apuntador a una **función binaria** (es decir, una función que recibe dos argumentos). La función `accumulate` se demuestra dos veces: una vez con un apuntador a una función y la otra con un objeto de función.

Objetos de función de la STL	Tipo	Objetos de función de la STL	Tipo
<code>divides< T ></code>	aritmético	<code>logical_or< T ></code>	lógico
<code>equal_to< T ></code>	relacional	<code>minus< T ></code>	aritmético
<code>greater< T ></code>	relacional	<code>modulus< T ></code>	aritmético
<code>greater_equal< T ></code>	relacional	<code>negate< T ></code>	aritmético
<code>less< T ></code>	relacional	<code>not_equal_to< T ></code>	relacional
<code>less_equal< T ></code>	relacional	<code>plus< T ></code>	aritmético
<code>logical_and< T ></code>	lógico	<code>multiplies< T ></code>	aritmético
<code>logical_not< T ></code>	lógico		

Figura 22.41 Objetos de función de la Biblioteca estándar.

```

1 // Fig. 22.42: Fig22_42.cpp
2 // Demostración de los objetos de función.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <vector> // definición de la plantilla de clase vector
8 #include <algorithm> // algoritmo de copia
9 #include <numeric> // algoritmo accumulate
10 #include <functional> // definición de binary_function
11 #include <iterator> // ostream_iterator
12
13 // función binaria que suma el cuadrado de su segundo argumento y el
14 // total actual en su primer argumento, y después devuelve la suma
15 int sumarCuadrados( int total, int valor )
16 {
17     return total + valor * valor;
18 } // fin de la función sumarCuadrados
19

```

Figura 22.42 | Objeto de función binaria. (Parte I de 2).

```

20 // plantilla de clases de función binaria que define el operator() sobrecargado
21 // que suma el cuadrado de su segundo argumento y el total actual
22 // en su primer argumento, y después devuelve la suma
23 template< typename T >
24 class claseSumarCuadrados : public std::binary_function< T, T, T >
25 {
26 public:
27     // suma el cuadrado de valor al total y devuelve el resultado
28     T operator()( const T &total, const T &valor )
29     {
30         return total + valor * valor;
31     } // fin de la función operator()
32 }; // fin de la clase claseSumarCuadrados
33
34 int main()
35 {
36     const int TAMANIO = 10;
37     int arreglo[ TAMANIO ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
38     std::vector< int > enteros( arreglo, arreglo + TAMANIO ); // copia de arreglo
39     std::ostream_iterator< int > salida( cout, " " );
40     int resultado;
41
42     cout << "el vector enteros contiene:\n";
43     std::copy( enteros.begin(), enteros.end(), salida );
44
45     // calcula la suma de los cuadrados de los elementos del vector
46     // enteros usando la función binaria sumarCuadrados
47     resultado = std::accumulate( enteros.begin(), enteros.end(),
48         0, sumarCuadrados );
49
50     cout << "\n\nSuma de los cuadrados de los elementos en enteros usando "
51         << "la función binaria sumarCuadrados: " << resultado;
52
53     // calcula la suma de los cuadrados de los elementos del vector
54     // enteros usando el objeto de función binaria
55     resultado = std::accumulate( enteros.begin(), enteros.end(),
56         0, claseSumarCuadrados< int >() );
57
58     cout << "\n\nSuma de los cuadrados de los elementos en enteros usando "
59         << "el objeto\nde función binaria de tipo "
60         << "claseSumarCuadrados< int >: " << resultado << endl;
61     return 0;
62 } // fin de main

```

```

el vector enteros contiene:
1 2 3 4 5 6 7 8 9 10

Suma de los cuadrados de los elementos en enteros usando la función
binaria sumarCuadrados: 385

Suma de los cuadrados de los elementos en enteros usando el objeto
de función binaria de tipo claseSumarCuadrados< int >: 385

```

Figura 22.42 | Objeto de función binaria. (Parte 2 de 2).

En las líneas 15 a 18 se define la función `sumarCuadrados`, que calcula el cuadrado de su segundo argumento llamado `valor`, suma ese cuadrado con su primer argumento `total` y devuelve la suma. La función `accumulate` pasará como el segundo argumento para `sumarCuadrados` a cada uno de los elementos de la secuencia sobre la cual va a iterar en el ejemplo. En la primera llamada a `sumarCuadrados`, el primer argumento será el valor inicial del `total` (que se suministra como el tercer argumento para `accumulate`: 0 en este programa). Todas las llamadas subsecuentes a `sumarCuadrados` reciben como primer argumento la suma actual devuelta por la llamada anterior a `sumarCuadrados`. Cuando `accumulate` termina, devuelve la suma de los cuadrados de todos los elementos en la secuencia.

En las líneas 23 a 32 se define una clase llamada `ClaseSumarCuadrados` que hereda de la clase `binary_function` (en el archivo de encabezado `<functional>`): una clase base vacía para crear objetos de función en los que `operator` recibe dos parámetros y devuelve un valor. La clase `binary_function` acepta tres parámetros de tipo que representan los tipos del primer argumento, el segundo argumento y el valor de retorno de `operator`, respectivamente. En este ejemplo, el tipo de esos parámetros es `T` (línea 24). En la primera llamada al objeto de función, el primer argumento será el valor inicial del `total` (que se suministra como el tercer argumento para `accumulate`: 0 en este programa) y el segundo argumento será el primer elemento en el vector `enteros`. Todas las llamadas subsiguientes a `operator` reciben como primer argumento el resultado devuelto por la llamada anterior al objeto de función, y el segundo argumento será el siguiente elemento en el vector. Cuando `accumulate` termina de ejecutarse, devuelve la suma de los cuadrados de todos los elementos en el vector.

En las líneas 47 y 48 se hace una llamada a la función `accumulate` con un apuntador a la función `sumarCuadrados` como su último argumento.

La instrucción en las líneas 55 y 56 llama a la función `accumulate` con un objeto de la clase `ClaseSumarCuadrados` como el último argumento. La expresión `ClaseSumarCuadrados< int >()` crea una instancia de la clase `ClaseSumarCuadrados` (un objeto de función) que se pasa a `accumulate`, la cual envía al objeto el mensaje (invoca a la función) `operator`. La instrucción podría haberse escrito como dos instrucciones separadas, como se muestra a continuación:

```
ClaseSumarCuadrados< int > objetoSumarCuadrados;
resultado = std::accumulate( enteros.begin(), enteros.end(),
    0, objetoSumarCuadrados );
```

En la primera línea se define un objeto de la clase `ClaseSumarCuadrados`. Ese objeto se pasa a continuación a la función `accumulate`.

22.8 Repaso

En este capítulo presentamos la Biblioteca de plantillas estándar y hablamos sobre sus tres componentes clave: contenedores, iteradores y algoritmos. Usted aprendió acerca de los contenedores de secuencia de la STL `vector`, `deque` y `list`, que representan estructuras de datos lineales. Hablamos sobre los contenedores asociativos `set`, `multiset`, `map` y `multimap`, que representan estructuras de datos no lineales. También vimos que los adaptadores de contenedores `stack`, `queue` y `priority_queue` pueden utilizarse para restringir las operaciones de los contenedores de secuencia para el propósito de implementar las estructuras de datos especializadas representadas por los adaptadores de contenedores. Despues demostramos muchos de los algoritmos de la STL, incluyendo los algoritmos matemáticos, los algoritmos básicos de búsqueda y ordenamiento, y las operaciones de conjuntos. Aprendió acerca de los tipos de iteradores que requiere cada algoritmo, y que cada algoritmo se puede utilizar con cualquier contenedor que soporte la mínima funcionalidad de iteradores que requiere el algoritmo. Aprendió además acerca de la clase `bitset`, que facilita la creación y manipulación de conjuntos de bits como un contenedor. Por último presentamos los objetos de función que trabajan en forma sintáctica y semántica como las funciones ordinarias, pero ofrecen ventajas tales como el rendimiento y la habilidad de almacenar datos.

En el siguiente capítulo hablaremos acerca de características más avanzadas de C++, incluyendo los operadores de conversión de tipos, los espacios de nombres, las palabras clave de operadores, los operadores de apuntador a miembro de clase, la herencia múltiple y las clases base `virtual`.

22.9 Recursos Web de la STL

Nuestro Centro de recursos de C++ (www.deitel.com/cplusplus/) se enfoca en la enorme cantidad de contenido gratuito sobre C++ disponible en línea. Empiece aquí su búsqueda de recursos, descargas, tutoriales, documentación, libros, libros electrónicos (e-books), diarios, artículos, blogs, transmisiones RSS (RSS feeds) y demás información que le ayudará a desarrollar aplicaciones en C++. El Centro de recursos de C++ incluye vínculos a muchos recursos y tutoriales de la STL.

Resumen

Sección 22.1 Introducción a la Biblioteca de plantillas estándar (STL)

- La Biblioteca de plantillas estándar define componentes reutilizables poderosos y basados en plantillas, que implementan muchas estructuras de datos comunes, además de algoritmos que se utilizan para procesar esas estructuras de datos.
- La STL tiene tres componentes clave: contenedores, iteradores y algoritmos.

- Los contenedores de la STL son estructuras de datos capaces de almacenar objetos de cualquier tipo. Hay tres estilos de clases contenedoras: contenedores de primera clase, adaptadores de contenedores y casi contenedores.
- Los algoritmos de la STL son funciones que realizan manipulaciones de datos comunes, tales como búsqueda, ordenamiento y comparación de elementos o de contenedores completos.

Sección 22.1.1 Introducción a los contenedores

- Los contenedores se dividen en contenedores de secuencia, contenedores asociativos y adaptadores de contenedores.
- Los contenedores de secuencia representan estructuras de datos lineales, como vectores y listas enlazadas.
- Los contenedores asociativos son contenedores no lineales que localizan con rapidez los elementos almacenados en ellos, como conjuntos de valores o pares clave/valor.
- Los contenedores de secuencia y los contenedores asociativos se conocen en forma colectiva como contenedores de primera clase.

Sección 22.1.2 Introducción a los iteradores

- La función `begin` de los contenedores de primera clase devuelve un iterador que apunta al primer elemento de un contenedor. La función `end` devuelve un iterador que apunta al primer elemento que está más allá del final del contenedor (un elemento que no existe y que generalmente se utiliza en un ciclo para indicar cuándo se debe terminar el procesamiento de los elementos del contenedor).
- Un `istream_iterator` es capaz de extraer los valores de un flujo de entrada en forma segura para los tipos. Un `ostream_iterator` es capaz de insertar valores en un flujo de salida.
- Los iteradores de entrada y salida sólo pueden desplazarse en dirección de avance (es decir, desde el inicio del contenedor hasta el final), un elemento a la vez.
- Un iterador de avance combina las herramientas de los iteradores de entrada y salida.
- Un iterador bidireccional tiene las herramientas de un iterador de avance y la habilidad de desplazarse en dirección hacia atrás (es decir, desde el final del contenedor hasta el principio).
- Un iterador de acceso aleatorio tiene las herramientas de un iterador bidireccional y la habilidad de acceder directamente a cualquier elemento del contenedor.

Sección 22.1.3 Introducción a los algoritmos

- Los contenedores que soportan iteradores de acceso aleatorio, como un `vector`, se pueden utilizar con todos los algoritmos en la STL.

Sección 22.2 Contenedores de secuencia

- La STL proporciona los contenedores de secuencia `vector`, `list` y `deque`. Las plantillas de clases `vector` y `deque` están basadas en arreglos. La plantilla de clase `list` implementa una estructura de datos tipo lista enlazada.

Sección 22.2.1 Contenedor de secuencia `vector`

- La función `capacity` devuelve el número de elementos que se pueden almacenar en un vector antes de que éste cambie su tamaño en forma dinámica, para dar cabida a más elementos.
- La función `push_back` de los contenedores de secuencia agrega un elemento al final de un contenedor.
- Para utilizar los algoritmos de la STL, debemos incluir el archivo de encabezado `<algorithm>`.
- El algoritmo `copy` copia cada elemento en un contenedor, empezando con la ubicación especificada por el iterador en su primer argumento, y hasta (pero sin incluir) la ubicación especificada por el iterador en su segundo argumento.
- La función `front` devuelve una referencia al primer elemento en un contenedor de secuencia. La función `begin` devuelve un iterador que apunta al principio de un contenedor de secuencia.
- La función `back` devuelve una referencia al último elemento en un contenedor de secuencia. La función `end` devuelve un iterador que apunta a un elemento más allá del final de un contenedor de secuencia.
- La función `insert` de los contenedores de secuencia inserta uno o varios valores antes del elemento en una ubicación específica.
- La función `erase` (en todos los contenedores de primera clase) elimina uno o varios elementos específicos del contenedor.
- La función `empty` (en todos los contenedores y adaptadores) devuelve `true` si el contenedor está vacío.
- La función `clear` (en todos los contenedores de primera clase) vacía el contenedor.

Sección 22.2.2 Contenedor de secuencia `list`

- El contenedor de secuencia `list` proporciona una implementación eficiente para las operaciones de inserción y eliminación en cualquier ubicación del contenedor. Debe incluirse el archivo de encabezado `<list>` para utilizar la plantilla de clase `list`.
- La función miembro `push_front` de `list` inserta valores al principio de una lista.
- La función miembro `sort` de `list` ordena los elementos de la lista en orden ascendente.

- La función miembro `splice` de `list` elimina elementos en un objeto `list` y los inserta en otro objeto `list` en una posición específica.
- La función miembro `unique` de `list` elimina los elementos duplicados en un objeto `list`.
- La función miembro `assign` de `list` reemplaza el contenido de un objeto `list` con el contenido de otro.
- La función miembro `remove` de `list` elimina todas las copias de un valor especificado de un objeto `list`.

Sección 22.2.3 Contenedor de secuencia deque

- La plantilla de clase `deque` proporciona las mismas operaciones que un objeto `vector`, pero agrega las funciones miembro `push_front` y `pop_front` para permitir la inserción y eliminación de elementos al principio de un objeto `deque`, respectivamente. Debe incluirse el archivo de encabezado `<deque>` para utilizar la plantilla de clase `deque`.

Sección 22.3 Contenedores asociativos

- Los contenedores asociativos de la STL proporcionan acceso directo para almacenar y obtener elementos a través de claves.
- Los cuatro contenedores asociativos son `multiset`, `set`, `multimap` y `map`.
- Las plantillas de clases `multiset` y `set` proporcionan operaciones para manipular conjuntos de valores, en donde los valores son las claves; no hay un valor separado asociado con cada clave. Debe incluirse el archivo de encabezado `<set>` para utilizar las plantillas de clases `set` y `multiset`.
- La principal diferencia entre un objeto `multiset` y un objeto `set` es que un `multiset` permite claves duplicadas y un `set` no.

Sección 22.3.1 Contenedor asociativo multiset

- El contenedor asociativo `multiset` proporciona operaciones rápidas de almacenamiento y obtención de claves, y permite claves duplicadas. El orden de los elementos se determina mediante un objeto de función de comparación.
- Las claves de un objeto `multiset` se pueden ordenar en forma ascendente, para lo cual se ordenan las claves con el objeto de función de comparación `less<T>`.
- El tipo de las claves en todos los contenedores asociativos debe soportar la comparación en forma apropiada, con base en el objeto de función de comparación especificado; por ejemplo, las claves ordenadas con `less<T>` deben soportar la comparación con el operador `<`.
- Un objeto `multiset` soporta los iteradores bidireccionales.
- Debe incluirse el archivo de encabezado `<set>` para utilizar la clase `multiset`.

Sección 22.3.2 Contenedor asociativo set

- El contenedor asociativo `set` se utiliza para operaciones rápidas de almacenamiento y obtención de claves únicas.
- Si se hace un intento de insertar una clave duplicada en un objeto `set`, se ignora el duplicado.
- Un objeto `set` soporta los iteradores bidireccionales.
- Debe incluirse el archivo de encabezado `<set>` para utilizar la clase `set`.

Sección 22.3.3 Contenedor asociativo multimap

- Las plantillas de clases `multimap` y `map` proporcionan operaciones para manipular los valores asociados con las claves.
- La principal diferencia entre un objeto `multimap` y un objeto `map` es que un `multimap` permite almacenar claves duplicadas con valores asociados, y un `map` sólo permite claves únicas con valores asociados.
- La función `count` (disponible para todos los contenedores asociativos) cuenta el número de ocurrencias del valor especificado actualmente en un contenedor.
- La función `find` (disponible para todos los contenedores asociativos) localiza un valor especificado en un contenedor.
- Las funciones `lower_bound` y `upper_bound` (disponibles en todos los contenedores asociativos) localiza la primera ocurrencia del valor especificado en un contenedor, y el elemento después de la última ocurrencia del valor especificado en un contenedor, respectivamente.
- La función `equal_range` (disponible en todos los contenedores asociativos) devuelve un objeto `pair` que contiene los resultados de una operación `lower_bound` y de una operación `upper_bound`.
- El contenedor asociativo `multimap` se utiliza para operaciones rápidas de almacenamiento y obtención de claves y valores asociados (que a menudo se les conoce como pares clave/valor).
- En un objeto `multimap` se permiten claves duplicadas, por lo que se pueden asociar múltiples valores con una sola clave. A esto se le conoce como relación de uno a varios.
- Debe incluirse el archivo de encabezado `<map>` para utilizar las plantillas de clases `map` y `multimap`.

Sección 22.3.4 Contenedor asociativo map

- No se permiten claves duplicadas en un objeto `map`, por lo que sólo se puede asociar un solo valor con cada clave. A esto se le conoce como asignación de uno a uno.
- Por lo general, a un objeto `map` se le conoce como arreglo asociativo.

Sección 22.4 Adaptadores de contenedores

- La STL proporciona tres adaptadores de contenedores: `stack`, `queue` y `priority_queue`.
- Los adaptadores no son contenedores de primera clase, ya que no proporcionan la implementación de la estructura de datos actual en la que se pueden almacenar elementos, y no soportan iteradores.
- Las tres plantillas de clases adaptadoras proporcionan las funciones miembro `push` y `pop` que insertan de manera apropiada un elemento en, y lo eliminan de, cada estructura de datos del adaptador, respectivamente.

Sección 22.4.1 Adaptador `stack`

- La plantilla de clase `stack` es una estructura de datos del tipo “último en entrar, primero en salir”. Debe incluirse el archivo de encabezado `<stack>` para utilizar la plantilla de clase `stack`.
- La función miembro `top` de `stack` devuelve una referencia al elemento superior del objeto `stack` (lo que se implementa mediante una llamada a la función `back` del contenedor subyacente).
- La función miembro `empty` de `stack` determina si el objeto `stack` está vacío (lo que se implementa mediante una llamada a la función `empty` del contenedor subyacente).
- La función miembro `size` de `stack` devuelve el número de elementos en el objeto `stack` (lo que se implementa mediante una llamada a la función `size` del contenedor subyacente).

Sección 22.4.2 Adaptador `queue`

- La plantilla de clase `queue` permite inserciones al final de la estructura de datos subyacente, y eliminaciones de la parte frontal de la estructura de datos subyacente (lo que comúnmente se conoce como estructura de datos del tipo “primero en entrar, primero en salir”). Debe incluirse el archivo de encabezado `<queue>` para utilizar un objeto `queue` o un `priority_queue`.
- La función miembro `front` de `queue` devuelve una referencia al primer elemento en el objeto `queue` (lo que se implementa mediante una llamada a la función `front` del contenedor subyacente).
- La función miembro `back` de `queue` devuelve una referencia al último elemento en el objeto `queue` (lo que se implementa mediante una llamada a la función `back` del contenedor subyacente).
- La función miembro `empty` de `queue` determina si el objeto `queue` está vacío (lo que se implementa mediante una llamada a la función `empty` del contenedor subyacente).
- La función miembro `size` de `queue` devuelve el número de elementos en el objeto `queue` (lo que se implementa mediante una llamada a la función `size` del contenedor subyacente).

Sección 22.4.3 Adaptador `priority_queue`

- La plantilla de clase `priority_queue` proporciona una funcionalidad que permite inserciones ordenadas en la estructura de datos subyacente, y eliminaciones de la parte frontal de la estructura de datos subyacente.
- Las operaciones comunes de `priority_queue` son `push`, `pop`, `top`, `empty` y `size`.

Sección 22.5.1 `fill`, `fill_n`, `generate` y `generate_n`

- Los algoritmos `fill` y `fill_n` establecen cada elemento en un rango de elementos del contenedor en un valor específico.
- Los algoritmos `generate` y `generate_n` utilizan una función generadora o un objeto de función para crear valores para cada elemento en un rango de elementos del contenedor.

Sección 22.5.2 `equal`, `mismatch` y `lexicographical_compare`

- El algoritmo `equal` compara la igualdad entre dos secuencias de valores.
- El algoritmo `mismatch` compara dos secuencias de valores y devuelve un par de iteradores que indican la ubicación en cada secuencia de los elementos que no coinciden.
- El algoritmo `lexicographical_compare` compara el contenido de dos secuencias.

Sección 22.5.3 `remove`, `remove_if`, `remove_copy` y `remove_copy_if`

- El algoritmo `remove` elimina todos los elementos con un valor específico en cierto rango.
- El algoritmo `remove_copy` copia todos los elementos que no tienen un valor específico en cierto rango.
- El algoritmo `remove_if` elimina todos los elementos que cumplen con la condición `if` en cierto rango.
- El algoritmo `remove_copy_if` copia todos los elementos que cumplen con la condición `if` en cierto rango.

Sección 22.5.4 `replace`, `replace_if`, `replace_copy` y `replace_copy_if`

- El algoritmo `replace` reemplaza todos los elementos con un valor específico en cierto rango.
- El algoritmo `replace_copy` copia todos los elementos con un valor específico en cierto rango.
- El algoritmo `replace_if` reemplaza todos los elementos que cumplen con la condición `if` en cierto rango.
- El algoritmo `replace_copy_if` copia todos los elementos que cumplen con la condición `if` en cierto rango.

Sección 22.5.5 Algoritmos matemáticos

- El algoritmo `random_shuffle` reordena al azar los elementos en cierto rango.
- El algoritmo `count` cuenta los elementos con un valor específico en cierto rango.
- El algoritmo `count_if` cuenta los elementos que cumplen con la condición `if` en cierto rango.
- El algoritmo `min_element` localiza el elemento más pequeño en cierto rango.
- El algoritmo `max_element` localiza el elemento más grande en cierto rango.
- El algoritmo `accumulate` suma los valores en cierto rango.
- El algoritmo `for_each` aplica una función general o un objeto de función a cada elemento en un rango.
- El algoritmo `transform` aplica una función general o un objeto de función a cada elemento en un rango, y reemplaza cada elemento con el resultado de la función.

Sección 22.5.6 Algoritmos básicos de búsqueda y ordenamiento

- El algoritmo `find` localiza un valor específico en cierto rango.
- El algoritmo `find_if` localiza el primer valor en cierto rango que cumpla con la condición `if`.
- El algoritmo `sort` ordena los elementos en cierto rango en orden ascendente, o en un orden especificado por un predicado.
- El algoritmo `binary_search` determina si un valor específico está en un rango ordenado de elementos.

Sección 22.5.7 `swap`, `iter_swap` y `swap_ranges`

- El algoritmo `swap` intercambia dos valores.
- El algoritmo `iter_swap` intercambia los dos elementos.
- El algoritmo `swap_ranges` intercambia los elementos en cierto rango.

Sección 22.5.8 `copy_backward`, `merge`, `unique` y `reverse`

- El algoritmo `copy_backward` copia los elementos en un rango y los coloca en un contenedor, empezando desde el final y avanzando hacia la parte inicial.
- El algoritmo `merge` combina dos secuencias en orden ascendente de valores en una tercera secuencia en orden ascendente.
- El algoritmo `unique` elimina los elementos duplicados en una secuencia ordenada de elementos en cierto rango.
- El algoritmo `reverse` invierte todos los elementos en cierto rango.

Sección 22.5.9 `inplace_merge`, `unique_copy` y `reverse_copy`

- El algoritmo `inplace_merge` combina dos secuencias ordenadas de elementos en el mismo contenedor.
- El algoritmo `unique_copy` realiza una copia de todos los elementos únicos en la secuencia ordenada de valores en cierto rango.
- El algoritmo `reverse_copy` realiza una copia inversa de todos los elementos en cierto rango.

Sección 22.5.10 Operaciones set

- La función `includes` de `set` compara dos conjuntos de valores ordenados para determinar si cada elemento del segundo objeto `set` está en el primer objeto `set`.
- La función `set_difference` de `set` busca los elementos del primer conjunto `set` de valores ordenados que no estén en el segundo conjunto `set` de valores ordenados (ambos conjuntos de valores deben estar en orden ascendente).
- La función `set_intersection` de `set` determina los elementos del primer conjunto `set` de valores ordenados que estén en el segundo conjunto `set` de valores ordenados (ambos conjuntos de valores deben estar en orden ascendente).
- La función `set_symmetric_difference` de `set` determina los elementos en el primer conjunto `set` que no estén en el segundo conjunto `set`, y los elementos en el segundo conjunto `set` que no estén en el primer conjunto `set` (ambos conjuntos de valores deben estar en orden ascendente).
- La función `set_union` de `set` crea un conjunto `set` de todos los elementos que estén en uno o ambos conjuntos `set` ordenados (ambos conjuntos de valores deben estar en orden ascendente).

Sección 22.5.11 `lower_bound`, `upper_bound` y `equal_range`

- El algoritmo `lower_bound` busca la primera ubicación en una secuencia ordenada de valores en la que el tercer argumento se debe insertar en la secuencia, de tal forma que la secuencia siga almacenada en orden ascendente.
- El algoritmo `upper_bound` busca la última ubicación en una secuencia ordenada de valores en los que el tercer argumento se podría insertar en la secuencia, de tal forma que ésta siga almacenada en orden ascendente.
- El algoritmo `equal_range` devuelve el límite inferior y el límite superior como un objeto `pair`.

Sección 22.5.12 Ordenamiento de montón (heapsort)

- El algoritmo `make_heap` recibe una secuencia de valores en un cierto rango, y crea un montón que se puede utilizar para producir una secuencia ordenada.
- El algoritmo `sort_heap` ordena una secuencia de valores en cierto rango que ya están ordenados en un montón.
- El algoritmo `pop_heap` elimina el elemento superior del montón.

Sección 22.5.13 *min* y *max*

- Los algoritmos *min* y *max* determinan el mínimo de dos elementos y el máximo de dos elementos, respectivamente.

Sección 22.6 La clase *bitset*

- La plantilla de clase *bitset* facilita la creación y manipulación de conjuntos de bits, que son útiles para representar un conjunto de banderas de bits.

Sección 22.7 Objetos de funciones

- Un objeto de función es una instancia de una clase que sobrecarga a *operator()*.
- La STL proporciona muchos objetos de función predefinidos, los cuales se encuentran en el encabezado *<functional>*.
- Los objetos de función binaria son objetos de función que reciben dos argumentos y devuelven un valor. La plantilla de clase *binary_function* es una clase base vacía para crear objetos de funciones binarias que proporciona nombres de tipos estándar para los parámetros y el resultado de la función.

Terminología

accumulate, algoritmo	for_each, algoritmo
adaptador	front, función miembro de contenedor de secuencia
adaptador de contenedor	front_inserter, plantilla de función
<algorithm>, archivo de encabezado	función binaria
algoritmo	<functional>, archivo de encabezado
algoritmo de secuencia mutante	functor
arreglo asociativo	generate, algoritmo
asignación de uno a uno	generate_n, algoritmo
assign, función miembro de <i>list</i>	heapsort, algoritmo de ordenamiento
back, función miembro de los contenedores de secuencia	includes, algoritmo
back_inserter, plantilla de función	inplace_merge, algoritmo
begin, función miembro de los contenedores de primera clase	insert, función miembro de contenedores
Biblioteca de plantillas estándar (STL)	inserter, plantilla de función
binary_function, plantilla de clase	istream_iterator
binary_search, algoritmo	iter_swap, algoritmo
capacity, función miembro de <i>vector</i>	iterador
clave de búsqueda	iterador bidireccional
const_iterator	iterador de avance
const_reverse_iterator	iterador de entrada
contenedor	iterador de salida
contenedor asociativo	less<int>
contenedor cercano	lexicographical_compare, algoritmo
contenedor de primera clase	<list>, archivo de encabezado
contenedor de secuencia	list, contenedor de secuencia
copy_backward, algoritmo	lower_bound, algoritmo
count, algoritmo	lower_bound, función de contenedor asociativo
count_if, algoritmo	make_heap, algoritmo
<deque>, archivo de encabezado	<map>, archivo de encabezado
deque, contenedor de secuencia	map, contenedor asociativo
empty, función miembro de los contenedores	max, algoritmo
end, función miembro de los contenedores	max_element, algoritmo
equal, algoritmo	merge, algoritmo
equal_range, algoritmo	min, algoritmo
equal_range, función de contenedor asociativo	min_element, algoritmo
erase, función miembro de contenedores	mismatch, algoritmo
fill, algoritmo	montón
fill_n, algoritmo	multimap, contenedor asociativo
find, algoritmo	multiset, contenedor asociativo
find, función de contenedor asociativo	<numeric>, archivo de encabezado
find_if, algoritmo	objeto de función
first, miembro de datos de pair	objeto de función binaria
flip, función de <i>bitset</i>	objeto de función de comparación
	ostream_iterator

par clave/valor	reverse_iterator
pop, función miembro de los adaptadores de contenedores	second, miembro de datos de pair
pop_back, función	secuencia
pop_front, función	secuencia de entrada
pop_heap, algoritmo	secuencia de salida
priority_queue, plantilla de clase de adaptador	<set>, archivo de encabezado
push, función miembro de los adaptadores de contenedores	set, contenedor asociativo
push_heap, algoritmo	set_difference, algoritmo
<queue>, archivo de encabezado	set_intersection, algoritmo
queue, plantilla de clase de adaptador	set_symmetric_difference, algoritmo
random_shuffle, algoritmo	set_union, algoritmo
random-access, iterador	size, función miembro de los contenedores
rango	sort, algoritmo
rbegin, función miembro de vector	sort, función miembro de list
remove, algoritmo	sort_heap, algoritmo
remove, función miembro de list	splice, función miembro de list
remove_copy, algoritmo	<stack>, archivo de encabezado
remove_copy_if, algoritmo	stack, plantilla de clase de adaptador
remove_if, algoritmo	swap, algoritmo
rend, función miembro de los contenedores	swap, función miembro de list
replace, algoritmo	swap_range, algoritmo
replace_copy, algoritmo	top, función miembro de los adaptadores de contenedores
replace_copy_if, algoritmo	unique, algoritmo
replace_if, algoritmo	unique, función miembro de list
reset, función de bitset	unique_copy, algoritmo
reverse, algoritmo	upper_bound, algoritmo
reverse_copy, algoritmo	

Ejercicios de autoevaluación

Indique si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.

- 22.1 (V/F) La STL utiliza abundantemente la herencia y las funciones *virtual*.
- 22.2 Los dos tipos de contenedores de primera clase de la STL son: contenedores de secuencia y contenedores _____.
- 22.3 Los cinco tipos principales de iteradores son _____, _____, _____, _____ y _____.
- 22.4 (V/F) Un iterador actúa como un apuntador a un elemento.
- 22.5 (V/F) Los algoritmos de la STL pueden operar con arreglos tipo C, basados en apuntadores.
- 22.6 (V/F) Los algoritmos de la STL se encapsulan como funciones miembro dentro de cada clase de contenedor.
- 22.7 (V/F) Al utilizar el algoritmo *remove* en un *vector*, no disminuye el tamaño del *vector* del que se van a eliminar elementos.
- 22.8 Los tres adaptadores de contenedores de la STL son _____, _____ y _____.
- 22.9 (V/F) La función miembro de contenedor *end()* devuelve la posición del último elemento en el contenedor.
- 22.10 Los algoritmos de la STL operan indirectamente sobre los elementos de los contenedores, mediante el uso de _____.
- 22.11 El algoritmo *sort* requiere de un iterador _____.

Respuestas a los ejercicios de autoevaluación

- 22.1 Falso. El uso de estos elementos se evita por cuestiones de rendimiento.
- 22.2 Asociativos.
- 22.3 De entrada, de salida, de avance, bidireccionales y de acceso aleatorio.
- 22.4 Falso. En realidad es al revés.
- 22.5 Verdadero.
- 22.6 Falso. Los algoritmos de la STL no son funciones miembro. Operan indirectamente sobre los contenedores, a través de los iteradores.

- 22.7** Verdadero.
- 22.8** `stack`, `queue`, `priority_queue`.
- 22.9** Falso. En realidad devuelve la posición que se encuentra justo después del final del contenedor.
- 22.10** Iteradores.
- 22.11** De acceso aleatorio.

Ejercicios

- 22.12** Escriba una plantilla de función llamada `palíndromo` que reciba un parámetro `vector` y devuelva `true` o `false`, dependiendo de si el vector se lee igual o no tanto hacia adelante como hacia atrás (por ejemplo, un vector que contenga 1, 2, 3, 2, 1 es un palíndromo, pero un vector que contenga 1, 2, 3, 4 no lo es).
- 22.13** Modifique la figura 22.40, la Criba de Eratóstenes, de forma que, si el número que introduce el usuario en el programa no es primo, éste muestre los factores primos del número. Recuerde que los factores de un número primo son sólo 1 y el número primo en sí. Cada número no primo tiene una factorización prima única. Por ejemplo, los factores de 54 son 2, 3, 3 y 3. Cuando estos factores se multiplican en conjunto, el resultado es 54. Para el número 54, los factores primos mostrados deben ser 2 y 3.
- 22.14** Modifique el ejercicio 22.13 de manera que, si el número que introduce el usuario en el programa no es primo, éste muestre los factores primos del número y el número de veces que aparezca el factor primo en la factorización prima única. Por ejemplo, la salida para el número 54 debe ser

La factorizacion prima unica de 54 es: 2 * 3 * 3 * 3

Lecturas recomendadas

- Ammeraal, L. *STL for C++ Programmers*. New York, NY: John Wiley, 1997.
- Austern, M. H. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Reading, MA: Addison Wesley, 1998.
- Glass, G. y B. Schuchert. *The STL <Primer>*. Upper Saddle River, NJ: Prentice Hall PTR, 1995.
- Henricson, M. y E. Nyquist. *Industrial Strength C++: Rules and Recommendations*. Upper Saddle River, NJ: Prentice Hall, 1997.
- Josuttis, N. *The C++ Standard Library: A Tutorial and Handbook*. Reading, MA: Addison-Wesley, 1999.
- Koenig, A. y B. Moo. *Ruminations on C++*. Reading, MA: Addison-Wesley, 1997.
- Meyers, S. *Effective STL: 50 Specific Ways to Improve your Use of the Standard Template Library*. Reading, MA: Addison-Wesley, 2001.
- Musser, D. R. y A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Reading, MA: Addison-Wesley, 1996.
- Musser, D. R. y A. A. Stepanov. "Algorithm-Oriented Generic Libraries", *Software Practice and Experience* Vol. 24, Núm. 7, julio de 1994.
- Nelson, M. *C++ Programmer's Guide to the Standard Template Library*. Foster City, CA: Programmer's Press, 1995.
- Pohl, I. *C++ Distilled: A Concise ANSI /ISO Reference and Style Guide*. Reading, MA: Addison-Wesley, 1997.
- Pohl, I. *Object-Oriented Programming Using C++, Second Edition*. Reading, MA: Addison-Wesley, 1997.
- Robson, R. *Using the STL: The C++ Standard Template Library*. New York, NY: Springer Verlag, 2000.
- Schildt, H. *STL Programming from the Ground Up*, New York: Osborne McGraw-Hill, 1999.
- Stepanov, A. y M. Lee. "The Standard Template Library", *Internet Distribution*. 31 de octubre de 1995 <www.cs.rpi.edu/~musser/doc.ps>.
- Stroustrup, B. "Making a vector Fit for a Standard", *The C++ Report*. Octubre de 1994.
- Stroustrup, B. *The Design and Evolution of C++*. Reading, MA: Addison-Wesley, 1994.
- Stroustrup, B. *The C++ Programming Language, Third Edition*. Reading, MA: Addison-Wesley, 1997.
- Vilot, M. J. "An Introduction to the Standard Template Library". *The C++ Report*, Vol. 6, Núm. 8, octubre de 1994.



*¡Venga, Watson, venga!
el juego ha comenzado.*

—Sir Arthur Conan Doyle

*Por uno, dos, tres strikes
estás fuera de la jugada.*

—Jack Norworth

El juego está arriba.

—William Shakespeare

*Si quieres evitar colisiones
extrañas, es mejor que
abandones el océano.*

—Henry Clay

*Donde hay mucha luz
hay una gran sombra.*

—Johann Wolfgang von Goethe

*La música de Wagner
es mejor de lo que suena.*

—Mark Twain

Programación de juegos con Ogre

OBJETIVOS

En este capítulo aprenderá a:

- Utilizar algunos fundamentos de la programación de juegos.
- Encontrar instrucciones acerca de cómo instalar el motor de gráficos Ogre 3D para trabajar con sus programas de C++ en las plataformas Windows, Linux y Mac.
- Crear juegos utilizando Ogre.
- Realizar la detección de colisiones.
- Utilizar Ogre para importar y mostrar gráficos.
- Utilizar OgreAL para integrar la biblioteca de audio OpenAL en sus juegos.
- Hacer que Ogre acepte la entrada mediante el teclado.
- Crear el juego simple Pong® con Ogre y OgreAL.
- Utilizar Ogre para regular la velocidad de un juego.

- 23.1** Introducción
- 23.2** Instalación de Ogre, OgreAL y OpenAL
- 23.3** Fundamentos de la programación de juegos
- 23.4** El juego de Pong: recorrido a través del código
 - 23.4.1** Inicialización de Ogre
 - 23.4.2** Creación de una escena
 - 23.4.3** Agregar elementos a la escena
 - 23.4.4** Animación y temporizadores
 - 23.4.5** Entrada del usuario
 - 23.4.6** Detección de colisiones
 - 23.4.7** Sonido
 - 23.4.8** Recursos
 - 23.4.9** Controlador de Pong
- 23.5** Repaso
- 23.6** Recursos Web de Ogre

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)

23.1 Introducción

Ahora vamos a presentar la programación de juegos y gráficos con el motor de gráficos Ogre 3D. Creado en el año 2000 por Steve Streeting, Ogre es un proyecto de código abierto, mantenido por el equipo de Ogre en www.ogre3d.org. En primer lugar hablaremos sobre las cuestiones básicas involucradas en la programación de juegos. Después mostraremos al lector cómo usar Ogre para crear un juego simple, que presenta una mecánica similar al videojuego clásico de Pong®, desarrollado originalmente por Atari en 1972. Demostraremos cómo crear una escena con gráficos en 3D a color, animar objetos con un movimiento uniforme, utilizar temporizadores para controlar la velocidad de animación, detectar colisiones entre objetos, agregar sonido, aceptar la entrada mediante el teclado y mostrar salida de texto.

23.2 Instalación de Ogre, OgreAL y OpenAL

Ogre es poderoso y fácil de usar, pero la instalación es un poco complicada y varía para las distintas plataformas y compiladores. Hay dos opciones de instalación: podemos instalar el SDK de Ogre o descargar el código fuente y compilarlo. En este libro utilizamos Ogre 1.4 (Eihort). También se deben instalar OgreAL y la biblioteca de audio OpenAL. OpenAL se encarga de la funcionalidad del sonido y OgreAL nos permite integrar las herramientas de OpenAL con el código de Ogre. Proporcionamos instrucciones detalladas para instalar el SDK de Ogre y todos los componentes de OgreAL. Las instrucciones también explican cómo configurar un proyecto de Visual C++ 2005 para utilizar Ogre y OgreAL. En la sección Recursos adicionales (Additional Resources) del sitio Web de este libro (www.deitel.com/books/cpphtp6) encontrará las instrucciones.

23.3 Fundamentos de la programación de juegos

En esta sección presentamos los fundamentos generales de la programación de juegos. En la sección 23.4 presentamos una implementación del juego de Pong, en la cual se utilizan las bibliotecas de Ogre y OgreAL.

Gráficos

Los gráficos son tal vez la característica más crucial de cualquier videojuego. La programación de gráficos, que anteriormente era una especialidad, se está volviendo cada vez más accesible, incluso para los programadores novatos. Hay muchos motores de gráficos en 3D disponibles; estos marcos de trabajo ocultan la compleja y a menudo tediosa programación requerida con las API de gráficos, y nos permiten manejar los gráficos con más facilidad.

Ogre (Object-oriented Graphics Rendering Engine. Motor de visualización de gráficos orientado a objetos), uno de los motores de gráficos más prominentes, se ha utilizado en muchos productos comerciales, incluyendo videojuegos. Proporciona una interfaz orientada a objetos para la programación de gráficos en 3D. Soporta las API de gráficos Direct3D y OpenGL, y se ejecuta en las plataformas Windows, Linux y Mac. Direct3D es la API de gráficos en 3D de

Microsoft Windows. OpenGL es una especificación de gráficos implementada por muchos distribuidores de tarjetas de video en todas las principales plataformas, incluyendo Windows.

Ogre es sólo un motor de visualización de gráficos; no soporta directamente el sonido, la física, la detección de colisiones, la conexión en red ni demás necesidades relacionadas con los juegos. La comunidad de Ogre ha proporcionado muchos complementos que permiten a los usuarios integrar otras bibliotecas con Ogre para soportar esas características.

Modelos en 3D

Un **modelo en 3D** es una representación computacional de un objeto que se puede dibujar en la pantalla; a este proceso se le conoce como **visualización (rendering)**. Los **materiales** determinan la apariencia de un objeto al establecer las propiedades de iluminación, los colores y las texturas. Una **textura** es una imagen que se envuelve alrededor del modelo.

La mayoría de los objetos que se muestran en gráficos en 3D, desde el terreno hasta los caracteres y los edificios, son modelos en 3D. Muchos modelos se crean en **herramientas de modelado en 3D**. Algunas herramientas populares de modelado en 3D son Maya (usa.autodesk.com/adsk/servlet/index?siteID=123112&id=7635018), SoftImage XSI (www.softimage.com/) y Blender (www.blender.org/). Todos están disponibles para las plataformas Windows, Linux y Mac. Blender es gratuito y Maya ofrece una versión sin costo. SoftImage tiene una versión de prueba de 30 días disponible. La comunidad de Ogre también ha producido varias herramientas para permitir a los usuarios **exportar modelos en 3D** de éstas y otras herramientas de modelado populares hacia Ogre.

Materiales, texturas y colores

Los colores se determinan mediante las intensidades de luz de color rojo, verde y azul, que pueden variar de 0 a 1.0; un valor de color de (1.0, 0, 0) creará un color rojo brillante, (0, 1.0, 0) creará un verde brillante, y (0, 0, 1.0) creará un azul brillante. El primer valor es la intensidad de rojo, el segundo es la intensidad de verde y el tercero es la intensidad de azul. Para crear blanco, se utilizan las máximas intensidades de los tres valores de colores, (1.0, 1.0, 1.0). Para crear negro (la ausencia de color) se utiliza (0, 0, 0). Los valores de colores incluyen algunas veces un **canal alfa** para representar la transparencia, que también varía de 0 a 1.0, en donde 0 es completamente transparente y 1 completamente opaco. En la figura 23.1 se muestran los colores comunes y sus valores de intensidad de rojo, verde y azul. En la Web también puede encontrar tablas de colores, como en www.tayloredmktg.com/rgb/.

En los gráficos en 3D, los materiales se utilizan para determinar el color de un modelo en 3D. Un material determina la forma en que el modelo debe reflejar distintos tipos de luz y aplica textura al modelo. Los materiales se pueden establecer para utilizar distintos **niveles de detalle (LoD)**, dependiendo de qué tan lejos se encuentre el observador del modelo. Al acercarse, el modelo se debe visualizar con el mayor detalle posible. Cuando el objeto en la escena está alejado, no tiene caso desperdiciar poder de cómputo en visualizar los detalles que el usuario no puede ver. Para incrementar el rendimiento, el objeto se puede visualizar con mucho menor detalle.

Color	Valor de rojo	Valor de verde	Valor de azul
Rojo	1.0	0.0	0.0
Verde	0.0	1.0	0.0
Azul	0.0	0.0	1.0
Naranja	1.0	0.784	0.0
Rosa	1.0	0.686	0.686
Cyan	0.0	1.0	1.0
Magenta	1.0	0.0	1.0
Amarillo	1.0	1.0	0.0
Negro	0.0	0.0	0.0
Blanco	1.0	1.0	1.0
Gris	0.5	0.5	0.5
Gris claro	0.75	0.75	0.75
Gris oscuro	0.25	0.25	0.25

Figura 23.1 | Intensidades de rojo, verde y azul de los colores comunes en Ogre.

Iluminación

Hay cuatro tipos distintos de luz en una escena en 3D: ambiental, difusa, emisiva y especular.¹ La **luz ambiental** es la iluminación general en la escena, que se ha reflejado desde tantas superficies que no parece tener un origen definido. La **luz difusa** parece venir de una dirección específica y se refleja en forma uniforme de cualquier superficie que toca. La **luz emisiva** parece venir de un objeto en la escena. Esta luz no afecta a los objetos alrededor de ella, pero hace que el objeto que la emite parezca más brillante. La **luz especular** proviene de una dirección específica y se refleja desde un objeto, con base en la dirección hacia el observador. Esta luz se utiliza para hacer que un objeto parezca reluciente.

Detección de colisiones y respuesta

La **detección de colisiones** es el proceso de determinar si dos objetos en un juego entraron en contacto. El programador debe saber cuáles objetos evaluar y debe lidiar con ciertas matemáticas complejas. Es relativamente sencillo comprobar si un cuadrado choca con otro si cada uno es paralelo a la línea del piso. Es más difícil comprobar colisiones en círculos y esferas; los cálculos matemáticos de las superficies curvas son más complejos.

Los objetos necesitan reaccionar en forma apropiada cuando chocan con otros objetos. Algunos objetos (como las paredes) son estacionarios, mientras que otros se desplazan a través de la escena. El modelado de la física del movimiento de los objetos puede ser un proceso complejo. Hay bibliotecas de detección de colisiones y modelado de la física que se encargan de estas complejidades por el programador. Dichas bibliotecas ayudan a crear una experiencia de juego realista.

Sonido

El sonido es crucial para la experiencia de juego. Los jugadores desean escuchar los láseres en las naves que explotan, o los motores de sus autos de carreras al arrancar en la línea de salida. Las bibliotecas de audio nos ayudan a enriquecer los juegos con sonido. Muchas de esas bibliotecas soportan **sonido en 3D**. En una escena en 3D, los objetos que emiten sonido pueden encontrarse a distintas distancias y direcciones respecto al usuario. Las bibliotecas de sonidos toman estos factores en cuenta cuando reproducen los sonidos. Un sonido proveniente de un objeto que esté cerca del oyente sonará más fuerte que el sonido de un objeto que esté más alejado. Además, los sonidos provenientes de un lado del oyente se reproducirán en forma distinta de los sonidos del otro lado.

Texto

A menudo, los juegos se comunican con el usuario mediante la visualización de texto. Esto puede ser para proporcionar mensajes al usuario, o tan sólo para reportar cuántos puntos ha obtenido hasta cierto momento. En muchos juegos, el texto es una forma imprescindible de comunicación entre los jugadores. En www.1001freefonts.com encontrará tipos de letra gratuitos que puede usar en sus juegos.

Temporizadores

La velocidad a la cual se ejecuta un juego puede variar de un sistema a otro, debido a las diferencias en las velocidades de los procesadores. Para resolver este problema, los programadores de juegos utilizan **temporizadores** para controlar la velocidad de animación. Si un objeto se desplaza la misma distancia cada **cuadro** (cada vez que se vuelve a dibujar la pantalla), entonces se puede desplazar a distintas velocidades en distintas computadoras. Un juego que se ejecuta a 100 cuadros por segundo (fps) sería dos veces más rápido que el mismo juego ejecutándose a 50 cuadros por segundo. Los temporizadores ayudan a mantener la consistencia del juego al regular la velocidad.

Experiencia del usuario

Los juegos deben ser divertidos y atractivos para el jugador en todas las formas posibles. Los fundamentos que hemos descrito contribuyen a la experiencia del usuario en general. Podemos obtener la atención del jugador a través de los gráficos y del sonido. A menudo, las acciones en los juegos tienen sonidos asociados. Muchos sitios Web ofrecen sonidos gratuitos que usted puede utilizar en sus juegos. Algunos sitios de sonidos populares son Sound Hunter (www.sound-hunter.com), Absolute Sound Effects Archive (www.grsites.com/sounds) y el motor de búsqueda FindSounds (www.findsounds.com). También puede reproducir una pista de audio en segundo plano. Asegúrese de obtener permiso para utilizar cualquier canción protegida por los derechos de autor si planea comercializar su juego.

Los jugadores necesitan interactuar con los juegos. Los dispositivos de entrada del usuario incluyen el teclado, ratón, palanca de mando (joystick) y controlador de juegos. Hay que mantener los controles simples (el juego debe ser fácil de usar, pero no fácil de vencer). Puede comunicarse con el usuario mediante texto.

1. D. Astel y K. Hawkins, *Beginning OpenGL: Game Programming*, 2005, pp. 104-110.

23.4 El juego de Pong: recorrido a través del código

En las siguientes secciones presentaremos una implementación completa en C++/Ogre de un juego simple, el cual presenta una mecánica de juego similar al videojuego clásico Pong®, desarrollado originalmente por Atari en 1972 (en la figura 23.2 podrá observar la máquina tragamonedas original de Pong). Haremos un recorrido a través del código, explicando las herramientas de Ogre a medida que se vayan encontrando. Éste es uno de los programas de ejemplo más grandes del libro. Es conveniente que pruebe el programa completamente antes de leer el recorrido del código. Encontrará todos los archivos para este programa en la carpeta cap23 de los ejemplos del libro. Copie la carpeta PongResources a la carpeta OgreSDK\media. Ogre lanzará una excepción en tiempo de ejecución si no puede encontrar alguno de los recursos. Abra el proyecto Pong de Visual C++. Si siguió las instrucciones de instalación de Ogre y OgreAL correctamente, el proyecto deberá generarse con éxito. El archivo ejecutable del proyecto se copia a su carpeta OgreSDK\bin\debug (o OgreSDK\bin\release, si generó el proyecto en modo Release). *Importante:* recuerde copiar los archivos OgreAL_d.dll (o OgreAL.dll para el modo Release) y alut.dll en la carpeta que contiene el archivo ejecutable de Pong.

Pong tiene cuatro objetos de juego principales, una pelota, dos controles de perilla y una caja rectangular (figura 23.3). La pelota rebota a través de la pantalla dentro de la caja, mientras los jugadores controlan las perillas para evitar que la pelota haga contacto con el lado izquierdo o derecho. Si la pelota hace contacto con el lado izquierdo o derecho de la caja, el jugador que “ataca” ese lado recibe un punto. La puntuación se muestra en la parte superior de la pantalla.

Juegue durante un tiempo, utilizando las teclas A y Z para controlar la perilla izquierda, y las flechas arriba y abajo para controlar la perilla derecha. Observe los colores de los objetos, la interacción de la pelota con los demás objetos y la puntuación que aparece en la parte superior de la pantalla. Haga clic en Esc para terminar; si cierra la ventana el programa no se detendrá.

23.4.1 Inicialización de Ogre

La clase Pong (figuras 23.4 y 23.5) representa el juego de Pong. La figura 23.4 contiene la definición de la clase Pong. En la línea 6 se incluye el archivo de encabezado Ogre.h. Este archivo incluye automáticamente los archivos de encabezado de Ogre que se utilizan con más frecuencia. En la línea 23 se encuentra el prototipo de la función ejecutar, que inicia y ejecuta el juego. En las líneas 26 y 27 se encuentran los prototipos de las funciones que manejan la entrada del usuario



Figura 23.2 | La máquina tragamonedas de Pong de Atari entró en producción comercial en noviembre de 1972, vendiendo un total de 38,000 máquinas. El juego fue diseñado completamente en circuitos TTL, sin utilizar CPU ni software de código de juego. Su “código” se implementó utilizando chips de “compuerta” simples, temporizadores y chips contadores. Este juego simple, divertido y adictivo fue la piedra angular de lo que se convirtió en la industria de los videojuegos. (Videojuego clásico PONG®, cortesía de Atari Interactive, Inc. © 2007 Atari Interactive, Inc. Todos los derechos reservados. Uso permitido).

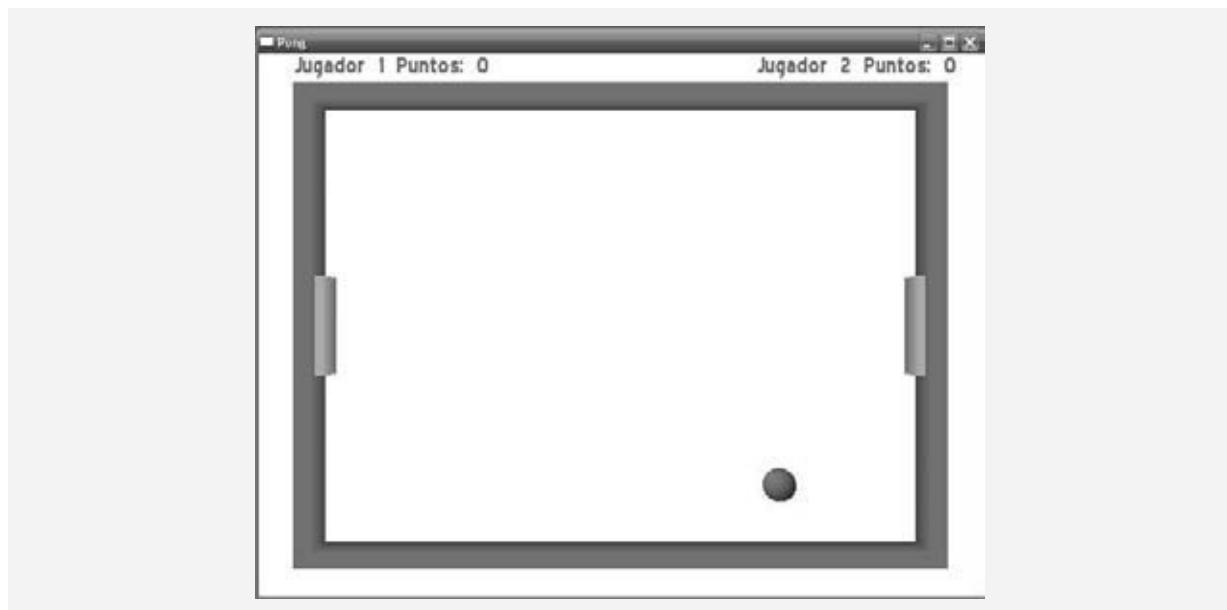


Figura 23.3 | Objetos del juego de Pong.

```

1 // Pong.h
2 // Definición de la clase Pong (representa un juego de Pong).
3 #ifndef PONG_H
4 #define PONG_H
5
6 #include <Ogre.h> // Definición de la clase Ogre
7 using namespace Ogre; // usa el espacio de nombres de Ogre
8
9 #include <OIS\OISEvents.h> // Definición de la clase OISEvents
0 #include <OIS\OISInputManager.h> // Definición de la clase OISInputManager
11 #include <OIS\OISKeyboard.h> // Definición de la clase OISKeyboard
12
13 class Pelota; // declaración anticipada de la clase Pelota
14 class Perilla; // declaración anticipada de la clase Perilla
15
16 enum Jugadores { JUGADOR1, JUGADOR2 };
17
18 class Pong : public FrameListener, public OIS::KeyListener
19 {
20 public:
21     Pong(); // constructor
22     ~Pong(); // destructor
23     void ejecutar(); // ejecuta un juego de Pong
24
25     // maneja los eventos keyPressed y keyReleased
26     bool keyPressed( const OIS::KeyEvent &keyEventRef );
27     bool keyReleased( const OIS::KeyEvent &keyEventRef );
28
29     // desplaza los objetos del juego y controla las interacciones entre los cuadros de animación
30     virtual bool frameStarted( const FrameEvent &frameEvent );
31     virtual bool frameEnded( const FrameEvent &frameEvent );
32     static void actualizarPuntuacion( Jugadores jugador ); // actualiza la puntuación
33
34 private:
35     void createScene(); // crea la escena a visualizar

```

Figura 23.4 | Definición de la clase Pong (representa un juego de Pong). (Parte I de 2).

```

36     static void actualizarPuntuacionTexto(); // actualiza la puntuación en la pantalla
37
38     // objetos de Ogre
39     Root *rootPtr; // apuntador al objeto Root de Ogre
40     SceneManager *sceneManagerPtr; // apuntador al objeto SceneManager
41     RenderWindow *windowPtr; // apuntador a RenderWindow para visualizar la escena
42     Viewport *viewportPtr; // apuntador a Viewport, el área que ve un objeto Camera
43     Camera *cameraPtr; // apuntador a un objeto Camera en la escena
44
45     // objetos de entrada de OIS
46     OIS::InputManager *inputManagerPtr; // apuntador al objeto InputManager
47     OIS::Keyboard *keyboardPtr; // apuntador al objeto Keyboard
48
49     // objetos del juego
50     Pelota *pelotaPtr; // apuntador al objeto Pelota
51     Perilla *perillaIzqPtr; // apuntador al objeto Perilla del jugador 1
52     Perilla *perillaDerPtr; // apuntador al objeto Perilla del jugador 2
53
54     // variables para controlar los estados del juego
55     bool salir, pausa; // ¿el usuario salió del juego o lo puso en pausa?
56     Real tiempo; // se utiliza para retrasar el movimiento de un nuevo objeto Pelota
57     static bool espera; // ¿Debe retrasarse el movimiento de la Pelota?
58
59     static int puntuacionJugador1; // puntuación del jugador 1
60     static int puntuacionJugador2; // puntuación del jugador 2
61 }; // fin de la clase Pong
62
63 #endif // PONG_H

```

Figura 23.4 | Definición de la clase Pong (representa un juego de Pong). (Parte 2 de 2).

mediante el teclado. En las líneas 30 y 31 se definen los prototipos de las funciones para llevar a cabo la lógica del juego entre un cuadro de animación y otro. También declaramos apuntadores a objetos importantes de Ogre (líneas 39 a 43), objetos para manejar la entrada (líneas 46 y 47) y los objetos del juego (líneas 50 a 52). En las líneas 55 a 57 se definen ciertas variables que se utilizan para controlar el comportamiento del juego; hablaremos sobre ellas más adelante.

```

1 // Pong.cpp
2 // Definiciones de las funciones miembro de la clase Pong.
3 #include <iostream>
4 using std::ostringstream;
5
6 #include <stdexcept>
7 using std::runtime_error;
8
9 #include <Ogre.h> // definición de la clase Ogre
10 using namespace Ogre; // usa el espacio de nombres de Ogre
11
12 #include <OgreAL.h> // definición de la clase OgreAL
13 #include <OgreStringConverter.h> // definición de la clase OgreStringConverter
14 #include <OIS\OISEvents.h> // definición de la clase OISEvents
15 #include <OIS\OISInputManager.h> // definición de la clase OISInputManager
16 #include <OIS\OISKeyboard.h> // definición de la clase OISKeyboard
17 #include "Pelota.h" // definición de la clase Pelota
18 #include "Perilla.h" // definición de la clase Perilla
19 #include "Pong.h" // definición de la clase Pong
20
21 int Pong::puntuacionJugador1 = 0; // inicializa la puntuación del jugador 1 en 0
22 int Pong::puntuacionJugador2 = 0; // inicializa la puntuación del jugador 2 en 0

```

Figura 23.5 | Definiciones de las funciones miembro de la clase Pong. (Parte 1 de 6).

```

23 bool Pong::espera = false; // inicializa espera con false
24
25 // direcciones para mover las Perillas
26 const Vector3 PERILLA_ABAJO = Vector3( 0, -15, 0 );
27 const Vector3 PERILLA_ARRIBA = Vector3( 0, 15, 0 );
28
29 // constructor
30 Pong::Pong()
31 {
32     rootPtr = new Root(); // inicializa el objeto Root
33
34     // Usa el cuadro de diálogo Ogre Config Dialog Box para seleccionar la configuración
35     if ( !rootPtr->showConfigDialog() ) // el usuario canceló el cuadro de diálogo
36         throw runtime_error( "El usuario cancelo el cuadro de dialogo Ogre Setup Dialog Box." );
37
38     // obtiene un apuntador al objeto RenderWindow
39     windowPtr = rootPtr->initialise( true, "Pong" );
40
41     // crea el objeto SceneManager
42     sceneManagerPtr = rootPtr->createSceneManager( ST_GENERIC );
43
44     // crea el objeto Camera
45     cameraPtr = sceneManagerPtr->createCamera( "PongCamera" );
46     cameraPtr->setPosition( Vector3( 0, 0, 200 ) ); // establece la posición del objeto Camera
47     cameraPtr->lookAt( Vector3( 0, 0, 0 ) ); // establece la dirección hacia la que ve el
        objeto Camera
48     cameraPtr->setNearClipDistance( 5 ); // distancia cercana que puede ver el objeto Camera
49     cameraPtr->setFarClipDistance( 1000 ); // distancia lejana que puede ver el objeto Camera
50
51     // crea el objeto Viewport
52     viewportPtr = windowPtr->addViewport( cameraPtr );
53     viewportPtr->setBackgroundColour( ColourValue( 1, 1, 1 ) );
54
55     // establece la proporción de aspecto del objeto Camera
56     cameraPtr->setAspectRatio( Real( viewportPtr->getActualWidth() ) /
57         ( viewportPtr->getActualHeight() ) );
58
59     // establece la luz ambiental de la escena
60     sceneManagerPtr->setAmbientLight( ColourValue( 0.75, 0.75, 0.75 ) );
61
62     // crea el objeto Light
63     Light *lightPtr = sceneManagerPtr->createLight( "Light" ); // un objeto Light
64     lightPtr->setPosition( 0, 0, 50 ); // establece la posición del objeto Light
65
66     unsigned long hWnd; // variable que contiene el manejador de ventana
67     windowPtr->getCustomAttribute( "WINDOW", &hWnd ); // obtiene el manejador de ventana
68     OIS::ParamList paramList; // crea un objeto ParamList de OIS
69
70     // agrega la ventana al objeto ParamList
71     paramList.insert( OIS::ParamList::value_type( "WINDOW",
72         Ogre::StringConverter::toString( hWnd ) ) );
73
74     // crea el objeto InputManager
75     inputManagerPtr = OIS::InputManager::createInputSystem( paramList );
76     keyboardPtr = static_cast< OIS::Keyboard*>( inputManagerPtr->
77         createInputObject( OIS::OISKeyboard, true ) ); // crea un objeto Keyboard
78     keyboardPtr->setEventCallback( this ); // agrega un objeto KeyListener
79
80     rootPtr->addFrameListener( this ); // agrega este objeto Pong como un FrameListener
81
82     // carga los recursos para Pong
83     ResourceGroupManager::getSingleton().addResourceLocation(

```

Figura 23.5 | Definiciones de las funciones miembro de la clase Pong. (Parte 2 de 6).

```

84     ".../media/RecursosPong", "FileSystem", "Pong" );
85 ResourceGroupManager::getSingleton().initialiseAllResourceGroups();
86
87     salir = pausa = false; // el jugador no salió ni puso en pausa el juego
88     tiempo = 0; // inicializa el tiempo desde que se restableció Pelota a 0
89 } // fin del constructor de Pong
90
91 // el destructor de Pong borra los objetos contenidos en un objeto Pong
92 Pong::~Pong()
93 {
94     // libera la memoria asignada en forma dinámica para Keyboard
95     inputManagerPtr->destroyInputObject( keyboardPtr );
96     OIS::InputManager::destroyInputSystem( inputManagerPtr );
97
98     // libera la memoria asignada en forma dinámica para Root
99     delete rootPtr; // libera la memoria a la que apunta el apuntador
100    rootPtr = 0; // apunta el apuntador a 0
101
102    // libera la memoria asignada en forma dinámica para Pelota
103    delete pelotaPtr; // libera la memoria a la que apunta el apuntador
104    pelotaPtr = 0; // apunta el apuntador a 0
105
106    // libera la memoria asignada en forma dinámica para Perilla
107    delete perillaIzqPtr; // libera la memoria a la que apunta el apuntador
108    perillaIzqPtr = 0; // apunta el apuntador a 0
109
110    // libera la memoria asignada en forma dinámica para Perilla
111    delete perillaDerPtr; // libera la memoria a la que apunta el apuntador
112    perillaDerPtr = 0; // apunta el apuntador a 0
113 } // fin del destructor de Pong
114
115 // crea la escena a mostrar
116 void Pong::createScene()
117 {
118     // obtiene un apuntador al objeto Overlay de Puntuacion
119     Overlay *scoreOverlayPtr =
120         OverlayManager::getSingleton().getByName( "Puntuacion" );
121     scoreOverlayPtr->show(); // muestra el objeto Overlay
122
123     // crea los objetos del juego
124     pelotaPtr = new Pelota( sceneManagerPtr ); // crea la Pelota
125     pelotaPtr->agregarAEscena(); // agrega la Pelota a la escena
126     perillaDerPtr = new Perilla( sceneManagerPtr, "PerillaDer", 90 );
127     perillaDerPtr->agregarAEscena(); // agrega una Perilla a la escena
128     perillaIzqPtr = new Perilla( sceneManagerPtr, "PerillaIzq", -90 );
129     perillaIzqPtr->agregarAEscena(); // agrega una Perilla a la escena
130
131     // crea las paredes
132     Entity *entityPtr = sceneManagerPtr->
133         createEntity( "ParedIzq", "cube.mesh" ); // crea la pared izquierda
134     entityPtr->setMaterialName( "pared" ); // establece el material para la pared izquierda
135     entityPtr->setNormaliseNormals( true ); // corrige las normales al aplicar una escala
136
137     // crea el objeto SceneNode para la pared izquierda
138     SceneNode *nodoPtr = sceneManagerPtr->getRootSceneNode()->
139         createChildSceneNode( "ParedIzq" );
140     nodoPtr->attachObject( entityPtr ); // adjunta la pared izquierda al objeto SceneNode
141     nodoPtr->setPosition( -95, 0, 0 ); // establece la posición de la pared izquierda
142     nodoPtr->setScale( .05, 1.45, .1 ); // establece el tamaño de la pared izquierda
143     entityPtr = sceneManagerPtr->createEntity( "ParedDer", "cube.mesh" );
144     entityPtr->setMaterialName( "pared" ); // establece el material para la pared derecha
145     entityPtr->setNormaliseNormals( true ); // corrige las normales al aplicar una escala

```

Figura 23.5 | Definiciones de las funciones miembro de la clase Pong. (Parte 3 de 6).

```

146 // crea el objeto SceneNode para la pared derecha
147 nodoPtr = sceneManagerPtr->getRootSceneNode()->
148     createChildSceneNode( "ParedDer" );
149 nodoPtr->attachObject( entityPtr ); // adjunta la pared izquierda al objeto SceneNode
150 nodoPtr->setPosition( 95, 0, 0 ); // establece la posición de la pared derecha
151 nodoPtr->setScale( .05, 1.45, .1 ); // establece el tamaño de la pared derecha
152 entityPtr = sceneManagerPtr->createEntity( "ParedInferior", "cube.mesh" );
153 entityPtr->setMaterialName( "pared" ); // establece el material para la pared inferior
154 entityPtr->setNormaliseNormals( true ); // corrige las normales al aplicar una escala
155
156 // crea el objeto SceneNode para la pared inferior
157 nodoPtr = sceneManagerPtr->getRootSceneNode()->
158     createChildSceneNode( "ParedInferior" );
159 nodoPtr->attachObject( entityPtr ); // adjunta la pared inferior al objeto SceneNode
160 nodoPtr->setPosition( 0, -70, 0 ); // establece la posición de la pared inferior
161 nodoPtr->setScale( 1.95, .05, .1 ); // establece el tamaño de la pared inferior
162 entityPtr = sceneManagerPtr->createEntity( "ParedSuperior", "cube.mesh" );
163 entityPtr->setMaterialName( "pared" ); // establece el material para la pared superior
164 entityPtr->setNormaliseNormals( true ); // corrige las normales al aplicar una escala
165
166 // crea el objeto SceneNode para la pared superior
167 nodoPtr = sceneManagerPtr->getRootSceneNode()->
168     createChildSceneNode( "ParedSuperior" );
169 nodoPtr->attachObject( entityPtr ); // adjunta la pared superior al objeto SceneNode
170 nodoPtr->setPosition( 0, 70, 0 ); // establece la posición de la pared superior
171 nodoPtr->setScale( 1.95, .05, .1 ); // establece el tamaño de la pared superior
172
173 } // fin de la función createScene
174
175 // inicia un juego de Pong
176 void Pong::ejecutar()
177 {
178     createScene(); // crea la escena
179     rootPtr->startRendering(); // empieza a visualizar los cuadros
180 } // fin de la función ejecutar
181
182 // actualiza la puntuación
183 void Pong::actualizarPuntuacion( Jugadores jugador )
184 {
185     // incrementa la puntuación del jugador correcto
186     if ( jugador == Jugadores::JUGADOR1 )
187         puntuacionJugador1++;
188     else
189         puntuacionJugador2++;
190
191     espera = true; // el juego debe esperar para reiniciar la Pelota
192     actualizarPuntuacionTexto(); // actualiza el texto de la puntuación en la pantalla
193 } // fin de la función actualizarPuntuacion
194
195 // actualiza el texto de la puntuación
196 void Pong::actualizarPuntuacionTexto()
197 {
198     ostringstream textoPuntuacion; // texto a mostrar
199
200     textoPuntuacion << "Jugador 2 Puntos: " << puntuacionJugador2; // crea el texto
201
202     // obtiene el objeto TextArea del jugador derecho
203     OverlayElement *elementoTextoPtr =
204         OverlayManager::getSingletonPtr()->getOverlayElement( "derecho" );
205     elementoTextoPtr->setCaption( textoPuntuacion.str() ); // establece el texto
206
207     textoPuntuacion.str( "" ); // restablece el texto en textoPuntuacion

```

Figura 23.5 | Definiciones de las funciones miembro de la clase Pong. (Parte 4 de 6).

```

208     textoPuntuacion << "Jugador 1 Puntos: " << puntuacionJugador1; // crea el texto
209
210     // obtiene el objeto TextArea del jugador izquierdo
211     elementoTextoPtr =
212         OverlayManager::getSingletonPtr()->getOverlayElement( "izquierdo" );
213     elementoTextoPtr->setCaption( textoPuntuacion.str() ); // establece el texto
214 } // fin de la función actualizarPuntuacionTexto
215
216 // responde a la entrada del usuario mediante el teclado
217 bool Pong::keyPressed( const OIS::KeyEvent &keyEventRef )
218 {
219     // si el juego no se pone en pausa
220     if ( !pausa )
221     {
222         // procesa los eventos KeyEvent que se aplican cuando el juego no está en pausa
223         switch ( keyEventRef.key )
224         {
225             case OIS::KC_ESCAPE: // se oprimió la tecla de escape: salir
226                 salir = true;
227                 break;
228             case OIS::KC_UP: // se oprimió la tecla flecha arriba: mueve la Perilla derecha hacia
229                 arriba
230                 perillaDerPtr->moverPerilla( PERILLA_ARRIBA );
231                 break;
232             case OIS::KC_DOWN: // se oprimió la tecla flecha abajo: mueve la Perilla derecha hacia
233                 abajo
234                 perillaDerPtr->moverPerilla( PERILLA_ABAJO );
235                 break;
236             case OIS::KC_A: // se oprimió A: mueve la Perilla izquierda hacia arriba
237                 perillaIzqPtr->moverPerilla( PERILLA_ARRIBA );
238                 break;
239             case OIS::KC_Z: // se oprimió Z: mueve la Perilla izquierda hacia abajo
240                 perillaIzqPtr->moverPerilla( PERILLA_ABAJO );
241                 break;
242             case OIS::KC_P: // se oprimió P: se pone el juego en pausa
243                 pausa = true; // establece pausa a true cuando el usuario pone en pausa el juego
244                 Overlay *pauseOverlayPtr =
245                     OverlayManager::getSingleton().getByName( "PausaOverlay" );
246                 pauseOverlayPtr->show(); // muestra el objeto Overlay de pausa
247                 break;
248         } // fin de switch
249     } // fin de if
250     else // el juego está en pausa
251     {
252         // el usuario oprimió 'R' en el teclado
253         if ( keyEventRef.key == OIS::KC_R )
254         {
255             pausa = false; // establece pausa a false cuando el usuario reanuda el juego
256             Overlay *pauseOverlayPtr =
257                 OverlayManager::getSingleton().getByName( "PausaOverlay" );
258             pauseOverlayPtr->hide(); // oculta el objeto Overlay de pausa
259         } // fin de if
260     } // fin de else
261     return true;
262 } // fin de la función keyPressed
263
264 // procesa los eventos de liberación de tecla
265 bool Pong::keyReleased( const OIS::KeyEvent &keyEventRef )
266 {
267     return true;
268 } // fin de la función keyReleased
269

```

Figura 23.5 | Definiciones de las funciones miembro de la clase Pong. (Parte 5 de 6).

```

268 // devuelve true si el programa debe visualizar el siguiente cuadro
269 bool Pong::frameEnded( const FrameEvent &frameEvent )
270 {
271     return !salir; // salir = false si el usuario no ha salido todavía
272 } // fin de la función frameEnded
273
274 // procesa la lógica del juego, devuelve true si se debe visualizar el siguiente cuadro
275 bool Pong::frameStarted( const FrameEvent &frameEvent )
276 {
277     keyboardPtr->capture(); // obtiene los eventos del teclado
278     // el juego no está en pausa y la Pelota se debe mover
279     if ( !espera && !pausa )
280     {
281         // mueve la Pelota
282         pelotaPtr->moverPelota( frameEvent.timeSinceLastFrame );
283     } // fin de if
284     // no mueve la Pelota si espera es true
285     else if ( espera )
286     {
287         // incrementa el tiempo si es menor de 4 segundos
288         if ( tiempo < 4 )
289             // agrega los segundos transcurridos desde el último cuadro
290             tiempo += frameEvent.timeSinceLastFrame;
291     } // fin de else
292     else
293     {
294         espera = false; // no debe esperar a mover la Pelota
295         tiempo = 0; // restablece la variable de control a 0
296     } // fin de else
297 } // fin de frameStarted
298
299 return !salir; // salir = false si el usuario no ha salido todavía
} // fin de la función frameStarted

```

Figura 23.5 | Definiciones de las funciones miembro de la clase Pong. (Parte 6 de 6).

Antes de poder mostrar gráficos, necesitamos inicializar la configuración del motor Ogre y crear ciertos objetos de Ogre. El cuadro de diálogo **OGRE Engine Rendering Setup** (figura 23.6) permite al usuario elegir la configuración de visualización, incluyendo el **sistema de visualización** a utilizar (Direct3D u OpenGL), la resolución, la profundidad de color, el modo de pantalla completa y demás opciones que están más allá del alcance de este capítulo. Direct3D es de uso exclusivo para Windows. OpenGL tiene soporte en todas las plataformas principales. La **resolución** se define mediante dos valores (anchura y altura), los cuales determinan el número de píxeles utilizados para dibujar la escena. Las opciones de resolución para los dos subsistemas de visualización pueden variar desde 640×400 hasta 1680×1050 o más, dependiendo del hardware que utilice el programador. Una mayor resolución producirá gráficos más detallados. Si el programador opta por desactivar el modo de pantalla completa, la resolución también determinará el tamaño de la ventana en la cual se muestra el juego. Nosotros ejecutamos el juego a una resolución de 800×600 . Una **profundidad de color** de n bits significa que se pueden mostrar en la pantalla 2^n colores. Una profundidad de color más baja hará que el programa requiera menos memoria, pero los gráficos pueden no ser tan buenos. Direct3D y OpenGL soportan profundidades de colores de 16 y 32 bits. En colores de 32 bits, sólo se utilizan 24 bits para el color; los otros ocho bits representan el valor alfa (es decir, la transparencia).

Para mostrar el cuadro de diálogo, debemos crear un **objeto Root** (figura 23.5, línea 32): el objeto de Ogre que se utiliza para iniciar el motor. El único objeto de Ogre que se puede crear antes de Root es **LogManager**, pero eso está más allá del alcance de este libro. A continuación, llamamos a la función **showConfigDialog** de la clase **Root** (línea 35) para mostrar el cuadro de diálogo. Una vez que el usuario haga clic en **Aceptar**, Ogre guarda la configuración y la utiliza como predeterminada la siguiente vez que se muestre el cuadro de diálogo. El programa debe terminar si el usuario selecciona **Cancelar**, ya que las opciones tal vez no estén configuradas en forma apropiada y se podrían producir errores. Se lanza una excepción **runtime_error** si **showConfigDialog** devuelve **false** (es decir, si el usuario seleccionó **Cancelar**).

Una vez que se hayan establecido las opciones del sistema de visualización y de la ventana, podemos crear el objeto **RenderWindow**, una ventana en la que Ogre visualizará los gráficos, mediante una llamada a la función **initialise** de la clase **Root** (línea 39). El primer parámetro (**true**) indica a Ogre que debe crear la ventana con la configuración que el



Figura 23.6 | El cuadro de diálogo OGRE Engine Rendering Setup.

usuario seleccionó en el cuadro de diálogo. Si se pasa `false` a este parámetro, el programador puede crear en forma manual la ventana posteriormente. El segundo parámetro ("Pong") es el nombre de la ventana dentro del motor y también aparecerá en la barra de título de la ventana, si no está seleccionada la opción de pantalla completa. Observe la ortografía británica de `initialise` y de la palabra "colour" en la figura 23.6, lo cual refleja el origen de Ogre en el Reino Unido.

23.4.2 Creación de una escena

Ahora que hemos inicializado Ogre y establecido una ventana para visualizar nuestros gráficos, vamos a agregar algunos objetos para crear nuestra escena: la colección de imágenes que mostraremos en la pantalla.

SceneManager

Para controlar la escena utilizamos el **objeto SceneManager** de Ogre (línea 42). El objeto **SceneManager** administra el **gráfico de escena**, una estructura de datos que contiene todos los objetos en la escena, tanto visibles como invisibles. El **objeto SceneManager** se utiliza para crear objetos que se agregarán al gráfico de escena y determina qué objetos se visualizarán. Al excluir los objetos que no se encuentran dentro de la escena visible para evitar que se visualicen (proceso conocido como **culling**), se reduce el tiempo de visualización y se incrementa el rendimiento. Esto se hace en forma automática. Mantendremos un apuntador a este objeto **SceneManager**, ya que se utilizará extensivamente a lo largo del juego.

Se han diseñado varios tipos de objetos **SceneManager** para manejar distintos tipos de escenas, como las escenas en exteriores o de paisajes expansivos. Una aplicación de Ogre puede utilizar más de un objeto **SceneManager**, por separado o al mismo tiempo. Para los fines de este capítulo, utilizamos sólo una instancia del tipo de escena genérico (**ST_GENERIC**), un objeto **SceneManager** que no está optimizado para cualquier tipo específico de escena.

Camera

Una vez que tenemos un objeto **SceneManager**, podemos empezar a construir nuestra escena. Primero agregamos un objeto **Camera**. En Ogre, un objeto **Camera** es el ojo a través del cual se ve la escena. Por lo general, una escena en 3D es demasiado grande como para mostrarla en una ventana. El objeto **Camera** ve dentro de la escena e indica a Ogre qué parte puede ver realmente el usuario. Se pueden colocar objetos **Camera** en cualquier ubicación de una escena, o se pueden adjuntar a objetos **SceneNode**; hablaremos sobre éstos en breve. Si se adjunta a un objeto **SceneNode**, el objeto **Camera** seguirá ese nodo si se desplaza dentro de la escena. Ogre soporta varios objetos **Camera** en una sola escena, pero sólo necesitamos uno.

Utilizamos el objeto **SceneManager** para crear el objeto **Camera** (línea 45), y después establecemos la posición, la orientación, las distancias de clip y el objeto **Viewport** (líneas 46 a 53). La posición es la ubicación del objeto **Camera** dentro de la escena. Posicionamos el objeto **Camera** a 200 unidades a partir del origen, a lo largo del eje **z**, en dirección hacia el

jugador. Esto coloca el objeto `Camera` lo bastante lejos del origen de la escena como para poder centrar el juego alrededor de éste. Podemos establecer todas las coordenadas `z` en 0 sin tener que cambiarlas. La orientación es la dirección en la que el objeto `Camera` está viendo. Hacemos que el objeto `Camera` vea hacia el origen (línea 47) debido a que centramos el juego alrededor de ese punto. Las distancias de clip definen qué tan cerca y qué tan lejos puede ver el objeto `Camera`. Si algo está más cerca del objeto `Camera` que la distancia de clip más cercana, o más alejado que la distancia de clip más lejana, entonces el objeto `Camera` no podrá verlo. El objeto `Viewport` es el área de la pantalla que se utiliza para mostrar lo que el objeto `Camera` puede ver. Establecemos el color de fondo del objeto `Viewport` en blanco. Observe que un objeto `Camera` puede tener más de un objeto `Viewport`, pero sólo utilizaremos uno para nuestro juego. Los objetos `Camera` pueden tener muchas otras características y funciones, pero esto es todo lo que necesitamos para nuestro juego.

Light

Uno de los aspectos más importantes de una escena en 3D es la iluminación. Ogre tiene tres tipos de objetos `Light`: `Point`, `Spot` y `Directional`. Las `luces Point` tienen una posición en el espacio e irradian luz en todas las direcciones. Las `luces Spot` tienen una posición en el espacio al igual que las luces `Point`, pero irradian luz sólo en una dirección; la fuerza de la luz se desvanece a medida que aumenta la distancia a partir del origen. Las `luces Directional` no tienen una posición en el espacio, sólo tienen una dirección en la cual brillan; se asume que la luz proviene de la misma dirección, sin importar en dónde se encuentre el jugador en la escena.

En nuestro juego utilizamos una luz `Point`. En las líneas 63 y 64 se utiliza el objeto `SceneManager` para crear el objeto `Light` y establecer su posición dentro de la escena. El argumento para la función `createLight` es el nombre mediante el cual nos referimos al objeto `Light`. Para establecer la posición del objeto `Light`, especificamos sus coordenadas `x`, `y` y `z`. Ahora nuestra escena está lista para utilizarse.

23.4.3 Agregar elementos a la escena

Como dijimos antes, Pong tiene cuatro objetos de juego principales: una pelota, dos perillas y una caja rectangular. Todos estos elementos se deben agregar a la escena para poder mostrarlos.

Agregar el objeto Pelota

La clase `Pelota` (figuras 23.7 y 23.8) representa la pelota que rebota alrededor de la pantalla. Tenemos que agregar la `Pelota` a la escena antes de poder mostrarla. La función miembro `agregarAEscena` (figura 23.8, líneas 27 a 52) crea un objeto `Entity` que representa a la `Pelota`, la agrega a la escena y crea ciertos sonidos asociados con la `Pelota`; hablaremos sobre los sonidos más adelante. Un objeto `Entity` es una instancia de una malla dentro de la escena. Una `malla` (`mesh`) es un archivo que contiene la información sobre la geometría de un modelo en 3D. Muchos objetos `Entity` pueden basarse en la misma malla, siempre y cuando cada uno de ellos tenga un nombre único. En las líneas 30 y 31 se crea el objeto `Entity`, que se referencia a través de un apuntador. El primer argumento es el nombre del objeto `Entity`. El segundo argumento, "`sphere.mesh`", es el archivo de malla que se utiliza para determinar la forma del objeto `Entity`. Utilizamos la malla de esfera que se proporciona con el SDK de Ogre. Puede encontrar esta malla en la carpeta `OgreSDK\media\models` en su computadora.

```

1 // Pelota.h
2 // Definición de la clase Pelota (representa una pelota rebotadora).
3 #ifndef PELOTA_H
4 #define PELOTA_H
5
6 #include <Ogre.h> // Definición de la clase Ogre
7 using namespace Ogre; // usa el espacio de nombres Ogre
8
9 #include <OgreAL.h> // Definición de la clase OgreAL
10
11 class Perilla; // declaración anticipada de la clase Perilla
12
13 const int RADIO = 5; // el radio de la Pelota
14
15 class Pelota
16 {
17 public:
```

Figura 23.7 | Definición de la clase `Pelota` (representa una pelota que rebota). (Parte I de 2).

```

18     Pelota( SceneManager *sceneManagerPtr ); // constructor
19     ~Pelota(); // destructor
20     void agregarAEscena(); // agrega la Pelota a la escena
21     void moverPelota( Real tiempo ); // desplaza la Pelota a través de la escena
22
23 private:
24     SceneManager *sceneManagerPtr; // apuntador al objeto SceneManager
25     SceneNode *nodoPtr; // apuntador al objeto SceneNode
26     OgreAL::SoundManager *soundManagerPtr; // apuntador al objeto SoundManager
27     OgreAL::Sound *sonidoParedPtr; // sonido que se reproduce cuando la Pelota choca en una
28     pared
29     OgreAL::Sound *sonidoPerillaPtr; // sonido que se reproduce cuando Pelota choca con una
30     Perilla
31     OgreAL::Sound *sonidoPuntoPtr; // sonido que se reproduce cuando alguien obtiene un punto
32     int velocidad; // velocidad de la Pelota
33     Vector3 direccion; // dirección de la Pelota
34
35     // funciones utilitarias private
36     void invertirDireccionHorizontal(); // cambia la dirección horizontal
37     void invertirDireccionVertical(); // cambia la dirección vertical
38     void chocarPerilla(); // controla cuando la Pelota choca con las Perillas
39 }; // fin de la clase Pelota
40
41 #endif // PELOTA_H

```

Figura 23.7 | Definición de la clase Pelota (representa una pelota que rebota). (Parte 2 de 2).

```

1 // Pelota.cpp
2 // Definiciones de las funciones miembro de la clase Pelota.
3 #include <Ogre.h> // definición de la clase Ogre
4 using namespace Ogre; // usa el espacio de nombres de Ogre
5
6 #include <OgreAL.h> // definición de la clase OgreAL
7 #include "Pelota.h" // definición de la clase Pelota
8 #include "Perilla.h" // definición de la clase Perilla
9 #include "Pong.h" // definición de la clase Pong
10
11 // constructor de Pelota
12 Pelota::Pelota( SceneManager *ptr )
13 {
14     sceneManagerPtr = ptr; // establece un apuntador al objeto SceneManager
15     soundManagerPtr = new OgreAL::SoundManager(); // crea el objeto SoundManager
16     velocidad = 100; // velocidad de la Pelota
17     direccion = Vector3( 1, -1, 0 ); // dirección de la Pelota
18 } // fin del constructor de Pelota
19
20 // destructor de Pelota
21 Pelota::~Pelota()
22 {
23     // cuerpo vacío
24 } // fin del destructor de Pelota
25
26 // agrega el objeto Pelota a la escena
27 void Pelota::agregarAEscena()
28 {
29     // crea un objeto Entity y lo adjunta a un nodo en la escena
30     Entity *entityPtr =
31         sceneManagerPtr->createEntity( "Pelota", "sphere.mesh" );
32     entityPtr->setMaterialName( "pelota" ); // establece el material para la Pelota
33     entityPtr->setNormaliseNormals( true ); // corrige las normales al aplicar escala

```

Figura 23.8 | Definiciones de las funciones miembro de la clase Pelota. (Parte I de 4).

```

34     nodoPtr = sceneManagerPtr->getRootSceneNode()->
35         createChildSceneNode( "Pelota" ); // crea un objeto SceneNode
36     nodoPtr->attachObject( entityPtr ); // adjunta el objeto Entity a SceneNode
37     nodoPtr->setScale( .05, .05, .05 ); // escala el objeto SceneNode
38
39     // adjunta sonidos a la Pelota, para que se reproduzcan desde donde está la Pelota
40     sonidoParedPtr = soundManagerPtr->
41         createSound( "wallSound", "wallSound.wav", false );
42     nodoPtr->attachObject( sonidoParedPtr ); // adjunta un sonido al objeto SceneNode
43     sonidoPerillaPtr = soundManagerPtr->
44         createSound( "paddleSound", "paddleSound.wav", false );
45     nodoPtr->attachObject( sonidoPerillaPtr ); // adjunta un sonido al objeto SceneNode
46     sonidoPuntoPtr = soundManagerPtr->
47         createSound( "cheer", "cheer.wav", false ); // crea un sonido
48
49     // adjunta el sonido de puntuación a su propio nodo centrado en ( 0, 0, 0 )
50     sceneManagerPtr->getRootSceneNode()->createChildSceneNode( "score" )->
51         attachObject( sonidoPuntoPtr );
52 } // fin de la función agregarAEscena
53
54 // mueve la Pelota por la pantalla
55 void Pelota::moverPelota( Real tiempo )
56 {
57     nodoPtr->translate( (direccion * (velocidad * tiempo)) ); // mueve la Pelota
58     Vector3 posicion = nodoPtr->getPosition(); // obtiene la nueva posición de la Pelota
59
60     // obtiene las posiciones de las cuatro paredes
61     Vector3 posicionSuperior = sceneManagerPtr->
62         getSceneNode( "ParedSuperior" )->getPosition();
63     Vector3 posicionInferior = sceneManagerPtr->
64         getSceneNode( "ParedInferior" )->getPosition();
65     Vector3 posicionIzq = sceneManagerPtr->
66         getSceneNode( "ParedIzq" )->getPosition();
67     Vector3 posicionDer = sceneManagerPtr->
68         getSceneNode( "ParedDer" )->getPosition();
69
70     const int ANCHURA_PARED = 5; // la anchura de las paredes
71
72     // comprueba si la Pelota chocó con el lado izquierdo
73     if ( (posicion.x - RADIO) <= posicionIzq.x + (ANCHURA_PARED / 2) )
74     {
75         nodoPtr->setPosition( 0, 0, 0 ); // coloca la Pelota en el centro de la pantalla
76         Pong::actualizarPuntuacion( Jugadores::JUGADOR2 ); // actualiza la puntuación
77         sonidoPuntoPtr->play(); // reproduce un sonido cuando el jugador 2 obtiene un punto
78     } // fin de if
79     // comprueba si la Pelota chocó con el lado derecho
80     else if (
81         (posicion.x + RADIO) >= posicionDer.x - (ANCHURA_PARED / 2) )
82     {
83         nodoPtr->setPosition( 0, 0, 0 ); // coloca la Pelota en el centro de la pantalla
84         Pong::actualizarPuntuacion( Jugadores::JUGADOR1 ); // actualiza la puntuación
85         sonidoPuntoPtr->play(); // reproduce un sonido cuando el jugador 1 obtiene un punto
86     } // fin de else
87     // comprueba si la Pelota chocó con la pared inferior
88     else if (
89         (posicion.y - RADIO) <= posicionInferior.y + (ANCHURA_PARED / 2) &&
90         direccion.y < 0 )
91     {
92         // coloca la Pelota en la pared inferior
93         nodoPtr->setPosition( posicion.x,
94             (posicionInferior.y + (ANCHURA_PARED / 2) + RADIO), posicion.z );
95         invertirDireccionVertical(); // hace que la Pelota empiece a moverse hacia arriba

```

Figura 23.8 | Definiciones de las funciones miembro de la clase Pelota. (Parte 2 de 4).

```

96     } // fin de else
97     // comprueba si la Pelota chocó con la pared superior
98     else if (
99         ( posicion.y + RADIO ) >= posicionSuperior.y - ( ANCHURA_PARED / 2 ) &&
100        direccion.y > 0 )
101    {
102        // coloca la Pelota en la pared superior
103        nodoPtr->setPosition( posicion.x,
104            ( posicionSuperior.y - ( ANCHURA_PARED / 2 ) - RADIO ), posicion.z );
105        invertirDireccionVertical(); // hace que la Pelota empiece a moverse hacia abajo
106    } // fin de else
107
108    chocarPerilla(); // comprueba si la Pelota chocó con una Perilla
109 } // fin de la función moverPelota
110
111 // invierte la dirección horizontal de la Pelota
112 void Pelota::invertirDireccionHorizontal()
113 {
114     direccion *= Vector3( -1, 1, 1 ); // invierte la dirección horizontal
115     sonidoPerillaPtr->play(); // reproduce el efecto de sonido "paddleSound"
116 } // fin de la función invertirDireccionHorizontal
117
118 // invierte la dirección vertical de la Pelota
119 void Pelota::invertirDireccionVertical()
120 {
121     direccion *= Vector3( 1, -1, 1 ); // invierte la dirección vertical
122     sonidoParedPtr->play(); // reproduce el efecto de sonido "wallSound"
123 } // fin de la función invertirDireccionVertical
124
125 // controla cuando la Pelota choca con la Perilla
126 void Pelota::chocarPerilla()
127 {
128     // obtiene la posición de los objetos Perilla y de la Pelota
129     Vector3 posicionPerillaIzquierda = sceneManagerPtr->
130         getSceneNode( "PerillaIzq" )->getPosition(); // Perilla izquierda
131     Vector3 posicionPerillaDerecha = sceneManagerPtr->
132         getSceneNode( "PerillaDer" )->getPosition(); // Perilla derecha
133     Vector3 posicionPelota = nodoPtr->getPosition(); // la posición de la Pelota
134
135     const int ANCHURA_PERILLA = 2; // anchura de la Perilla
136     const int ALTURA_PERILLA = 30; // altura de la Perilla
137
138     // ¿está la Pelota en el rango de la Perilla izquierda?
139     if ( ( posicionPelota.x - RADIO ) <
140         ( posicionPerillaIzquierda.x + ( ANCHURA_PERILLA / 2 ) ) )
141    {
142        // comprueba si hay colisión con la Perilla izquierda
143        if ( ( posicionPelota.y - RADIO ) <
144            ( posicionPerillaIzquierda.y + ( ALTURA_PERILLA / 2 ) ) &&
145            ( posicionPelota.y + RADIO ) >
146            ( posicionPerillaIzquierda.y - ( ALTURA_PERILLA / 2 ) ) )
147        {
148            invertirDireccionHorizontal(); // invierte la dirección de la Pelota
149
150            // coloca la Pelota en el borde de la Perilla
151            nodoPtr->setPosition(
152                ( posicionPerillaIzquierda.x + ( ANCHURA_PERILLA / 2 ) + RADIO ),
153                posicionPelota.y, posicionPelota.z );
154        } // fin de if
155    } // fin de if
156    // ¿está la Pelota en el rango de la Perilla derecha?
157    else if ( ( posicionPelota.x + RADIO ) >

```

Figura 23.8 | Definiciones de las funciones miembro de la clase Pelota. (Parte 3 de 4).

```

158     ( posicionPerillaDerecha.x - ( ANCHURA_PERILLA / 2 ) ) )
159 {
160     // comprueba si hay colisión con la Perilla derecha
161     if ( ( posicionPelota.y - RADIO ) <
162         ( posicionPerillaDerecha.y + ( ALTURA_PERILLA / 2 ) ) &&
163         ( posicionPelota.y + RADIO ) >
164         ( posicionPerillaDerecha.y - ( ALTURA_PERILLA / 2 ) ) )
165     {
166         invertirDireccionHorizontal(); // invierte la dirección de la Pelota
167
168         // coloca la Pelota en el borde de la Perilla
169         nodoPtr->setPosition(
170             ( posicionPerillaDerecha.x - ( ANCHURA_PERILLA / 2 ) - RADIO ),
171             posicionPelota.y, posicionPelota.z );
172     } // fin de if
173 } // fin de else
174 } // fin de la función chocarPerilla

```

Figura 23.8 | Definiciones de las funciones miembro de la clase Pelota. (Parte 4 de 4).

En la línea 32 se establece el material utilizado para colorear el objeto `Entity`. El argumento "pelota" es el nombre del material utilizado para colorear el objeto `Pelota`. En Ogre, un material se crea comúnmente con una secuencia de comandos (script), aunque también se puede crear directamente en el programa. Un `script` de material (figura 23.9) es un archivo de texto que Ogre utiliza para crear un material. Guarde el archivo de texto con una extensión `.material`.

Observe que la estructura del material de `pelota` tiene una apariencia similar al código de C++, con llaves que encierran cada sección. En la línea 2 indicamos que vamos a definir un `material` llamado `pelota`. Dentro del `material` definimos un objeto `technique`; es decir, cómo visualizar el objeto (líneas 5 a 15). Podemos definir varios objetos `technique` para un material, pero eso está más allá de nuestro alcance. Dentro de cada objeto `technique` se definen uno o más objetos `pass` (líneas 8 a 14). Cada objeto `pass` define un paso individual en el proceso de visualización del `material`. Utilizar varios objetos `pass` es algo que está más allá de nuestro alcance. El color se determina en el objeto `pass` mediante el establecimiento de valores de color para los distintos tipos de iluminación en la escena. Queremos que la `Pelota` sea color violeta, por lo cual establecemos los valores de color a (0.58, 0, 0.827) (líneas 11 a 13). Los números después de cada tipo de luz (ambiental, difusa y especular) representan valores de color. El cuarto número en `specular` (120) determina qué tan brillante es la `Pelota`. Puede ser cualquier valor mayor que 0. Entre mayor sea el valor, más brillante será el objeto.

Observe que "pelota" no es el nombre del archivo en el cual está definido el `material`, sino el nombre del `material` dentro de ese archivo. Un archivo de `material` puede definir varios objetos `material`. Se puede utilizar el mismo `material` para varios objetos `Entity`, pero cada `material` definido debe tener un nombre único.

Hemos creado el objeto `Entity` para nuestra `Pelota`, pero todavía no es parte de la escena. En las líneas 34 a 36 (figura 23.8) se agrega a la escena, para visualizarla en la pantalla. Utilizamos el objeto `SceneManager` para crear un

```

1 // definición del material pelota
2 material pelota
3 {
4     // define una técnica para visualizar la Pelota
5     technique
6     {
7         // visualiza la Pelota en una pasada
8         pass
9         {
10            // aplica el color violeta a la Pelota
11            ambient 0.58 0 0.827
12            diffuse 0.58 0 0.827
13            specular 0.58 0 0.827 120
14        }
15    }
16 }

```

Figura 23.9 | Secuencia de comandos (script) del material `pelota`.

objeto **SceneNode**: un objeto Node dentro del gráfico de escena que contiene información acerca de un objeto y su posición en la escena, visible o invisible. Un objeto **SceneNode** puede tener muchos objetos **SceneNode** hijos adjuntos, pero sólo puede tener un objeto **Node** padre. El argumento "Pelota" es el nombre mediante el cual se hace referencia al objeto **SceneNode** en el motor Ogre. Cada objeto **SceneNode** en el gráfico de escena debe tener un nombre único. La llamada a **getRootSceneNode** obtiene el nodo del nivel superior dentro del gráfico de escena. El **nodo raíz** es el padre de todos los demás nodos. Al crear un hijo del nodo raíz, su posición inicial es (0, 0, 0). En la línea 36 se adjunta el objeto **Entity** que representa la Pelota al objeto **SceneNode** recién creado. La Pelota es ahora parte de la escena y se visualizará. Observe que todas las funciones utilizadas para agregar la Pelota a la escena son funciones miembro del objeto **SceneManager**. Ésta es la razón por la cual el constructor de Pelota recibe un apuntador al objeto **SceneManager** como parámetro; el objeto Pelota debe ser capaz de acceder al objeto **SceneManager** para agregarse a sí mismo a la escena.

La malla de esfera que se proporciona con el SDK de Ogre tiene un radio de 100. Esto es mucho más de lo que necesitamos. En la línea 37 se modifica el tamaño del objeto **Entity** adjunto al objeto **SceneNode**, pero no afecta al tamaño de la malla real en la que se basa el objeto **Entity** del nodo. Proporcionamos un factor de escala para cada eje (*x*, *y* y *z*). Pasamos .05 como el factor de escala para los tres ejes. Al utilizar el mismo factor de escala para los tres ejes se escala la malla en forma uniforme, de manera que mantenga su figura original. La función multiplica el radio de la esfera en cada eje por el factor de escala para modificar el radio de 100 a 5. Al escalar una malla, los efectos de iluminación se distorsionan un poco. Para corregir eso, hacemos que el objeto **Entity** calcule las nuevas normales para la malla cada vez que se escala (línea 33). Una **normal** en este caso se refiere a la dirección a la que apunta cada faceta (es decir, sección pequeña) de la superficie del objeto. Si la faceta apunta hacia la luz, es más brillante. Si está apuntando hacia el lado contrario de la luz, es más oscura.

También hay una función llamada **scale** para modificar el tamaño del objeto **Entity**. La diferencia es que **scale** modifica el tamaño con base en el tamaño actual, mientras que **setScale** lo modifica con base en el tamaño original del nodo. Estas funciones también escalan a todos los hijos del objeto **SceneNode** con base en el mismo factor. Para modificar esto, hay que indicar a cada hijo del nodo padre que no deseamos que se escale cuando se escala el padre; para ello, hay que hacer una llamada a la función **setInheritScale** y pasárle el valor **false**.

Agregar los objetos Perilla

La clase **Perilla** (figuras 23.10 y 23.11) representa los objetos Perilla. Podemos agregar un objeto **Perilla** a la escena en forma muy parecida al proceso de agregar la Pelota. La función miembro **addToScene** (figura 23.11, líneas 24 a 35) utiliza las mismas cinco primeras funciones de Ogre, pero con distintos argumentos.

```

1 // Perilla.h
2 // Definición de la clase Perilla (representa una perilla en el juego).
3 #ifndef PERILLA_H
4 #define PERILLA_H
5
6 #include <Ogre.h> // Definición de la clase Ogre
7 using namespace Ogre; // usa el espacio de nombres Ogre
8
9 class Perilla
10 {
11 public:
12     // constructor
13     Perilla( SceneManager *sceneManagerPtr, String nombrePerilla,
14             int posicionX);
15     ~Perilla(); // destructor
16     void agregarAEscena(); // agrega una Perilla a la escena
17     void moverPerilla( const Vector3 &direccion ); // desplaza una Perilla
18
19 private:
20     SceneManager* sceneManagerPtr; // apuntador al objeto SceneManager
21     SceneNode *nodoPtr; // apuntador a un objeto SceneNode
22     String nombre; // nombre de la Perilla
23     int x; // coordenada x de la Perilla
24 }; // fin de la clase Perilla
25
26 #endif // PERILLA_H

```

Figura 23.10 | Definición de la clase **Paddle** (representa una perilla en el juego).

```

1 // Perilla.cpp
2 // Definición de las funciones miembro de la clase Perilla.
3 #include <Ogre.h> // definición de la clase Ogre
4 using namespace Ogre; // usa el espacio de nombres de Ogre
5
6 #include "Perilla.h" // definición de la clase Perilla
7
8 // constructor
9 Perilla::Perilla( SceneManager *ptr, String nombrePerilla,
10                  int posicionX )
11 {
12     sceneManagerPtr = ptr; // establece el apuntador al objeto SceneManager
13     nombre = nombrePerilla; // establece el nombre de la Perilla
14     x = posicionX; // establece la coordenada x de la Perilla
15 } // fin del constructor de Perilla
16
17 // destructor
18 Perilla::~Perilla()
19 {
20     // cuerpo vacío
21 } // fin del destructor predeterminado de Perilla
22
23 // agrega la Perilla a la escena
24 void Perilla::agregarAEscena()
25 {
26     Entity *entityPtr = sceneManagerPtr->
27         createEntity( nombre, "cube.mesh" ); // crea un objeto Entity
28     entityPtr->setMaterialName( "perilla" ); // establece el material de la Perilla
29     entityPtr->setNormaliseNormals( true ); // corrige las normales al aplicar una escala
30     nodoPtr = sceneManagerPtr->getRootSceneNode()->
31         createChildSceneNode( nombre ); // crea un objeto SceneNode para la Perilla
32     nodoPtr->attachObject( entityPtr ); // adjunta la Perilla al objeto SceneNode
33     nodoPtr->setScale( .02, .3, .1 ); // establece el tamaño de la Perilla
34     nodoPtr->setPosition( x, 0, 0 ); // establece la posición de la Perilla
35 } // fin de la función agregarAEscena
36
37 // desplaza la Perilla hacia arriba y hacia abajo de la pantalla
38 void Perilla::moverPerilla( const Vector3 &direccion )
39 {
40     nodoPtr->translate( direccion ); // mueve la Perilla
41     if ( nodoPtr->getPosition().y > 52.5 ) // parte superior de la caja
42         nodoPtr->setPosition( x, 52.5, 0 ); // coloca la Perilla en la parte superior de la caja
43     else if ( nodoPtr->getPosition().y < -52.5 ) // parte inferior de la caja
44         // coloca la Perilla en la parte inferior de la caja
45         nodoPtr->setPosition( x, -52.5, 0 );
46 } // fin de la función moverPerilla

```

Figura 23.11 | Definiciones de las funciones miembro de la clase Perilla.

Primero creamos un objeto **Entity** para representar la **Perilla** en la pantalla (líneas 26 y 27). Aquí utilizamos el nombre que se suministra al constructor como el nombre del objeto **Entity**. No podemos tan sólo utilizar "Perilla" de la misma forma en que utilizamos "Pelota", ya que cada objeto **Entity** debe tener un nombre único y hay dos objetos **Perilla** en el juego. Utilizamos la malla de cubo que se proporciona con el SDK de Ogre como el modelo para la **Perilla**. La malla de cubo se encuentra en la carpeta **OgreSDK\media\models**. Añadimos el color naranja oscuro a cada objeto **Perilla** con el mismo **material** (figura 23.12). Esta secuencia de comandos **material** es casi idéntica a la que se utiliza para la pelota. Las únicas diferencias son el nombre del **material** (línea 2) y los valores de color (líneas 11 a 13).

Después creamos un objeto **SceneNode** hijo del nodo raíz para que contenga los datos de la **Perilla** (figura 23.11, líneas 30 y 31). Utilizamos el nombre que se proporciona al constructor para el nombre del objeto **Node**, como hicimos para el objeto **Entity**. Esto se permite, ya que los objetos **Node** y **Entity** son de tipos separados, por lo que no hay un conflicto de nombres.

```

1 // definición del material perilla
2 material perilla
3 {
4     // define una técnica para visualizar una Perilla
5     technique
6     {
7         // visualiza una Perilla en una pasada
8         pass
9         {
10            // aplica el color naranja oscuro a la Perilla
11            ambient 1 0.549 0
12            diffuse 1 0.549 0
13            specular 1 0.549 0 120
14        }
15    }
16 }
```

Figura 23.12 | Secuencia de comandos del material de Perilla.

A continuación, adjuntamos el objeto `Entity` al nodo (línea 32) y escalamos el nodo a un tamaño apropiado (línea 33). La malla de cubo es de $100 \times 100 \times 100$, pero la escalamos a $2 \times 30 \times 10$ para que tenga un tamaño apropiado para un objeto `Perilla`. También establecemos el objeto `Entity` para recalcular sus normales (línea 29), como hicimos con la `Pelota`. La única nueva función de Ogre que utilizamos es `setPosition` (línea 34). Esta función coloca el nodo en las coordenadas dadas en la escena, relativo a su padre. No tuvimos que utilizar esta función en la clase `Pelota`, ya que queríamos que el objeto `SceneNode` de `Pelota` empezara en $(0, 0, 0)$, que es la posición predeterminada de cualquier nodo adjunto al nodo raíz. Queremos que el objeto `Perilla` se posicione en el borde de la pantalla, por lo que tenemos que desplazarlo ahí. En la línea 34, `x` es un miembro de datos de la clase `Perilla` que define la coordenada `x` de la `Perilla`.

Agregar las paredes

Ahora vamos a agregar la caja que contiene la `Pelota` rebotadora y los objetos `Perilla` que se desplazan hacia arriba y hacia abajo. Creamos esta caja en la función `createScene` de la clase `Pong` (figura 23.5, líneas 132 a 172). Utilizamos la misma malla de cubo, que se proporciona con el SDK de Ogre, para las cuatro paredes; escalamos las paredes en forma apropiada para hacer la caja y recalculamos las normales para la iluminación. Las paredes se agregan a la escena en forma similar a la `Pelota` y los objetos `Perilla`. Creamos un objeto `Entity` utilizando la malla de cubo para representar cada pared. Utilizamos un material simple para colorear todas las paredes de cyan (figura 23.13). La secuencia de comandos del material tiene la misma apariencia que las otras dos que hemos visto, sólo que el nombre y los valores de color difieren.

Ahora vamos a posicionar y escalar las paredes. Las paredes izquierda y derecha se colocan a 95 unidades a partir del origen, en la dirección `x`. Las paredes superior e inferior se colocan a 70 unidades a partir del origen, en la dirección `y`.

```

1 // definición del material pared
2 material pared
3 {
4     // define una técnica para visualizar una pared
5     technique
6     {
7         // visualiza una pared en una pasada
8         pass
9         {
10            // aplica el color cyan a la pared
11            ambient 0 0.545 0.545
12            diffuse 0 0.545 0.545
13            specular 0 0.545 0.545 120
14        }
15    }
16 }
```

Figura 23.13 | Secuencia de comandos del material de una pared.

Después, cada pared se escala al tamaño correcto. Las paredes superior e inferior se colocan a 140 unidades de distancia, en la dirección *y*. Otorgamos a cada pared una anchura de 5 unidades. Esta anchura es un valor arbitrario. Podemos modificar la anchura para hacer que el juego tenga la apariencia deseada. Si modificamos esto, también tendremos que modificar el código de detección de colisiones; veremos esto cuando hablaremos sobre la lógica de colisiones. Para que las paredes izquierda y derecha se extiendan hasta la paredes superior e inferior, deben tener 145 unidades de longitud (140 más la mitad de la anchura de cada pared). Por lo tanto, el factor de escala *x* para las paredes izquierda y derecha es de 1.45. Las paredes izquierda y derecha también reciben una anchura de 5 unidades y se posicionan a 185 unidades de distancia en dirección *x*. Para que las paredes superior e inferior se extiendan entre las paredes izquierda y derecha, deben tener 195 unidades de longitud, por lo que su factor de escala *y* es de 1.95.

Agregar el texto

Utilizamos un objeto *Overlay* de Ogre para mostrar la puntuación del juego como texto. Un objeto *Overlay* se refiere a algo que el programador desea visualizar encima de los elementos en 3D de la escena. Utilizamos objetos *Overlay* sólo para texto en este capítulo. El objeto *Overlay* se define mediante una secuencia de comandos guardada en un archivo *.overlay*.

Los objetos *Overlay* están compuestos de objetos *OverlayElement*. El primer elemento en un objeto *Overlay* debe ser un *OverlayContainer*. Un objeto *OverlayContainer* puede contener cualquier tipo de objeto *OverlayElement*. Un objeto *TextAreaOverlayElement* contiene el texto que se mostrará en la pantalla. Cada objeto en un *Overlay* tiene tres atributos principales: modo de métrica, posición y tamaño. La posición se determina con base en la esquina superior izquierda del objeto, y siempre es relativa al objeto *OverlayContainer* padre del objeto. El tamaño se determina con base en la anchura y altura. El **modo de métrica** determina la forma en que se coloca y ajusta el tamaño del objeto. El **modo de píxel** ajusta el objeto con base en la anchura y altura declarada en píxeles. El **modo relativo** coloca y ajusta el tamaño del objeto en forma relativa al tamaño del objeto *OverlayContainer* padre (o de la ventana, si es el objeto *OverlayContainer* más exterior). En modo relativo, los valores de tamaño y posición varían de 0.0 a 1.0; considérelo como un porcentaje del tamaño del objeto *OverlayElement* padre. Si coloca un elemento en (0.0, 0.0), estará en la esquina superior izquierda del elemento padre; (0.5, 0.0) sería un 50 por ciento a lo largo de la parte superior.

Para mostrar la puntuación, creamos un objeto *Overlay* (figura 23.14). En la línea 2 se asigna el nombre *Puntuacion* a este objeto *Overlay*. Un solo archivo de overlay puede contener varias definiciones *Overlay*. Ogre hace referencia a cada *Overlay* por su nombre, en vez de hacerlo por el archivo. El *orden-z* del objeto *Overlay* (línea 5) se utiliza para definir sobre qué elementos se debe visualizar este objeto *Overlay*. Al utilizar varios objetos *Overlay*, un objeto *Overlay* con un *orden-z* mayor se visualizará encima de cualquier objeto *Overlay* con un *orden-z* menor. En las líneas 8 a 58 se crea un contenedor *PanelOverlayElement* que contiene dos objetos *TextAreaOverlayElement*. El objeto *OverlayContainer* se coloca en la esquina superior izquierda de la pantalla (líneas 13 y 14) y abarca toda la anchura (línea 17). El contenedor tiene un 10 por ciento de la altura de la ventana (línea 18). El primer objeto *TextAreaOverlayElement* (líneas 21 a 38) se coloca en la parte superior del contenedor, a una distancia de un 5 por ciento del lado izquierdo, abarca la mitad de la anchura y tiene la misma altura que el contenedor (líneas 28 a 31). El otro objeto *TextAreaOverlay-*

```

1 // Un Overlay para mostrar la puntuación
2 Puntuacion
3 {
4     // establece un orden-z alto, se muestra encima de cualquier cosa con un orden-z menor
5     zorder 500
6
7     // crea un contenedor PanelOverlayElement para contener las áreas de texto
8     container Panel(PanelPuntuacion)
9     {
10        // usa el modo de métrica relativa para posicionar este contenedor en
11        // la esquina superior izquierda de la pantalla
12        metrics_mode relative
13        left 0.0
14        top 0.0
15
16        // lo hace de 1/10 de la altura y la anchura completa de la pantalla
17        width 1.0
18        height .1
19

```

Figura 23.14 | Secuencia de comandos *Overlay* para mostrar la puntuación. (Parte I de 2).

```

20 // crea un objeto TextAreaOverlayElement para mostrar la puntuación del jugador 1
21 element TextArea(izquierdo)
22 {
23     // coloca y ajusta el tamaño del elemento relativo al contenedor
24     metrics_mode relative
25
26     // lo coloca en la parte superior del contenedor, a un 5% de la izquierda y
27     // le otorga la misma altura y la mitad de la longitud del contenedor
28     left 0.05
29     top 0.0
30     width 0.5
31     height 1.0
32
33     // establece el tipo de letra a usar para la leyenda y establece el tamaño y color
34     font_name BlueBold
35     char_height .05
36     colour 1.0 0 0
37     caption Jugador 1 Puntos: 0
38 }
39 // crea un objeto TextAreaOverlayElement para mostrar la puntuación del jugador 2
40 element TextArea(derecho)
41 {
42     // coloca y ajusta el tamaño del elemento relativo al contenedor
43     metrics_mode relative
44
45     // lo coloca en la parte superior del contenedor, a un 69% de la izquierda y
46     // le otorga la misma altura que el contenedor; lo estira hasta el final
47     left 0.69
48     top 0.0
49     width 0.5
50     height 1.0
51
52     // establece el tipo de letra a usar para la leyenda y establece el tamaño y color
53     font_name BlueBold
54     char_height 0.05
55     colour 1.0 0 0
56     caption Jugador 2 Puntos: 0
57 }
58 }
59 }
```

Figura 23.14 | Secuencia de comandos Overlay para mostrar la puntuación. (Parte 2 de 2).

Element (líneas 40 a 57) se coloca a una distancia del 69 por ciento a lo largo de la parte superior del contenedor, y abarca hasta el final. Los objetos TextAreaOverlayElement también declaran un tipo de letra a usar (que se define mediante una secuencia de comandos en un archivo .fontdef de la carpeta RecursosPong), la altura de los caracteres, el color del tipo de letra (observe la ortografía británica, “colour”) y la leyenda (líneas 34 a 37 y 53 a 56).

La figura 23.15 es el archivo .fontdef que define el tipo de letra **BlueBold**. En la línea 2 se proporciona un nombre al tipo de letra, al que Ogre hará referencia. En la línea 5 se indica a Ogre qué tipo de letra es. True Type es un formato común de archivos de tipos de letra (un archivo .ttf). El origen (línea 8) es el archivo que contiene el tipo de letra. Colocamos el archivo .ttf en la misma carpeta que el archivo .fontdef. Si coloca los dos archivos en ubicaciones distintas, tendrá que especificar la ruta al archivo .ttf en la línea 8 o agregar la carpeta que contiene el archivo .ttf como una ubicación de recursos (lo cual veremos en la sección 23.4.8).

En las líneas 119 a 121 de la clase Pong (figura 23.5) se muestra la puntuación en la pantalla. Utilizamos la función miembro estática **getSingleton** de la clase **OverlayManager** para obtener un apuntador al objeto **OverlayManager**. Utilizamos ese objeto para obtener un apuntador al objeto **Overlay** de la puntuación, y después llamamos a la función **show** para mostrarlo en la pantalla. Cuando un jugador obtiene un punto, necesitamos actualizar el texto dentro del objeto **Overlay** para reflejar el cambio (líneas 196 a 214). Primero creamos el nuevo texto. Despues obtenemos un apuntador al objeto **TextAreaOverlayElement** apropiado del objeto **OverlayManager**, y utilizamos la función miembro **setCaption** de **TextAreaOverlayElement** para reemplazar el texto.

```

1 // define el tipo de letra BlueBold
2 BlueBold
3 {
4     // define el tipo del tipo de letra
5     type truetype
6
7     // establece el archivo de origen para el tipo de letra
8     source bluebold.ttf
9
10    // establece el tamaño del tipo de letra
11    size 16
12
13    // establece la resolución del tipo de letra (96 es el estándar)
14    resolution 96
15 }

```

Figura 23.15 | Secuencia de comandos de definición del tipo de letra BlueBold.

23.4.4 Animación y temporizadores

Ahora que sabemos cómo dibujar una Pelota en la pantalla, el proceso de animarla y hacer que se mueva alrededor de la pantalla es simple. La función `moverPelota` (figura 23.8, líneas 55 a 109) desplaza la Pelota alrededor de la pantalla. En la mayoría de los juegos de Pong, la pelota puede viajar a muchos ángulos distintos. No obstante, como estamos apenas empezando con Ogre, queremos mantener las cosas lo más simples que sea posible. Por esta razón, en nuestro juego de Pong la pelota sólo tiene cuatro posibles direcciones de recorrido: abajo-derecha, arriba-derecha, abajo-izquierda y arriba-izquierda; todas a 45 grados con los ejes *x* y *y* en nuestro programa.

En la línea 57 es donde realmente se hace que la Pelota se mueva; el resto de la función controla las colisiones con varios objetos dentro de la escena, como veremos en breve. La función `translate` recibe como argumento un `Vector3`, que es un tipo de vector tridimensional definido por Ogre. El vector representa la dirección y la distancia para mover la Pelota. Pasamos a la función `translate` la dirección de la Pelota multiplicada por la distancia que debe recorrer ($velocidad \times tiempo$) para determinar el vector final. El parámetro `velocidad` es el número de unidades que se moverá la pelota por segundo. El parámetro `tiempo` es el número de segundos transcurridos desde la última vez que se movió la Pelota. En unos momentos veremos de dónde proviene esto. Las traslaciones de objetos `SceneNode` se realizan en el espacio del padre de manera predeterminada. Esto significa que el nodo se mueve respecto a la posición y orientación de su nodo padre (es decir, la dirección a la que apunta el nodo). Éstas también se pueden llevar a cabo en espacio mundial o local, para lo cual se agrega otro parámetro a la función `translate` (`TS_LOCAL` o `TS_WORLD`). Las traslaciones en espacio mundial se realizan respecto al origen de la escena (0, 0, 0). Las traslaciones en espacio local se realizan respecto al origen del nodo (en cualquier parte donde esté colocado el nodo, y en cualquier dirección a la que apunte).²

Para mover en forma continua la Pelota a través de la pantalla, hay que llamar a la función `moverPelota` cada vez que se visualiza un nuevo cuadro de animación. En la figura 23.4 se define la clase `Pong`, nuestra clase controladora principal del juego, la cual hereda de la clase `FrameListener` de Ogre. `FrameListener` es una clase que procesa eventos `Ogre::FrameEvent`. Un evento `FrameEvent` ocurre cada vez que empieza o termina un cuadro de animación. Cada objeto `FrameListener` tiene dos funciones: `frameStarted` y `frameEnded` (líneas 30 y 31). Ambas funciones devuelven un valor `bool`. Ogre sigue visualizando cuadros hasta que una de estas funciones devuelve `false`. Utilizamos la función `frameStarted` (figura 23.5, líneas 275 a 299) para controlar la animación de nuestra Pelota, en específico la línea 282. Esta función es llamada por Ogre antes de visualizar cada nuevo cuadro de animación. Antes de cada cuadro, la función `frameStarted` llama a la función miembro `moverPelota` de la clase `Pelota`, que mueve en forma continua la Pelota a través de la pantalla. Como vimos antes, es imprescindible controlar la velocidad de la animación para crear un movimiento uniforme. Las velocidades de cuadro (es decir, qué tan rápido se vuelve a dibujar la escena) pueden variar considerablemente en distintos equipos, por lo que la Pelota se podría mover a una velocidad distinta en cada uno. Por esta razón pasamos el miembro de datos `timeSinceLastFrame` de `FrameEvent` (en segundos) a la función `moverPelota`. Multiplicamos este tiempo por la velocidad de la Pelota para determinar la distancia que debe moverse a través de la pantalla. Éste es un ejemplo de cómo usar un temporizador para controlar la animación.

2. Junker, Gregory, *Pro OGRE 3D Programming*, 2006, pp. 82-89.

23.4.5 Entrada del usuario

Ahora hablaremos sobre cómo mover la Perilla hacia arriba y hacia abajo con la función `moverPerilla` de la clase `Perilla` (figura 23.11, líneas 38 a 47). Para mover la Perilla, utilizamos de nuevo la función `translate` de `SceneNode` (línea 40). En vez de mover la Perilla con base en el tiempo, la movemos con base en la entrada del usuario mediante el teclado. El usuario especifica una dirección (arriba o abajo) al oprimir la tecla correspondiente, y la Perilla se mueve de manera acorde. La dirección se pasa a `moverPerilla` como un `Vector3`.

Ogre no soporta directamente la entrada del usuario de dispositivos tales como el teclado, ratón o palanca de mandos. El SDK de Ogre incluye el *Sistema de entrada orientado a objetos* (OIS) para manejar la entrada del usuario. No es obligatorio utilizar OIS para la entrada de datos con Ogre, pero es una buena elección.

Necesitamos crear un objeto `InputManager`, un objeto `Keyboard` y un objeto `KeyListener` para manejar la entrada del usuario y controlar las llamadas a `moverPerilla`. El objeto `InputManager` se utiliza para crear los diversos dispositivos de entrada. Creamos el objeto `InputManager` en el constructor de la clase `Pong` (figura 23.5, línea 75). Para crear el `InputManager` debemos proporcionarle una ventana en la que pueda recolectar (líneas 66 a 72).

Creamos un objeto `Keyboard` que representa el teclado de la computadora. Para recolectar eventos `KeyEvent`, debemos llamar a la función `capture` de la clase `Keyboard`. Queremos llamar a esta función repetidas veces, por lo cual la colocamos en la función `frameStarted`, la cual se llama al principio de cada cuadro. La clase `Pong` hereda de la clase `KeyListener`, una clase del OIS que maneja la entrada mediante el teclado. La registramos con el objeto `Keyboard` (línea 78) para recibir eventos `KeyEvent`, lo cual es una indicación de que el jugador ha oprimido una tecla. Un objeto `KeyListener` define dos funciones miembro (figura 23.4, líneas 26 y 27); sólo utilizamos una de ellas (línea 26). Debemos implementar la otra también, debido a que ambas se declaran como puras `virtual` en la clase `KeyListener`.

La implementación de la función para manejar las teclas está en las líneas 217 a 260. Cada vez que capturamos un evento de pulsación de tecla, el objeto `Keyboard` envía el evento `KeyEvent` a esta función miembro. OIS define una enumeración de todas las teclas en el teclado, la cual utilizamos para determinar cuál tecla se oprimió. La instrucción `switch` (líneas 223 a 246) responde sólo a ciertas teclas. Extraemos la numeración de la tecla del objeto `KeyEvent` y la pasamos a la instrucción `switch` (línea 223). Si se oprimen las teclas *A* o *Z*, la Perilla del lado izquierdo se debe mover hacia arriba o hacia abajo, respectivamente. De igual forma, si el usuario oprime la tecla de flecha arriba o abajo, la Perilla del lado derecho se debe mover en la dirección correspondiente. Las direcciones que se pasan a la función `moverPerilla` se definen como objetos `Vector3` constantes (líneas 26 y 27).

Permitimos al usuario poner en pausa el juego al oprimir la tecla *P* (líneas 240 a 245), con lo cual se establece el miembro de datos `pausa` de `Pong` en `true`. La instrucción `if` (línea 220) omite la instrucción `switch` que controla el movimiento de la Perilla cuando `pausa` es `true`. El miembro de datos `pausa` también detiene el movimiento de la Pelota cuando es `true` (línea 279). También utilizamos un objeto `Overlay` (figura 23.16) para mostrar "Juego en pausa" en la pantalla. El juego se reanuda cuando el jugador oprime la tecla *R*.

```

1 // Un Overlay para mostrar "Juego en pausa" cuando el jugador pone en pausa el juego
2 PausaOverlay
3 {
4     // establece un orden-z alto, se muestra encima de cualquier cosa con un orden-z menor
5     zorder 500
6
7     // crea un contenedor PanelOverlayElement para almacenar el área de texto
8     container Panel(Pausa)
9     {
10        // usa el modo de métrica relativa para posicionar y ajustar el tamaño de este contenedor
11        metrics_mode relative
12
13        // coloca el contenedor en la esquina superior izquierda de la ventana
14        left 0.0
15        top 0.0
16
17        // hace al contenedor del mismo tamaño que la ventana
18        width 1.0
19        height 1.0
20

```

Figura 23.16 | Secuencia de comandos `Overlay` para mostrar "Juego en pausa" cuando el jugador pone en pausa el juego. (Parte I de 2).

```

21     // crea un objeto TextAreaOverlayElement para mostrar el texto
22     element TextArea(textoPausa)
23     {
24         // posiciona y ajusta el tamaño del elemento relativo a su contenedor
25         metrics_mode relative
26
27         // lo centra verticalmente en el contenedor
28         vert_align center
29
30         // coloca la esquina izquierda a 4/10 de la izquierda del contenedor y
31         // lo hace 2/10 de la anchura del contenedor y 1/10 de su altura
32         left 0.4
33         width 0.2
34         height 0.1
35
36         // establece el tipo de letra a usar para la leyenda y establece el tamaño y color
37         font_name BlueBold
38         char_height 0.05
39         colour 0 0 0
40         caption Juego en pausa
41     }
42 }
43 }
```

Figura 23.16 | Secuencia de comandos Overlay para mostrar "Juego en pausa" cuando el jugador pone en pausa el juego. (Parte 2 de 2).

Si el usuario oprime la tecla *Esc*, el juego termina estableciendo el miembro de datos `salir` en `true` (figura 23.5, líneas 225 a 227). Recuerde que Ogre continúa visualizando cuadros hasta que la función `frameStarted` o la función `frameEnded` devuelven `false`. Ambas devuelven `!salir`, por lo que cuando establecemos `salir` a `true`, las funciones devuelven `false` e indican a Ogre que debe cerrar el programa. Si no utilizamos la tecla *Esc* para salir, el programa no se detendrá en forma apropiada; seguirá ejecutándose en segundo plano. Asegúrese de utilizar la tecla *Esc*.

23.4.6 Detección de colisiones

La Pelota choca con varios objetos a medida que rebota a través de la pantalla. Necesitamos detectar estas colisiones para hacer que la Pelota interactúe correctamente con los objetos que la rodean. En las líneas 61 a 106 de la figura 23.8 se controlan las colisiones entre la Pelota y las paredes del área de juego. La llamada a la función miembro `getPosition` de `SceneNode` devuelve un objeto `Vector3` que representa la posición del nodo relativa a su nodo padre. Como todos nuestros nodos son hijos directos del nodo raíz, cuya posición es `(0, 0, 0)`, la posición que se devuelve siempre es relativa al origen. También hay una función `_getDerivedPosition` que devuelve la posición relativa al origen de cualquier nodo, sin importar la posición de su padre.

Podemos obtener cualquier nodo dentro del gráfico de la escena al pasar el nombre del nodo a la función miembro `getSceneNode` de `SceneManager`. Obtenemos los nodos de las cuatro paredes (líneas 61 a 68) y utilizamos sus posiciones para comprobar las colisiones con la Pelota. Si la coordenada *x* de la Pelota (menos el radio) es menor o igual que la coordenada *x* de la pared izquierda (más la mitad de la anchura de la pared), entonces la Pelota ha chocado con la pared izquierda. Una vez que se determina la colisión, se toman las acciones apropiadas. La Pelota se coloca en la parte media de la pantalla para la siguiente ronda (línea 75). El jugador 2 recibe un punto (línea 76) al llamar a la función miembro `actualizarPuntuacion` de la clase `Pong`. Hicimos que la función `actualizarPuntuacion` sea `static` para poder llamarla desde la clase `Pelota` sin una referencia a una instancia de la clase `Pong`. Por último, se reproduce un sonido para indicar que un jugador ha obtenido un punto (línea 77); más adelante explicaremos esa llamada a la función en la sección 23.4.7. El proceso es el mismo para determinar si la Pelota chocó con el lado derecho. La coordenada *x* de la Pelota se compara con la coordenada *x* de la pared derecha. Si la Pelota choca con el lado derecho, se toman las acciones apropiadas. La figura 23.17 muestra al jugador 1 obteniendo un punto. Observe que en realidad la pelota no está entrando en la pared; es una ilusión provocada por los gráficos en 3D.

Después se revisa si la Pelota chocó contra las paredes superior e inferior. Se utiliza la misma lógica de colisiones. Si la coordenada *y* de la Pelota (más o menos el radio, dependiendo de cuál sea la pared con la que chocó) después de haberse movido cruzara la coordenada *y* interna de la pared (lo cual es físicamente imposible, ya que ambos son objetos

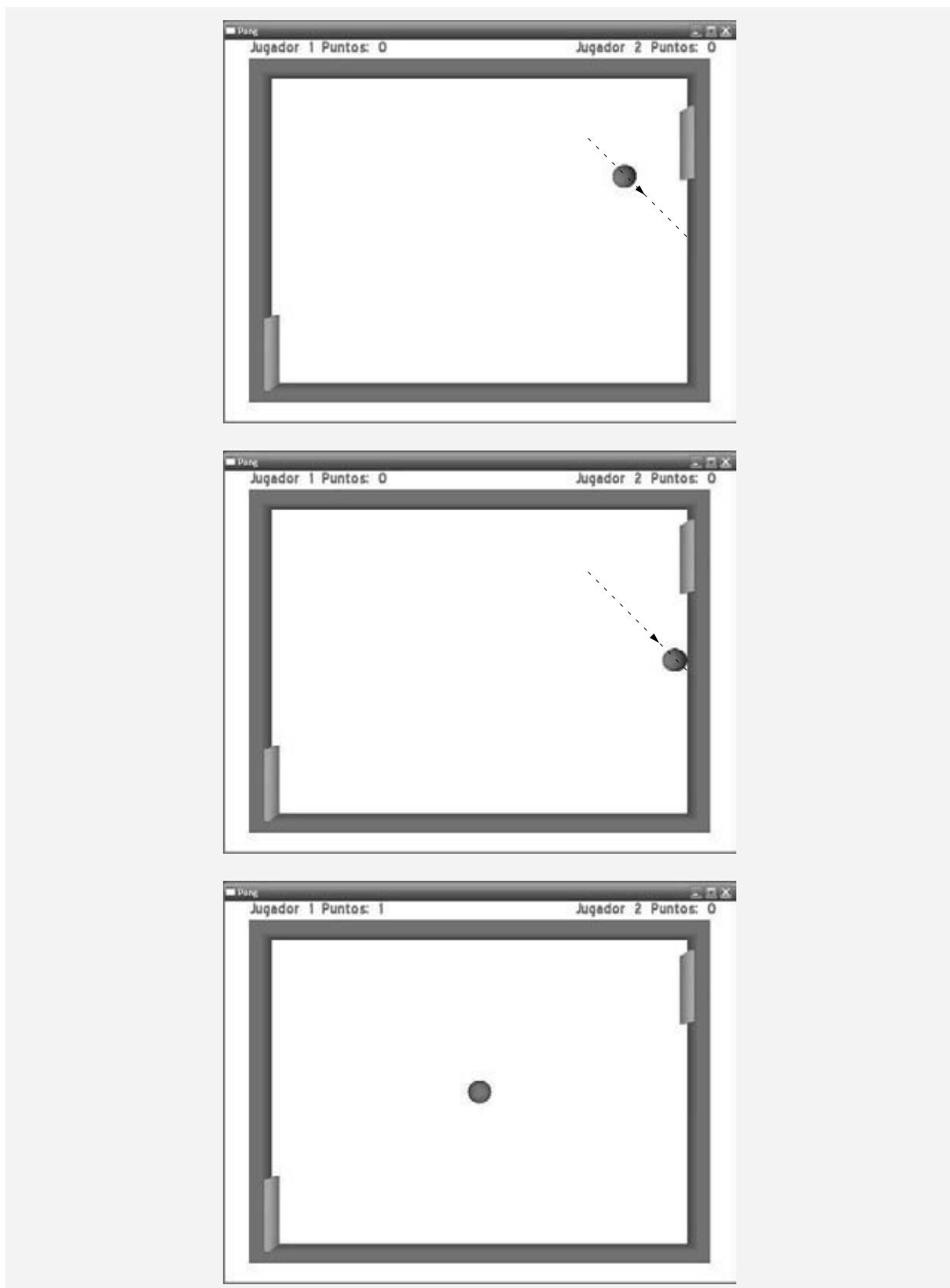


Figura 23.17 | El jugador 1 obteniendo un punto.

sólidos), entonces la Pelota ha chocado con la pared superior o la inferior. Para evitar que la Pelota se traslape con la pared, la colocamos en el borde de la pared después de una colisión (líneas 93 y 94, y líneas 103 y 104). Técnicamente, esto viola la física de la Pelota al modificar la distancia que se movió en el intervalo de tiempo dado. Para ser precisos, tendríamos que determinar la distancia y dirección en que se movió después de chocar contra la pared, y dibujarla en ese punto. Con el fin de mantener el código simple, no lidiaremos con esta cuestión. La escena se vuelve a dibujar tan rápidamente que la distancia que se mueve la Pelota en cada cuadro es extremadamente pequeña. El efecto sobre el movimiento de la Pelota es imperceptible. En la figura 23.18 se muestra cómo rebota la Pelota de la pared superior.

Al final de moverPelota llamamos a la función chocarPerilla (líneas 126 a 174), para comprobar colisiones entre la Pelota y los objetos Perilla, y tomar las acciones apropiadas cuando ocurra una. En las líneas 129 y 130 se obtiene el nodo al que está unida la Perilla izquierda, y después se devuelve la posición del nodo. En las líneas 131 y 132 se hace lo mismo para la Perilla derecha. Utilizamos estas posiciones para detectar colisiones entre los objetos Perilla y la Pelota. La lógica es similar a la que se utiliza para comprobar las paredes. Primero comprobamos si la coordenada x de la Pelota está más allá de la de los objetos Perilla. También comprobamos que la Pelota esté dentro de las coordenadas y de los objetos Perilla. En la figura 23.19 se muestra cómo rebota la Pelota de una Perilla.

Considere las líneas que modifican la dirección de la Pelota. La línea 114 hace que la Pelota empiece a moverse a la izquierda si actualmente se mueve a la derecha, y que empiece a moverse a la derecha si actualmente se mueve a la

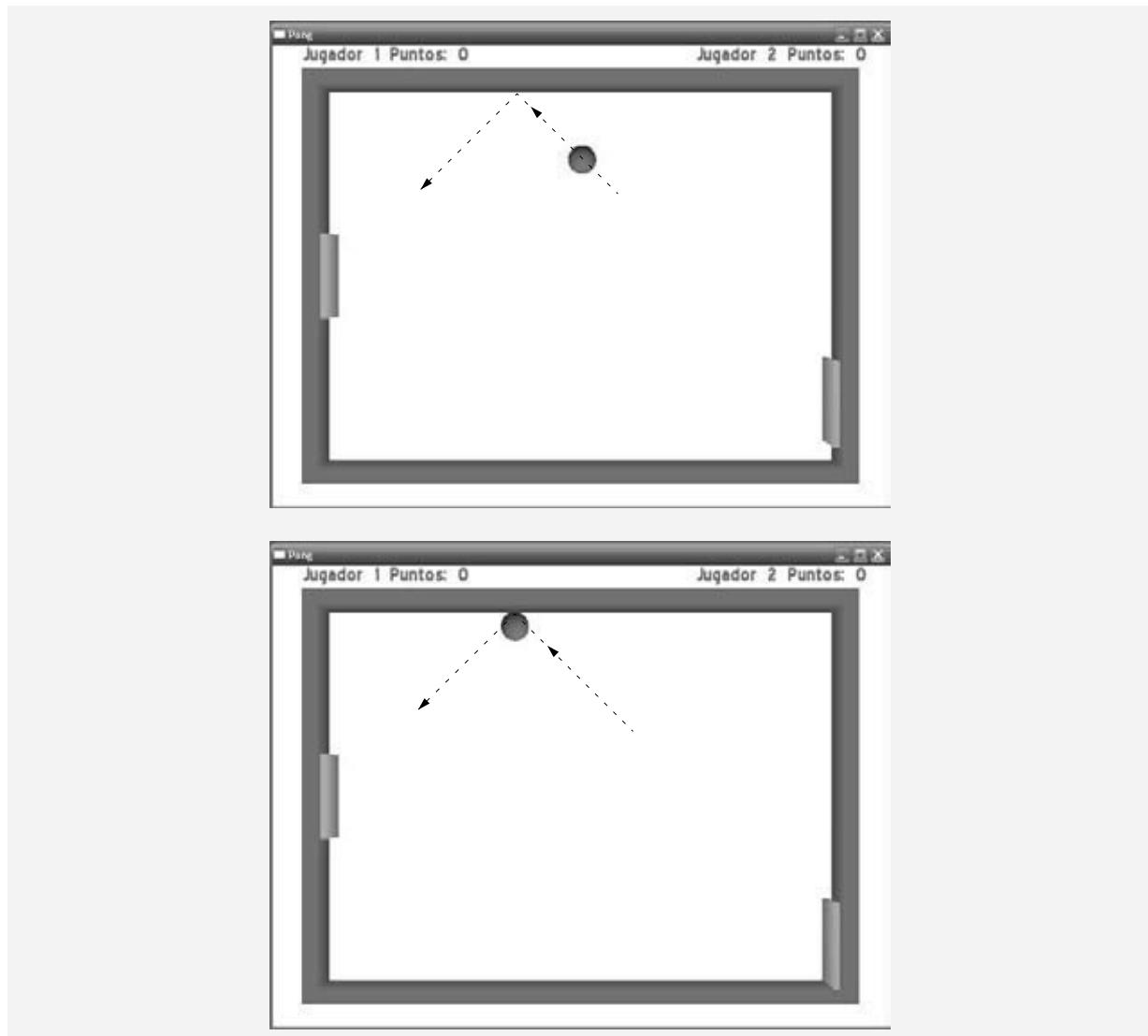


Figura 23.18 | La Pelota rebotando de la pared superior. (Parte I de 2).

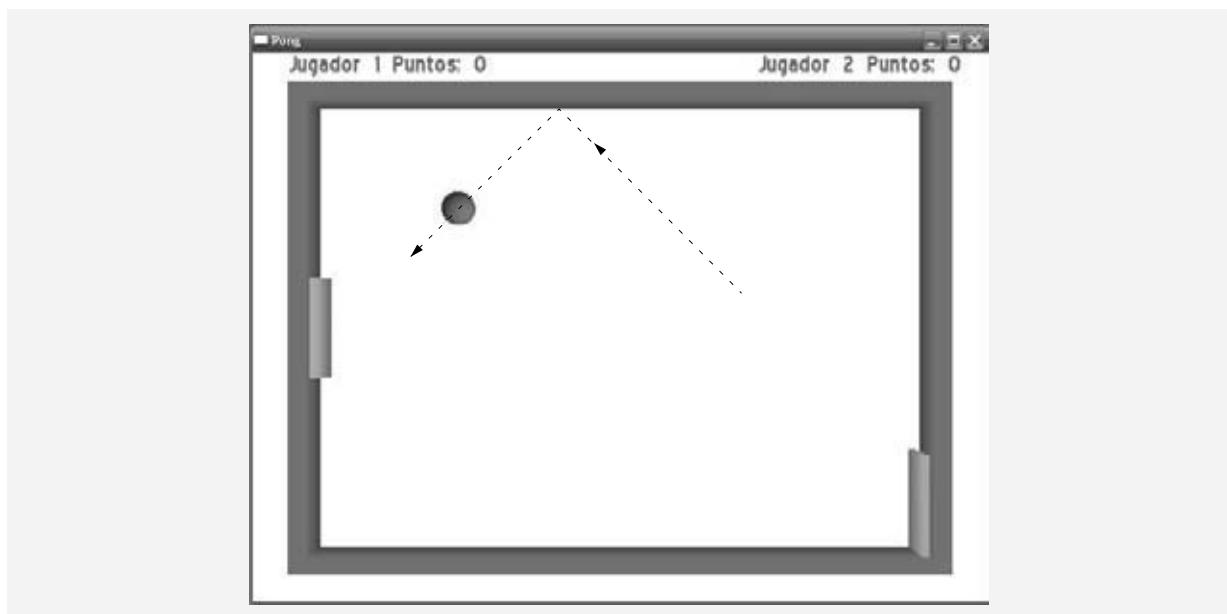


Figura 23.18 | La Pelota rebotando de la pared superior. (Parte 2 de 2).

izquierda. La línea 121 hace que la Pelota se mueva hacia arriba si actualmente se mueve hacia abajo, y que se mueva hacia abajo si actualmente se mueve hacia arriba. ¿Por qué funciona esto? La dirección de la Pelota se determina mediante un Vector3. Cada valor representa una distancia a lo largo de los ejes x , y o z . Un valor de x positivo significa que la Pelota se moverá a la derecha por el eje x , y un valor negativo moverá la Pelota a la izquierda. Si la Pelota se mueve a la derecha, al multiplicar su valor de x por -1 se cambiará el signo y se invertirá la dirección. Hacemos lo mismo para cambiar la dirección vertical.

Las colisiones en nuestro juego son casos bastante simples, por lo que hemos mantenido la lógica simple. Hay bibliotecas completas dedicadas al manejo de colisiones y la física, como Open Dynamics Engine (ODE, www.ode.org/), Bullet (www.continuousphysics.com/Bullet/), Newton Game Dynamics (www.newtondynamics.com/) y PhysX (www.ageia.com/). Estas bibliotecas tienen vinculaciones con Ogre disponibles en la página de complementos de la comunidad de Ogre (Ogre Community Add-ons).

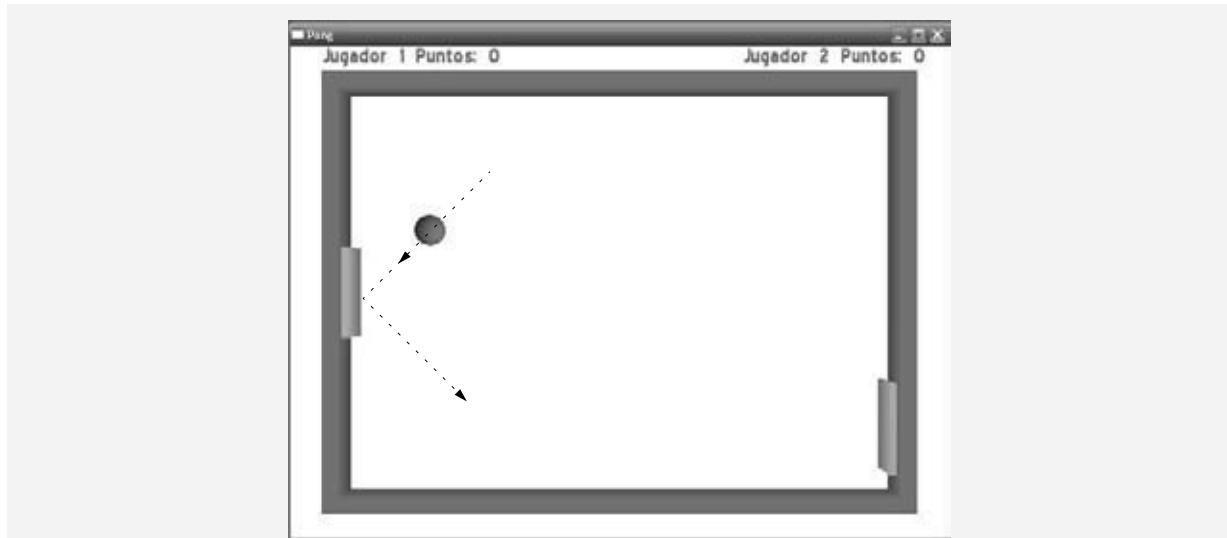


Figura 23.19 | La Pelota rebotando de la Perilla izquierda. (Parte 1 de 2).

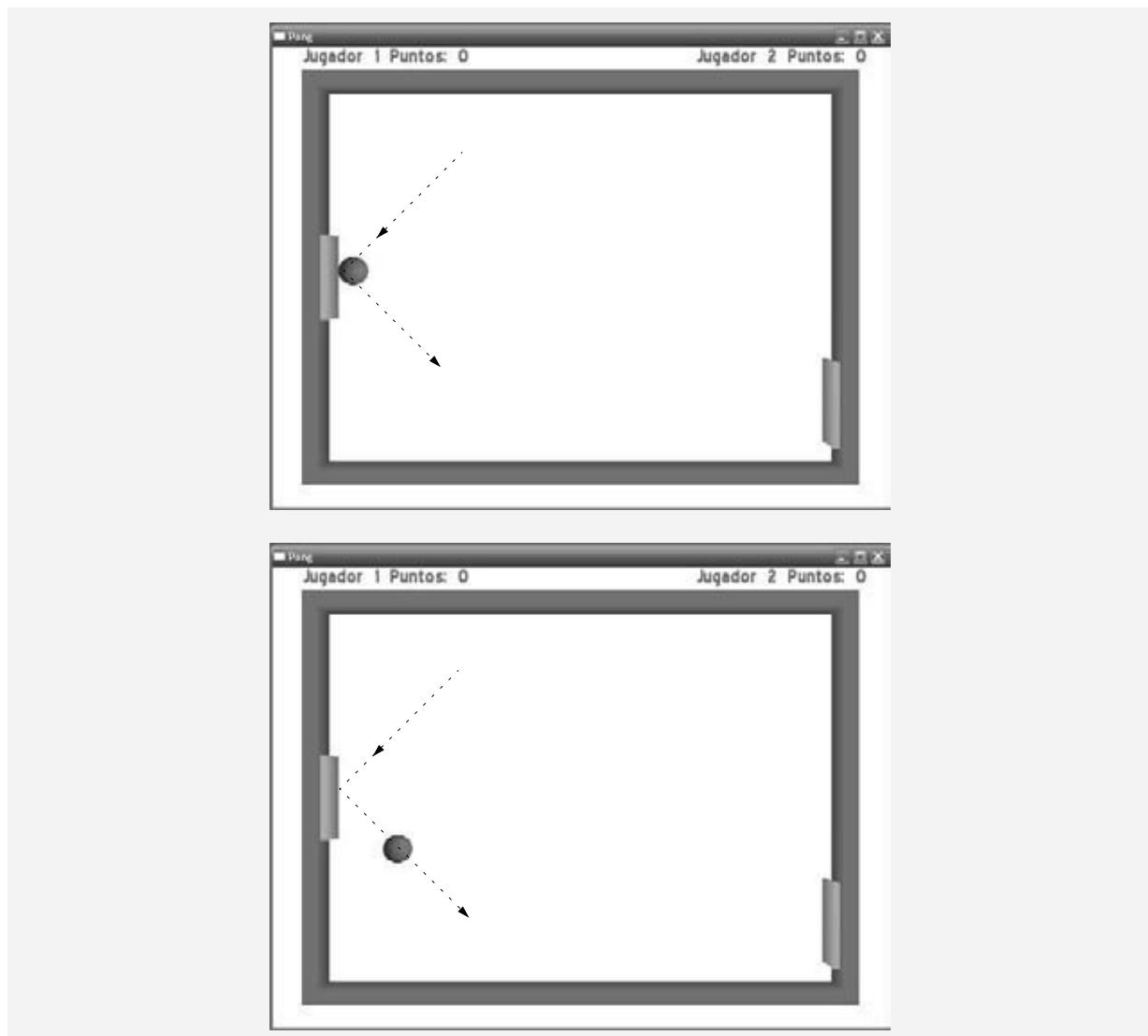


Figura 23.19 | La Pelota rebotando de la Perilla izquierda. (Parte 2 de 2).

23.4.7 Sonido

Ahora hablaremos sobre cómo importar sonidos y reproducir archivos de sonido en programas de Ogre, lo cual utilizaremos para mejorar nuestro juego de Pong. Reproducimos un sonido “boing” cada vez que la Pelota choca contra una pared, reproducimos un sonido “boing” distinto cada vez que la Pelota choca con una Perilla y reproducimos un sonido de vitoreo cada vez que un jugador obtiene un punto.

Utilizaremos OgreAL para agregar sonido a nuestro juego. **OgreAL** es una envoltura alrededor de la biblioteca de audio **OpenAL**. OgreAL fue creada y es mantenida por Casey Borders (www.mooproductions.org), un miembro de la comunidad de Ogre. La biblioteca OpenAL es mantenida por Creative Labs, developer.creative.com. La envoltura nos permite integrar la funcionalidad del sonido al código de Ogre, al adjuntar los sonidos a nodos dentro del gráfico de la escena. Debido a que todos los sonidos que reproducimos están relacionados con la Pelota, colocamos el código de OgreAL en la clase Pelota. Las funciones de OgreAL que se utilizan para importar y reproducir sonidos son análogas a las que se utilizan para importar y mostrar modelos en Ogre.

Al igual que con Ogre, incluimos el encabezado **OgreAL.h**. OgreAL administra los sonidos usando una clase **SoundManager**, que cumple con el esquema de administración de recursos de Ogre. Creamos un objeto **SoundManager** en el constructor de **Pelota** (figura 23.8, línea 15). Sólo puede haber un objeto **SoundManager**. Éste se utiliza para crear instancias de objetos **Sound**, los objetos de OgreAL que contienen los datos de sonido. Creamos tres objetos **Sound** y

los adjuntamos a los nodos (líneas 40 a 51). La función `createSound` recibe tres parámetros. El primero es un objeto `Ogre::String` que será el nombre del objeto `Sound`. El segundo es el nombre del archivo de sonido asociado con el objeto `Sound`. El tercero es un valor `bool` que determina si el objeto `Sound` se debe poner en ciclo para continuar reproduciéndolo. Si se pasa `false`, se reproducirá el objeto `Sound` una vez y luego se detendrá. Al pasar `true` el sonido se reproducirá continuamente hasta que el usuario lo detenga. Adjuntamos los objetos `Sound` a un nodo en la misma forma que adjuntaríamos un objeto `Entity` a un nodo con la función `attachToObject`.

El primer objeto `Sound` que creamos (líneas 40 y 41) se reproducirá cada vez que la `Pelota` rebote de la pared superior o inferior. Lo adjuntamos al nodo de la `Pelota`. OpenAL soporta sonido en 3D, por lo que cuando se adjunta el objeto `Sound` a la `Pelota`, se reproducirá desde cualquier parte en la que se encuentre la `Pelota`. Nuestra escena es relativamente pequeña, por lo que tal vez el usuario no detecte que el objeto `Sound` se reproduce en 3D, pero si escucha con cuidado, sonará un poco diferente. Como colocamos la llamada a la función `play` (línea 122) dentro de nuestra función que invierte la dirección vertical de la `Pelota`, el sonido “boing” se reproducirá cada vez que se invierta la dirección vertical de la misma; en otras palabras, cada vez que la pelota choque con la pared superior o inferior.

El segundo objeto `Sound` se crea de la misma forma que el primero, y de nuevo se adjunta al nodo de la `Pelota`. Este sonido se reproducirá cada vez que la `Pelota` rebote de uno de los objetos `Perilla`. Reproducimos este objeto `Sound` dentro de la función `invertirDireccionHorizontal` (línea 115) por la misma razón que reproducimos el otro sonido de `invertirDireccionVertical`.

El tercer sonido se reproducirá cada vez que un jugador obtenga un punto. No hay una ubicación específica desde la que se reproduzca este objeto `Sound`; lo adjuntamos directamente al nodo raíz del gráfico de la escena, que está colocado en el origen. Reproducimos el objeto `Sound` desde la función `moverPelota` cada vez que se determina que un jugador ha obtenido un punto (líneas 77 y 85).

Hay varias cosas que tener en cuenta acerca de los objetos `Sound` en OgreAL. Cada objeto `Sound` debe tener un nombre único, de igual forma que los objetos `Entity` y `Node`. Un objeto `Sound` debe terminar de reproducirse antes de que pueda volver a reproducirse.

23.4.8 Recursos

Como dijimos antes, Ogre utiliza secuencias de comandos para crear objetos `Material`, `Overlay` y algunas otras características avanzadas que están más allá del alcance de este libro. Ogre también utiliza archivos `.mesh` para representar objetos en 3D. OgreAL utiliza archivos de sonido. Todos estos recursos se deben cargar antes de poder utilizarlos. Ogre lanzará una excepción en tiempo de ejecución si tratamos de usar un recurso que no se haya cargado. Para administrar los recursos del juego utilizamos un objeto `ResourceGroupManager`. Para cargar los recursos de nuestro juego, primero indicamos al objeto `ResourceGroupManager` en donde puede encontrarlos. La función `addResourceLocation` (figura 23.5, líneas 83 y 84) recibe tres argumentos `Ogre::String`. El primero es la ubicación de los recursos. Colocamos todos los recursos en una carpeta llamada `PongResources` dentro de la carpeta `media` del SDK de Ogre. Por lo general, los recursos se organizan en distintas carpetas según su tipo; por ejemplo, materiales, modelos y overlays. Pero por cuestión de simpleza, mantenemos una carpeta en la que se guardan todos los recursos que necesitaremos para el juego. El segundo argumento es el tipo de archivo en el que se encuentran los recursos. El tercero es el grupo de recursos al que pertenecen estos archivos. Colocaremos estos archivos en el grupo “Pong”. Ahora cargamos los recursos en la ubicación que acabamos de agregar (línea 83).

23.4.9 Controlador de Pong

El último paso es escribir una función `main` (figura 23.20). Ogre soporta varias plataformas, por lo que no debemos tratar de escribir código específico para una plataforma cuando podamos evitarlo. La envoltura `if else` del preprocesador (líneas 6 a 15) determinará si el programa se ejecuta en una plataforma Windows. Si es así, incluirá el encabezado `windows.h` y definirá la función `WinMain`. Si no es así, definirá la función `main` normal. Esto permite que el código se ejecute en varias plataformas sin necesidad de modificarlo. Tal vez usted no haya visto el código específico para Windows antes. La directiva del preprocesador para incluir el encabezado `windows.h` proporciona al programa el acceso necesario a la API de Windows para ejecutar nuestro programa. La definición de `WIN32_LEAN_AND_MEAN` (línea 7) excluirá los encabezados que se utilizan pocas veces en el encabezado `windows.h`. Esto agilizará el tiempo de compilación para nuestro programa.

```

1 // PongMain.cpp
2 // Programa controlador para el juego de Pong
3 #include <iostream>
4

```

Figura 23.20 | Programa controlador para el juego de Pong. (Parte I de 2).

```

5 #include "Pong.h" // definición de la clase Pong
6
7 // Si se ejecuta en Windows, incluye windows.h y define la función WinMain
8 #if OGRE_PLATFORM==PLATFORM_WIN32 || OGRE_PLATFORM==OGRE_PLATFORM_WIN32
9 #define WIN32_LEAN_AND_MEAN
10 #include "windows.h"
11
12 int WINAPI WinMain( HINSTANCE hInst, HINSTANCE, LPSTR strCmdLine, INT )
13
14 // Si no, define la función main normal
15 #else
16 int main( int argc, char **argv )
17 #endif
18 {
19     try
20     {
21         Pong game; // crea un objeto Pong
22         game.ejecutar(); // inicia el juego de Pong
23     } // fin de try
24
25     catch ( std::runtime_error &error )
26     {
27 #if OGRE_PLATFORM==PLATFORM_WIN32 || OGRE_PLATFORM==OGRE_PLATFORM_WIN32
28         MessageBoxA( NULL, error.what(), "Se lanza una excepcion!",
29                     MB_OK | MB_ICONERROR | MB_TASKMODAL );
30 #else
31         std::cerr << "Se lanza excepcion: " << error.what() << std::endl;
32 #endif
33     } // fin de catch
34
35     return 0;
36 } // fin de main

```

Figura 23.20 | Programa controlador para el juego de Pong. (Parte 2 de 2).

La función `main` crea el objeto `Pong` inicial en un bloque `try` (líneas 17 a 21). Recuerde que el constructor de `Pong` lanza una excepción si el usuario cancela el cuadro de diálogo de configuración de Ogre. Si el usuario hace clic en **Aceptar** en el cuadro de diálogo, el objeto `Pong` se crea y llamamos a la función miembro `run` (línea 20). La función miembro `run` de la clase `Pong` (figura 23.5, líneas 176 a 180) crea primero la escena del juego (línea 178) y después llama a la función miembro `startRendering` (línea 179) de la clase `Root` para visualizar la escena repetidas veces, hasta que una de las funciones `frameStarted` o `frameEnded` devuelva `false`.

23.5 Repaso

En este capítulo aprendió acerca de los fundamentos de la creación de juegos de computadora con Ogre. Hablamos sobre los conceptos básicos de los gráficos, describimos brevemente los modelos, la iluminación y los colores. Vio cómo utilizar el motor de visualización gratuito Ogre3D para producir un juego en 3D. Le mostramos cómo utilizar el objeto `SceneManager` para crear y administrar su escena. Aprendió a utilizar un objeto `Camera` para ver su escena. Hablamos sobre cómo responder a la entrada del usuario mediante el teclado con OIS. Demostramos cómo mover un objeto a una velocidad constante. Cubrimos los fundamentos de la detección de colisiones y mostramos qué tan importante es para la programación de juegos. Aprendió a mostrar texto en la pantalla usando objetos `Overlay`. Mostramos cómo utiliza Ogre las secuencias de comandos para administrar los materiales y objetos `Overlay` sin tener que recompilar cada vez que se modifican. También mostramos cómo agregar sonido a sus juegos usando la envoltura OgreAL para OpenAL.

Este capítulo debe verse sólo como una introducción. Aquí le presentamos un ejemplo básico de Pong. Úselo como la base para su propia versión. Vaya y encuentre sus propios sonidos que pueda utilizar. Agregue nuevas características al juego. Explore las demás herramientas de Ogre y cree algunos efectos visuales impresionantes. Haga que este juego sea realmente suyo. La programación de juegos consiste fundamentalmente en la creatividad.

El siguiente capítulo habla sobre el futuro de C++. Un nuevo estándar, conocido como C++0x, saldrá en 2009. Usted aprenderá sobre las nuevas bibliotecas y características básicas del lenguaje que se van a agregar a C++. También

aprenderá acerca de las Bibliotecas Boost, en las cuales se basan muchas de las bibliotecas que se van a agregar a C++0x. En el siguiente capítulo le demostraremos cómo utilizar dos de las nuevas bibliotecas para trabajar con expresiones regulares y apuntadores inteligentes.

23.6 Recursos Web de Ogre

www.ogre3d.org/

La página inicial de Ogre. Aquí puede encontrar las noticias más recientes sobre Ogre, descargar este software o herramientas relacionadas con el mismo, explorar la documentación o ver proyectos que utilizan Ogre.

www.ogre3d.org/index.php?option=com_content&task=view&id=411&Itemid=131

La página de descarga del SDK previamente generado. Hay SDKs disponibles para Code::Blocks + MinGW C++ Toolbox, Visual C++ .Net 2003 y Visual C++ .Net 2005 (debe instalar el Service Pack 1).

www.ogre3d.org/index.php?option=com_content&task=view&id=412&Itemid=132

La página de descarga del código fuente de Ogre. Hay código fuente disponible para Windows, Linux y Mac OS X. También puede descargar el paquete de dependencias de terceros para su plataforma. Además hay un vínculo a una guía para generar Ogre a partir del código fuente.

www.ogre3d.org/index.php?option=com_content&task=view&id=415&Itemid=144

Instrucciones acerca de cómo obtener el código fuente de Ogre mediante el directorio CVS.

www.ogre3d.org/wiki/index.php/Installing_An_SDK

Instrucciones de instalación para el SDK de Ogre en Windows con Visual C++, Code::Blocks + MinGW, Code::Blocks + MinGW + STLPort, Eclipse + MinGW + STLPort y GCC & Make/Any IDE. Linux, Debian, Gentoo, Fedora y Ubuntu. Mac OS X.

www.ogre3d.org/wiki/index.php/Building_From_Source

Instrucciones para generar el código fuente de Ogre en Windows con Visual C++, Visual C++ Toolkit 2003 & Code::Blocks, y GCC. Linux con GCC & Make, Debian, Fedora, Gentoo, Ubuntu/Kubuntu. Mac OS X con Xcode.

www.ogre3d.org/wiki/index.php/BuildFAQ

Soluciones a errores comunes al generar del código fuente. Los errores incluyen el no poder buscar archivos, símbolos externos sin resolver y otros tipos de errores.

www.ogre3d.org/wiki/index.php/SettingUpAnApplication

Guía para configurar un proyecto de Aplicación Ogre en Visual C++, Code::Blocks, GCC, Autotools, Scons, Eclipse, Anjuta IDE, KDevelop IDE.

www.ogre3d.org/phpBB2addons/viewtopic.php?t=3293

Instrucciones para descargar e instalar OgreAL.

developer.creative.com/landing.asp?cat=1&sbcat=31&top=38

Vínculos para descargar e instalar OpenAL.

[www.openal.org/downloads.html](http://openal.org/downloads.html)

Página de descarga de OpenAL.

www.wreckedgames.com/wiki/index.php/WreckedLibs:OIS

La página wiki del Sistema de entrada orientado a objetos (OIS) incluye vínculos para el manual de OIS y la referencia a la API.

www.tayloredmktg.com/rgb/

Tabla de códigos de colores. Proporciona los valores RGB en hexadecimal y decimal. Los colores se dividen en un rango de colores general (por ejemplo, grises, azules, verdes, naranjas).

www.htmlcenter.com/tutorials/tutorials.cfm/89/General/

Una tabla de colores que proporciona valores de colores RGB y hexadecimales.

Tutoriales

www.ogre3d.org/wiki/index.php/Ogre_Tutorials

La página de tutoriales de Ogre. Los tutoriales varían desde básicos hasta avanzados, sobre temas que incluyen una introducción a Ogre, a los objetos FrameListener, la animación de varios objetos SceneManager y la creación de contenido.

www.blender.org/tutorials-help/

Página de tutoriales de Blender.

en.wikibooks.org/wiki/Blender_3D:_Noob_to_Pro

El libro wiki “Blender 3D:Noob to Pro” guía a los nuevos usuarios de Blender a través del proceso del modelado en 3D. Este libro enseña cómo trabajar con modelos, iluminación, visualización, animación, partículas y cuerpos suaves. También cuenta con tutoriales avanzados sobre secuencias de comandos de python y animación avanzada.

www.cegui.org.uk/wiki/index.php/Tutorials

Muchos tutoriales acerca de cómo usar el sistema de GUI Crazy Eddie (CEGUI) que soporta Ogre.

Herramientas

www.ogre3d.org/index.php?option=com_content&task=view&id=413&Itemid=133

Aquí se pueden descargar herramientas de exportación de modelos de Blender, SoftImage XSI y 3DS Max.

usa.autodesk.com/adsk/servlet/index?siteID=123112&id=7639525

Página del software Autodesk Maya Learning Edition. Versión gratuita de Maya.

www.softimage.com/downloads/default.aspx

Página de descarga del software SoftImage XSI. Hay una versión sin costo de 30 días disponible.

www.blender.org/download/get-blender/

Página de descarga de Blender.

Ejemplos de código

www.ogre3d.org/wiki/index.php/CodeSnippets#HOWTO

El Libro de recetas (Cookbook) de Ogre contiene ejemplos de código que explican cómo realizar varias tareas relacionadas con la geometría, la visualización, los materiales, las texturas, la animación, la GUI de entrada y el sonido.

www.ogre3d.org/phpBB2/viewtopic.php?t=27326

Asteroid Wars. Un juego que se escribió utilizando Ogre para los gráficos. El código fuente está disponible.

www.ogre3d.org/phpBB2/viewtopic.php?t=27806

Cinco juegos escritos con Ogre. Está disponible el código fuente para todos los juegos.

Libros

www.amazon.com/Pro-OGRE-3D-Programming/dp/1590597109/ref=pd_bbs_sr_1/102-2583408-2260151?ie=UTF8&s=books&qid=1173888297&sr=1-1

Pro OGRE 3D Programming, por Gregory Junker.

Foros

www.ogre3d.org/phpBB2/viewtopic.php?t=5706

Un mensaje en el foro que describe cómo instalar Ogre en Debian GNU/Linux.

www.ogre3d.org/phpBB2/viewforum.php?f=2

Foro de ayuda de Ogre. Aquí puede obtener ayuda de los usuarios de Ogre sobre cualquier problema que encuentre al utilizar Ogre.

www.ogre3d.org/phpBB2/

Varios foros en el sitio de Ogre, incluyendo la ayuda, el uso de Ogre en la práctica, la creación de contenido, fundamentos de programación y mucho más.

www.ogre3d.org/phpBB2addons/

Foros de los complementos de Ogre. Varios foros dedicados a los complementos más populares de Ogre, incluyendo OgreAL, OgreODE, NxOgre, PyOgre y mucho más.

www.ogre3d.org/phpBB2addons/viewform.php?f=10

Foro de OgreAL en el sitio de Ogre. Información acerca de cómo instalar y utilizar OgreAL. Excelente sitio para buscar ayuda.

www.wreckedgames.com/forum/viewforum.php?f=6&sid=dc5f903554a80ac5194213329f5e46e4

Foro de OIS. Obtenga ayuda acerca del uso de OIS.

Resumen

Sección 23.3 Fundamentos de la programación de juegos

- Los motores de gráficos en 3D ocultan la programación tediosa y compleja requerida con las API de gráficos.
- Ogre soporta las API de gráficos Direct3D y OpenGL, y se ejecuta en las plataformas Windows, Linux y Mac.

- Ogre es estrictamente un motor de visualización de gráficos. La comunidad de Ogre ha producido muchos complementos que permiten a los usuarios integrar otras bibliotecas con Ogre para dar soporte a esas características.
- Un modelo en 3D es una representación computacional de un objeto que se puede dibujar en la pantalla.
- Los materiales determinan la apariencia de un objeto al establecer las propiedades de iluminación, colores y texturas.
- Una textura es una imagen que está envuelta alrededor del modelo.
- Los colores se determinan mediante las intensidades de rojo, verde y azul, y un valor alfa opcional para representar la transparencia. Los valores pueden variar de 0 a 1.0.
- Hay cuatro tipos distintos de luz en una escena en 3D: ambiental, difusa, emisiva y especular.
- La detección de colisiones es el proceso de determinar si dos objetos en un juego se están tocando y reaccionar de forma apropiada.
- Hay bibliotecas de detección de colisiones y modelado de propiedades físicas que se encargan de las complejidades por el programador.
- Las bibliotecas de audio enriquecen nuestros juegos con sonido. Muchas de esas bibliotecas soportan sonido en 3D.
- A menudo los juegos se comunican con el usuario mostrando texto.
- Los temporizadores controlan la velocidad de animación y hacen que las animaciones parezcan más naturales.
- Los dispositivos de entrada del usuario incluyen el teclado, ratón, palanca de mandos y controlador de juegos.

Sección 23.4.1 Inicialización de Ogre

- Root es el objeto base que se utiliza en Ogre para iniciar el motor. No se pueden hacer llamadas a Ogre hasta que se haya creado el objeto Root.
- Llame a la función `showConfigDialog` de la clase `Root` para mostrar el cuadro de diálogo. El cuadro de diálogo **OGRE Engine Rendering Setup** permite al usuario elegir las opciones de visualización.
- La resolución se define mediante dos valores, anchura y altura, que determinan el número de píxeles utilizados para dibujar la escena. Una resolución más alta producirá gráficos más detallados.
- Una profundidad de color de n bits indica que se pueden mostrar 2^n posibles colores en la pantalla.
- El objeto `RenderWindow` es una ventana en la que Ogre visualiza los gráficos.

Sección 23.4.2 Creación de una escena

- Una escena es una colección de imágenes que conforman nuestros gráficos.
- El objeto `SceneManager` administra el gráfico de la escena, una estructura de datos que contiene todos los objetos de la escena.
- El objeto `SceneManager` se utiliza para crear objetos y determinar qué objetos se van a visualizar. Una aplicación de Ogre puede utilizar más de un objeto `SceneManager`.
- Un objeto `Camera` es el ojo a través del cual se ve la escena. Pueden colocarse objetos `Camera` en cualquier ubicación en la escena, o adjuntarse a objetos `SceneNode`. Ogre soporta varios objetos `Camera` en una sola escena.
- El objeto `Viewport` es el área de la pantalla que se utiliza para mostrar lo que el objeto `Camera` puede ver. Un objeto `Camera` puede tener más de un objeto `Viewport`.
- Ogre tiene tres tipos de objetos `Light`: `Point`, `Spot` y `Directional`. Los objetos `Light` se crean con la función `createLight` de la clase `SceneManager`.

Sección 23.4.3 Agregar elementos a la escena

- Un objeto `Entity` es una instancia de una malla dentro de la escena. Una malla es un archivo que contiene la información geométrica de un modelo en 3D. Muchos objetos `Entity` se pueden basar en la misma malla, siempre y cuando cada uno tenga un nombre único.
- Utilice el objeto `SceneManager` para crear objetos `SceneNode` que contienen información acerca de un objeto y su posición en la escena.
- El nodo raíz es el padre de todos los demás nodos. Al crear un hijo del nodo raíz, su posición inicial es (0, 0, 0).
- Para adjuntar objetos `Entity` a objetos `SceneNode` se utiliza la función `attachObject` de la clase `SceneNode`.
- `scale` cambia el tamaño del objeto `Entity` adjunto al objeto `SceneNode`, pero no afecta el tamaño de la malla actual en la que se basa el objeto `Entity` del nodo. `setScale` cambia el tamaño con base en el tamaño original del objeto `Entity`. Estas funciones también escalan a los hijos del objeto `SceneMode` por el mismo factor. Para modificar eso, hay que llamar a la función `setInheritScale` y pasarle `false`.
- La función `setPosition` coloca el nodo en las coordenadas proporcionadas en la escena.
- Ogre utiliza una secuencia de comandos de `material` para crear un `material`. Guarde el archivo con una extensión `.material`. Un archivo `material` puede definir varios materiales; cada `material` debe tener un nombre único.
- Un objeto `Overlay` se define mediante una secuencia de comandos que se guarda en un archivo `.overlay`. Un solo archivo `.overlay` puede guardar varias definiciones de objetos `Overlay`. Cada objeto en un `Overlay` tiene tres atributos principales: modo de métrica, posición y tamaño.

- Los objetos `Overlay` están compuestos de objetos `OverlayElement`. El primer elemento en un `Overlay` debe ser un `OverlayContainer`. Un objeto `OverlayContainer` puede contener cualquier objeto `OverlayElement`. Un objeto `TextAreaOverlayElement` contiene texto. Llame a la función `show` para mostrar el objeto `Overlay` en la pantalla.
- Use un objeto `TextAreaOverlayElement` para mostrar texto. Llame a `setCaption` para modificar el texto en la pantalla.
- Un objeto `Overlay` con un orden-z mayor se visualizará en la parte superior de un objeto `Overlay` con un orden-z menor.
- Los tipos de letra se definen mediante una secuencia de comandos en un archivo `fontdef`.
- Use la función miembro estática `getSingleton` de la clase `OverlayManager` para obtener el objeto `OverlayManager`.

Sección 23.4.4 Animación y temporizadores

- La función `translate` mueve un objeto `SceneNode`.
- Las traslaciones de objetos `SceneNode` se realizan en el espacio del padre de manera predeterminada. Las traslaciones en el espacio del padre se realizan respecto al origen del padre. Las traslaciones en el espacio de palabra se realizan respecto al origen de la escena (0, 0, 0). Las traslaciones en el espacio local se realizan respecto al origen del nodo.
- Un objeto `FrameListener` procesa eventos `Ogre::FrameEvent`. Este tipo de eventos ocurre cada vez que empieza o termina un cuadro de animación.

Sección 23.4.5 Entrada del usuario

- Ogre no soporta directamente la entrada del usuario desde dispositivos tales como el teclado, ratón o palanca de mandos.
- Utilice el Sistema de entrada orientado a objetos (OIS) para manejar la entrada del usuario.
- El objeto `InputManager` se utiliza para crear los diversos dispositivos de entrada. Para crear el objeto `InputManager` debemos proporcionarle una ventana en la que pueda recolectar la entrada.
- Un objeto `Keyboard` recolecta eventos `KeyEvent` y los envía a un objeto `KeyListener`.
- OIS define una enumeración de todas las teclas en el teclado, que utilizamos para determinar qué tecla se oprimió.

Sección 23.4.6 Detección de colisiones

- `getPosition` devuelve un objeto `vector3` que representa la posición del nodo relativa a su nodo padre; `_getDerivedPosition` devuelve la posición relativa al origen.
- El objeto `SceneManager` puede obtener cualquier nodo dentro del gráfico de escena, para lo cual hace referencia al nombre que se proporcionó al nodo al momento de crearlo.
- La dirección de la `Pelota` se determina mediante un objeto `Vector3`. Un valor *x* positivo indica que la `Pelota` se desplazará a la derecha a lo largo del eje *x*, y un valor negativo desplazará a la `Pelota` a la izquierda. Si la `pelota` se está moviendo a la derecha, al multiplicar su valor *x* por -1 se cambiará el signo y se invertirá la dirección.
- Hay bibliotecas completas dedicadas al manejo de colisiones y las propiedades físicas.

Sección 23.4.7 Sonido

- OgreAL es una envoltura alrededor de la biblioteca de audio OpenAL. Esta envoltura nos permite integrar la funcionalidad del sonido al código de Ogre, al adjuntar los sonidos a nodos dentro del gráfico de la escena.
- Debemos tener la directiva del preprocesador para incluir el encabezado `OgreAL.h`.
- Sound es el objeto de OgreAL que contiene los datos del sonido. Use la función `createSound` de la clase `SoundManager` para crear sonidos. Sólo puede haber un objeto `SoundManager`.
- La función `createSound` recibe tres parámetros. El primero es un objeto `Ogre::String` que será el nombre del objeto `Sound` dentro del sistema OgreAL. El segundo es el nombre del archivo de sonido asociado con el objeto `Sound`. El tercero es un valor `bool` que determina si el objeto `Sound` debe configurarse para reproducirse en forma continua. Al pasar `false` se reproducirá el objeto `Sound` sólo una vez, y después se detendrá. Al pasar `true` se reproducirá continuamente el sonido hasta que el programador lo detenga.
- Adjunte los objetos `Sound` a un nodo con la función `attachObject`.
- Cada objeto `Sound` debe tener un nombre único.
- Un objeto `Sound` debe terminar de reproducirse antes de que se pueda volver a reproducir.

Sección 23.4.8 Recursos

- Todos los recursos deben cargarse antes de poder utilizarlos.
- Use un objeto `ResourceGroupManager` para administrar los recursos del juego.
- La función `addResourceLocation` recibe tres argumentos `Ogre::String`. El primero es la ubicación de los recursos. El segundo es el tipo de archivo en el que se encuentran los recursos. El tercero es el grupo de recursos al que pertenecen estos archivos.

Sección 23.4.9 Controlador de Pong

- Ogre soporta varias plataformas, por lo que no debemos tratar de escribir código específico para una plataforma si podemos evitarlo.

Terminología

animación	nodo raíz
<code>attachObject</code> , función de la clase <code>SceneNode</code>	normal
<code>Camera</code> , clase	Ogre
canal alfa	Ogre Application Wizard (asistente de aplicaciones Ogre)
color	OgreAL
complementos	OIS (Sistema de entrada orientado a objetos)
<code>createSound</code> , función de la clase <code>SoundManager</code>	OpenAL, biblioteca de audio
cuadro	OpenGL
culling	orden-z
detección de colisiones	<code>Overlay</code> , clase
Direct3D	<code>OverlayContainer</code> , clase
<code>Directional</code> , luces	<code>OverlayElement</code> , clase
<code>Entity</code> , clase	<code>OverlayManager</code> , clase
entrada del usuario	<code>PanelOverlayElement</code> , clase
escena	Pongmodo relativo
espacio del padre	profundidad de color
espacio local	<code>RenderWindow</code> , clase
espacio mundial	Resolución
<code>EventProcessor</code> , clase	<code>ResourceGroupManager</code> , clase
factor de escala	<code>Root</code> , clase
<code>FrameEvent</code> , clase	<code>scale</code> , función de la clase <code>SceneNode</code>
<code>FrameListener</code> , clase	<code>SceneManager</code> , clase
<code>getPosition</code> , función de la clase <code>SceneNode</code>	<code>SceneNode</code> , clase
<code>getRootSceneNode</code> , función de la clase <code>SceneManager</code>	secuencias de comandos
<code>getWorldPosition</code> , función de la clase <code>SceneNode</code>	<code>setCaption</code> , función de la clase <code>TextAreaOverlayElement</code>
gráfico de la escena	<code>setInheritScale</code> , función de la clase <code>SceneNode</code>
gráficos	<code>setPosition</code> , función de la clase <code>SceneNode</code>
herramienta de modelado en 3D	<code>setScale</code> , función de la clase <code>SceneNode</code>
iluminación	<code>showConfigDialog</code> , función de la clase <code>Root</code>
<code>initialise</code> , función de la clase <code>Root</code>	sonido
<code>InputManager</code> , clase	sonido en 3D
<code>Keyboard</code> , clase	<code>Sound</code> , clase
<code>KeyEvent</code> , clase	<code>SoundManager</code> , clase
<code>KeyListener</code> , clase	<code>Spot</code> , luz
<code>Light</code> , clase	subsistema de visualización
LoD (nivel de detalle)	temporizador
luz ambiental	<code>TextAreaOverlayElement</code> , clase
luz difusa	texto
luz emisiva	textura
luz especular	<code>timeSinceLastFrame</code>
malla	<code>translate</code> , función de la clase <code>SceneNode</code>
material	<code>Vector3</code>
modelo en 3D	<code>Viewport</code> , clase
modo de métrica	visualizar
modo de píxeles	<code>WIN32_LEAN_AND_MEAN</code>
Point, luz	<code>WinMain</code>
motores gráficos en 3D	
Node, clase	

Ejercicios de autoevaluación

23.1 Complete los siguientes enunciados:

- El encabezado _____ incluye los archivos de encabezado de Ogre que se utilizan con más frecuencia.
- El objeto _____ debe crearse antes de llamar a cualquier otra función de Ogre (que no sea para registro).
- El tipo principal definido por OgreAL para apuntar a los datos de archivos de sonido es _____.
- Un objeto _____ se utiliza para representar un color en Ogre.
- El encabezado _____ incluye los archivos de encabezado de OgreAL que se utilizan con más frecuencia.
- Las _____ se utilizan para definir materiales y overlays para los programas de Ogre.

- g) El objeto _____ se utiliza para cargar los recursos para los programas de Ogre.
- h) Ogre utiliza un objeto _____ para administrar la escena.
- i) Un modelo en 3D se define en un archivo _____ de Ogre.
- 23.2** Indique si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.
- Las coordenadas (0, 0) se refieren a la esquina inferior izquierda de un objeto `OverlayContainer`.
 - Si Ogre trata de cargar un archivo externo que no exista, se producirá un error en tiempo de ejecución.
 - Los valores de color en Ogre varían entre 0 y 255.
 - Al pasar un valor de `false` a la función `createSound`, el archivo de sonido tendrá que reproducirse en forma continua.
 - Un objeto `Overlay` que dibuja texto en la pantalla debe especificar un tipo de letra en el que se debe dibujar ese texto.
 - Cada objeto `Entity` debe tener un nombre único.
- 23.3** Escriba instrucciones para realizar cada una de las siguientes acciones:
- Adjuntar un apuntador `Entity` llamado `entityPtr` a un apuntador `SceneNode` llamado `nodoPtr`.
 - Escalar el objeto `Entity` de la pregunta anterior a la mitad de su tamaño original.
 - Crear el objeto `Sound` llamado `muestra` que reproduzca en forma continua el archivo `.wav` de sonido.
 - Si se está oprimiendo la barra espaciadora, establecer el valor de la variable `int` llamada `numero` en 0.
 - Establecer un objeto `OverlayElement` para que se posicione en forma relativa al tamaño de su objeto `Container` padre.
 - Agregar una carpeta llamada `sonidos` en la carpeta `media` del SDK de Ogre como una ubicación de recursos "General".
 - Desplazar un objeto `SceneNode` 15 unidades a la izquierda, 4 unidades hacia arriba y 8 unidades hacia usted.
- 23.4** Busque el error en cada una de las siguientes instrucciones:
- `SceneNode nodo;`
 - `ColourValue(0, 0, 255);`
 - `Root *rootPtr = new Root();`
`rootPtr->initialize(true, "Ventana");`
 - `viewportPtr = sceneManagerPtr->addViewport(cameraPtr);`

Respuestas a los ejercicios de autoevaluación

- 23.1** a) `Ogre.h`. b) `Root`. c) `Sound`. d) `ColourValue`. e) `OgreAL.h`. f) secuencias de comandos. g) `ResourceGroupManager` o `ResourceManager`. h) `SceneManager`. i) `.mesh`.
- 23.2** a) Falso. Las coordenadas (0, 0) se refieren a la esquina superior izquierda de un objeto `OverlayContainer`.
- b) Verdadero.
- c) Falso. Los valores de color en Ogre varían de 0.0 a 1.0.
- d) Falso. El sonido se reproducirá una vez y luego se detendrá.
- e) Verdadero.
- f) Verdadero.
- 23.3** a) `nodoPtr->attachObject(entityPtr);`
- b) `nodoPtr->setScale(.5, .5, .5);`
- c) `soundManagerPtr->createSound("muestra", "sonido.wav", true);`
- d) `if (keyEvent.key == OIS::KC_SPACE)`
`numero = 0;`
- e) `metrics_mode relative;`
- f) `ResourceGroupManager::getSingleton().addResourceLocation("../../../../media/sonidos", "FileSystem", "General");`
- g) `sceneNodePtr->translate(-15, 4, 8);`
- 23.4** a) La variable `nodo` debe declararse como apuntador a un objeto `SceneNode`. Todas las funciones de la clase `SceneNode` de Ogre reciben un apuntador como parámetro, o devuelven un apuntador.
- b) El objeto `ColourValue` puede aceptar parámetros solamente con valores entre 0 y 1.
- c) Ogre utiliza la ortografía británica, la función se escribe como `initialise`. Además, las opciones de visualización deben establecerse antes de poder llamar a `initialise`.
- d) `addViewport` es una función de la clase `RenderWindow`, no de `SceneManager`.

Ejercicios

23.5 Explore los recursos disponibles en nuestro Centro de recursos de programación de juegos en www.deitel.com/computergames/gameprogramming/ y el Centro de recursos de programación de juegos en C++ en www.deitel.com/CplusplusGameProgramming/.

23.6 Modifique el juego de Pong, de manera que cuando un jugador llegue a 21 puntos, el juego termine y muestre un mensaje que indique que ganó el jugador izquierdo o derecho.

23.7 En la mayoría de los juegos de Pong, cuando una competencia entre los dos jugadores dura mucho tiempo, la pelota empieza a aumentar su velocidad para poder evitar un empate indefinido. Modifique el juego de Pong de manera que la velocidad de la pelota aumente por cada diez veces que se le pegue en una competencia. Cuando cualquiera de los jugadores obtenga un punto, la pelota deberá regresar a su velocidad original.

23.8 Algunos juegos de Pong también modifican la velocidad de una o ambas perillas de los jugadores, en un esfuerzo por mantener el juego balanceado. Modifique el juego de Pong, de manera que cuando un jugador tenga ventaja de por lo menos 5 puntos, su perilla empiece a disminuir su velocidad. Entre mayor sea la ventaja de ese jugador, más lenta deberá moverse su perilla. Si la ventaja del jugador se reduce a menos de 5 puntos, su perilla deberá regresar a la velocidad normal.

23.9 Modifique el juego de Pong, de manera que antes de que empiece el juego aparezca un menú en la pantalla que permita a los jugadores elegir distintas velocidades para la pelota y las perillas.

23.10 Escriba un programa que dibuje la malla `sphere.mesh` en el centro de la pantalla. Cuando el usuario oprima una de las teclas de flecha, la malla deberá moverse diez unidades en esa dirección.

23.11 Modifique el programa del ejercicio 23.10, de manera que si el usuario mantiene oprimida una tecla de flecha, la esfera se mueva sólo una vez cada segundo.

23.12 (*El juego de la serpiente*). El objeto del juego de la serpiente es maniobrar a la serpiente a través del área de juego, tratando de comer trozos de alimento. La serpiente se representa mediante una cadena de esferas contiguas en el área de juego, que es una rejilla bidimensional. La serpiente se puede mover hacia arriba, hacia abajo, a la izquierda o a la derecha. Si la serpiente come un trozo de alimento (que se muestra mediante la "F"), crece agregando otra esfera al final (figura 23.21). Si la serpiente choca con una pared del área de juego (es decir, quedaría fuera del arreglo), el jugador pierde (figura 23.22). Si la serpiente choca contra sí misma, el jugador pierde (figura 23.23).

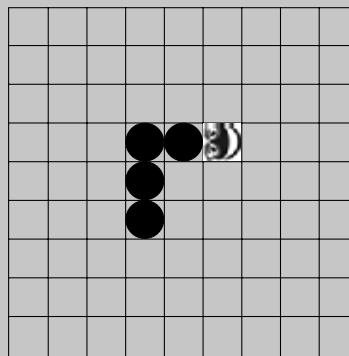
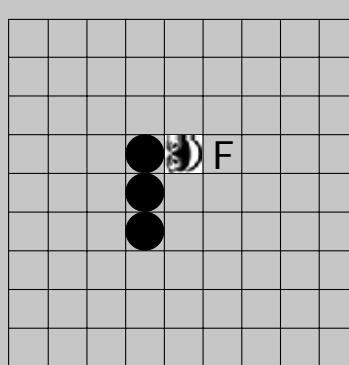


Figura 23.21 | La serpiente crece cuando come.

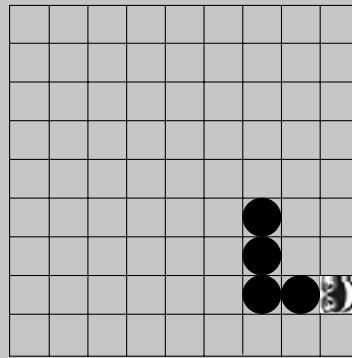


Figura 23.22 | La serpiente muere si choca con una pared.

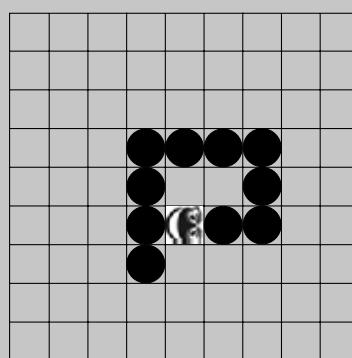


Figura 23.23 | La serpiente muere si choca contra sí misma.

23.13 Modifique el programa del ejercicio 23.12 para agregar obstáculos al área de juego (figura 23.24). Si la serpiente choca contra un obstáculo, el jugador pierde.

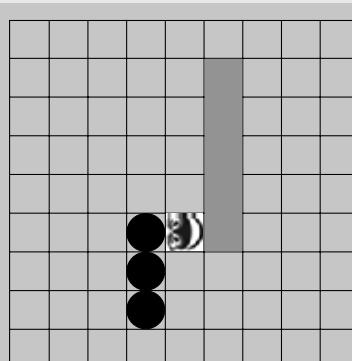


Figura 23.24 | La serpiente muere si choca con un obstáculo.



Los líderes sobresalientes se extralimitan por impulsar la autoestima de su personal.

—Sam Walton

La práctica y el pensamiento podrían gradualmente forjar muchas artes.

—Virgilio

Pienso que “Sin comentarios” es una expresión espléndida.

—Sir Winston Spencer Churchill

Mientras que estés seguro, contarás con muchos amigos.

—Ovidio

El peligro de las computadoras no es que en un momento dado se vuelvan tan inteligentes como los hombres, sino que mientras eso pase estaremos de acuerdo en bajarnos a su nivel.

—Bernard Avishai

Bibliotecas Boost, Reporte técnico I y C++0x

OBJETIVOS

En este capítulo aprenderá a:

- Evaluar el futuro de C++.
- Utilizar las Bibliotecas Boost.
- Obtener información acerca del proyecto de código fuente abierto Boost, cómo se agregan nuevas bibliotecas a Boost, y cómo instalar Boost.
- Utilizar `Boost.Regex` para realizar búsquedas en cadenas, validar datos y reemplazar partes de cadenas mediante el uso de expresiones regulares.
- Evitar fugas de memoria mediante el uso de `Boost.Smart_ptr` para administrar la asignación y desasignación dinámica de memoria.
- Identificar las bibliotecas Boost (y otras) que se incluyen en el Reporte técnico I (TRI): una descripción de las adiciones a la Biblioteca estándar de C++.
- Identificar las modificaciones al lenguaje básico y la Biblioteca estándar que se incluyen en el nuevo Estándar de C++: C++0x.
- Seguir los Centros de recursos de C++ en línea de Deitel para obtener actualizaciones acerca de la evolución de Boost, los Reportes técnicos y C++0x.

- 24.1** Introducción
- 24.2** Centros de recursos de C++ (y relacionados) en línea de Deitel
- 24.3** Bibliotecas Boost
- 24.4** Cómo agregar una nueva biblioteca a Boost
- 24.5** Instalación de las Bibliotecas Boost
- 24.6** Las Bibliotecas Boost en el Reporte Técnico I (TRI)
- 24.7** Uso de expresiones regulares con la biblioteca `Boost.Regex`
 - 24.7.1** Ejemplo de una expresión regular
 - 24.7.2** Cómo validar la entrada del usuario mediante expresiones regulares
 - 24.7.3** Cómo reemplazar y dividir cadenas
- 24.8** Apuntadores inteligentes con `Boost.Smart_ptr`
 - 24.8.1** Uso de `shared_ptr` y conteo de referencias
 - 24.8.2** `weak_ptr`: observador de `shared_ptr`
- 24.9** Reporte técnico I
- 24.10** C++0x
- 24.11** Cambios en el lenguaje básico
- 24.12** Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)

24.1 Introducción

En este capítulo consideraremos el futuro de C++. Presentamos las Bibliotecas Boost de C++, el Reporte técnico 1 (TR1) y C++0x. Las **Bibliotecas Boost de C++** son bibliotecas gratuitas de código fuente abierto, creadas por miembros de la comunidad de C++. Boost proporciona a los programadores de C++ bibliotecas útiles y bien diseñadas, que funcionan bien con la Biblioteca estándar de C++ existente. Las bibliotecas Boost pueden ser utilizadas por los programadores de C++ que trabajan en una amplia variedad de plataformas, con muchos compiladores distintos. Aquí veremos las generalidades acerca de las bibliotecas incluidas en el TR1 y proporcionaremos ejemplos de código para las bibliotecas de expresiones regulares y apuntadores inteligentes. El **Reporte técnico 1** describe los cambios propuestos a la Biblioteca estándar de C++, muchos de los cuales están basados en las bibliotecas Boost actuales. Estas bibliotecas agregan una funcionalidad útil a C++. **C++0x** es el nombre funcional para la siguiente versión del Estándar de C++. Incluye algunas adiciones al lenguaje básico, muchas de las adiciones de bibliotecas descritas en el TR1 y otras mejoras a la biblioteca.

24.2 Centros de recursos de C++ (y relacionados) en línea de Deitel

Con frecuencia publicamos Centros de recursos en línea acerca de programación clave, software, Web 2.0 y temas de negocios en Internet en www.deitel.com/resourcecenters.html. C++0x no se ha completado todavía, y se agregan con frecuencia nuevas bibliotecas a Boost, con lo cual queda en un estado de flujo. Hemos creado varios Centros de recursos en línea que proporcionan vínculos a información clave sobre cada uno de estos temas. Visite el Centro de recursos de las Bibliotecas Boost de C++ en www.deitel.com/CPlusPlusBoostLibraries/ para encontrar información actual acerca de las bibliotecas disponibles y nuevas versiones. Puede encontrar información actual acerca del TR1 y C++0x en la sección C++0x del Centro de recursos de C++ en www.deitel.com/cplusplus/ (haga clic en **C++0x** en la lista **Categories**). Utilizamos Visual C++ 2005 Express Edition para compilar los ejemplos de código en este capítulo; para obtener más información acerca de Visual C++, visite nuestro Centro de recursos de Visual C++ en www.deitel.com/VisualCPlusPlus/.

24.3 Bibliotecas Boost

La idea de un repositorio en línea de bibliotecas de C++ gratuitas de código fuente abierto se propuso por primera vez en un artículo escrito por Beman Dawes en 1998.¹ Él y Robert Klarer obtuvieron la idea mientras asistían a una reunión del Comité de Estándares de C++. El artículo sugería un sitio Web en el que los programadores de C++ pudieran buscar y compartir bibliotecas y patrocinar el posterior desarrollo de C++. Esa idea se desarrolló eventualmente para convertirse

1. “Proposal for a C++ Library Repository Web Site”, Beman G. Dawes, mayo 6, 1998, www.boost.org/more/proposal.pdf.

en las Bibliotecas Boost en www.boost.org. Boost ha crecido hasta tener 70 bibliotecas, y se agregan más con frecuencia. Hoy en día hay miles de programadores en la comunidad Boost.

Al momento de escribir este libro, el código fuente abierto para Boost se hospeda en SourceForge (sourceforge.net): el repositorio de código fuente abierto más grande del mundo, el cual hospeda más de 140,000 proyectos de código fuente abierto. SourceForge proporciona servicios de desarrollo de proyectos gratuitos, incluyendo el control de revisiones tal como Subversion (SVN), servicios de descarga y hospedaje de sitios Web para los proyectos. SourceForge es propiedad del Grupo de tecnología de código fuente abierto (OSTG; www.ostg.com), el cual posee y opera muchos sitios Web de tecnología populares, incluyendo slashdot.org, www.linux.com, www.newsforge.com y www.devchannel.com. Para obtener información acerca del software de código fuente abierto y su desarrollo, visite el Centro de recursos de código fuente abierto en www.deitel.com/OpenSource/.

24.4 Cómo agregar una nueva biblioteca a Boost

Boost acepta bibliotecas portables útiles y bien diseñadas, de cualquier persona que deseé contribuir. Las bibliotecas Boost potenciales deben conformarse al Estándar de C++ y usar la Biblioteca Estándar de C++ (u otras bibliotecas Boost apropiadas). Hay un proceso de aceptación formal para asegurar que las bibliotecas cumplan con los altos estándares de calidad y portabilidad de Boost.

El interés de la comunidad en una biblioteca se determina mediante el proceso de publicar mensajes en listas de correo y leer las respuestas. Si hay interés en una biblioteca, se publica una versión preliminar de la biblioteca en **Boost Sandbox** (svn.boost.org/svn/boost/sandbox/): un repositorio de código para bibliotecas que se encuentran en desarrollo. Sandbox permite a otros usuarios experimentar con la biblioteca y proporcionar retroalimentación.

Cuando la biblioteca está lista para una revisión formal, la sumisión de código se publica en Sandbox Vault y se selecciona un administrador de revisión de una lista de voluntarios aprobados. El administrador de revisión se asegura que el código esté listo para una revisión formal, establece el itinerario de revisión, lee todas las reseñas de los usuarios y toma la decisión final de aceptar o no la biblioteca. El administrador de revisión puede aceptar la biblioteca con ciertas correcciones o mejoras que deben implementarse antes de que ésta se agregue oficialmente a Boost. Una vez que se ha aceptado una biblioteca, el autor es responsable de su mantenimiento.

La licencia de software Boost

La Licencia de Software Boost (www.boost.org/more/license_info.html) otorga los derechos de copiar, modificar, utilizar y distribuir el código fuente de Boost y los archivos binarios para cualquier uso comercial o no comercial. El único requerimiento es que se distribuya la información de copyright y licencia con cualquier código fuente que se haga público, aunque no es obligatorio liberar el código fuente. Estas condiciones permiten utilizar las bibliotecas Boost en cualquier aplicación. Cada biblioteca Boost se debe conformar a estas condiciones.

24.5 Instalación de las Bibliotecas Boost

Las bibliotecas Boost se pueden utilizar con un proceso mínimo de instalación en muchas plataformas y compiladores. Una guía de instalación disponible en www.boost.org/more/getting_started/index.html (en inglés) proporciona instrucciones de instalación para muchos compiladores y plataformas. En el sitio Web para este libro (www.deitel.com/books/cpphtp6) proporcionamos instrucciones de instalación de Boost para Visual C++ 2005 Express.

24.6 Las Bibliotecas Boost en el Reporte técnico I (TR1)

Varias bibliotecas Boost se han aceptado como parte del Reporte técnico 1. El TR1 es una descripción de los cambios y adiciones propuestos a la Biblioteca estándar de C++. GCC (Colección de compiladores de GNU) proporciona una implementación parcial del TR1; el sitio gcc.gnu.org/onlinedocs/libstdc++/ext/tr1.html presenta las generalidades acerca de las características de TR1 que se soportan actualmente. GCC, que forma parte del proyecto GNU, proporciona compiladores gratuitos para varios lenguajes de programación, incluyendo C, C++ y Java. Boost también ha liberado un subconjunto de bibliotecas que implementan la mayor parte de la funcionalidad del TR1; este subconjunto se incluye en la última revisión de las bibliotecas Boost. Aquí presentamos las bibliotecas Boost en las que se basan las correspondientes extensiones del TR1. Más adelante proporcionaremos ejemplos de código para algunas de las bibliotecas.

Array²

Boost.Array es una clase de envoltura para los arreglos de tamaño fijo que mejora a los arreglos integrados al soportar la mayor parte de la interfaz de contenedores de la STL, descrita en la sección 22.1. **Boost.Array** nos permite utilizar arreglos de tamaño fijo en las aplicaciones de STL, en vez de los objetos **vector** (arreglos que ajustan su tamaño en forma dinámica).

2. Documentación para Boost.Array, Nicolai Josuttis, www.boost.org/doc/html/array.html.

Bind³

Boost.Bind extiende la funcionalidad de las funciones estándar `std::bind1st` y `std::bind2nd`. Las funciones `bind1st` y `bind2nd` se utilizan para adaptar funciones binarias (es decir, funciones que reciben dos argumentos) que se utilizan con los algoritmos estándar que reciben funciones unarias (es decir, funciones que reciben un argumento). **Boost.Bind** mejora esa funcionalidad, al permitir al programador que adapte las funciones que reciben hasta nueve argumentos. **Boost.Bind** también facilita la labor de reordenar los argumentos que se pasan a la función, mediante el uso de receptáculos.

Function⁴

Boost.Function nos permite almacenar apuntadores a funciones, apuntadores a funciones miembro y objetos de función en una envoltura de función. También podemos almacenar una referencia a un objeto de función, mediante el uso de las funciones `ref` y `cref` que se agregaron al encabezado `<utility>`. Esto nos permite evitar las costosas operaciones de copia. Una función `boost::function` puede contener cualquier función cuyos argumentos y tipo de valor de retorno pueden convertirse para que coincidan con la firma de la envoltura de la función. Por ejemplo, si se creó la envoltura de función para contener una función que recibe un objeto `string` y devuelve un objeto `string`, también puede contener una función que reciba un objeto `char*` y devuelva un `char *`, ya que un `char*` se puede convertir en un `string` mediante el uso de un constructor de conversión.

Mem_fn⁵

Boost.mem_fn mejora las funciones `std::mem_fun` y `std::mem_fun_ref`. `mem_fun` y `mem_fun_ref` reciben un apuntador a una función miembro o una referencia a una función miembro, respectivamente, y crea un objeto de función que llama a esa función miembro. La función miembro no puede recibir argumentos (ni un solo argumento). Los objetos de función se utilizan comúnmente con la función `for_each` de la Biblioteca estándar y con otros algoritmos de la STL. En la sección 22.7 hablamos sobre los objetos de función. **Boost.mem_fn** mejora las funciones estándar al permitir al programador crear el objeto de función con un apuntador, una referencia o un apuntador inteligente (sección 24.8) a una función miembro. También permite que las funciones miembro reciban más de un argumento. `mem_fn` es una versión más flexible de `mem_fun` y `mem_fun_ref`.

Random⁶

Boost.Random nos permite crear una variedad de generadores de números aleatorios y distribuciones de números aleatorios. Las funciones `std::rand` y `std::srand` en la Biblioteca Estándar de C++ generan números seudoaleatorios. Un generador de números seudoaleatorios utiliza un estado inicial para producir números que parecen aleatorios; si se utiliza el mismo estado inicial, se produce la misma secuencia de números. La función `rand` siempre utiliza el mismo estado inicial, por lo tanto produce la misma secuencia de números todo el tiempo. La función `srand` nos permite establecer el estado inicial para variar la secuencia. Los números seudoaleatorios se utilizan comúnmente en las pruebas; la predictibilidad nos permite confirmar los resultados. **Boost.Random** proporciona generadores de números seudoaleatorios, así como generadores que pueden producir números aleatorios no determinísticos: un conjunto de números aleatorios que no se pueden predecir. Dichos generadores de números aleatorios se utilizan en simulaciones y escenarios de seguridad, en donde la predictibilidad es indeseable.

Boost.Random también nos permite especificar la distribución de los números generados. Una distribución común es la **distribución uniforme**, la cual asigna la misma probabilidad a cada número dentro de un rango dado. Esto es similar a tirar un dado o lanzar una moneda; cada posible resultado tiene la misma probabilidad. El programador puede establecer este rango en tiempo de compilación. **Boost.Random** nos permite utilizar una distribución en combinación con cualquier generador de números aleatorios, e incluso hasta crear nuestras propias distribuciones.

Ref⁷

La biblioteca **Boost.Ref** proporciona envolturas de referencia que nos permiten pasar referencias a algoritmos que por lo general reciben sus argumentos por valor. El objeto `reference_wrapper` contiene la referencia y permite al algoritmo utilizarla como valor. El uso de referencias en vez de valores mejora el rendimiento al pasar objetos grandes a un algoritmo.

3. Documentación para `Boost.Bind`, Peter dimov, www.boost.org/libs/bind/bind.html.

4. Documentación para `Boost.Function`, Douglas Gregor, www.boost.org/doc/html/function.html.

5. Documentación para `Boost.Mem_fn`, Peter Dimov, www.boost.org/libs/bind/mem_fn.html.

6. Jens Maurer, "A Proposal to Add an Extensible Random Number Facility to the Standard Library", Documento número N1452, Abril 10, 2003, www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1452.html.

7. Documentación para `Boost.Ref`, Jaakko Javi, Peter Dimov, Douglas Gregor y Dave Abrahams, www.boost.org/doc/html/ref.html.

Regex⁸

Boost.Regex ofrece soporte para procesar expresiones regulares en C++. Las expresiones regulares se utilizan para relacionar patrones de caracteres específicos en el texto. Muchos lenguajes modernos de programación tienen soporte integrado para las expresiones regulares, pero C++ no. Con **Boost.Regex** podemos buscar una expresión específica en un objeto **string**, reemplazar partes de un objeto **string** que coincidan con una expresión regular y dividir un objeto **string** en tokens mediante el uso de expresiones regulares para definir los delimitadores. Estas técnicas se utilizan comúnmente para el procesamiento y análisis de texto, y para validar la entrada. En la sección 24.7 hablaremos con más detalle sobre la biblioteca **Boost.Regex**.

Result_of⁹

La plantilla de clase **result_of** se puede utilizar para especificar el tipo de valor de retorno de la expresión de una llamada; es decir, una expresión que llama a una función, o que llama al operador paréntesis sobrecargado de un objeto de función. El tipo de valor de retorno de la expresión de la llamada se determina con base en los tipos de los argumentos que se pasan a la expresión de la llamada. Esto puede ser útil para convertir en plantillas los tipos de valores de retorno de las funciones y los objetos de funciones.

Smart_ptr¹⁰

Boost.Smart_ptr define apuntadores inteligentes que nos pueden ayudar a administrar los recursos que se asignan en forma dinámica (por ejemplo, memoria, archivos y conexiones de bases de datos). A menudo los programadores se confunden acerca de cuándo liberar la memoria, o simplemente olvidan hacerlo, en especial cuando más de un apuntador hace referencia a la memoria. Los apuntadores inteligentes se encargan de estas tareas en forma automática. TR1 incluye dos apuntadores inteligentes de la biblioteca **Boost.Smart_ptr**. Los objetos **shared_ptr** se encargan de la administración de los objetos que se asignan en forma dinámica, durante el tiempo que existan. La memoria se libera cuando no hay objetos **shared_ptr** que hagan referencia a ella. Los objetos **weak_ptr** nos permiten observar el valor que contiene un objeto **shared_ptr** sin asumir responsabilidades administrativas. En la sección 24.8 hablaremos sobre la biblioteca **Boost.Smart_ptr** con más detalle.

Tuple¹¹

Un **tuple** es una colección de objetos. **Boost.Tuple** nos permite crear conjuntos de objetos en forma genérica y permite que las funciones genéricas actúen sobre esos conjuntos. La biblioteca nos permite crear tuples de hasta 10 objetos; ese límite se puede extender. **Boost.Tuple** es básicamente una extensión para la plantilla de clase **std::pair** de la STL. Los tuples se utilizan comúnmente para devolver múltiples valores de una función. También se pueden utilizar para almacenar conjuntos de elementos en un contenedor de la STL, en donde cada conjunto de elementos es un elemento del contenedor. Otra característica útil es la habilidad de establecer los valores de las variables mediante el uso de los elementos de un tuple.

Type_traits¹²

La biblioteca **Boost.Type_traits** ayuda a abstraer las diferencias entre los tipos para permitir optimizar implementaciones de programación genérica. Las clases de **type_traits** nos permiten determinar rasgos específicos de un tipo (por ejemplo, si es un tipo de apuntador o de referencia, o si el tipo tiene un calificador **const**) y realizar transformaciones de tipos para permitir utilizar el objeto en código genérico. Dicha información se puede utilizar para optimizar el código genérico. Por ejemplo, algunas veces es más eficiente copiar una colección de objetos utilizando la función **memcpy** de C, en vez de iterar a través de todos los elementos de la colección, como se hace con el algoritmo **copy** de la STL. Con la biblioteca **Boost.Type_traits**, los algoritmos genéricos se pueden optimizar comprobando primero los rasgos de los tipos que se van a procesar, y después se puede llevar a cabo el algoritmo de manera acorde.

8. Documentación para **Boost.Regex**, John Maddock, www.boost.org/libs/regex/doc/index.html.

9. Documentación para **Boost.Result_of**, Doug Gregor, www.boost.org/libs/utility/utility.htm#result_of.

10. Documentación para **Boost.Smart_ptr**, Greg Colvin y Beman Dawes, www.boost.org/libs/smart_ptr/smart_ptr.htm.

11. Documentación para **Boost.Tuple**, Jaakko Järvi, www.boost.org/libs/tuple/doc/tuple_users_guide.html.

12. Documentación para **Boost.Type_traits**, Steve Cleary, Beman Dawes, Howard Hinnant y John Maddock, www.boost.org/doc/html/boost_typetraits.html.

24.7 Uso de expresiones regulares con la biblioteca Boost.Regex

Las expresiones regulares son objetos `string` con formato especial, que se utilizan para buscar patrones en el texto. Se pueden utilizar para validar datos y asegurar que se encuentren en un formato específico. Por ejemplo, un código postal debe consistir de cinco dígitos, y un apellido paterno debe empezar con letra mayúscula.

La biblioteca `Boost.Regex` proporciona varias clases y algoritmos para reconocer y manipular expresiones regulares. La plantilla de clase `basic_regex` (en el espacio de nombres `boost`) representa una expresión regular. El algoritmo `regex_match` de `Boost.Regex` devuelve verdadero si un objeto `string` coincide con la expresión regular. Con `regex_match`, toda la cadena debe coincidir con la expresión regular. `Boost.Regex` también proporciona el algoritmo `regex_search`, el cual devuelve `true` si cualquier parte de un objeto `string` arbitrario coincide con la expresión regular. Para utilizar la biblioteca `Boost.Regex`, debemos incluir el archivo de encabezado "`boost/regex.hpp`". Observe que los archivos de encabezado de Boost utilizan la extensión `.hpp`.

Clases de caracteres de las expresiones regulares

La tabla en la figura 24.1 especifica algunas clases de caracteres que se pueden utilizar con las expresiones regulares. Una clase de carácter no es una clase de C++; es simplemente una secuencia de escape que representa un grupo de caracteres que podrían aparecer en un objeto `string`.

Un carácter de palabra es cualquier carácter alfanumérico o guion bajo. Un carácter de espacio en blanco es un espacio, tabulador, retorno de carro, nueva línea o avance de página. Un dígito es cualquier carácter numérico. Las expresiones regulares no se limitan a las clases de caracteres de la figura 24.1. En la figura 24.2 verá que las expresiones regulares pueden utilizar otras notaciones para buscar patrones complejos en objetos `string`.

Clase de carácter	Coincide con	Clase de carácter	Coincide con
<code>\d</code>	cualquier dígito decimal	<code>\D</code>	cualquiera que no sea dígito
<code>\w</code>	cualquier carácter de palabra	<code>\W</code>	cualquier carácter que no sea de palabra
<code>\s</code>	cualquier carácter de espacio en blanco	<code>\S</code>	cualquier carácter que no sea de espacio en blanco

Figura 24.1 | Clases de caracteres.

24.7.1 Ejemplo de una expresión regular

El programa de la figura 24.2 trata de relacionar cumpleaños con una expresión regular. Para fines de demostración, la expresión en la línea 15 sólo coincide con cumpleaños que no ocurren en Abril y que pertenecen a las personas cuyos nombres empiezan con "J".

En la línea 15 se crea un objeto `regex` (un `typedef` de la plantilla de clase `basic_regex`), para lo cual se pasa una expresión regular al constructor de `regex`. Observe que anteponemos a cada carácter de barra diagonal inversa una barra diagonal inversa adicional. Recuerde que C++ considera a una barra diagonal inversa en una literal de cadena como el principio de una secuencia de escape. Para insertar una barra diagonal inversa literal en una cadena, debemos escapar el carácter de barra diagonal inversa con otra barra diagonal inversa. Por ejemplo, la clase de carácter `\d` debe representarse como `\\\d` en una literal de cadena de C++.

```

1 // Fig. 24.2: CoincidenciasRegex.cpp
2 // Demostración de las expresiones regulares.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
```

Figura 24.2 | Expresiones regulares para comprobar cumpleaños. (Parte I de 2).

```

9
10 #include "boost/regex.hpp"
11
12 int main()
13 {
14     // crea una expresión regular
15     boost::regex expresion( "J.*\\d[0-35-9]-\\d\\d-\\d\\d" );
16
17     // crea un objeto string para probarlo
18     string string1 = "Jane cumple anios el 05-12-75\n"
19         "Dave cumple anios el 11-04-68\n"
20         "John cumple anios el 04-28-73\n"
21         "Joe cumple anios el 12-17-77";
22
23     // crea un objeto boost::smatch para guardar los resultados de la búsqueda
24     boost::smatch coincidencia;
25
26     // relaciona la expresión regular con el objeto string e imprime todas las coincidencias
27     while ( boost::regex_search( string1, coincidencia, expresion,
28         boost::match_not_dot_newline ) )
29     {
30         cout << coincidencia << endl; // imprime el objeto string que coincide
31
32         // elimina la subcadena que coincidió del objeto string
33         string1 = coincidencia.suffix();
34     } // fin de while
35
36     return 0;
37 } // fin de la función main

```

```

Jane cumple anios el 05-12-75
Joe cumple anios el 12-17-77

```

Figura 24.2 | Expresiones regulares para comprobar cumpleaños. (Parte 2 de 2).

El primer carácter en la expresión regular "J" es un carácter literal. Es obligatorio que cualquier objeto **string** que coincide con esta expresión regular empiece con "J". En una expresión regular, el carácter punto "." coincide con cualquier carácter individual. Cuando el carácter punto va seguido de un asterisco, como en ".*", la expresión regular coincide con cualquier número de caracteres no especificados. En general, cuando se aplica el operador "*" a un patrón, éste coincidirá con *ceros o más* ocurrencias. Por ejemplo, tanto "A*" como "A+" coincidirán con "A", pero sólo "A*" coincidirá con un objeto **string** vacío.

Como se indica en la figura 24.1, "\d" coincide con cualquier dígito decimal. Para especificar conjuntos de caracteres aparte de los que pertenecen a una clase de carácter predefinida, pueden listarse entre corchetes, []. Por ejemplo, el patrón "[aeiou]" coincide con cualquier vocal. Para representar los rangos de caracteres se coloca un guión corto (-) entre dos caracteres. En el ejemplo, "[0-35-9]" coincide sólo con los dígitos en los rangos especificados por el patrón; es decir, cualquier dígito entre 0 y 3 o entre 5 y 9; por lo tanto, coincide con cualquier dígito excepto 4. También podemos especificar que un patrón debe coincidir con cualquier carácter que no sea uno de los que se incluyen en los corchetes. Para ello, debemos colocar ^ como el primer carácter en los corchetes. Es importante tener en cuenta que "[^4]" no es lo mismo que "[0-35-9]"; "[^4]" coincide con cualquier carácter que no sea dígito y con dígitos distintos a 4.

Aunque el carácter "-" indica un rango cuando se encierra entre corchetes, las instancias del carácter "-" fuera de expresiones de agrupamiento se consideran como caracteres literales. Por ende, la expresión regular en la línea 15 busca un objeto **string** que empiece con la letra "J", seguido de cualquier número de caracteres, seguido de un número de dos dígitos (de los cuales el segundo dígito no puede ser 4), seguido de un guión corto, de otro número de dos dígitos, de otro guión corto y de otro número de dos dígitos.

En la línea 24 se crea un objeto **smatch** (se pronuncia como "ess-match"; un **typedef** para **match_results**). Cuando un objeto **match_results** se pasa como argumento a uno de los algoritmos de **Boost.Regex**, almacena la coincidencia de la expresión regular. Un objeto **smatch** almacena un objeto de tipo **string::const_iterator** que podemos usar para acceder al objeto **string** que coincide. Hay definiciones **typedef** para dar soporte a otras representaciones de cadena tales como **char*** (**cmatch**).

La instrucción `while` (líneas 27 a 34) busca en `string1` las coincidencias con la expresión regular, hasta que no pueda encontrar ninguna. Utilizamos la llamada a `regex_search` como la condición para la instrucción `while` (líneas 27 y 28). `regex_search` devuelve `true` si el objeto `string` (`string1`) contiene una coincidencia con la expresión regular (`expresion`). También pasamos un objeto `smatch` a `regex_search` para poder acceder al objeto `string` que coincide. El último argumento (`match_not_dot_newline`) evita que el carácter `"."` coincida con un carácter de nueva línea. El cuerpo de la instrucción `while` imprime la subcadena que coincidió con la expresión regular (línea 30) y la elimina del objeto `string` en el que se está realizando la búsqueda (línea 33). La llamada a la función miembro `suffix` de `match_results` devuelve un objeto `string` del final de la coincidencia hasta el final del objeto `string` en el que se está realizando la búsqueda. Los resultados de la figura 24.2 muestran las dos coincidencias que se encontraron en `string1`. Observe que ambas coincidencias se conforman con el patrón especificado de la expresión regular.

Cuantificadores

El asterisco (*) en la línea 15 de la figura 24.2 se conoce más formalmente como **cuantificador**. La figura 24.3 lista varios cuantificadores que podemos colocar después de un patrón en una expresión regular y el propósito de cada cuantificador.

Ya hemos hablado acerca de cómo trabajan los cuantificadores asterisco (*) y suma (+). El cuantificador signo de interrogación (?) coincide con cero o una ocurrencias del patrón que cuantifica. Un conjunto de llaves que contiene un número (`{n}`) coincide con exactamente n ocurrencias del patrón que cuantifica. En el siguiente ejemplo demostramos este cuantificador. Si se incluye una coma después del número encerrado entre llaves, el patrón coincide con al menos n ocurrencias del patrón cuantificado. El conjunto de llaves que contiene dos números (`{n,m}`) coincide con un número entre m y n ocurrencias (inclusive) del patrón que cuantifica. Todos los cuantificadores son **avaros**; coincidirán con todas las ocurrencias posibles del patrón, hasta que ya no haya una coincidencia. Si un cuantificador va seguido de un signo de interrogación (?), se vuelve **perezoso** y coincidirá con el menor número posible de ocurrencias, mientras que haya una coincidencia exitosa.

Cuantificador	Coincidencias
*	Coincide con cero o más ocurrencias del patrón que le antecede.
+	Coincide con una o más ocurrencias del patrón que le antecede.
?	Coincide con cero o una ocurrencias del patrón que le antecede.
{n}	Coincide exactamente con n ocurrencias del patrón que le antecede.
{n,}	Coincide con al menos n ocurrencias del patrón que le antecede.
{n,m}	Coincide con entre n y m (inclusive) ocurrencias del patrón que le antecede.

Figura 24.3 | Cuantificadores utilizados en expresiones regulares.

24.7.2 Cómo validar la entrada del usuario mediante expresiones regulares

El programa de la figura 24.4 presenta un ejemplo más complicado que utiliza expresiones regulares para validar la información del nombre, dirección y número telefónico introducida por un usuario.

```

1 // Fig. 24.4: Validar.cpp
2 // Validación de la entrada del usuario con expresiones regulares.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 #include <string>
9 using std::string;
10
11 #include "boost/regex.hpp"

```

Figura 24.4 | Validación de la entrada del usuario con expresiones regulares. (Parte I de 3).

```
12
13 bool validar( const string&, const string& ); // prototipo de validar
14 string introducirDatos( const string&, const string& ); // prototipo de introducirDatos
15
16 int main()
17 {
18     // escribe el apellido paterno
19     string apellidoPaterno = introducirDatos( "apellido paterno", "[A-Z][a-zA-Z]*" );
20
21     // escribe el primer nombre
22     string primerNombre = introducirDatos( "primer nombre", "[A-Z][a-zA-Z]*" );
23
24     // escribe la dirección
25     string direccion = introducirDatos( "direccion",
26         "[0-9]+\\s+([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" );
27
28     // escribe la ciudad
29     string ciudad =
30         introducirDatos( "ciudad", "([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" );
31
32     // escribe el estado
33     string estado = introducirDatos( "estado",
34         "([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" );
35
36     // escribe el código postal
37     string codigoPostal = introducirDatos( "codigo postal", "\\d{5}" );
38
39     // escribe el número telefónico
40     string numeroTelefonico = introducirDatos( "numero telefonico",
41         "[1-9]\\d{2}-[1-9]\\d{2}-\\d{4}" );
42
43     // muestra los datos validados
44     cout << "\nDatos validados\n\n"
45         << "Apellido paterno: " << apellidoPaterno << endl
46         << "Primer nombre: " << primerNombre << endl
47         << "Direccion: " << direccion << endl
48         << "Ciudad: " << ciudad << endl
49         << "Estado: " << estado << endl
50         << "Codigo postal: " << codigoPostal << endl
51         << "Numero telefonico: " << numeroTelefonico << endl;
52
53     return 0;
54 } // fin de la función main
55
56 // valida el formato de los datos usando una expresión regular
57 bool validar( const string &datos, const string &expresion )
58 {
59     // crea un objeto regex para validar los datos
60     boost::regex expresionValidacion = boost::regex( expresion );
61     return boost::regex_match( datos, expresionValidacion );
62 } // fin de la función validar
63
64 // recolecta la entrada del usuario
65 string introducirDatos( const string &nombreCampo, const string &expresion )
66 {
67     string datos; // almacena los datos recolectados
68
69     // solicita los datos al usuario
70     cout << "Escriba " << nombreCampo << ": ";
71     getline( cin, datos );
72 }
```

Figura 24.4 | Validación de la entrada del usuario con expresiones regulares. (Parte 2 de 3).

```

73 // valida los datos
74 while ( !( validar( datos, expresion ) ) )
75 {
76     cout << "Invalido: " << nombreCampo << ".\n";
77     cout << "Escriba " << nombreCampo << ": ";
78     getline( cin, datos );
79 } // fin de while
80
81     return datos;
82 } // fin de la función introducirDatos

```

```

Escriba apellido paterno: Doe
Escriba primer nombre: John
Escriba direccion: 123 Una Calle
Escriba ciudad: Una Ciudad
Escriba estado: Un Estado
Escriba codigo postal: 12345
Escriba numero telefonico: 123-456-7890

```

Datos validados

```

Apellido paterno: Doe
Primer nombre: John
Direccion: 123 Una Calle
Ciudad: Una Ciudad
Estado: Un Estado
Codigo postal: 12345
Numero telefonico: 123-456-7890

```

Figura 24.4 | Validación de la entrada del usuario con expresiones regulares. (Parte 3 de 3).

El programa primero pide al usuario que introduzca un apellido paterno (línea 19), para lo cual llama a la función `introducirDatos`. Esta función (líneas 65 a 82) recibe dos argumentos, el nombre de los datos que se van a introducir y una expresión regular con la que deben coincidir. La función pide al usuario (línea 70) que introduzca los datos específicos. Después, `introducirDatos` comprueba si la entrada tiene el formato correcto, para lo cual llama a la función `validar` (líneas 57 a 62). Esta función (líneas 57 a 62) recibe dos argumentos: el objeto `string` a validar y la expresión regular con la que debe coincidir. La función utiliza primero la expresión para crear un objeto `regex` (línea 60). Después llama a `regex_match` para determinar si el objeto `string` coincide con la expresión. Si la entrada no es válida, `introducirDatos` pide al usuario que introduzca la información de nuevo. Una vez que el usuario introduce una entrada válida, los datos se devuelven como un objeto `string`. El programa repite ese proceso hasta que se han validado todos los campos de datos (líneas 19 a 41). Después mostramos toda la información (líneas 44 a 51).

En el ejemplo anterior, buscamos subcadenas en un objeto `string` que coincidieran con una expresión regular. En este ejemplo deseamos asegurar que el objeto `string` completo para cada entrada se conforme a una expresión regular específica. Por ejemplo, deseamos aceptar "Smith" como apellido paterno, pero no "9@Smith#". Aquí utilizamos `regex_match` en vez de `regex_search`; `regex_match` sólo devuelve `true` si todo el objeto `string` coincide con la expresión regular. De manera alternativa, podemos usar una expresión regular que empiece con un carácter "^" y termine con un carácter "\$". Los caracteres "^" y "\$" representan el inicio y el final de un objeto `string`, respectivamente. En conjunto, estos caracteres obligan a que una expresión regular devuelva una coincidencia sólo si todo el objeto `string` que se está procesando coincide con la expresión regular.

La expresión regular en la línea 19 utiliza el corchete y la notación de rango para relacionar una primera letra en mayúsculas seguida de letras tanto en mayúscula como en minúscula; `a-z` coincide con cualquier letra minúscula, y `A-Z` coincide con cualquier letra mayúscula. El cuantificador `*` indica que el segundo rango de caracteres puede ocurrir cero o más veces en el objeto `string`. Por ende, esta expresión sólo coincide con cualquier objeto `string` que consiste en una letra mayúscula, seguida de cero o más letras adicionales.

La notación `\s` coincide con un solo carácter de espacio en blanco (líneas 26, 30 y 34). La expresión `\d{5}`, que se utiliza para el objeto `string` llamado `codigoPostal` (línea 37), coincide con cinco dígitos cualesquiera. El carácter "`|`" (líneas 26, 30 y 34) coincide con la expresión a su izquierda *o* con la expresión a su derecha. Por ejemplo, `Hola (John|Jane)` coincide tanto con `Hola John`, como con `Hola Jane`. En la línea 26, utilizamos el carácter "`|`" para indicar que la dirección puede contener una palabra de uno o más caracteres *o* una palabra de uno o más caracteres seguida de

un espacio y otra palabra de uno o más caracteres. Observe el uso de los paréntesis para agrupar partes de la expresión regular. Pueden aplicarse cuantificadores a los patrones encerrados entre paréntesis para crear expresiones regulares más complejas.

Las variables `apellidoPaterno` y `primerNombre` (líneas 19 y 22) aceptan objetos `string` de cualquier longitud que empiecen con una letra mayúscula. La expresión regular para el objeto `string direccion` (línea 26) coincide con un número de por lo menos un dígito, seguido de un espacio y después de una o más letras, o de una o más letras seguidas por un espacio y otra serie de una o más letras. Por lo tanto, "10 Broadway" y "10 Main Street" son direcciones válidas. Según su composición actual, la expresión regular en la línea 26 no coincide con una dirección que no empiece con un número, o que tenga más de dos palabras. Las expresiones regulares para los objetos `string ciudad` (línea 30) y `estado` (línea 34) coinciden con cualquier palabra de al menos un carácter o, de manera alternativa, con dos palabras cualesquiera de al menos un carácter, si éstas van separadas por un solo espacio. Esto significa que tanto `Waltham` como `West Newton` son coincidencias. De nuevo, estas expresiones regulares no aceptan nombres que tengan más de dos palabras. La expresión regular para el objeto `string codigoPostal` (línea 37) asegura que el código postal sea un número de cinco dígitos. La expresión regular para el objeto `string numeroTelefonico` (línea 41) indica que el número telefónico debe ser de la forma `xxx-yyy-yyyy`, en donde las `x` representan el código de área y las `y` el número. La primera `x` y la primera `y` no pueden ser cero, según lo especificado por el rango [1-9] en cada caso.

24.7.3 Cómo reemplazar y dividir cadenas

Algunas veces es útil reemplazar partes de un objeto `string` con otro, o dividir un objeto `string` de acuerdo con una expresión regular. Para este fin, la biblioteca `Boost.Regex` proporciona el algoritmo `regex_replace` y la clase `regex_token_iterator`, que demostramos en la figura 24.5.

El algoritmo `regex_replace` reemplaza texto en un objeto `string` con texto nuevo, en cualquier parte en donde el objeto `string` original coincida con una expresión regular. En la línea 23, `regex_replace` reemplaza cada instancia de "*" en `stringPrueba1` con "^". Observe que la expresión regular ("*") coloca una barra diagonal inversa (\) antes del carácter "*". Por lo general, * es un cuantificador que indica que una expresión regular debe coincidir con cualquier número de ocurrencias de un patrón anterior. Sin embargo, en este caso queremos encontrar todas las ocurrencias del carácter literal "*"; para ello, debemos escapar el carácter "*" con el carácter "\". Al escapar un carácter de una expresión regular especial con una \, indicamos al motor para relacionar expresiones regulares que debe buscar el carácter "*" actual, en vez de usarlo como cuantificador. En las líneas 27 y 28 se utiliza `regex_replace` para reemplazar la cadena "estrellas" en `stringPrueba1` con la cadena "asteriscos". En las líneas 33 y 34 se utiliza `regex_replace` para reemplazar cada palabra en `stringPrueba1` con la cadena "palabra".

```

1 // Fig. 24.5: SustitucionRegex.cpp
2 // Uso del algoritmo regex_replace.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 #include "boost/regex.hpp"
11
12 int main()
13 {
14     // crea las cadenas de prueba
15     string stringPrueba1 = "Este enunciado termina con 5 estrellas *****";
16     string stringPrueba2 = "1, 2, 3, 4, 5, 6, 7, 8";
17     string salida;
18
19     cout << "Cadena original: " << stringPrueba1 << endl;
20
21     // reemplaza cada * con un ^
22     stringPrueba1 =
23         boost::regex_replace( stringPrueba1, boost::regex( "\\*" ), "^" );
24     cout << "^ se sustituyo por *: " << stringPrueba1 << endl;

```

Figura 24.5 | Uso del algoritmo `regex_replace`. (Parte 1 de 2).

```

25 // reemplaza "estrellas" con "intercaladores"
26 stringPrueba1 = boost::regex_replace(
27     stringPrueba1, boost::regex( "estrellas" ), "intercaladores" );
28 cout << "\"intercaladores\" se sustituyeron por \"estrellas\": "
29     << stringPrueba1 << endl;
30
31 // reemplaza cada palabra con "palabra"
32 stringPrueba1 = boost::regex_replace(
33     stringPrueba1, boost::regex( "\w+" ), "palabra" );
34 cout << "Cada palabra se reemplazo por \"palabra\": " << stringPrueba1 << endl;
35
36 // reemplaza los primeros tres dígitos con "digito"
37 cout << "\nCadena original: " << stringPrueba2 << endl;
38 string copiaStringPrueba2 = stringPrueba2;
39
40 for ( int i = 0; i < 3; i++ ) // itera tres veces
41 {
42     copiaStringPrueba2 = boost::regex_replace( copiaStringPrueba2,
43         boost::regex( "\d" ), "digito", boost::format_first_only );
44 } // fin de for
45
46 cout << "Reemplaza los primeros 3 digitos por \"digito\": "
47     << copiaStringPrueba2 << endl;
48
49 // divide la cadena en las comas
50 cout << "la cadena se dividio en las comas [";
51
52 boost::sregex_token_iterator iteradorToken( stringPrueba2.begin(),
53     stringPrueba2.end(), boost::regex( ",\s" ), -1 ); // iterador de token
54 boost::sregex_token_iterator end; // iterador vacío
55
56 while ( iteradorToken != end ) // iteradorToken no está vacío
57 {
58     salida += "" + *iteradorToken + "\", "; // agrega el token a salida
59     iteradorToken++; // avanza el iterador
60 } // fin de while
61
62 // elimina la ", " al final del objeto string salida
63 cout << salida.substr( 0, salida.length() - 2 ) << "]" << endl;
64
65 return 0;
66 } // fin de la función main

```

```

Cadena original: Este enunciado termina con 5 estrellas *****
^ se sustituyo por *: Este enunciado termina con 5 estrellas ^^^^^
"intercaladores" se sustituyeron por "estrellas": Este enunciado termina con 5 intercaladores
^^^^^
Cada palabra se reemplazo por "palabra": palabra palabra palabra palabra palabra ^^^^^^
Cadena original: 1, 2, 3, 4, 5, 6, 7, 8
Reemplaza los primeros 3 digitos por "digito": digito, digito, digito, 4, 5, 6, 7, 8
la cadena se dividio en las comas ["1", "2", "3", "4", "5", "6", "7", "8"]

```

Figura 24.5 | Uso del algoritmo `regex_replace`. (Parte 2 de 2).

En las líneas 41 a 45 se reemplazan las tres primeras instancias de un dígito ("`\d`") en `stringPrueba2` con el texto "`digito`". Pasamos un argumento adicional (`boost::format_first_only`) a `regex_replace` (líneas 43 y 44). Este argumento indica a `regex_replace` que reemplace sólo la primera subcadena que coincide con la expresión regular. Por lo general, `regex_replace` reemplaza todas las ocurrencias del patrón. Colocamos esta llamada dentro de un ciclo `for` que se ejecuta tres veces; cada vez reemplaza la primera instancia de un dígito con el texto "`digito`". Utilizamos una copia de `stringPrueba2` (línea 39), para poder usar el objeto `stringPrueba2` original para la siguiente parte del ejemplo.

A continuación, utilizamos un objeto `regex_token_iterator` para dividir un objeto `string` en varias subcadenas. Un `regex_token_iterator` itera a través de las partes de un objeto `string` que coinciden con una expresión regular. En las líneas 53 y 55 se utiliza `sregex_token_iterator`, una definición `typedef` que indica que los resultados se van a manipular mediante un objeto `string::const_iterator`. Para crear el iterador (líneas 53 y 54) pasamos dos iteradores al constructor (`stringPrueba2.begin()` y `stringPrueba2.end()`), que representan el inicio y final del objeto `string` en el que se va a iterar, y la expresión regular que se va a buscar. En nuestro caso, queremos iterar a través de las partes del objeto `string` que *no* coinciden con la expresión regular. Para ello, pasamos `-1` al constructor. Esto indica que debe iterar sobre cada subcadena que no coincide con la expresión regular. El objeto `string` original se divide en los delimitadores que coinciden con la expresión regular especificada. Utilizamos una instrucción `while` (líneas 57 a 61) para agregar cada subcadena al objeto `string` salida (línea 17). El iterador `regex_token_iterator` end (línea 55) es un iterador vacío. Hemos iterado sobre todo el objeto `string` completo cuando `iteratorToken` es igual a `end` (línea 57).

24.8 Apunadores inteligentes con Boost.Smart_ptr

Muchos errores comunes en el código de C y C++ se relacionan con los apunadores. Los **apunadores inteligentes** nos ayudan a evitar errores al proporcionar funcionalidad adicional a los apunadores estándar. Esta funcionalidad por lo general refuerza el proceso de asignación y desasignación de memoria. Los apunadores inteligentes también nos ayudan a escribir código a prueba de excepciones. Si un programa lanza una excepción antes de llamar a `delete` en un apuntador, crea una fuga de memoria. Después de lanzar una excepción, de todas formas se hará una llamada al destructor de un apuntador inteligente, el cual llama a `delete` en el apuntador por el programador.

La Biblioteca estándar de C++ proporciona un apuntador inteligente básico: `std::auto_ptr`. Un apuntador `auto_ptr` es responsable de administrar la memoria asignada en forma dinámica y llama automáticamente a `delete` para liberar la memoria dinámica cuando se destruye el `auto_ptr` o cuando queda fuera de alcance. Estos apunadores inteligentes son perfectos para variables automáticas. En la sección 16.12 hablaremos de `auto_ptr`.

`auto_ptr` tiene algunas limitaciones. Por ejemplo, un `auto_ptr` no puede apuntar a un arreglo. Suprimiendo un apuntador de un arreglo debemos utilizar `delete []` para asegurar que los destructores son llamados para todos los objetos en el arreglo, pero `auto_ptr` utiliza `delete`. Otra limitación de `auto_ptr` es que no puede ser utilizado con los contenedores STL, los elementos en un contenedor STL deben ser capaces de ser copiados sin peligro. Cuando un `auto_ptr` es copiado, la propiedad de la memoria es transferida a nuevo `auto_ptr` y el original es puesto a NULL. Un contenedor STL puede hacer copias de sus elementos, entonces no puede garantizar que una copia válida del `auto_ptr` permanecerá después del algoritmo que trata los elementos finales del contenedor.

La biblioteca `Boost.Smart_ptr` proporciona apunadores inteligentes adicionales para llenar los huecos donde `auto_ptr` no trabaja. TR1 incluye dos de los seis tipos de apunadores inteligentes en la biblioteca `Boost.Smart_ptr`, llamada `shared_ptr` y `weak_ptr`. No se destina que estos apunadores inteligentes sustituyan a `auto_ptr`. En cambio, ellos proporcionan opciones adicionales con diferente funcionalidad.

24.8.1 Uso de `shared_ptr` y conteo de referencias

Los apunadores `shared_ptr` contienen un apuntador interno a un recurso (por ejemplo, un objeto asignado en forma dinámica) que se puede compartir con otros objetos en el programa. Podemos tener cualquier número de apunadores `shared_ptr` al mismo recurso. Los apunadores `shared_ptr` realmente comparten el recurso; si modificamos el recurso con un `shared_ptr`, los cambios también serán “vistos” por los otros apunadores `shared_ptr`. El apuntador interno se elimina una vez que se destruye el último apuntador `shared_ptr` al recurso. Los apunadores `shared_ptr` utilizan el **conteo de referencias** para determinar cuántos de ellos apuntan al recurso. Cada vez que se crea un nuevo apuntador `shared_ptr` al recurso se incrementa la **cuenta de referencia**, y cada vez que se destruye uno se decrementa la cuenta de referencia. Cuando la cuenta de referencia llega a cero, se elimina el apuntador interno y se libera la memoria.

Los apunadores `shared_ptr` son útiles en situaciones en las que se necesitan varios apunadores al mismo recurso, como en los contenedores de la STL. Los apunadores `auto_ptr` no funcionan en los contenedores de la STL, ya que los contenedores (o los algoritmos que los manipulan) podrían copiar los elementos almacenados. Las copias de los apunadores `auto_ptr` no son iguales, ya que el original se establece en NULL después de ser copiado. Por otro lado, los apunadores `shared_ptr` pueden copiarse y utilizarse en forma segura en los contenedores de la STL.

Los apunadores `shared_ptr` también nos permiten determinar la forma en que se destruirá el recurso. Para la mayoría de los objetos que se asignan en forma dinámica, se utiliza `delete`. Sin embargo, algunos recursos requieren de un proceso de limpieza más complejo. En ese caso, podemos suministrar una función `delete` personalizada (o un objeto de función) al constructor del `shared_ptr`. El eliminador determina la forma en que debe destruir el recurso. Cuando la cuenta de referencia llega a cero y el recurso está listo para ser destruido, el apuntador `shared_ptr` llama a la función eliminadora personalizada. Esta funcionalidad permite a un apuntador `shared_ptr` administrar casi cualquier tipo de recurso.

Ejemplo acerca del uso de `shared_ptr`

En las figuras 24.6 y 24.7 se define una clase simple para representar un *Libro* con un objeto `string` que representa el título del *Libro*. El destructor para la clase *Libro* (figura 24.7, líneas 16 a 19) muestra un mensaje en la pantalla, el cual indica que se va a destruir una instancia. Utilizamos esta clase para demostrar la funcionalidad común de `shared_ptr`.

El programa de la figura 24.8 utiliza apuntadores `shared_ptr` para administrar varias instancias de la clase *Libro*. Incluimos el archivo de encabezado "boost/shared_ptr.hpp" para poder utilizar apuntadores `shared_ptr`. También creamos una definición `typedef` llamada *LibroPtr* como alias para el tipo `boost::shared_ptr< Libro >` (línea 13).

```

1 // Fig. 24.6: Libro.h
2 // Declaración de la clase Libro.
3 #ifndef LIBRO_H
4 #define LIBRO_H
5 #include <string>
6 using std::string;
7
8 class Libro
9 {
10 public:
11     Libro( const string &tituloLibro ); // constructor
12     ~Libro(); // destructor
13     string titulo; // titulo del Libro
14 };
15 #endif // LIBRO_H

```

Figura 24.6 | Archivo de encabezado de *Libro*.

```

1 // Fig. 24.7: Libro.cpp
2 // Definiciones de las funciones miembro para la clase.=
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 #include "Libro.h"
11
12 Libro::Libro( const string &tituloLibro ) : titulo( tituloLibro )
13 {
14 }
15
16 Libro::~Libro() // destructor
17 {
18     cout << "Destruyendo Libro: " << titulo << endl;
19 } // fin del destructor

```

Figura 24.7 | Definiciones de las funciones miembro de *Libro*.

```

1 // Fig. 24.8: fig24_8.cpp
2 // Demostración del uso de los apuntadores shared_ptr.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <vector>
8 using std::vector;
9

```

Figura 24.8 | Ejemplo de Programa de `shared_ptr`. (Parte I de 3).

```
10 #include "Libro.h"
11 #include "boost/shared_ptr.hpp"
12
13 typedef boost::shared_ptr< Libro > LibroPtr; // apuntador shared_ptr a un Libro
14
15 // una función delete personalizada para un apuntador a un Libro
16 void eliminarLibro( Libro* libro )
17 {
18     cout << "Eliminador personalizado para un Libro, ";
19     delete libro; // elimina el apuntador al Libro
20 } // fin de eliminarLibro
21
22 // compara los títulos de los dos objetos Libro
23 bool compararTitulos( LibroPtr libroPtr1, LibroPtr libroPtr2 )
24 {
25     return ( libroPtr1->titulo < libroPtr2->titulo );
26 } // fin de compararTitulos
27
28 int main()
29 {
30     // crea un apuntador shared_ptr a un Libro y muestra la cuenta de referencias
31     LibroPtr libroPtr( new Libro( "C++ How to Program" ) );
32     cout << "La cuenta de referencias para Libro " << libroPtr->titulo << " es: "
33         << libroPtr.use_count() << endl;
34
35     // crea otro apuntador shared_ptr al Libro y muestra la cuenta de referencias
36     LibroPtr libroPtr2( libroPtr );
37     cout << "La cuenta de referencias para Libro " << libroPtr->titulo << " es: "
38         << libroPtr.use_count() << endl;
39
40     // cambia el título del Libro y lo utiliza desde ambos apuntadores
41     libroPtr2->titulo = "Java How to Program";
42     cout << "\nEl título del Libro se cambio para ambos apuntadores: "
43         << "\nlibroPtr: " << libroPtr->titulo
44         << "\nlibroPtr2: " << libroPtr2->titulo << endl;
45
46     // crea un objeto std::vector de apuntadores shared_ptr a objetos Libro (LibroPtr)
47     vector< LibroPtr > libros;
48     libros.push_back( LibroPtr( new Libro( "C How to Program" ) ) );
49     libros.push_back( LibroPtr( new Libro( "VB How to Program" ) ) );
50     libros.push_back( LibroPtr( new Libro( "C# How to Program" ) ) );
51     libros.push_back( LibroPtr( new Libro( "C++ How to Program" ) ) );
52
53     // imprime los objetos Libro en el vector
54     cout << "\nlibros antes de ordenar: " << endl;
55     for ( int i = 0; i < libros.size(); i++ )
56         cout << ( libros[ i ] )->titulo << "\n";
57
58     // ordena el vector por título de Libro e imprime el vector ordenado
59     sort( libros.begin(), libros.end(), compararTitulos );
60     cout << "\nlibros despues de ordenar: " << endl;
61     for ( int i = 0; i < libros.size(); i++ )
62         cout << ( libros[ i ] )->titulo << "\n";
63
64     // crea un apuntador shared_ptr con un eliminador personalizado
65     cout << "\nshared_ptr con un eliminador personalizado." << endl;
66     LibroPtr libroPtr3( new Libro( "Small C++ How to Program" ), eliminarLibro );
67     libroPtr3.reset(); // libera el Libro que administra este apuntador shared_ptr
68
69     // los apuntadores shared_ptr están quedando fuera de alcance
70     cout << "\nTodos los objetos shared_ptr estan quedando fuera de alcance." << endl;
```

Figura 24.8 | Ejemplo de Programa de shared_ptr. (Parte 2 de 3).

```

71
72     return 0;
73 } // fin de main

```

La cuenta de referencias para Libro C++ How to Program es: 1
 La cuenta de referencias para Libro C++ How to Program es: 2

El título del Libro se cambio para ambos apuntadores:

LibroPtr: Java How to Program
 LibroPtr2: Java How to Program

Libros antes de ordenar:

C How to Program
 VB How to Program
 C# How to Program
 C++ How to Program

Libros despues de ordenar:

C How to Program
 C# How to Program
 C++ How to Program
 VB How to Program

shared_ptr con un eliminador personalizado.

Eliminador personalizado para un Libro, Destruyendo Libro: Small C++ How to Program

Todos los objetos shared_ptr están quedando fuera de alcance.

Destruyendo Libro: C How to Program
 Destruyendo Libro: C# How to Program
 Destruyendo Libro: C++ How to Program
 Destruyendo Libro: VB How to Program
 Destruyendo Libro: Java How to Program

Figura 24.8 | Ejemplo de Programa de shared_ptr. (Parte 3 de 3).

En la línea 31 se crea un apuntador `shared_ptr` a un `Libro` titulado "C++ How to Program" (utilizando la definición `typedef LibroPtr`). El constructor de `shared_ptr` recibe como argumento un apuntador a un objeto. Le pasamos el apuntador devuelto del operador `new`. Esto crea un apuntador `shared_ptr` que administra el objeto `Libro` y establece el conteo de referencias en uno. El constructor también puede recibir otro `shared_ptr`, en cuyo caso comparte la propiedad del recurso con el otro `shared_ptr` y se incrementa el conteo de referencias en uno. El primer apuntador `shared_ptr` a un recurso debe crearse siempre mediante el operador `new`. Un `shared_ptr` creado con un apuntador regular asume que es el primer apuntador `shared_ptr` asignado a ese recurso, y empieza el conteo de referencias en uno. Si el programador crea varios apuntadores `shared_ptr` con el mismo apuntador, estos apuntadores no se reconocerán entre sí y el conteo de referencias estará equivocado. Cuando se destruyen los apuntadores `shared_ptr`, ambos llaman a `delete` en el recurso.

En las líneas 32 y 33 se muestra el `título` del `Libro` y el número de apuntadores `shared_ptr` que hacen referencia a esa instancia. Observe que utilizamos el operador `->` para acceder al miembro de datos `título` del `Libro`, de igual forma que con un apuntador regular. Los apuntadores `shared_ptr` proporcionan los operadores de apuntador `*` y `->`. Para obtener la cuenta de referencias utilizamos la función miembro `use_count` de `shared_ptr`, la cual devuelve el número de apuntadores `shared_ptr` al recurso. Después creamos otro apuntador `shared_ptr` a la instancia de la clase `Libro` (línea 36). Aquí utilizamos el constructor de `shared_ptr`, con el apuntador `shared_ptr` original como argumento. También podemos usar el operador de asignación (`=`) para crear un apuntador `shared_ptr` al mismo recurso. En las líneas 37 y 38 se imprime la cuenta de referencias del apuntador `shared_ptr` original, para mostrar que la cuenta se incrementó en uno al crear el segundo `shared_ptr`. Como dijimos antes, los cambios realizados al recurso de un apuntador `shared_ptr` son "vistos" por todos los apuntadores `shared_ptr` a ese recurso. Al cambiar el título del `Libro` usando `LibroPtr2` (línea 41), podemos ver el cambio cuando usamos `LibroPtr` (líneas 42 a 44).

A continuación demostramos cómo usar apuntadores `shared_ptr` en un contenedor de la STL. Creamos un `vector` de apuntadores `LibroPtr` (línea 47) y agregamos cuatro elementos (recuerde que `LibroPtr` es una definición `typedef` para un `shared_ptr<Libro>`, línea 13). Esto demuestra una ventaja clave de utilizar `shared_ptr` en vez de `auto_ptr`; como dijimos antes, no se pueden utilizar apuntadores `auto_ptr` en contenedores de la STL. En las líneas 54 a 56 se imprime el contenido del `vector`. Después ordenamos los objetos `Libro` en el `vector` por título (línea 59).

Utilizamos la función `compararTitulos` (líneas 23 a 26) en el algoritmo `sort` para comparar los datos miembro `titulo` de cada `Libro` en orden alfábético.

En la línea 66 se crea un apuntador `shared_ptr` con un eliminador personalizado. Definimos la función eliminadora personalizada `eliminarLibro` (líneas 16 a 20) y la pasamos al constructor de `shared_ptr`, junto con un apuntador a una nueva instancia de la clase `Libro`. Cuando el apuntador `shared_ptr` destruye la instancia de la clase `Libro`, llama a `eliminarLibro` con el apuntador `Libro *` interno como argumento. Observe que `eliminarLibro` recibe un apuntador `Libro *`, no un `shared_ptr`. Una función eliminadora personalizada debe recibir un argumento del tipo de apuntador interno del apuntador `shared_ptr`. `eliminarLibro` muestra un mensaje para indicar que se hizo una llamada al eliminador personalizado, y después elimina el apuntador. Llamamos a la función miembro `reset` (línea 67) de `shared_ptr` para mostrar el eliminador personalizado en acción. La función `reset` libera el recurso actual y establece el apuntador `shared_ptr` en `NULL`. Si no hay otros apuntadores `shared_ptr` al recurso, se destruye. También podemos pasar un apuntador o `shared_ptr` que represente un nuevo recurso a la función `reset`, en cuyo caso el `shared_ptr` administrará el nuevo recurso. Pero, al igual que con el constructor, sólo debemos utilizar un apuntador regular devuelto por el operador `new`.

Todos los apuntadores `shared_ptr` y el vector quedan fuera de alcance al final de la función `main` y se destruyen. Cuando el vector se destruye, también se destruyen los apuntadores `shared_ptr` dentro de él. Los resultados del programa indican que cada instancia de la clase `Libro` se destruye automáticamente mediante los apuntadores `shared_ptr`. No hay necesidad de eliminar (mediante `delete`) cada apuntador colocado en el vector.

24.8.2 weak_ptr: observador de shared_ptr

Un apuntador `weak_ptr` apunta a un recurso administrado por un apuntador `shared_ptr` sin asumir responsabilidad por el mismo. La cuenta de referencias para un apuntador `shared_ptr` no incrementa cuando un `weak_ptr` hace referencia. Esto significa que el recurso de un apuntador `shared_ptr` puede eliminarse mientras aún haya apuntadores `weak_ptr` apuntando a él. Cuando se destruye el último `shared_ptr`, el recurso se elimina y todos los apuntadores `weak_ptr` restantes se establecen en `NULL`. Un uso para los apuntadores `weak_ptr`, como veremos más adelante en esta sección, es evitar las fugas de memoria producidas por las referencias circulares.

Un apuntador `weak_ptr` no puede acceder directamente al recurso al que apunta; el programador debe crear un apuntador `shared_ptr` desde el apuntador `weak_ptr` para acceder al recurso. Hay dos maneras de hacer esto. Podemos pasar el apuntador `weak_ptr` al constructor de `shared_ptr`. Eso crea un apuntador `shared_ptr` al recurso al que apunta el `weak_ptr` e incrementa de manera apropiada la cuenta de referencias. Si el recurso ya se ha eliminado, el constructor de `shared_ptr` lanzará una excepción `boost::bad_weak_ptr`. También podemos llamar a la función miembro `lock` de `weak_ptr`, la cual devuelve un apuntador `shared_ptr` al recurso del `weak_ptr`. Si el apuntador `weak_ptr` apunta a un recurso eliminado (por decir, `NULL`), `lock` devolverá un apuntador `shared_ptr` vacío (es decir, un apuntador `shared_ptr` a `NULL`). Debemos usar `lock` cuando un apuntador `shared_ptr` vacío no se considera como error. Podremos acceder al recurso una vez que tengamos un apuntador `shared_ptr` al mismo. Los apuntadores `weak_ptr` se deben utilizar en cualquier situación en la que es necesario observar el recurso, pero no queremos asumir ninguna responsabilidad de administración del mismo. El siguiente ejemplo demuestra el uso de apuntadores `weak_ptr` en datos referenciales circulares, una situación en la que dos objetos hacen referencia uno al otro en forma interna.

Ejemplo sobre el uso de weak_ptr

En las figuras 24.9 a 24.12 se definen las clases `Autor` y `Libro`. Cada clase tiene un apuntador a una instancia de la otra clase. Esto crea una referencia circular entre las dos clases. Observe que utilizamos apuntadores `weak_ptr` y `shared_ptr` para guardar la referencia cruzada a cada clase (figuras 24.9 y 24.10, líneas 21 y 22 en cada figura). Si establecemos los apuntadores `shared_ptr`, se crea una fuga de memoria; pronto le explicaremos por qué y le mostraremos cómo puede utilizar los apuntadores `weak_ptr` para corregir este problema.

```

1 // Fig. 24.9: Autor.h
2 // Definición de la clase Autor.
3 #ifndef AUTOR_H
4 #define AUTOR_H
5 #include <string>
6 using std::string;
7

```

Figura 24.9 | Definición de la clase Autor. (Parte 1 de 2).

```

8 #include "boost/shared_ptr.hpp"
9 #include "boost/weak_ptr.hpp"
10
11 class Libro; // declaración anticipada de la clase Libro
12
13 // definición de la clase Autor
14 class Autor
15 {
16 public:
17     Autor( const string &nombreAutor ); // constructor
18     ~Autor(); // destructor
19     void imprimirTituloLibro(); // imprime el título del Libro
20     string nombre; // nombre del Autor
21     boost::weak_ptr< Libro > libroWeakPtr; // Libro que escribió el Autor
22     boost::shared_ptr< Libro > libroSharedPtr; // Libro que escribió el Autor
23 };
24 #endif // AUTOR_H

```

Figura 24.9 | Definición de la clase Autor. (Parte 2 de 2).

```

1 // Fig. 24.10: Libro.h
2 // Definición de la clase Libro.
3 #ifndef LIBRO_H
4 #define LIBRO_H
5 #include <string>
6 using std::string;
7
8 #include "boost/shared_ptr.hpp"
9 #include "boost/weak_ptr.hpp"
10
11 class Autor; // declaración anticipada de la clase Autor
12
13 // Definición de la clase Libro
14 class Libro
15 {
16 public:
17     Libro( const string &tituloLibro ); // constructor
18     ~Libro(); // destructor
19     void imprimirNombreAutor(); // imprime el nombre del Autor
20     string titulo; // titulo del Libro
21     boost::weak_ptr< Autor > autorWeakPtr; // Autor del Libro
22     boost::shared_ptr< Autor > autorSharedPtr; // Autor del Libro
23 };
24 #endif // LIBRO_H

```

Figura 24.10 | Definición de la clase Libro.

Las clases **Autor** y **Libro** definen destructores, cada uno de los cuales muestra un mensaje para indicar cuándo se destruye una instancia de cada clase (figuras 24.11 y 24.12, líneas 19 a 22). Cada clase también define una función miembro para imprimir el **título del Libro** y el **nombre del Autor** (líneas 25 a 38 en cada figura). Recuerde que no puede tener acceso al recurso directamente a través de un apuntador **weak_ptr**, por lo que primero creamos un apuntador **shared_ptr** desde el miembro de datos **weak_ptr** (línea 28 en cada figura). Si el recurso al que hace referencia el apuntador **weak_ptr** no existe, la llamada a la función **lock** devuelve un apuntador **shared_ptr** que apunta a **NULL** y la condición falla. En caso contrario, el nuevo apuntador **shared_ptr** contiene un apuntador válido al recurso del apuntador **weak_ptr**, y podemos acceder a ese recurso. Si la condición en la línea 28 es verdadera (es decir, si **libroPtr** y **autorPtr** no son ambos **NULL**), imprimimos la cuenta de referencias para mostrar que se incrementó con la creación del nuevo **shared_ptr**, y después imprimimos el **título del Libro** y el **nombre del Autor**. El apuntador **shared_ptr** se destruye cuando la función termina, por lo que la cuenta de referencias se decrementa en uno.

```

1 // Fig. 24.11: Autor.cpp
2 // Definiciones de las funciones miembro para la clase Autor.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 #include "Autor.h"
11 #include "Libro.h"
12 #include "boost/shared_ptr.hpp"
13 #include "boost/weak_ptr.hpp"
14
15 Autor::Autor( const string &nombreAutor ) : nombre( nombreAutor )
16 {
17 }
18
19 Autor::~Autor()
20 {
21     cout << "Destruyendo Autor: " << nombre << endl;
22 } // fin del destructor
23
24 // imprime el título del Libro que escribió este Autor
25 void Autor::imprimirTituloLibro()
26 {
27     // si libroWeakPtr.lock() devuelve un shared_ptr no vacío
28     if ( boost::shared_ptr< Libro > libroPtr = libroWeakPtr.lock() )
29     {
30         // muestra el incremento en la cuenta de referencias e imprime el título del Libro
31         cout << "La cuenta de referencias para el Libro " << libroPtr->titulo
32         << " es " << libroPtr.use_count() << "." << endl;
33         cout << "El autor " << nombre << " escribió el libro " << libroPtr->titulo
34         << "\n" << endl;
35     } // fin de if
36     else // libroWeakPtr apunta a NULL
37         cout << "Este Autor no tiene un Libro." << endl;
38 } // fin de imprimirTituloLibro

```

Figura 24.11 | Definiciones de las funciones miembro de Autor.

```

1 // Fig. 24.12: Libro.cpp
2 // Definiciones de las funciones miembro para la clase Libro.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 #include "Autor.h"
11 #include "Libro.h"
12 #include "boost/shared_ptr.hpp"
13 #include "boost/weak_ptr.hpp"
14
15 Libro::Libro( const string &tituloLibro ) : titulo( tituloLibro )
16 {
17 }
18
19 Libro::~Libro()

```

Figura 24.12 | Definiciones de las funciones miembro de Libro. (Parte I de 2).

```

20  {
21      cout << "Destruyendo Libro: " << titulo << endl;
22  } // fin del destructor
23
24 // imprime el nombre del Autor de este Libro
25 void Libro::imprimirNombreAutor()
26 {
27     // si autorWeakPtr.lock() devuelve un shared_ptr no vacío
28     if ( boost::shared_ptr< Autor > autorPtr = autorWeakPtr.lock() )
29     {
30         // muestra el incremento en la cuenta de referencias e imprime el nombre del Autor
31         cout << "La cuenta de referencias para el Autor " << autorPtr->nombre
32         << " es " << autorPtr.use_count() << "." << endl;
33         cout << "El libro " << titulo << " fue escrito por "
34         << autorPtr->nombre << "\n" << endl;
35     } // fin de if
36     else // autorWeakPtr apunta a NULL
37     cout << "Este Libro no tiene Autor." << endl;
38 } // fin de imprimirNombreAutor

```

Figura 24.12 | Definiciones de las funciones miembro de *Libro*. (Parte 2 de 2).

En la figura 24.13 se define una función `main` que demuestra la fuga de memoria ocasionada por la referencia circular entre las clases `Autor` y `Libro`. En las líneas 14 a 16 se crean apuntadores `shared_ptr` a una instancia de cada clase. Los datos miembro de `weak_ptr` se establecen en las líneas 19 y 20. En las líneas 23 y 24 se establecen los datos miembro de `shared_ptr` para cada clase. Las instancias de las clases `Autor` y `Libro` ahora se hacen referencia una a la otra. Después imprimimos la cuenta de referencias para los apuntadores `shared_ptr`, con lo cual demostramos que cada instancia es referenciada por dos apuntadores `shared_ptr` (líneas 27 a 30), los que creamos en la función `main` y el miembro de datos de cada instancia. Recuerde que los apuntadores `weak_ptr` no afectan a la cuenta de referencias. Después llamamos a la función miembro de cada clase para imprimir la información almacenada en el miembro de datos `weak_ptr` (líneas 35 y 36). Las funciones también muestran el hecho de que se creó otro apuntador `shared_ptr` durante la llamada a la función. Por último, imprimimos las cuentas de referencias de nuevo, para mostrar que los apuntadores `shared_ptr` creados en las funciones miembro `imprimirNombreAutor` e `imprimirTituloLibro` se destruyen cuando termina la función.

```

1 // Fig. 24.13: fig24_13.cpp
2 // Demuestra el uso de boost::weak_ptr.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Autor.h"
8 #include "Libro.h"
9 #include "boost/shared_ptr.hpp"
10
11 int main()
12 {
13     // crea un Libro y un Autor
14     boost::shared_ptr< Libro > libroPtr( new Libro( "C++ How to Program" ) );
15     boost::shared_ptr< Autor > autorPtr(
16         new Autor( "Deitel & Deitel" ) );
17
18     // hace que Libro y Autor se referencien entre si
19     libroPtr->autorWeakPtr = autorPtr;
20     autorPtr->libroWeakPtr = libroPtr;
21
22     // establece los datos miembro de shared_ptr para crear la fuga de memoria
23     libroPtr->autorSharedPtr = autorPtr;

```

Figura 24.13 | Los apuntadores `shared_ptr` producen una fuga de memoria en los datos referenciales circulares. (Parte 1 de 2).

```

24 autorPtr->libroSharedPtr = libroPtr;
25
26 // la cuenta de referencias para libroPtr y autorPtr es uno
27 cout << "La cuenta de referencias para el Libro " << libroPtr->titulo << " es "
28     << libroPtr.use_count() << endl;
29 cout << "La cuenta de referencias para el Autor " << autorPtr->nOMBRE << " es "
30     << autorPtr.use_count() << "\n" << endl;
31
32 // accede a las referencias cruzadas para imprimir los datos a los que apuntan
33 cout << "Acceso al nombre del Autor y al titulo del Libro mediante "
34     << "apuntadores weak_ptr." << endl;
35 libroPtr->imprimirNombreAutor();
36 autorPtr->imprimirTituloLibro();
37
38 // la cuenta de referencias para cada shared_ptr regresa a uno
39 cout << "La cuenta de referencias para el Libro " << libroPtr->titulo << " es "
40     << libroPtr.use_count() << endl;
41 cout << "La cuenta de referencias para el Autor " << autorPtr->nOMBRE << " es "
42     << autorPtr.use_count() << "\n" << endl;
43
44 // los apuntadores shared_ptr quedan fuera de alcance, el Libro y Autor se destruyen
45 cout << "Los apuntadores shared_ptr estan quedando fuera de alcance." << endl;
46
47 return 0;
48 } // fin de main

```

La cuenta de referencias para el Libro C++ How to Program es 2
 La cuenta de referencias para el Autor Deitel & Deitel es 2

Acceso al nombre del Autor y al titulo del Libro mediante apuntadores weak_ptr.

La cuenta de referencias para el Autor Deitel & Deitel es 3.

El libro C++ How to Program fue escrito por Deitel & Deitel

La cuenta de referencias para Libro C++ How to Program es 3.

El autor Deitel & Deitel escribio el libro C++ How to Program

La cuenta de referencias para el Libro C++ How to Program es 2

La cuenta de referencias para el Autor Deitel & Deitel es 2

Los apuntadores shared_ptr estan quedando fuera de alcance.

Figura 24.13 | Los apuntadores shared_ptr producen una fuga de memoria en los datos referenciales circulares.
 (Parte 2 de 2).

Al final de `main`, los apuntadores `shared_ptr` a las instancias de `Autor` y `Libro` que creamos quedan fuera de alcance y se destruyen. Observe que los resultados no muestran a los destructores para las clases `Autor` y `Libro`. El programa tiene una fuga de memoria; las instancias de `Autor` y `Libro` no se destruyen debido a los datos miembro de `shared_ptr`. Cuando `libroPtr` se destruye al final de la función `main`, la cuenta de referencias para la instancia de la clase `Libro` se vuelve uno; la instancia de `Autor` aún tiene un apuntador `shared_ptr` a la instancia de `Libro`, por lo que no se elimina. Cuando `autorPtr` queda fuera de alcance y se destruye, la cuenta de referencias para la instancia de la clase `Autor` también se vuelve uno; la instancia de `Libro` aún tiene un apuntador `shared_ptr` a la instancia de `Autor`. No se elimina ninguna instancia debido a que la cuenta de referencias para cada una sigue siendo uno.

Ahora, comente las líneas 23 y 24 colocando los signos `//` al principio de cada línea. Esto evita que el código establezca los datos miembro de `shared_ptr` para las clases `Autor` y `Libro`. Vuelva a compilar el código y ejecute el programa de nuevo. En la figura 24.14 se muestran los resultados. Observe que la cuenta de referencias inicial para cada instancia es ahora uno en vez de dos, ya que no establecimos los datos miembro `shared_ptr`. Las últimas dos líneas de los resultados muestran que las instancias de las clases `Autor` y `Libro` se destruyeron al final de la función `main`. Eliminamos la fuga de memoria utilizando los datos miembro de `weak_ptr` en vez de los datos miembro de `shared_ptr`. Los apuntadores `weak_ptr` no afectan a la cuenta de referencias, pero de todas formas nos permiten acceder al recurso cuando lo necesitamos, creando un apuntador `shared_ptr` temporal al recurso. Cuando los apuntadores `shared_ptr` que creamos en `main` se destruyen, las cuentas de referencias se vuelven cero y las instancias de las clases `Autor` y `Libro` se eliminan de forma apropiada.

```

La cuenta de referencias para el Libro C++ How to Program es 1
La cuenta de referencias para el Autor Deitel & Deitel es 1

Acceso al nombre del Autor y al titulo del Libro mediante apuntadores weak_ptr.
La cuenta de referencias para el Autor Deitel & Deitel es 2.
El libro C++ How to Program fue escrito por Deitel & Deitel

La cuenta de referencias para el Libro C++ How to Program es 2.
El autor Deitel & Deitel escribio el libro C++ How to Program

La cuenta de referencias para el Libro C++ How to Program es 1
La cuenta de referencias para el Autor Deitel & Deitel es 1

Los apuntadores shared_ptr estan quedando fuera de alcance.
Destruyendo Autor: Deitel & Deitel
Destruyendo Libro: C++ How to Program

```

Figura 24.14 | Apuntadores `weak_ptr` utilizados para evitar una fuga de memoria en datos referenciales circulares.

24.9 Reporte técnico I

El Reporte técnico 1 (TR1) describe las adiciones propuestas a la Biblioteca estándar de C++. Muchas de las bibliotecas en el TR1 serán aceptadas por el Comité de estándares de C++, pero no se considerarán como parte del estándar de C++ sino hasta que se finalice la siguiente versión. Las adiciones a la biblioteca proporcionan soluciones para muchos problemas de programación comunes. En la sección 24.6 describimos las 11 bibliotecas Boost en el TR1; a continuación presentaremos descripciones de tres bibliotecas adicionales del TR1. Muchas bibliotecas no se consideraron en el TR1 debido a restricciones de tiempo. El **Reporte técnico 2 (TR2)**, que saldrá a la luz poco después de C++0x, contiene proposiciones de bibliotecas adicionales que no se incluyeron en el TR1. La liberación del TR2 traerá aún más funcionalidad a la biblioteca estándar, sin tener que esperar a otro nuevo estándar.

Contenedores asociativos desordenados¹³

La biblioteca de Contenedores asociativos desordenados define cuatro nuevos contenedores: `unordered_set`, `unordered_map`, `unordered_multiset` y `unordered_multimap`. Estos contenedores asociativos se implementan como tablas de hash. Una **tabla de hash** se divide en secciones, a las que algunas veces se les conoce como “cubetas”. Una clave se utiliza para determinar en dónde se va a almacenar un elemento en el contenedor. La clave se pasa a una **función de hash**, la cual devuelve un valor `size_t`. El valor `size_t` devuelto por la función de hash determina la “cubeta” en la que se coloca el valor. Si dos valores son iguales, también lo son los valores `size_t` devueltos por la función de hash. Se pueden colocar múltiples valores en la misma “cubeta”. Para obtener un elemento del contenedor, se utiliza la clave de la misma forma que con un objeto `set` o `map`. La clave determina en qué “cubeta” se colocó el valor, y después se busca el valor en esa “cubeta”.

Con `unordered_set` y `unordered_multiset`, se utiliza el mismo elemento como la clave. `unordered_map` y `unordered_multimap` utilizan una clave separada para determinar en dónde se debe colocar el elemento; los argumentos se pasan como un objeto `pair<const Clave, Valor>`. `unordered_set` y `unordered_map` requieren que todas las claves que se utilicen sean únicas; `unordered_multiset` y `unordered_multimap` no implementan esa restricción. Los contenedores se definen en los encabezados `<unordered_set>` y `<unordered_map>`.

Funciones matemáticas especiales¹⁴

Esta biblioteca incorpora funciones matemáticas que se agregaron al **C99** (el estándar de C publicado en 1999) y que no están presentes en el Estándar de C++. C99 proporciona funciones trigonométricas, hiperbólicas, exponenciales, logarítmicas, de potencias y especiales. Esta biblioteca agrega esas funciones, entre otras, a C++ en el encabezado `<cmath>`.

Compatibilidad mejorada con C99¹⁵

C++ evolucionó del lenguaje de programación C. La mayoría de los compiladores de C++ pueden también compilar programas de C, pero hay ciertas incompatibilidades entre los lenguajes. El objetivo de esta biblioteca es incrementar la

-
- 13. Matthew Austern, “A Proposal to Add Hash Tables to the Standard Library”, Documento número N1456=03-0039, abril 9, 2003, www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1456.html.
 - 14. Walter E. Brown, “A Proposal to Add Mathematical Special Functions to the C++ Standard Library”, Documento número N1422=03-0004, febrero 24, 2003, std.dkuug.dk/jtc1/sc22/wg21/docs/papers/2003/n1422.html.
 - 15. P. J. Plauger, “Proposed Additions to TR-1 to Improve Compatibility With C99”, Documento número N1568=04-0008, www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1568.htm.

compatibilidad entre C++ y C99. La mayor parte de esta biblioteca involucra la adición de elementos a los encabezados de C++ para soportar las características de C99; a menudo esto se logra al incluir los encabezados correspondientes de C99.

24.10 C++0x

El Comité de estándares de C++ está revisando actualmente el Estándar de C++. El último estándar se publicó en 1998. El trabajo en el nuevo estándar, que actualmente se conoce como C++0x, comenzó en el 2003. El nuevo estándar, que probablemente se liberará en el 2009, incluye las bibliotecas y adiciones al lenguaje básico del TR1. Explore la sección sobre C++0x en el Centro de recursos de C++ de Deitel en www.deitel.com/cplusplus/ y haga clic en **C++0x** en la lista **Categories** para obtener información actual acerca de C++0x.

Proceso de estandarización

La **Organización internacional para la estandarización (ISO)** supervisa la creación de estándares internacionales para los lenguajes de programación, incluyendo los de C y C++. Cada adición o modificación al estándar actual de C++ debe ser aprobada por el Grupo de trabajo 21 (WG21) de ISO/IEC JTC 1/SC 22, el comité que mantiene el estándar de C++. Este comité de voluntarios de la comunidad de programación de C++ se reúne dos veces al año para hablar acerca de las cuestiones relacionadas con el estándar. Con frecuencia se llevan a cabo reuniones no oficiales más pequeñas para considerar las proposiciones entre las reuniones oficiales del comité. ISO requiere por lo menos 5 años entre cada nuevo borrador de un estándar.

Objetivos para C++0x¹⁶

Bjarne Stroustrup, creador del lenguaje de programación C++, ha expresado su visión para el futuro de C++; los objetivos principales para el nuevo estándar son facilitar el aprendizaje de C++, mejorar las herramientas para generar bibliotecas e incrementar la compatibilidad con el lenguaje de programación C.

24.11 Cambios en el lenguaje básico

En el sitio www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2228.html. Encontrará una lista de cambios propuestos al lenguaje básico. También hay vínculos a los documentos asociados con cada proposición. Hablaremos brevemente sólo acerca de los cambios en el lenguaje básico que han sido aceptados en el borrador de trabajo del nuevo estándar. Es probable que el número de proposiciones que se incluyen en el borrador de trabajo aumente antes de finalizar el estándar. El compilador GCC de C++ tiene un modo opcional de C++0x, el cual nos permite experimentar con varios de los cambios en el lenguaje básico (gcc.gnu.org/gcc-4.3/cxx0x_status.html).

Referencia rvalue¹⁷

El tipo de referencia *rvalue* en C++0x nos permite enlazar un *rvalue* (objeto temporal) a una referencia no *const*. Una referencia *rvalue* se declara como *T&&* (en donde *T* es el tipo del objeto al que se hace referencia) para diferenciarla de una referencia normal *T&* (que ahora se conoce como referencia *lvalue*). Una referencia *rvalue* se puede utilizar para implementar de manera efectiva la semántica de movimiento. Esto mejora el rendimiento al evitar que el compilador realice copias adicionales de objetos grandes, una operación costosa. Las referencias *rvalue* también se pueden utilizar en las “funciones de avance”: objetos de función que adaptan una función para que reciba menos argumentos (por ejemplo, *std::bind1st* o los objetos de función creados mediante el uso de *Boost.Bind*). Por lo general, cada parámetro de referencia necesitaría una versión *const* y una versión no *const* para tomar en cuenta los *lvalues*, los *lvalues const* y los *rvalues*. Con las referencias *rvalue* sólo se necesita una función de avance.

Aclaración de la inicialización de objetos de clases mediante rvalues¹⁸

Esta proposición aclara la formulación en el estándar relacionada con el uso de *rvalues* como argumentos para funciones que reciben sus argumentos por valor. La nueva formulación permitiría a los escritores de compiladores implementar la semántica de movimiento en vez de la semántica de copia, si se puede determinar que el *rvalue* se va a eliminar de la memoria después de la copia. Esto aumentaría el rendimiento al eliminar una llamada adicional al constructor de copia.

-
- 16. Bjarne Stroustrup, “The Design of C++0x”, mayo 2005, www.research.att.com/~bs/rules.pdf.
 - 17. Howard E. Hinnant, “A Proposal to Add an *rvalue* Reference to the C++ Language”, octubre 19, 2006, Documento número N2118=06-0188, www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2118.html.
 - 18. David Abrahams y Gary Powell, “Clarification of Initialization of Class Objects by *rvalues*”, Documento número N1610=04-0050, febrero 14, 2004, www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1610.html.

static_assert¹⁹

La declaración **static_assert** nos permite evaluar ciertos aspectos del programa en tiempo de compilación. Una declaración **static_assert** recibe una expresión integral constante y una literal **string**. Si la expresión se evalúa como 0 (false), el compilador reporta el error. El mensaje de error incluye la literal **string** que se proporciona en la declaración. La declaración **static_assert** se puede utilizar en alcance de espacio de nombres, de clase o de bloque.

La adición de **static_assert** facilita el aprendizaje de C++. Las aserciones se pueden utilizar para proporcionar mensajes de error más informativos cuando los programadores novatos cometan errores comunes, como utilizar el tipo incorrecto de argumento en la llamada a una función o en el instanciamiento de plantillas. También son útiles en el desarrollo de bibliotecas; el uso incorrecto de la biblioteca se puede reportar de una manera mucho más efectiva.

extern template²⁰

La palabra clave **extern** indica que una variable o función está definida, ya sea antes en el archivo actual, o en un archivo separado. Esta proposición permite que la palabra clave **extern** proporcione la misma funcionalidad para las plantillas, lo cual ayudaría a evitar instanciamientos adicionales de la plantilla. Esto puede mejorar los tiempos de compilación y reducir el tamaño del código objeto de un programa.

Declaraciones friend extendidas²¹

Esta proposición aclara la formulación en el estándar C++ respecto a las declaraciones **friend** y las plantillas.

Sincronización del preprocesador de C++ con C99²²

Esta proposición es un esfuerzo por hacer que C++0x sea más compatible con C99, al cambiar los requerimientos del preprocesador de C++. Estos cambios de compatibilidad implican, en su mayor parte, copiar texto directamente del estándar C99 y colocarlo en el estándar de C++. Los cambios afectan áreas que incluyen macros predefinidas, el operador **pragma**, la concatenación de cadenas y los nombres de archivos de encabezado y de inclusión, y cambios en los límites de traslación. Al incorporar estos cambios en el estándar, se forzaría a los implementadores de compiladores a crear un compilador y un preprocesador de C++ con las mismas reglas y características que el compilador y preprocesador de C99, con lo cual los lenguajes serían más compatibles.

Comportamiento soportado en forma condicional²³

Esta proposición agrega el **comportamiento soportado en forma condicional** a C++. Muchos implementadores de C++ agregan una funcionalidad opcional. Utilizar esta funcionalidad con una implementación distinta que no la soporta se considera actualmente como un comportamiento indefinido. No hay garantía acerca de cómo actuará el programa cuando ocurre un comportamiento indefinido. En la actualidad, los compiladores no tienen que proporcionar advertencias en estas situaciones. Bajo la categoría del comportamiento soportado en forma condicional, cualquier característica opcional que no esté soportada por la implementación actual hace que el compilador devuelva un reporte de que el programa utiliza una característica no soportada. Esto ayuda a los programadores novatos, que pueden llegar a utilizar sin saber las características no estándar. Ellos recibirán un mensaje informativo, en vez de tener que diagnosticar los efectos del comportamiento indefinido.

Cambiar el comportamiento indefinido por errores diagnosticables²⁴

Otra proposición, inspirada por las discusiones relacionadas con el comportamiento soportado en forma condicional, cambia ciertas situaciones que producen un comportamiento indefinido y las convierte en errores diagnosticables (un error que el compilador puede reportar). Las áreas que requerirán un reporte de diagnóstico incluyen los enteros que

19. Robert Klarer, Dr. John Maddock, Beman Dawes y Howard Hinnant, “Proposal to Add Static Assertions to the Core Language”, Documento número N1720, octubre 20, 2004, www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1987.htm.
20. John Spicer, “Adding **extern template**”, Documento número N1987, abril 6, 2006, www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1987.htm.
21. William M. Miller, “Extended **friend** Declarations”, Documento número N1791, mayo 1, 2005, www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1791.pdf.
22. Clark Nelson, “Working Draft Changes for C99 Preprocessor Synchronization”, Documento número N1653, julio 16, 2004, www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1653.htm.
23. William M. Miller, “Conditionally-Supported Behavior”, Documento número N1627, abril 4, 2004, www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1627.pdf.
24. William M. Miller, “Changing Undefined Behavior into Diagnosable Errors”, Documento número N1727, noviembre 8, 2004, www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1727.pdf.

se exceden del tamaño permitido, los escapes de caracteres no soportados y el paso de objetos que no son POD (“datos antiguos simples”) a las funciones “variadic” (funciones que reciben un número no especificado de argumentos).

Agregar el tipo `long long` a C++²⁵

El nuevo tipo `long long` es un tipo de entero que debe tener por lo menos 64 bits (8 bytes). Este tipo se está introduciendo para mejorar la compatibilidad con el Estándar C99, que ya incluye el tipo `long long`.

Agregar tipos de enteros extendidos a C++²⁶

La adición de tipos de enteros extendidos a C++ es parte del esfuerzo de agregar características de C99 a C++0x. Un entero extendido es un tipo de entero definido por una implementación, además de los tipos de enteros requeridos por el Estándar de C++. La implementación de un tipo de entero extendido no es obligatoria, pero deben seguirse ciertas reglas cuando un distribuidor opta por proporcionarlos. Cualquier tipo de entero extendido con signo debe tener también un tipo correspondiente sin signo del mismo tamaño. Observe que al utilizar de manera explícita un tipo de entero extendido se ve afectada la portabilidad de los programas.

Delegación de constructores²⁷

Esta característica permite delegar un constructor a otro de los constructores de la clase (es decir, llamar a otro de los constructores de la clase). Esto facilita la escritura de constructores sobrecargados. En la actualidad, un constructor sobre cargado debe duplicar el código común para el otro constructor. Esto produce código repetitivo y propenso a errores. Un error en un constructor podría ocasionar inconsistencia en la inicialización de objetos. Al llamar a otra versión del constructor, el código común no necesita repetirse y se reduce la probabilidad de error.

Signos >²⁸

Es necesario colocar un espacio entre los signos `>` al utilizar plantillas anidadas. Sin el espacio, el compilador asume que los dos signos `>>` son el operador de desplazamiento a la derecha (`>>`). Esto significa que al escribir `vector<class<T>` se produce un error del compilador. La instrucción tendría que escribirse como `vector<class<T>>`. Muchos novatos se tropiezan con esta peculiaridad de C++. La proposición es cambiar el compilador de C++ para que reconozca cuando `>>` forma parte de una plantilla, en vez de ser el operador de desplazamiento a la derecha.

Deducción del tipo de variable a partir de su inicializador²⁹

Esta proposición define una nueva funcionalidad para la palabra clave `auto`: determina automáticamente los tipos de las variables, con base en la expresión inicializadora. `auto` se puede utilizar en vez de los tipos largos y complicados cuyo tipo no se puede determinar en forma manual. `auto` también se puede utilizar con los calificadores `const` y `volatile`. Podemos crear apuntadores y referencias con `auto` de igual forma que con el nombre completo del tipo. `auto` soporta la declaración de múltiples variables en una instrucción (por ejemplo, `auto x = 1, y = 2`). El objetivo de la palabra clave `auto` es ahorrar tiempo, facilitar el proceso de aprendizaje y mejorar la programación genérica. El siguiente código crea un vector de instancias de una clase hipotética llamada `Class< T >`.

```
vector< Class< T > > miVector;
vector< Class< T > >::const_iterator iterador = miVector.begin()
```

Mediante el uso de `auto`, la declaración de `iterador` se puede escribir así:

```
auto iterador = miVector.begin();
```

El tipo de `iterador` es `vector<Class< T > >::const_iterator`. También podemos crear dos variables del mismo tipo en una declaración. Ambas variables en

```
auto iteradorBegin = miVector.begin(), iteradorEnd = miVector.end();
```

- 25. J. Stephen Adamczyk, “Adding the `long long` Type to C++”, Documento número N1811, abril 29, 2005, www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1811.pdf.
- 26. J. Stephen Adamczyk, “Adding Extended Integer Types to C++”, Documento número N1988, abril 19, 2006, www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1988.pdf.
- 27. Herb Sutter y Francis Glassborow, “Delegating Constructors”, Documento número N1986=06-0056, abril 6, 2006, www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1986.pdf.
- 28. Daveed Vandevoorde, “Right Angle Brackets”, Documento número N1857=05-0017, enero 14, 2005, www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1757.html.
- 29. Jaakko Järvi, Bjarne Stroustrup y Gabriel Dos Reis, “Deducing the Type of Variable From Its Initializer Expression”, Documento número N1984=06-0054, abril 6, 2006, www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1984.pdf.

se crean con el tipo `vector<Class< T >>::const_iterator`. También podemos usar `auto` con los calificadores `const` o `volatile` y crear apuntadores o referencias. La instrucción

```
const auto & iteratorPtr = miVector.begin();
```

crea una referencia `const` a un `vector< Class< T >>::const_iterator`. `auto` puede ahorrarnos mucho tiempo, al determinar de manera automática el tipo de la variable que se está declarando.

*Plantillas variadic*³⁰

Una plantilla variadic acepta cualquier número de argumentos. Los parámetros de la plantilla se colocan en un paquete de parámetros de tipo de plantilla. Los argumentos que se pasan a una función se colocan en un paquete de parámetros de función. Estos paquetes trabajan en conjunto para administrar los tipos y los valores de los parámetros, respectivamente. Tres puntos suspensivos (“...”) a la izquierda del nombre de un parámetro de plantilla o función lo declaran como paquete de parámetros de tipo de plantilla o paquete de parámetros de función, respectivamente. Los parámetros se deben sacar de los paquetes para poder utilizarlos. Tres puntos suspensivos a la derecha de un parámetro de función extraen todos los elementos del paquete de parámetros. El n -ésimo elemento de un paquete de parámetros de tipo de plantilla es el tipo del n -ésimo elemento en un paquete de parámetros de función. Las plantillas variadic facilitan la acción de expresar clases y funciones que pueden recibir un número arbitrario de argumentos. Muchas bibliotecas, como la biblioteca `tuple` en el TR1, pueden beneficiarse del uso de plantillas variadic.

*Alias de plantillas*³¹

A menudo las bibliotecas utilizan plantillas con muchos parámetros para implementar la programación genérica. Puede haber situaciones en las que sería conveniente poder especificar ciertos argumentos para la plantilla que permanecen consistentes, pero que aun así el resto pueda variar. Esto se puede hacer mediante un alias de plantilla. Un alias de plantilla es similar a una definición `typedef`; introduce un nombre que se utiliza para hacer referencia a una plantilla. En una `typedef`, se especifican todos los parámetros de plantilla. Al usar un alias de plantilla, ciertos parámetros se especifican y otros aun pueden variar. Podemos usar una plantilla de propósito general en un papel más específico, en el que muchos de los argumentos siempre son iguales, mediante el uso de un alias de plantilla para establecer los parámetros consistentes y poder seguir variando aquellos parámetros que cambian en cada instanciamiento.

*Nuevos tipos de caracteres*³²

Esta proposición agrega soporte de caracteres Unicode a C++. Unicode ya es soportado por el estándar de C, y la mayor parte de ese trabajo se duplica en el estándar de C++. Se introducen dos nuevos tipos: `char16_t` y `char32_t`. Estos tipos representan los caracteres Unicode de 16 bits y 32 bits. Se pueden utilizar los prefijos `u` y `U` para denotar una literal tipo carácter `char16_t` o `char32_t`, respectivamente.

*Extensión de sizeof*³³

El operador `sizeof` devuelve el número de bytes utilizados para almacenar un objeto. C++0x permite llamar a `sizeof` en un miembro de datos sin acceder al mismo a través de una instancia de la clase de la cual forma parte. `sizeof` también se puede utilizar de una forma más natural con datos miembro `static`.

*Alternativa a los puntos de secuencia*³⁴

El estándar de C++ describe el orden en el que ocurren las operaciones en un programa, utilizando el término “puntos de secuencia”. Muchos expertos en C++ piensan que la formulación no describe con claridad la secuencia requerida de eventos en la ejecución de un programa. Esta proposición cambia la formulación en el estándar para definir los requerimientos de secuencia con más precisión. Esta cuestión es importante para C++, ya que cualquier intento futuro de agregar la concurrencia (es decir, el subprocesamiento múltiple) requiere una sólida comprensión del orden de los eventos en un programa.

30. Douglas Gregor, Jaakko Järvi y Gary Powell, “Variadic Templates”, Documento número N2080=06-0150, septiembre 9, 2006, www.osf.iu.edu/~dgregor/cpp/variadictemplates.pdf.
31. Gabriel Dos Reis y Mat Marcus, “Proposal to Add Template Aliases to C++”, Documento número N1449=03-0032, abril 7, 2003, www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1449.pdf.
32. Lawrence Crowl, “New Character Types in C++”, Documento número N2149=07-0009, enero 10, 2007, www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2149.html.
33. Jens Maurer, “Extending sizeof to Apply to Non-Static Data Members Without an Object”, Documento número N2150=07-0010, enero 7, 2007, www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2150.html.
34. Clark Nelson, “A Finer-Grained Alternative to Sequence Points”, Documento número N1944=06-0014, febrero 17, 2006, www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1944.htm.

24.12 Repaso

En este capítulo hablamos sobre varios aspectos del futuro de C++. Presentamos las Bibliotecas Boost de C++ y describimos las bibliotecas Boost que se incluyen en el TR1: las adiciones a la Biblioteca estándar de C++.

Hablamos sobre la biblioteca `Boost.Regex` y los símbolos que se utilizan para formar expresiones regulares. Proporcionamos ejemplos acerca de cómo utilizar las clases de `Boost.Regex`, incluyendo `regex`, `match_results` y `regex_token_iterator`. Aprendió a buscar patrones en una cadena, y a relacionar cadenas completas con patrones mediante los algoritmos `regex_search` y `regex_match` de `Boost.Regex`. Demostramos cómo reemplazar caracteres en una cadena con `regex_replace` y cómo dividir cadenas en tokens con un `regex_token_iterator`.

Vimos ejemplos acerca del uso de la biblioteca `Boost.Smart_ptr`. Hablamos sobre los apuntadores inteligentes incluidos en el TR1, a saber, `shared_ptr` y `weak_ptr`. Le mostramos cómo usar estas clases para evitar fugas de memoria al utilizar memoria asignada en forma dinámica. Demostramos cómo utilizar funciones eliminadoras personalizadas para permitir que los apuntadores `shared_ptr` administren los recursos que requieren procedimientos de destrucción especiales. También explicamos cómo se pueden utilizar los apuntadores `weak_ptr` para evitar fugas de memoria en datos referenciales circulares.

Vimos las generalidades acerca del estándar revisado que está por venir, C++0x, con discusiones acerca del TR1 y de los cambios al lenguaje básico. Presentamos cada una de las bibliotecas aceptadas en el TR1. Describimos las nuevas características del lenguaje básico, incluyendo la palabra clave `auto`, la referencia `rvalue`, las mejoras en la compatibilidad con el C99, los tipos de enteros adicionales y el nuevo concepto del comportamiento soportado en forma condicional. Recuerde que Boost, el TR1 y C++0x están cambiando en forma continua; visite nuestros Centros de recursos para que permanezca actualizado con los tres.

Resumen

Sección 24.2 Centros de recursos de C++ (y relacionados) en línea de Deitel

- Visite el Centro de recursos de las Bibliotecas Boost de C++ en www.deitel.com/CPlusPlusBoostLibraries/ para encontrar información actual acerca de las bibliotecas disponibles y nuevas versiones.
- Puede encontrar información actual acerca del TR1 y C++0x en la sección **C++0x** del Centro de recursos de C++ en www.deitel.com/cplusplus/.
- Para obtener más información acerca de Visual C++, visite nuestro Centro de recursos de Visual C++ en www.deitel.com/VisualCPlusPlus/.

Sección 24.3 Bibliotecas Boost

- Las Bibliotecas Boost en www.boost.org proporcionan bibliotecas gratuitas de C++, revisadas por miembros de la comunidad de C++.
- El código fuente abierto Boost se hospeda en SourceForge (sourceforge.net).
- Para obtener información acerca del software de código fuente abierto y su desarrollo, visite nuestro Centro de recursos de código fuente abierto en www.deitel.com/OpenSource/.

Sección 24.4 Cómo agregar una nueva biblioteca a Boost

- Las bibliotecas Boost deben conformarse al estándar de C++ y usar la Biblioteca Estándar de C++ (u otras bibliotecas Boost apropiadas).
- Se publica una submisión preliminar de cada biblioteca Boost en el Boost Sandbox Vault.
- El administrador de revisiones se asegura que el código esté listo para la revisión formal, establece el itinerario de revisión, lee todas las reseñas de los usuarios, y toma la decisión final en cuanto a aceptar o no la biblioteca.
- La Licencia de Software Boost otorga el derecho de copiar, modificar, utilizar y distribuir el código fuente de Boost y los archivos binarios para uso comercial y no comercial.

Sección 24.5 Instalación de las bibliotecas Boost

- En www.boost.org/more/getting_started/index.html hay una guía de instalación disponible.
- Descargue las bibliotecas Boost y el sistema de generación `Boost.Jam` de sourceforge.net/project/showfiles.php?group_id=7856.
- Para instalar `Boost.Jam`, extraiga el archivo y copie el archivo `bjam.exe` a cualquier carpeta accesible por la variable de entorno `path` en su sistema.
- Extraiga las bibliotecas Boost del archivo que descargó.

- Abra una ventana de símbolo del sistema y cambie al directorio que contiene las bibliotecas Boost. Para instalar, escriba `bjam --toolset=msvc install`; sustituya "msvc" con cualquier otro conjunto de herramientas que deseé utilizar.
- Agregue el directorio de inclusión (`C:\Boost\include\boost-1_34`) a los directorios de inclusión adicionales.
- Agregue el directorio de biblioteca (`C:\Boost\lib`) a los directorios de biblioteca adicionales.
- Algunas bibliotecas requieren que se agregue el archivo de biblioteca como una dependencia adicional al proyecto.

Sección 24.6 Las bibliotecas Boost en el Reporte técnico 1 (TR1)

- El TR1 es una descripción de los cambios y adiciones a la Biblioteca estándar de C++.
- GCC (Colección de compiladores de GNU) proporciona una implementación parcial del TR1. Para ver las generalidades acerca de las características soportadas del TR1, visite gcc.gnu.org/onlinedocs/libstdc++/ext/tr1.html.
- `Boost.Array` proporciona arreglos de tamaño fijo que soportan la interfaz de contenedores de la STL.
- `Boost.Bind` extiende la funcionalidad proporcionada por las funciones estándar `bind1st` y `bind2nd`. Nos permite adaptar funciones que ocupan hasta nueve argumentos. También facilita el reordenamiento de los argumentos que se pasan a la función.
- `Boost.Function` nos permite almacenar apuntadores a funciones, apuntadores a funciones miembro y objetos de función en una envoltura de función. También podemos almacenar una referencia a un objeto de función, utilizando las funciones `ref` y `cref` agregadas al encabezado `<utility>`. Un objeto `function` puede contener cualquier función cuyos argumentos y tipo de valor de retorno puedan convertirse para que coincidan con la firma de la envoltura de función.
- `Boost.mem_fn` nos permite crear un objeto de función con un apuntador, referencia o apuntador inteligente a una función miembro. `mem_fn` es una versión más flexible de `mem_fun` y `mem_fun_ref`.
- `Boost.Random` nos permite crear una variedad de generadores de números aleatorios y distribuciones.
- Un generador de números seudoaleatorios utiliza un estado inicial para producir números que parecen aleatorios; si se utiliza el mismo estado inicial se produce la misma secuencia de números.
- La biblioteca `Boost.Ref` proporciona envolturas de referencias que nos permiten utilizar referencias en algoritmos que generalmente reciben sus argumentos por valor. El uso de referencias mejora el rendimiento cuando se pasan objetos grandes a un algoritmo.
- Las expresiones regulares se utilizan para relacionar patrones de caracteres en un texto.
- Con `Boost.Regex` podemos buscar una expresión específica en un objeto `string`, reemplazar partes de un objeto `string` que coincidan con una expresión regular y dividir un objeto `string` en tokens mediante el uso de expresiones regulares.
- La plantilla de clase `result_of` determina el tipo de valor de retorno de una expresión de llamada, con base en los parámetros que se pasan a la expresión de llamada.
- `Boost.Smart_ptr` define apuntadores inteligentes que nos ayudan a administrar los recursos asignados en forma dinámica.
- Los apuntadores `shared_ptr` manejan la administración del tiempo de vida de los objetos asignados en forma dinámica. La memoria se libera automáticamente cuando no hay apuntadores `shared_ptr` que hagan referencia a ella.
- Un apuntador `weak_ptr` nos permite observar el valor de un apuntador `shared_ptr` sin asumir responsabilidades administrativas.
- `Boost.Tuple` nos permite crear conjuntos de objetos que pueden ser utilizados por funciones genéricas.
- Las clases `type_traits` nos permiten especificar rasgos de un tipo y realizar transformaciones de tipos para permitir utilizar el objeto en código genérico.

Sección 24.7 Uso de expresiones regulares con la biblioteca Boost.Regex

- Las expresiones regulares son objetos `string` con formato especial que se utilizan para buscar patrones en el texto.
- `basic_regex` representa una expresión regular.
- El algoritmo `regex_match` devuelve `true` sólo si un objeto `string` completo coincide con la expresión regular.
- El algoritmo `regex_search` devuelve `true` si cualquier parte de un objeto `string` coincide con la expresión regular.
- Para utilizar la biblioteca `Boost.Regex` debemos incluir el archivo de encabezado "`boost/regex.hpp`".
- Una clase de carácter representa un grupo de caracteres.
- Un carácter de palabra (`\w`) es cualquier carácter alfanumérico o guión bajo. Un carácter de espacio en blanco (`\s`) es un espacio, tabulador, retorno de carro, nueva línea o avance de página. Un dígito (`\d`) es cualquier carácter numérico.

Sección 24.7.1 Ejemplo de una expresión regular

- Debemos anteponer al carácter de barra diagonal inversa de una clase de carácter un carácter de barra diagonal inversa adicional en las cadenas de C++.
- Para especificar conjuntos de caracteres que no sean los que pertenecen a una clase de carácter predefinida, debemos listar los caracteres entre corchetes (`[]`). Los rangos de caracteres se representan colocando un `"-"` entre dos caracteres. Las instancias del carácter `"-"` fuera de los caracteres `[]` se tratan como literales.
- Debemos colocar `^` como el primer carácter en los corchetes para especificar que un patrón debe coincidir con cualquier cosa que no sean los caracteres entre los corchetes.
- Un objeto `match_results` contiene una coincidencia para una expresión regular. La definición `typedef smatch` presenta un objeto `match_results` que proporciona acceso al resultado de la coincidencia a través de un iterador `string::const_iterator`.

- `match_not_dot_newline` evita que el carácter `.` coincida con un carácter de nueva línea.
- La función miembro `suffix` de `match_results` devuelve un objeto `string` desde el final de la coincidencia hasta el final del objeto `string` en el que se realiza la búsqueda.
- El cuantificador `"*"` coincide con *cero o más* ocurrencias.
- El cuantificador `"+"` coincide con *una o más* ocurrencias.
- El cuantificador `"?"` coincide con *cero o una* ocurrencias.
- Un conjunto de llaves que contienen un número `{n}` coincide con *exactamente n* ocurrencias.
- Al incluir una coma después del número encerrado entre llaves, se buscan coincidencias con *por lo menos n* ocurrencias.
- La notación `{n,m}` coincide con *entre n y m* ocurrencias (inclusive).
- Los cuantificadores son avaros; coinciden con todas las ocurrencias posibles del patrón, hasta que ya no haya una coincidencia.
- Un cuantificador seguido de un signo de interrogación `(?)` se vuelve perezoso y coincidirá con el menor número posible de ocurrencias, mientras que haya una coincidencia exitosa.

Sección 24.7.2 Cómo validar la entrada del usuario mediante expresiones regulares

- Los caracteres `"^"` y `"$"` representan el inicio y el final de un objeto `string`, respectivamente.
- El carácter `"|"` coincide con la expresión a su izquierda *o* con la expresión a su derecha.
- Podemos aplicar cuantificadores a patrones entre paréntesis para crear expresiones regulares más complejas.

Sección 24.7.3 Cómo reemplazar y dividir cadenas

- El algoritmo `regex_replace` reemplaza texto en un objeto `string` con texto nuevo, en cualquier parte en donde el objeto `string` original coincida con una expresión regular.
- Al escapar un carácter `"*"` con una `\` indicamos al motor para relacionar expresiones regular que debe buscar el carácter `"*"` actual, en vez de usarlo como cuantificador.
- `format_first_only` indica a `regex_replace` que debe reemplazar sólo la primera subcadena que coincide con la expresión regular. Por lo general, `regex_replace` reemplaza todas las ocurrencias del patrón.
- Un iterador `regex_token_iterator` itera a través de las partes de un objeto `string` que coinciden con la expresión regular.
- Para crear un `regex_token_iterator` debemos pasar al constructor dos iteradores que representan el inicio y el final del objeto `string` sobre el cual se va a iterar, y la expresión regular con la que debe haber coincidencias.
- Debemos pasar `-1` al `regex_token_iterator` para indicar que debe iterar sobre cada subcadena que *no* coincide con la expresión regular.

Sección 24.8 Apuntadores inteligentes con Boost.Smart_ptr

- Los apuntadores inteligentes evitan errores al reforzar el proceso de asignación y desasignación de memoria.
- Una vez que se lanza una excepción, el destructor de un apuntador inteligente llamará a `delete` en el apuntador por el programador.
- Un apuntador `auto_ptr` no puede apuntar a un arreglo. Tampoco se puede utilizar con los contenedores de la STL.
- El TR1 incluye a `shared_ptr` y a `weak_ptr`.

Sección 24.8.1 Uso de `shared_ptr` y conteo de referencias

- Los apuntadores `shared_ptr` contienen un apuntador interno a un recurso (por ejemplo, un objeto asignado en forma dinámica) que puede compartirse con otros objetos en el programa.
- Los cambios al recurso de un apuntador `shared_ptr` serán “vistos” por los otros apuntadores `shared_ptr` a ese recurso.
- Los apuntadores `shared_ptr` utilizan el conteo de referencias para determinar cuántos apuntadores `shared_ptr` apuntan al recurso. Cuando la cuenta de referencias llega a cero, el apuntador interno se elimina.
- Los apuntadores `shared_ptr` pueden copiarse con seguridad y se pueden utilizar en los contenedores de la STL.
- Podemos crear un apuntador `shared_ptr` con una función eliminadora personalizada, que especifique cómo destruir el recurso. Una función eliminadora personalizada debe recibir un argumento del tipo del apuntador interno.
- Para usar apuntadores `shared_ptr`, debemos incluir el archivo de encabezado `"boost/shared_ptr.hpp"`.
- El constructor de `shared_ptr` recibe un apuntador a un objeto. El constructor también puede recibir otro apuntador `shared_ptr`, en cuyo caso comparte la propiedad del recurso con el otro `shared_ptr` y la cuenta de referencias se incrementa en uno.
- El primer apuntador `shared_ptr` a un recurso debe crearse siempre mediante el operador `new`.
- Los apuntadores `shared_ptr` proporcionan los operadores de apuntadores `*` y `->`.
- La función miembro `use_count` de `shared_ptr` devuelve el número de apuntadores `shared_ptr` al recurso.
- La función `reset` libera el recurso actual y establece el apuntador `shared_ptr` a `NULL`. También podemos pasar un apuntador o `shared_ptr` a la función `reset`; el apuntador `shared_ptr` administrará el nuevo recurso.

Sección 24.8.2 `weak_ptr`: observador de `shared_ptr`

- Un apuntador `weak_ptr` se utiliza para apuntar al recurso administrado por un `shared_ptr` sin asumir responsabilidad por él; la cuenta de referencias para el apuntador `shared_ptr` no se incrementa.

- Cuando se destruye el último `shared_ptr`, el recurso se elimina y cualquier apuntador `weak_ptr` restante se establece en `NULL`.
- Un apuntador `weak_ptr` no puede acceder al recurso al que apunta; debemos crear un apuntador `shared_ptr` desde el `weak_ptr` para acceder al recurso. Podemos pasar el apuntador `weak_ptr` al constructor de `shared_ptr`. También podemos llamar a la función `lock` de `weak_ptr`, la cual devuelve un apuntador `shared_ptr` al recurso del `weak_ptr`.
- Para usar apuntadores `weak_ptr`, debemos incluir el archivo de encabezado "boost/weak_ptr.hpp".

Sección 24.9 Reporte técnico 1

- El Reporte técnico 1 (TR1) describe las adiciones a la Biblioteca estándar de C++.
- El Reporte técnico 2 (TR2) contiene proposiciones adicionales a la biblioteca que no se incluyeron en el TR1.
- La biblioteca de Contenedores asociativos desordenados define cuatro nuevos contenedores: `unordered_set`, `unordered_map`, `unordered_multiset` y `unordered_multimap`. Estos contenedores asociativos se implementan como tablas de hash y se definen en `<unordered_set>` y `<unordered_map>`.
- `unordered_set` y `unordered_multiset` utilizan el elemento como la clave. `unordered_map` y `unordered_multimap` almacenan pares clave-valor.
- `unordered_set` y `unordered_map` requieren claves únicas, `unordered_multiset` y `unordered_multimap` no implementan esa restricción.
- El TR1 incluye funciones trigonométricas, hiperbólicas, exponenciales, logarítmicas, de potencia y especiales del C99.
- El TR1 incluye encabezados del C99 para mejorar la compatibilidad entre C++ y C99.

Sección 24.10 C++0x

- El nuevo estándar, conocido como C++0x, incluye las bibliotecas del TR1 y los cambios al lenguaje básico.
- La Organización internacional para la estandarización (ISO) supervisa la creación de estándares de lenguajes de programación internacionales. El Grupo de trabajo 21 del ISO mantiene el estándar de C++.
- Los principales objetivos para el nuevo estándar son facilitar el aprendizaje de C++, mejorar las herramientas para generar bibliotecas y aumentar la compatibilidad con el lenguaje de programación C.

Sección 24.11 Cambios al lenguaje básico

- El compilador GNU C++ tiene un modo opcional de C++0x, el cual nos permite experimentar con varios de los cambios al lenguaje básico (gcc.gnu.org/gcc-4.3/cxx0x_status.html).
- El tipo de referencia `rvalue` en C++0x nos permite enlazar un `rvalue` (objeto temporal) con una referencia no `const`.
- Una referencia `rvalue` se declara como `T&&` (en donde `T` es el tipo del objeto al que se hace referencia).
- Una referencia `rvalue` se puede utilizar para implementar la semántica de movimiento.
- La declaración `static_assert` nos permite evaluar ciertos aspectos del programa en tiempo de compilación.
- Una declaración `static_assert` recibe una expresión integral constante y un objeto `string`. Si la expresión se evalúa como 0 (false), el compilador reporta el error usando el objeto `string` que se proporciona en la declaración.
- La palabra clave `extern` indica que una variable, función o plantilla se define más adelante en el archivo actual, o en un archivo separado.
- El nuevo tipo `long long` (que ya se encuentra en el C99) es un tipo de entero que es más grande que el tipo `long`.
- Un entero extendido es un tipo de entero definido por una implementación, pero no es requerido por el estándar de C++. Cualquier tipo de entero extendido con signo debe tener su correspondiente tipo sin signo del mismo tamaño. Si se utiliza un tipo de entero extendido de manera explícita, se ve afectada la portabilidad de los programas.
- Un constructor puede llamar a otro de los constructores de la clase directamente.
- El compilador de C++ reconocerá cuando `>>` forme parte de una plantilla.
- La palabra clave `auto` determina de manera automática el tipo de una variable, con base en su expresión inicializadora. `auto` ocupa el lugar del nombre del tipo completo.
- Una plantilla variadic acepta cualquier número de argumentos. Los parámetros de la plantilla se colocan en un paquete de parámetros de tipo de plantilla. Los argumentos que se pasan a una función se colocan en un paquete de parámetros de función. El n -ésimo elemento de un paquete de parámetros de tipo de plantilla es el tipo del n -ésimo elemento en un paquete de parámetros de función.
- Tres puntos suspensivos a la izquierda de un parámetro de plantilla o de función lo declaran como un paquete de parámetros de tipo de plantilla o de función. Tres puntos suspensivos a la derecha de un paquete de parámetro de función extraen los elementos.
- `char16_t` y `char32_t` representan a los caracteres Unicode de 16 bits y 32 bits. Los prefijos `u` y `U` se pueden utilizar para denotar un carácter `char16_t` o `char32_t`, respectivamente.
- C++0x permite llamar a `sizeof` en un miembro de datos sin acceder a éste a través de una instancia de su clase. `sizeof` también se puede utilizar de una forma más natural con los datos miembro `static`.
- La formulación en el estándar C++0x define los requerimientos de secuenciamiento con más precisión. Esta cuestión es importante para C++, ya que cualquier intento futuro de agregar concurrencia requiere una sólida comprensión del orden de los eventos en un programa.

Terminología

*, cuantificador (0 o más)	char16_t
+, cuantificador (1 o más)	char32_t
?, cuantificador (0 o 1)	datos referenciales circulares
\$, final de una cadena	delegación de constructores
^, inicio de una cadena	delete, función
\d, clase de carácter (cualquier dígito decimal)	dígito
\w, clase de carácter (cualquier carácter de palabra)	distribución de números aleatorios
\s, clase de carácter (cualquier carácter de espacio en blanco)	distribución uniforme
\D, clase de carácter (cualquier carácter que no sea un dígito)	entero extendido
\W, clase de carácter (cualquier carácter que no sea de palabra)	expresión regular
\S, clase de carácter (cualquier carácter que no sea de espacio en blanco)	extern template
{n}, cuantificador (por lo menos n)	format_first_only
{n,m}, cuantificador (entre n y m)	friend, declaración
{n}, cuantificador (exactamente n)	fuga de memoria
<math.h>	función binaria
>, signo	función de hash
Algoritmo	función eliminadora personalizada
alias de plantilla	función unaria
apuntador inteligente	function, clase
asignación dinámica de memoria	generador de números aleatorios
auto, palabra clave	generador de números seudoaleatorios
auto_ptr	Licencia de software Boost
bad_weak_ptr, excepción	lock, función miembro de la clase weak_ptr
basic_regex, clase	long long, tipo de entero
Bibliotecas Boost de C++	match_not_dot_newline
bind1st, función	match_results, clase
bind2nd, función	mem_fun, función
bjam.exe	mem_fun_ref, función
boost, espacio de nombres	Microsoft Platform SDK
Boost Sandbox Vault	números aleatorios no determinísticos
Boost.Array, biblioteca	objeto de función
Boost.Bind, biblioteca	Organización internacional de estándares
Boost.Function, biblioteca	pair
Boost.Jam, sistema de generación	paquete de parámetros de función
Boost.mem_fn, biblioteca	paquete de parámetros de tipo de plantilla
Boost.Random, biblioteca	plantilla variadic
Boost.Ref, biblioteca	rand, función
Boost.Regex, biblioteca	reference_wrapper, plantilla de clase
Boost.Smart_ptr, biblioteca	regex, definición typedef
Boost.Tuple, biblioteca	regex_match, algoritmo
Boost.Type_traits, biblioteca	regex_replace, algoritmo
C++0x	regex_search, algoritmo
C99	regex_token_iterator
carácter de espacio en blanco	Reporte técnico 1 (TR1)
carácter de palabra	Reporte técnico 2 (TR2)
carácter Unicode	reset, función miembro de la clase shared_ptr
clases de caracteres	result_of, plantilla de clase
clave (tabla de hash)	rvalue, referencia
comportamiento soportado en forma condicional	shared_ptr, clase
conteo de referencias	sizeof, operador
cuantificador	smatch, definición typedef
cuantificador avaro	sourceforge.net
cuantificador perezoso	srand, función
cubeta (tabla de hash)	static_assert, declaración
cuenta de referencias	suffix, función miembro de la clase match_results
	tabla de hash
	tuple

unordered_map, contenedor
 unordered_multimap, contenedor
 unordered_multiset, contenedor

unordered_set, contenedor
 use_count, función miembro de la clase shared_ptr
 weak_ptr, clase

Ejercicios de autoevaluación

24.1 Complete los siguientes enunciados:

- El _____ describe los cambios propuestos a la Biblioteca estándar de C++.
- La biblioteca _____ ayuda a administrar la liberación de memoria asignada en forma dinámica para evitar fugas de memoria.
- Boost.Bind mejora las funciones _____ y _____ de la biblioteca estándar.
- Los apuntadores shared_ptr utilizan un(a) _____ para determinar cuándo deben eliminar el recurso.
- La clase _____ representa una expresión regular en Boost.Regex.
- La clase regex_token_iterator se encuentra en el espacio de nombres _____.
- El algoritmo _____ de Boost.Regex cambia todas las ocurrencias de un patrón en un objeto string por un objeto string especificado.
- El cuantificador _____ de expresiones regulares coincide con cero o más ocurrencias de una expresión.
- El operador _____ de expresiones regulares dentro de corchetes no coincidirá con ninguno de los caracteres en ese conjunto de corchetes.
- La palabra clave _____ en C++0x determina en forma automática el tipo de una variable al momento de inicializarla.
- La semántica de movimiento y las funciones de avance en C++0x se pueden escribir mediante el uso de _____.

24.2 Indique si cada uno de los siguientes enunciados es *verdadero* o *falso*. En caso de ser *falso*, explique por qué.

- Un apuntador auto_ptr se puede utilizar en los contenedores de la STL.
- Al crear un apuntador weak_ptr a un recurso se incrementa la cuenta de referencias.
- Una expresión regular relaciona un objeto string con un patrón.
- La expresión \d en una expresión regular denota a todas las letras.

24.3 Escriba instrucciones para realizar cada una de las siguientes tareas:

- Cree una expresión regular para relacionar una palabra de cinco letras o un número de cinco dígitos.
- Cree una expresión regular para relacionar un número telefónico en la forma de (123) 456-7890.
- Cree un apuntador shared_ptr al valor int 5, llamado intPtr.
- Cree un apuntador weak_ptr a intPtr, llamado weakIntPtr.
- Acceda al valor de la variable int mediante el uso del apuntador weakIntPtr.

Respuestas a los ejercicios de autoevaluación

24.1 a) TR1. b) Boost.Smart_ptr. c) bind1st, bind2nd. d) cuenta de referencias. e) regex o basic_regex. f) boost. g) regex_replace. h) *. i) ^. j) auto. k) referencias rvalue.

24.2 a) Falso. Un apuntador auto_ptr no se puede copiar con seguridad, y por lo tanto no se puede utilizar en contenedores de la STL. b) Falso. Un apuntador weak_ptr no asume la propiedad de su recurso y no afecta a la cuenta de referencias. c) Verdadero. d) Falso. La expresión \d en una expresión regular denota a todos los dígitos decimales.

24.3

```
a) boost::regex( "\w{5}|\d{5}");
b) boost::regex( "\(\d{3}\)\s\d{3}-\d{4}");
c) boost::shared_ptr< int > intPtr( new int( 5 ) );
d) boost::weak_ptr< int > weakIntPtr( intPtr );
e) boost::shared_ptr< int > sharedIntPtr = weakIntPtr.lock();
    *sharedIntPtr;
```

Ejercicios

24.4 (*Latín cerdo*) Escriba una aplicación que codifique frases en español a frases en latín cerdo. El latín cerdo es una forma de lenguaje codificado, que a menudo se utiliza para fines de entretenimiento. Existen muchas variaciones en los métodos utilizados para formar frases en latín cerdo. Por cuestiones de simpleza, utilice el siguiente algoritmo:

Para traducir cada palabra en español a una palabra en latín cerdo, coloque la primera letra de la palabra en español al final de la palabra, y agregue las letras “ae”. De esta forma, la palabra “salta” se convierte a “altasae”, la palabra “el” se convierte

en “leae” y la palabra “computadora” se convierte en “omputadorace”. Los espacios en blanco entre las palabras permanecen como espacios en blanco. Suponga que la frase en español consiste en palabras separadas por espacios en blanco, que no hay signos de puntuación y que todas las palabras tienen dos o más letras. Permita que el usuario introduzca un enunciado. Use un iterador `regex_token_iterator` para dividir el enunciado en palabras separadas. La función `obtenerLatinCerdo` deberá traducir una sola palabra en latín cerdo.

24.5 Escriba un programa que utilice expresiones regulares para convertir la primera letra de todas las palabras en mayúscula. Haga esto para una cadena arbitraria introducida por el usuario.

24.6 Utilice una expresión regular para contar el número de dígitos, caracteres y caracteres de espacio en blanco en una cadena.

24.7 Escriba una expresión regular que busque en una cadena y obtenga una coincidencia con un número válido. Un número puede tener cualquier cantidad de dígitos, pero sólo puede tener dígitos y un punto decimal. El punto decimal es opcional, pero si aparece en el número, sólo debe haber uno y debe tener dígitos a su izquierda y a su derecha. Debe haber espacio en blanco o un carácter de inicio o fin de línea en cualquiera de los lados de un número válido. Los números negativos deben incluir un signo negativo antes de ellos.

24.8 Escriba un programa que reciba HTML como entrada y que muestre en pantalla el número de etiquetas de HTML en la cadena. El programa debe utilizar expresiones regulares para contar el número de elementos anidados en cada nivel. Por ejemplo, el HTML:

```
<p><strong>hola</strong></p>
```

tiene un elemento `p` (nivel de anidamiento 0; es decir, no está anidado en otra etiqueta) y un elemento `strong` (nivel de anidamiento 1). Por cuestión de simpleza, use HTML en el que ninguno de los elementos contengan elementos anidados del mismo tipo; por ejemplo, un elemento `table` no debe contener otro elemento `table`.

Esta solución requiere un concepto de expresiones regulares conocido como referencia inversa, para determinar las etiquetas inicial y final de un elemento de HTML. Para encontrar estas etiquetas, debe aparecer la misma palabra en las etiquetas inicial y final. Una referencia inversa nos permite utilizar una coincidencia anterior en la expresión en otra parte de la expresión regular. Al encerrar una porción de una expresión regular entre paréntesis, la coincidencia para esa subexpresión se almacena de manera automática. Después podemos acceder al resultado de esa expresión, utilizando la sintaxis `\dígito`, en donde `dígito` es un número en el rango de 1 a 9. Por ejemplo, la expresión regular

```
^(7*).*\$
```

coincide con una cadena completa que inicia y termina con uno o más números 7. Las cadenas "777abcd777" y "7abcdef7" coinciden con esta expresión regular. El `\1` en la expresión regular anterior es una referencia inversa, la cual indica que lo que haya coincidido con la subexpresión `(7*)` también debe aparecer al final de la cadena. Se aplica una referencia inversa a la primera subexpresión entre paréntesis con el `\1`, a la segunda con el `\2`, etcétera.

Necesitará una función recursiva para poder procesar los elementos anidados de HTML. En cada llamada recursiva, tendrá que pasar el contenido de un elemento como la cadena a procesar en esa llamada; por ejemplo, el contenido del elemento `p` en el HTML de este ejemplo sería:

```
<strong>hola</strong>
```

Utilice paréntesis para almacenar el contenido que aparezca entre las etiquetas inicial y final de una cadena que coincide con su expresión regular. Este valor se almacena en el objeto `match_results` y se puede utilizar mediante el operador `[]` en ese objeto. Al igual que con las referencias inversas, las coincidencias de la subexpresión se indizan de 1 a 9.

24.9 Escriba un programa que pida al usuario introducir un enunciado y que utilice una expresión regular para comprobar si ese enunciado contiene más de un espacio entre las palabras. De ser así, el programa deberá eliminar los espacios adicionales. Por ejemplo, "Hola Programador" deberá ser "Hola Programador".

24.10 Responda a las siguientes preguntas acerca de los apuntadores inteligentes:

- Describa brevemente las ventajas de `shared_ptr`, en comparación con `auto_ptr`.
- Describa una situación en la que se utilizaría una función eliminadora personalizada.
- Describa una situación en la que utilizaría un apuntador `weak_ptr` que no sea responsable por la administración del tiempo de vida de su recurso.



*¿Qué hay en un nombre?
A eso que llamamos rosa,
si le diéramos otro nombre
conservaría su misma
fragancia dulce.*

—William Shakespeare

*¡Oh, diamante, diamante!
¡Qué poco sabes acerca del
daño que has hecho!*

—Sir Isaac Newton

Otros temas

En este capítulo aprenderá a:

- Utilizar `const_cast` para tratar temporalmente un objeto `const` como un objeto no `const`.
- Utilizar los espacios de nombres (`namespace`).
- Utilizar las palabras clave de operadores.
- Utilizar miembros `mutable` en objetos `const`.
- Utilizar los operadores de apuntadores a miembros de clases, `.*` y `->*`.
- Utilizar la herencia múltiple.
- Comprender el papel de las clases base `virtual` en la herencia múltiple.

- 25.1 Introducción
- 25.2 Operador `const_cast`
- 25.3 Espacios de nombres
- 25.4 Palabras clave de operadores
- 25.5 Miembros de clases `mutable`
- 25.6 Apuntadores a miembros de clases (`.*` y `->*`)
- 25.7 Herencia múltiple
- 25.8 Herencia múltiple y clases base `virtual`
- 25.9 Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)

25.1 Introducción

Ahora consideraremos varias características avanzadas de C++. En primer lugar, aprenderá acerca del operador `const_cast`, el cual permite a los programadores agregar o eliminar la calificación `const` de una variable. Después, hablaremos sobre los espacios de nombres (`namespace`), que se pueden utilizar para asegurar que cada identificador en un programa tenga un nombre único, y que pueda ayudar a resolver los conflictos de nombres producidos por el uso de bibliotecas que tienen los mismos nombres de variables, funciones o clases. Después presentaremos varias palabras clave de operadores que son útiles para los programadores que tienen teclados que no soportan ciertos caracteres utilizados en los símbolos de operadores, como `!`, `&`, `^`, `~` y `|`. Continuaremos nuestra discusión con el especificador de clases de almacenamiento `mutable`, el cual permite a un programador indicar que un miembro de datos siempre debe ser modificable, aun y cuando aparezca en un objeto que el programa trate en un momento dado como objeto `const`. Posteriormente presentaremos dos operadores especiales que podemos utilizar con apuntadores a miembros de clases, para acceder a un miembro de datos o a una función miembro sin conocer su nombre de antemano. Por último, presentaremos la herencia múltiple, que permite a una clase derivada heredar los miembros de varias clases base. Como parte de esta introducción, hablaremos sobre los problemas potenciales con la herencia múltiple y cómo se puede utilizar la herencia `virtual` para resolver esos problemas.

25.2 Operador `const_cast`

C++ proporciona el operador `const_cast` para eliminar mediante una conversión la calificación `const` o `volatile`. Declaramos una variable con el calificador `volatile` cuando esperamos que sea modificada por el hardware u otros programas no conocidos por el compilador. Al declarar una variable como `volatile`, indicamos que el compilador no debe optimizar el uso de esa variable, ya que ello podría afectar a la habilidad de los otros programas para utilizarla y modificarla.

En general, es peligroso utilizar el operador `const_cast`, ya que permite que un programa modifique una variable que se declaró como `const`, y que por ende no se supone que deba modificarse. Hay casos en los que es conveniente (o incluso necesario) eliminar el calificador `const`. Por ejemplo, las bibliotecas anteriores de C y C++ podrían proporcionar funciones que tengan parámetros no `const` y que no modifiquen sus parámetros. Si el programador desea pasar datos `const` a esa función, tendría que eliminar el calificador `const` de esos datos; en caso contrario, el compilador reportaría mensajes de error.

De manera similar, podríamos pasar datos no `const` a una función que considere esos datos como si fueran constantes, y que después devuelva esos datos como una constante. En dichos casos, tal vez sea necesario eliminar el calificador `const` de los datos devueltos, como se demuestra en la figura 25.1.

En este programa la función `maximo` (líneas 11 a 14) recibe dos cadenas estilo C como parámetros `const char *` y devuelve un apuntador `const char *` que apunta a la mayor de las dos cadenas. La función `main` declara las dos cadenas estilo C como arreglos `char` no `const` (líneas 18 y 19); por ende, estos arreglos son modificables. En `main` deseamos imprimir la mayor de las dos cadenas estilo C, y después modificar esa cadena estilo C para convertirla en letras minúsculas.

Los dos parámetros de la función `maximo` son de tipo `const char *`, por lo que el tipo de valor de retorno de la función también debe declararse como `const char *`. Si el tipo de valor de retorno se especifica como sólo `char *`, el compilador genera un mensaje de error, indicando que el valor que se devuelve no puede convertirse de `const char *` a `char *`; una conversión peligrosa, ya que intenta tratar a los datos que la función cree que son `const` como si fueran datos no `const`.

```

1 // Fig. 24.1: fig24_01.cpp
2 // Demostración de const_cast.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // contiene los prototipos para las funciones strcmp y strlen
8 #include <cctype> // contiene el prototipo para la función toupper
9
10 // devuelve la mayor de dos cadenas estilo C
11 const char *maximo( const char *primera, const char *segunda )
12 {
13     return ( strcmp( primera, segunda ) >= 0 ? primera : segunda );
14 } // fin de la función maximo
15
16 int main()
17 {
18     char s1[] = "hola"; // arreglo modifiable de caracteres
19     char s2[] = "adios"; // arreglo modifiable de caracteres
20
21     // se requiere const_cast para permitir asignar el const char * devuelto
22     // por maximo a la variable char * maxPtr
23     char *maxPtr = const_cast< char * >( maximo( s1, s2 ) );
24
25     cout << "La cadena mas grande es: " << maxPtr << endl;
26
27     for ( size_t i = 0; i < strlen( maxPtr ); i++ )
28         maxPtr[ i ] = toupper( maxPtr[ i ] );
29
30     cout << "La cadena mas grande en mayusculas es: " << maxPtr << endl;
31     return 0;
32 } // fin de main

```

```

La cadena mas grande es: hola
La cadena mas grande en mayusculas es: HOLA

```

Figura 25.1 | Demostración del operador `const_cast`.

Aun y cuando la función `maximo` cree que los datos son constantes, sabemos que los arreglos originales en `main` no contienen datos constantes. Por lo tanto, `main` debe poder modificar el contenido de esos arreglos según sea necesario. Como sabemos que estos arreglos son modificables, utilizamos `const_cast` (línea 23) para quitar el calificador `const` del apuntador devuelto por `maximo`, de manera que podamos entonces modificar los datos en el arreglo que representa a la mayor de las dos cadenas estilo C. Así, podemos usar el apuntador como el nombre de un arreglo de caracteres en la instrucción `for` (líneas 27 y 28) para convertir el contenido de la cadena más grande a mayúsculas. Sin el operador `const_cast` en la línea 23, este programa no se compilará debido a que no se nos permite asignar un apuntador de tipo `const char *` a un apuntador de tipo `char *`.



Tip para prevenir errores 25.1

En general, debemos utilizar un operador `const_cast` sólo cuando se sabe de antemano que los datos originales no son constantes. En caso contrario, podrían producirse resultados inesperados.

25.3 Espacios de nombres

Un programa incluye muchos identificadores definidos en distintos alcances. Algunas veces una variable de un alcance se “traslapa” (es decir, entra en conflicto) con una variable del mismo nombre en un alcance distinto, con lo que existe la posibilidad de crear un conflicto de nombres. Dicho traslapamiento puede ocurrir en muchos niveles. El traslapamiento de identificadores ocurre con frecuencia en bibliotecas desarrolladas por terceros, que por casualidad utilizan los mismos nombres para los identificadores globales (como las funciones). Esto puede producir errores de compilación.



Buena práctica de programación 25.1

Evite los identificadores que empiezan con el carácter de guión bajo, ya que pueden producir errores de vinculación. Muchas bibliotecas de código utilizan nombres que empiezan con guiones bajos.

El estándar de C++ resuelve este problema mediante los espacios de nombres (palabra clave `namespace`). Cada espacio de nombres define un alcance en el que se colocan los identificadores y las variables. Para utilizar un **miembro de un espacio de nombres**, el nombre del miembro se debe calificar con el nombre `namespace` y el operador binario de resolución de alcance (`::`), como en

`MiEspacioDeNombres::miembro`

o debe aparecer una declaración `using` o una directiva `using` antes de utilizar el nombre en el programa. Por lo general, dichas instrucciones `using` se colocan al inicio del archivo en el que se utilizan los miembros del espacio de nombres. Por ejemplo, al colocar la siguiente directiva `using` en el inicio de un archivo de código fuente:

`using namespace MiEspacioDeNombres;`

específicamente que los miembros del espacio de nombres `MiEspacioDeNombres` se pueden utilizar en el archivo sin tener que anteponer a cada miembro el nombre `MiEspacioDeNombres` y el operador de resolución de alcance (`::`).

Una declaración `using` (por ejemplo, `using std::cout;`) lleva un nombre al alcance en el que aparece la declaración. Una directiva `using` (por ejemplo, `using namespace std;`) lleva todos los nombres del espacio de nombres especificado al alcance en el que aparece la directiva.



Observación de Ingeniería de Software 25.1

Idealmente, en los programas cada entidad se debe declarar en una clase, función, bloque o espacio de nombres. Esto ayuda a aclarar el papel de cada entidad.



Tip para prevenir errores 25.2

Hay que anteponer el nombre del namespace y el operador de resolución de alcance (`::`) a un miembro, si existe la posibilidad de un conflicto de nombres.

No se garantiza que todos los espacios de nombres serán únicos. Dos distribuidores independientes podrían utilizar en forma inadvertida los mismos identificadores para los nombres de sus espacios de nombres. En la figura 25.2 se demuestra el uso de los espacios de nombres.

```

1 // Fig. 25.2: fig24_02.cpp
2 // Demostración de los espacios de nombres.
3 #include <iostream>
4 using namespace std; // usa el espacio de nombres std
5
6 int entero1 = 98; // variable global
7
8 // crea el espacio de nombres Ejemplo
9 namespace Ejemplo
10 {
11     // declara dos constantes y una variable
12     const double PI = 3.14159;
13     const double E = 2.71828;
14     int entero1 = 8;
15
16     void imprimirValores(); // prototipo
17
18     // espacio de nombres anidado
19     namespace Interno
20     {
21         // define una enumeración
22         enum Anios { FISCAL1 = 1990, FISCAL2, FISCAL3 };

```

Figura 25.2 | Demostración del uso de namespace. (Parte I de 2).

```

23 } // fin del espacio de nombres Interno
24 } // fin del espacio de nombres Ejemplo
25
26 // crea espacio de nombres sin nombre
27 namespace
28 {
29     double doubleEnSinNombre = 88.22; // declara una variable variable
30 } // fin del espacio de nombres sin nombre
31
32 int main()
33 {
34     // imprime el valor doubleEnSinNombre del espacio de nombres sin nombre
35     cout << "doubleEnSinNombre = " << doubleEnSinNombre;
36
37     // imprime la variable global
38     cout << "\n(global) entero1 = " << entero1;
39
40     // imprime los valores del espacio de nombres Ejemplo
41     cout << "\nPI = " << Ejemplo::PI << "\nE = " << Ejemplo::E
42         << "\nentero1 = " << Ejemplo::entero1 << "\nFISCAL3 = "
43         << Ejemplo::Interno::FISCAL3 << endl;
44
45     Ejemplo::imprimirValores(); // invoca a la función imprimirValores
46     return 0;
47 } // fin de main
48
49 // muestra los valores de las variables y constantes
50 void Ejemplo::imprimirValores()
51 {
52     cout << "\nEn imprimirValores:\nentero1 = " << entero1 << "\nPI = "
53         << PI << "\nE = " << E << "\ndoubleEnSinNombre = "
54         << doubleEnSinNombre << "\n(global) entero1 = " << ::entero1
55         << "\nFISCAL3 = " << Interno::FISCAL3 << endl;
56 } // fin de imprimirValores

```

```

doubleEnSinNombre = 88.22
(global) entero1 = 98
PI = 3.14159
E = 2.71828
entero1 = 8
FISCAL3 = 1992

En imprimirValores:
entero1 = 8
PI = 3.14159
E = 2.71828
doubleEnSinNombre = 88.22
(global) entero1 = 98
FISCAL3 = 1992

```

Figura 25.2 | Demostración del uso de `namespace`. (Parte 2 de 2).

Uso del espacio de nombres `std`

En la línea 4 se informa al compilador que se está usando el espacio de nombres `std`. Todo el contenido del archivo de encabezado `<iostream>` se define como parte del espacio de nombres `std`. [Nota: la mayoría de los programadores de C++ consideran una mala práctica escribir una directiva `using` como la línea 4, debido a que se incluye todo el contenido del espacio de nombres, con lo cual se incrementa la probabilidad de un conflicto de nombres].

La directiva `namespace` especifica que se utilizarán los miembros de un espacio de nombres con frecuencia en un programa. Esto nos permite acceder a todos los miembros del espacio de nombres y escribir instrucciones más concisas, tales como

```
cout << "double1 = " << double1;
```

en vez de

```
std::cout << "double1 = " << double1;
```

Si la línea 4, cada cout y endl en la figura 25.2 tendría que calificarse con std::, o deben incluirse declaraciones using individuales para cada cout y endl, como en:

```
using std::cout;
using std::endl;
```

La directiva using namespace se puede utilizar para los espacios de nombres predefinidos (por ejemplo, std) o los espacios de nombres definidos por el programador.

Definición de espacios de nombres

En las líneas 9 a 24 se utiliza la palabra clave namespace para definir el espacio de nombres Ejemplo. El cuerpo de un espacio de nombres está delimitado por llaves ({}). Los miembros del espacio de nombres Ejemplo consisten en dos constantes (PI y E en las líneas 12 y 13), un int (entero1 en la línea 14), una función (imprimirValores en la línea 16) y un espacio de nombres anidado (Interno en las líneas 19 a 23). Observe que el miembro entero1 tiene el mismo nombre que la variable global entero1 (línea 6). Las variables que tienen el mismo nombre deben tener distintos alcances; en caso contrario, se producen errores de compilación. Un espacio de nombres puede contener constantes, datos, clases, espacios de nombres anidados, funciones, etcétera. Las definiciones de los espacios de nombres deben ocupar el alcance global, o anidarse con otros espacios de nombres.

En las líneas 27 a 30 se crea un espacio de nombres sin nombre que contiene el miembro doubleEnSinNombre. El espacio de nombres sin nombre tiene una directiva using implícita, por lo que sus miembros parecen ocupar el espacio de nombres global, se pueden utilizar directamente y no tienen que calificarse con el nombre de un espacio de nombres. Las variables globales también forman parte del espacio de nombres global, y son accesibles en todos los alcances después de la declaración en el archivo.



Observación de Ingeniería de Software 25.2

Cada unidad de compilación separada tiene su propio espacio de nombres sin nombre único; es decir, el espacio de nombres sin nombre reemplaza al especificador de vinculación static.

Cómo acceder a los miembros de un espacio de nombres con nombres calificados

En la línea 35 se imprime el valor de la variable doubleEnSinNombre, que se puede utilizar directamente como parte del espacio de nombres sin nombre. En la línea 38 se imprime el valor de la variable global entero1. Para ambas variables, el compilador primero intenta localizar una declaración local de las variables en main. Como no hay declaraciones locales, el compilador asume que esas variables están en el espacio de nombres global.

En las líneas 41 a 43 se imprimen los valores de PI, e, entero1 y FISCAL3 del espacio de nombres Ejemplo. Observe que cada uno debe calificarse con Ejemplo:: debido a que el programa no proporciona una directiva using ni declaraciones que indiquen que utilizará a los miembros del espacio de nombres Ejemplo. Además, el miembro entero1 debe calificarse, ya que una variable global tiene el mismo nombre. En caso contrario, se imprime el valor de la variable global. Observe que FISCAL3 es un miembro del espacio de nombres anidado Interno, por lo que debe calificarse con Ejemplo::Interno::.

La función imprimirValores (definida en las líneas 50 a 56) es miembro de Ejemplo, por lo que puede acceder a otros miembros del espacio de nombres Ejemplo directamente, sin utilizar un calificador de espacio de nombres. La instrucción de salida en las líneas 52 a 55 imprime entero1, PI, E, doubleEnSinNombre, la variable global entero1 y FISCAL3. Observe que PI y E no se califican con Ejemplo. La variable doubleEnSinNombre aún sigue accesible, ya que se encuentra en el espacio de nombres sin nombre, y el nombre de la variable no entra en conflicto con ningún otro miembro del espacio de nombres Ejemplo. La versión global de entero1 debe calificarse con el operador unario de resolución de alcance (::), debido a que su nombre entra en conflicto con un miembro del espacio de nombres Ejemplo. Además, FISCAL3 debe calificarse con Interno::. Al acceder a los miembros de un espacio de nombres anidado, los miembros deben calificarse con el nombre del espacio de nombres (a menos que el miembro se esté utilizando dentro del espacio de nombres sin nombre).



Error común de programación 25.1

Colocar a main en un espacio de nombres es un error de compilación.

Alias para los nombres de espacios de nombres

Los espacios de nombres pueden tener alias. Por ejemplo, la instrucción

```
namespace CPPHTTP6E = CplusPlusHowToProgram6E;
```

crea el alias CPPHTTP6E para CPlusPlusHowToProgram6E.

25.4 Palabras clave de operadores

El estándar de C++ proporciona **palabras clave de operadores** (figura 25.3) que pueden utilizarse en lugar de varios operadores de C++. Las palabras clave de operadores son útiles para los programadores que tienen teclados que no soportan ciertos caracteres tales como !, &, ^, ~, |, etcétera.

En la figura 25.4 se demuestran las palabras clave de los operadores. Este programa se compiló con Microsoft Visual C++ 2005, el cual requiere el encabezado `<iostream.h>` (línea 8) para utilizar las palabras clave de los operadores. En GNU C++, la línea 8 se debe eliminar y el programa debe compilarse de la siguiente manera:

```
g++ -foperator-names Fig24_04.cpp -o Fig24_04
```

La opción del compilador `-foperator-names` indica que el compilador debe habilitar el uso de las palabras clave de los operadores en la figura 25.3. Tal vez otros compiladores no requieran que se incluya un archivo de encabezado, o que se utilice una opción del compilador para habilitar el soporte para estas palabras clave. Por ejemplo, el compilador Borland C++ 5.6.4 permite de manera implícita estas palabras clave.

Operador	Palabra clave del operador	Descripción
<i>Palabras clave de operadores lógicos</i>		
<code>&&</code>	<code>and</code>	AND lógico
<code> </code>	<code>or</code>	OR lógico
<code>!</code>	<code>not</code>	NOT lógico
<i>Palabra clave del operador de desigualdad</i>		
<code>!=</code>	<code>not_eq</code>	Desigualdad
<i>Palabras clave de operadores a nivel de bits</i>		
<code>&</code>	<code>bitand</code>	AND a nivel de bits
<code> </code>	<code>bitor</code>	OR inclusivo a nivel de bits
<code>^</code>	<code>xor</code>	OR exclusivo a nivel de bits
<code>~</code>	<code>compl</code>	complemento a nivel de bits
<i>Palabras clave de operadores de asignación a nivel de bits</i>		
<code>&=</code>	<code>and_eq</code>	AND de asignación a nivel de bits
<code> =</code>	<code>or_eq</code>	OR de asignación inclusivo a nivel de bits
<code>^=</code>	<code>xor_eq</code>	OR de asignación exclusivo a nivel de bits

Figura 25.3 | Alternativas de palabras clave de operadores para los símbolos de los mismos.

```

1 // Fig. 25.4: fig24_04.cpp
2 // Demostración de las palabras clave de los operadores.
3 #include <iostream>
4 using std::boolalpha;
5 using std::cout;
6 using std::endl;
```

Figura 25.4 | Demostración de las palabras clave de operadores. (Parte 1 de 2).

```

7
8 #include <iostream> // habilita las palabras clave de los operadores en Microsoft Visual C++
9
10 int main()
11 {
12     bool a = true;
13     bool b = false;
14     int c = 2;
15     int d = 3;
16
17     // opción pegajosa que hace que los valores bool se muestren como true o false
18     cout << boolealpha;
19
20     cout << "a = " << a << "; b = " << b
21         << "; c = " << c << "; d = " << d;
22
23     cout << "\n\nPalabras clave de operadores lógicos:";
24     cout << "\n    a and a: " << ( a and a );
25     cout << "\n    a and b: " << ( a and b );
26     cout << "\n    a or a: " << ( a or a );
27     cout << "\n    a or b: " << ( a or b );
28     cout << "\n    not a: " << ( not a );
29     cout << "\n    not b: " << ( not b );
30     cout << "\n    a not_eq b: " << ( a not_eq b );
31
32     cout << "\n\nPalabras clave de operadores a nivel de bits:";
33     cout << "\n    c bitand d: " << ( c bitand d );
34     cout << "\n    c bit_or d: " << ( c bitor d );
35     cout << "\n    c xor d: " << ( c xor d );
36     cout << "\n    compl c: " << ( compl c );
37     cout << "\n    c and_eq d: " << ( c and_eq d );
38     cout << "\n    c or_eq d: " << ( c or_eq d );
39     cout << "\n    c xor_eq d: " << ( c xor_eq d ) << endl;
40
41 } // fin de main

```

a = true; b = false; c = 2; d = 3

Palabras clave de operadores lógicos:

```

a and a: true
a and b: false
a or a: true
a or b: true
not a: false
not b: true
a not_eq b: true

```

Palabras clave de operadores a nivel de bits:

```

c bitand d: 2
c bit_or d: 3
c xor d: 1
compl c: -3
c and_eq d: 2
c or_eq d: 3
c xor_eq d: 0

```

Figura 25.4 | Demostración de las palabras clave de operadores. (Parte 2 de 2).

El programa declara e inicializa dos variables `bool` y dos variables enteras (líneas 12 a 15). Las operaciones lógicas (líneas 24 a 30) se llevan a cabo con las variables `bool` `a` y `b`, usando las diversas palabras clave de los operadores lógicos. Las operaciones a nivel de bits (líneas 33 a 39) se llevan a cabo con las variables `int` `c` y `d`, usando las diversas palabras clave de los operadores a nivel de bits. Se muestra en pantalla el resultado de cada operación.

25.5 Miembros de clases mutable

En la sección 25.2 presentamos el operador `const_cast`, el cual nos permitía eliminar el calificador `const` de un tipo. Una operación `const_cast` también se puede aplicar a un miembro de datos de un objeto `const` desde el cuerpo de una función miembro `const` de la clase de ese objeto. Esto permite a la función miembro `const` modificar el miembro de datos, aun y cuando se considera que el objeto es `const` en el cuerpo de esa función. Dicha operación podría realizarse cuando la mayoría de los miembros de datos de un objeto deben considerarse `const`, pero cierto miembro de datos específico necesita modificarse.

Como ejemplo, considere una lista enlazada que mantiene su contenido en orden. Para realizar búsquedas en la lista enlazada no se requieren modificaciones en los datos de ésta, por lo que la función de búsqueda podría ser una función miembro `const` de la clase de lista enlazada. Sin embargo, es concebible que un objeto tipo lista enlazada, en un esfuerzo por realizar búsquedas futuras de una manera más eficiente, podría llevar el registro de la ubicación de la última coincidencia exitosa. Si la siguiente operación de búsqueda intenta localizar un elemento que aparece más adelante en la lista, la búsqueda podría empezar desde la ubicación de la última coincidencia exitosa, en vez de hacerlo desde el inicio de la lista. Para ello, la función miembro `const` que realiza la búsqueda debe poder modificar el miembro de datos que lleva el registro de la última búsqueda exitosa.

Si un miembro de datos tal como el antes descrito debe ser siempre modificable, C++ proporciona el especificador de clase de almacenamiento `mutable` como alternativa para `const_cast`. Un miembro de datos `mutable` siempre es modificable, incluso en una función miembro `const` o en un objeto `const`. Esto reduce la necesidad de eliminar el calificador `const`.



Tip de portabilidad 25.1

El efecto de tratar de modificar un objeto que se definió como constante, sin importar que esa modificación se haya hecho posible mediante una operación `const_cast` o una conversión estilo C, varía de un compilador a otro.

`mutable` y `const_cast` se utilizan en contextos diferentes. Para un objeto `const` sin miembros de datos `mutable`, debe utilizarse el operador `const_cast` cada vez que se va a modificar un miembro. Esto reduce de manera considerable la probabilidad de que un miembro se modifique de manera accidental, debido a que el miembro no es modificable en forma permanente. Las operaciones que involucran a `const_cast` por lo general se ocultan en la implementación de una función miembro. El usuario de una clase tal vez no esté consciente de que se va a modificar un miembro.



Observación de Ingeniería de Software 25.3

Los miembros `mutable` son útiles en las clases que tienen detalles de implementación “secretos” que no contribuyen al valor lógico de un objeto.

Demostración mecánica de un dato miembro `mutable`

En la figura 25.5 se demuestra el uso de un miembro `mutable`. El programa define la clase `PruebaMutable` (líneas 8 a 22), la cual contiene un constructor, la función `getValor` y un miembro de datos `private` llamado `valor`, que se declara como `mutable`. En las líneas 16 a 19 se define la función `getValor` como una función miembro `const` que devuelve una copia de `valor`. Observe que la función incrementa el miembro de datos `mutable` llamado `valor` en la instrucción `return`. Por lo general, una función miembro `const` no puede modificar miembros de datos, a menos que el objeto en el que opera la función (es decir, el objeto al que apunta `this`) se convierta (mediante `const_cast`) a un tipo no `const`. Debido a que `valor` es `mutable`, esta función `const` puede modificar los datos.

```

1 // Fig. 25.5: fig24_05.cpp
2 // Demostración del especificador de clase de almacenamiento mutable.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // definición de la clase PruebaMutable
8 class PruebaMutable
9 {
10 public:

```

Figura 25.5 | Demostración de un miembro de datos `mutable`. (Parte I de 2).

```

11     PruebaMutable( int v = 0 )
12     {
13         valor = v;
14     } // fin del constructor de PruebaMutable
15
16     int getValor() const
17     {
18         return valor++; // incrementa el valor
19     } // fin de la función getValor
20 private:
21     mutable int valor; // miembro mutable
22 }; // fin de la clase PruebaMutable
23
24 int main()
25 {
26     const PruebaMutable prueba( 99 );
27
28     cout << "Valor inicial: " << prueba.getValor();
29     cout << "\nValor modificado: " << prueba.getValor() << endl;
30     return 0;
31 } // fin de main

```

```

Valor inicial: 99
Valor modificado: 100

```

Figura 25.5 | Demostración de un miembro de datos `mutable`. (Parte 2 de 2).

En la línea 26 se declara el objeto `PruebaMutable` llamado `prueba` y se inicializa con 99. En la línea 28 se hace una llamada a la función miembro `const` llamada `getValor`, la cual suma uno a `valor` y devuelve su contenido anterior. Observe que el compilador permite la llamada a la función miembro `getValor` en el objeto `prueba` debido a que es un objeto `const`, y `getValor` es una función miembro `const`. Sin embargo, `getValor` modifica la variable `valor`. Así, cuando en la línea 29 se invoca a `getValor` de nuevo, se muestra en pantalla el nuevo `valor` (100) para demostrar que indudablemente se modificó el miembro de datos `mutable`.

25.6 Apuntadores a miembros de clases (. * y ->*)

C++ proporciona los operadores `.*` y `->*` para acceder a los miembros de una clase por medio de apuntadores. Ésta es una herramienta que los programadores avanzados de C++ utilizan raras veces. Aquí proporcionamos sólo un ejemplo mecánico del uso de apuntadores a miembros de una clase. En la figura 25.6 se demuestran los operadores de apuntadores a miembros de una clase.

El programa declara la clase `Prueba` (líneas 8 a 17), la cual proporciona la función miembro `public` llamada `prueba` y el miembro de datos `public` llamado `valor`. En las líneas 19 a 20 se proporcionan prototipos para las funciones `flecha-Estrella` (definida en las líneas 32 a 36) y `puntoEstrella` (definida en las líneas 39 a 43), las cuales demuestran el uso de los operadores `->*` y `.*`, respectivamente. En la línea 24 se crea el objeto `prueba` y en la línea 25 se asigna 8 a su miembro de datos `valor`. En las líneas 26 y 27 se hacen llamadas a las funciones `flechaEstrella` y `puntoEstrella` con la dirección del objeto `prueba`.

```

1 // Fig. 25.6: fig24_06.cpp
2 // Demostración de los operadores .* y ->*.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // definición de la clase Prueba
8 class Prueba
9 {

```

Figura 25.6 | Demostración de los operadores `.*` y `->*`. (Parte 1 de 2).

```

10 public:
11     void prueba()
12     {
13         cout << "En la función prueba\n";
14     } // fin de la función prueba
15
16     int valor; // miembro de datos public
17 }; // fin de la clase Prueba
18
19 void flechaEstrella( Prueba * ); // prototipo
20 void puntoEstrella( Prueba * ); // prototipo
21
22 int main()
23 {
24     Prueba prueba;
25     prueba.valor = 8; // asigna el valor 8
26     flechaEstrella( &prueba ); // pasa la dirección a flechaEstrella
27     puntoEstrella( &prueba ); // pasa la dirección a puntoEstrella
28     return 0;
29 } // fin de main
30
31 // accede a la función miembro del objeto Prueba usando ->*
32 void flechaEstrella( Prueba *pruebaPtr )
33 {
34     void ( Prueba::*memPtr )() = &Prueba::prueba; // declara apuntador a la función
35     ( pruebaPtr->*memPtr )(); // invoca a la función directamente
36 } // fin de flechaEstrella
37
38 // accede a los miembros del miembro de datos del objeto Prueba usando .*
39 void puntoEstrella( Prueba *pruebaPtr2 )
40 {
41     int Prueba::*vPtr = &Prueba::valor; // declara un apuntador
42     cout << ( *pruebaPtr2 ).*vPtr << endl; // accede al valor
43 } // fin de puntoEstrella

```

En la función prueba

8

Figura 25.6 | Demostración de los operadores `.*` y `->*`. (Parte 2 de 2).

La línea 34 en la función `flechaEstrella` declara e inicializa la variable `memPtr` como un apuntador a una función miembro. En esta declaración, `Prueba::*` indica que la variable `memPtr` es un apuntador a un miembro de la clase `Prueba`. Para declarar un apuntador a una función hay que anteponer un `*` al nombre del apuntador y encerrarlo entre paréntesis, como en `(Prueba::*memPtr)`. Un apuntador a una función debe especificar (como parte de su tipo) tanto el tipo de valor de retorno de la función a la que apunta, como la lista de parámetros de esa función. El tipo de valor de retorno de la función aparece a la izquierda del paréntesis izquierdo, y la lista de parámetros aparece en un conjunto separado de paréntesis a la derecha de la declaración del apuntador. En este caso, la función tiene un tipo de valor de retorno `void` y no tiene parámetros. El apuntador `memPtr` se inicializa con la dirección de la función miembro `prueba` de la clase `Prueba`. Observe que el encabezado de la función debe coincidir con la declaración del apuntador a la función; es decir, la función `prueba` debe tener un tipo de valor de retorno `void` y no debe tener parámetros. Observe que el lado derecho de la asignación utiliza el operador dirección (`&`) para obtener la dirección de la función miembro `prueba`. Observe además que ni el lado izquierdo ni el lado derecho de la asignación en la línea 34 hace referencia a un objeto específico de la clase `Prueba`. Sólo se utiliza el nombre de la clase con el operador binario de resolución de alcance (`::`). En la línea 35 se invoca a la función miembro almacenada en `memPtr` (es decir, `prueba`) utilizando el operador `->*`. Como `memPtr` es un apuntador a un miembro de una clase, se debe utilizar el operador `->*` en vez del operador `->` para invocar a la función.

En la línea 41 se declara e inicializa `vPtr` como un apuntador a un miembro de datos `int` de la clase `Prueba`. El lado derecho de la asignación especifica la dirección del miembro de datos `valor`. En la línea 42 se desreferencia el apuntador `pruebaPtr2` y después se utiliza el operador `.*` para acceder al miembro al que apunta `vPtr`. Observe que el código cliente

puede crear apuntadores a miembros de la clase, sólo para aquellos miembros que sean accesibles para el código cliente. En este ejemplo, tanto la función miembro `prueba` como el miembro de datos `valor` son accesibles públicamente.



Error común de programación 25.2

Declarar un apuntador a una función miembro sin encerrar el nombre del apuntador entre paréntesis es un error de sintaxis.



Error común de programación 25.3

Declarar un apuntador a una función miembro sin anteponer al nombre del apuntador el nombre de una clase, seguido del operador de resolución de alcance (:), es un error de sintaxis.



Error común de programación 25.4

*Al tratar de utilizar el operador -> o * con un apuntador a un miembro de la clase, se generan errores de sintaxis.*

25.7 Herencia múltiple

En los capítulos 9 y 10 hablamos sobre la herencia múltiple, en la que cada clase se deriva exactamente de una clase base. En C++, una clase se puede derivar de más de una clase base: una técnica conocida como **herencia múltiple**, en la que una clase derivada hereda los miembros de dos o más clases base. Esta poderosa herramienta fomenta varios métodos interesantes de reutilización de software, pero puede producir una variedad de problemas de ambigüedad. La herencia múltiple es un concepto difícil que sólo deben utilizar los programadores experimentados. De hecho, algunos de los problemas asociados con la herencia múltiple son tan sutiles que los lenguajes de programación más recientes, como Java y C#, no permiten que una clase se derive de más de una clase base.



Buena práctica de programación 25.2

La herencia múltiple es una herramienta poderosa cuando se utiliza en forma apropiada. La herencia múltiple debe utilizarse cuando existe una relación del tipo “es un” entre un nuevo tipo y dos o más tipos existentes (es decir, el tipo A es un tipo B, y el tipo A es un tipo C).



Observación de Ingeniería de Software 25.4

La herencia múltiple puede introducir complejidad en un sistema. Se requiere un extremo cuidado en el diseño de un sistema para poder utilizar la herencia múltiple en forma apropiada; no debe utilizarse cuando la herencia simple y/o la composición pueden encargarse de la tarea.

Un problema común con la herencia múltiple es que cada una de las clases base podrían contener miembros de datos o funciones miembro con el mismo nombre. Esto puede producir problemas de ambigüedad a la hora de compilar. Considere el ejemplo de herencia múltiple (figuras 25.7, 25.8, 25.9, 25.10, 25.11). La clase `Base1` (figura 25.7) contiene un miembro de datos `protected int` llamado `valor` (línea 20), un constructor (líneas 10 a 13) que establece `valor` y la función miembro `public` llamada `getDatos` (líneas 15 a 18) que devuelve `valor`.

La clase `Base2` (figura 25.8) es similar a la clase `Base1`, excepto que sus datos `protected` son una variable `char` llamada `letra` (línea 20). Al igual que la clase `Base1`, `Base2` tiene una función miembro `public` llamada `getDatos`, pero esta función devuelve el valor del miembro de datos `char` llamado `letra`.

La clase `Derivada` (figuras 25.9 a 25.10) hereda de la clase `Base1` y de la clase `Base2` a través de la herencia múltiple. La clase `Derivada` tiene un miembro de datos `private` de tipo `double` llamado `real` (línea 21), un constructor para inicializar todos los datos de la clase `Derivada` y una función miembro `public` llamada `getReal` que devuelve el valor de la variable `double` llamada `real`.

Para indicar la herencia múltiple colocamos una lista separada por comas de clases base después del punto y coma (:) que sigue de `class Derivada` (línea 14). En la figura 25.10 podemos observar que el constructor `Derivada` llama en forma explícita a los constructores de cada una de sus clases base (`Base1` y `Base2`) utilizando la sintaxis de inicializador de miembros (línea 9). Los constructores de las clases base se llaman en el orden en el que se especifica la herencia, no en el orden en el que se mencionan sus constructores; además, si los constructores de la clase base no se llaman en forma explícita en la lista inicializadora de miembros, se harán llamadas implícitas a sus constructores predeterminados.

```

1 // Fig. 25.7: Base1.h
2 // Definición de la clase Base1
3 #ifndef BASE1_H
4 #define BASE1_H
5
6 // definición de la clase Base1
7 class Base1
8 {
9 public:
10     Base1( int valorParametro )
11     {
12         valor = valorParametro;
13     } // fin del constructor de Base1
14
15     int getDatos() const
16     {
17         return valor;
18     } // fin de la función getDatos
19 protected: // accesible para las clases derivadas
20     int valor; // heredado por la clase derivada
21 }; // fin de la clase Base1
22
23 #endif // BASE1_H

```

Figura 25.7 | Demostración de la herencia múltiple: Base1.h.

```

1 // Fig. 25.8: Base2.h
2 // Definición de la clase Base2
3 #ifndef BASE2_H
4 #define BASE2_H
5
6 // definición de la clase Base2
7 class Base2
8 {
9 public:
10     Base2( char datosCaracter )
11     {
12         letra = datosCaracter;
13     } // fin del constructor de Base2
14
15     char getDatos() const
16     {
17         return letra;
18     } // fin de la función getDatos
19 protected: // accesible para las clases derivadas
20     char letra; // heredado por la clase derivada
21 }; // fin de la clase Base2
22
23 #endif // BASE2_H

```

Figura 25.8 | Demostración de la herencia múltiple: Base2.h.

```

1 // Fig. 25.9: Derivada.h
2 // Definición de la clase Derivada que hereda
3 // varias clases base (Base1 y Base2).
4 #ifndef DERIVADA_H
5 #define DERIVADA_H
6
7 #include <iostream>

```

Figura 25.9 | Demostración de la herencia múltiple: Derivada.h. (Parte I de 2).

```

8  using std::ostream;
9
10 #include "Base1.h"
11 #include "Base2.h"
12
13 // definición de la clase Derivada
14 class Derivada : public Base1, public Base2
15 {
16     friend ostream &operator<<( ostream &, const Derivada & );
17 public:
18     Derivada( int, char, double );
19     double getReal() const;
20 private:
21     double real; // datos privados de la clase derivada
22 }; // fin de la clase Derivada
23
24 #endif // DERIVADA_H

```

Figura 25.9 | Demostración de la herencia múltiple: `Derivada.h`. (Parte 2 de 2).

El operador de inserción de flujo sobrecargado (figura 25.10, líneas 18 a 23) utiliza su segundo parámetro (una referencia a un objeto `Derivada`) para mostrar los datos del objeto `Derivada`. Esta función operador es amiga (`friend`) de `Derivada`, por lo que `operator<<` puede acceder directamente a todos los miembros `protected` y `private` de la clase `Derivada`, incluyendo el miembro de datos `protected` llamado `valor` (heredado de la clase `Base1`), el miembro de datos `protected` llamado `letra` (heredado de la clase `Base2`) y el miembro de datos `private` llamado `real` (declarado en la clase `Derivada`).

```

1 // Fig. 25.10: Derivada.cpp
2 // Definiciones de las funciones miembro para la clase Derivada
3 #include "derivada.h"
4
5 // el constructor para Derivada llama a los constructores
6 // para las clases Base1 y Base2.
7 // usa inicializadores de miembros para llamar a los constructores de las clases base
8 Derivada::Derivada( int entero, char caracter, double double1 )
9     : Base1( entero ), Base2( caracter ), real( double1 ) { }
10
11 // devuelve real
12 double Derivada::getReal() const
13 {
14     return real;
15 } // fin de la función getReal
16
17 // muestra todos los miembros de datos de Derivada
18 ostream &operator<<( ostream &salida, const Derivada &derivada )
19 {
20     salida << "    Entero: " << derivada.valor << "\n    Caracter: "
21         << derivada.letra << "\nNumero real: " << derivada.real;
22     return salida; // habilita las llamadas en cascada
23 } // fin de operator<<

```

Figura 25.10 | Demostración de la herencia múltiple: `Derivada.cpp`.

Ahora vamos a examinar la función `main` (figura 25.11) que prueba las clases de las figuras 25.7 a 25.10. En la línea 13 se crea el objeto `Base1` llamado `base1` y se inicializa con el valor `int 10`, después se crea el apuntador `base1Ptr` y se inicializa con el apuntador nulo (es decir, 0). En la línea 14 se crea el objeto `Base2` llamado `base2` y se inicializa con el valor `char 'Z'`, después se crea el apuntador `base2Ptr` y se inicializa con el apuntador nulo. En la línea 15 se crea el objeto `Derivada` llamado `derivada` y se inicializa para contener el valor `int 7`, el valor `char 'A'` y el valor `double 3.5`.

```

1 // Fig. 25.11: fig24_11.cpp
2 // Controlador para el ejemplo de herencia múltiple.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Base1.h"
8 #include "Base2.h"
9 #include "Derivada.h"
10
11 int main()
12 {
13     Base1 base1( 10 ), *base1Ptr = 0; // crea un objeto Base1
14     Base2 base2( 'Z' ), *base2Ptr = 0; // crea un objeto Base2
15     Derivada derivada( 7, 'A', 3.5 ); // crea un objeto Derivada
16
17     // imprime los miembros de datos de los objetos de las clases base
18     cout << "El objeto base1 contiene el entero " << base1.getDatos()
19         << "\nEl objeto base2 contiene el carácter " << base2.getDatos()
20         << "\nEl objeto derivada contiene:\n" << derivada << "\n\n";
21
22     // imprime los miembros de datos del objeto de la clase derivada
23     // el operador de resolución de alcance resuelve la ambigüedad de getDatos
24     cout << "Se puede acceder individualmente a los miembros de datos de la clase Derivada:"
25         << "\n    Entero: " << derivada.Base1::getDatos()
26         << "\n    Carácter: " << derivada.Base2::getDatos()
27         << "\nNúmero real: " << derivada.getReal() << "\n\n";
28     cout << "Derivada se puede tratar como un objeto de cualquiera de sus clases base:\n";
29
30     // trata a Derivada como un objeto Base1
31     base1Ptr = &derivada;
32     cout << "base1Ptr->getDatos() produce " << base1Ptr->getDatos() << '\n';
33
34     // trata a Derivada como un objeto Base2
35     base2Ptr = &derivada;
36     cout << "base2Ptr->getDatos() produce " << base2Ptr->getDatos() << endl;
37     return 0;
38 } // fin de main

```

```

El objeto base1 contiene el entero 10
El objeto base2 contiene el carácter Z
El objeto derivada contiene:
    Entero: 7
    Carácter: A
    Número real: 3.5

Se puede acceder individualmente a los miembros de datos de la clase Derivada:
    Entero: 7
    Carácter: A
    Número real: 3.5

Derivada se puede tratar como un objeto de cualquiera de sus clases base:
base1Ptr->getDatos() produce 7
base2Ptr->getDatos() produce A

```

Figura 25.11 | Demostración de la herencia múltiple.

En las líneas 18 a 20 se muestran los valores de los datos de cada objeto. Para los objetos `base1` y `base2`, invocamos a la función miembro `getDatos` de cada objeto. Aun y cuando hay dos funciones `getDatos` en este ejemplo, las llamadas no son ambiguas. En la línea 18, el compilador sabe que `base1` es un objeto de la clase `Base1`, por lo que se hace una llamada a la función `getDatos` de la clase `Base1`. En la línea 19 el compilador sabe que `base2` es un objeto de la clase `Base2`, por lo que se hace una llamada a la función `getDatos` de la clase `Base2`. En la línea 20 se muestra el contenido del objeto `derivada`, usando el operador de inserción de flujo sobrecargado.

Cómo resolver problemas de ambigüedad que surgen cuando una clase derivada hereda funciones miembro con el mismo nombre de varias clases base

En las líneas 24 a 27 se muestra el contenido del objeto derivada otra vez, usando las funciones miembro `get` de la clase `Derivada`. Sin embargo hay un problema de ambigüedad, ya que este objeto contiene dos funciones `getDatos`, una que hereda de la clase `Base1` y la otra que hereda de la clase `Base2`. El problema es fácil de resolver mediante el uso del operador binario de resolución de alcance. La expresión `derivada.Base1::getDatos()` obtiene el valor de la variable heredada de la clase `Base1` (es decir, la variable `int` llamada `valor`) y `derivada.Base2::getDatos()` obtiene el valor de la variable heredada de la clase `Base2` (es decir, la variable `char` llamada `letra`). El valor `double` en `real` se imprime sin ambigüedad con la llamada `derivada.getReal()`; no hay otras funciones miembro con ese nombre en la jerarquía.

Demostración de las relaciones “es un” en la herencia múltiple

Las relaciones “*es un*” de la herencia simple también se aplican en las relaciones de herencia múltiple. Para demostrar esto, en la línea 31 se asigna la dirección del objeto `derivada` al apuntador `Base1` llamado `base1Ptr`. Esto se permite debido a que un objeto de la clase `Derivada` *es un* objeto de la clase `Base1`. En la línea 32 se invoca a la función miembro `getDatos` de `Base1` a través de `base1Ptr` para obtener el valor de sólo la parte correspondiente a `Base1` del objeto `derivada`. En la línea 35 se asigna la dirección del objeto `derivada` al apuntador `Base2` llamado `base2Ptr`. Esto se permite debido a que un objeto de la clase `Derivada` *es un* objeto de la clase `Base2`. En la línea 36 se invoca a la función miembro `getDatos` de `Base2` a través de `base2Ptr` para obtener el valor de sólo la parte correspondiente a `Base2` del objeto `derivada`.

25.8 Herencia múltiple y clases base virtual

En la sección 25.7 hablamos sobre la herencia múltiple, el proceso por el cual una clase hereda de dos o más clases. Por ejemplo, la herencia múltiple se utiliza en la biblioteca estándar de C++ para formar la clase `basic_iostream` (figura 25.12).

La clase `basic_ios` es la clase base para `basic_istream` y `basic_ostream`, cada una de las cuales se forma con herencia simple. La clase `basic_iostream` hereda de `basic_istream` y `basic_ostream`. Esto permite a los objetos `basic_iostream` proveer la funcionalidad de los objetos `basic_istream` y `basic_ostream`. En las jerarquías de herencia múltiple, la situación descrita en la figura 25.12 se conoce como **herencia de diamante**.

Como las clases `basic_istream` y `basic_ostream` heredan de `basic_ios`, existe un problema potencial para `basic_iostream`. La clase `basic_iostream` podría contener dos copias de los miembros de la clase `basic_ios` (uno se hereda a través de la clase `basic_istream` y otro de la clase `basic_ostream`). Dicha situación sería ambigua y produciría un error de compilación, debido a que el compilador no sabría qué versión de los miembros de la clase `basic_ios` utilizar. Desde luego que `basic_iostream` no sufre realmente del problema que mencionamos. En esta sección veremos cómo al utilizar clases base `virtual` se resuelve el problema de heredar copias duplicadas de una clase base indirecta.

Errores de compilación que se producen cuando surge la ambigüedad en la herencia de diamante

En la figura 25.13 se demuestra la ambigüedad que puede ocurrir en la herencia de diamante. El programa define la clase `Base` (líneas 9 a 13), que contiene la función `virtual` pura llamada `imprimir` (línea 12). Las clases `DerivadaUno` (líneas 16 a 24) y `DerivadaDos` (líneas 27 a 35) heredan públicamente de la clase `Base` y sobrescriben a la función `imprimir`. Las clases `DerivadaUno` y `DerivadaDos` contienen lo que el estándar C++ denomina **subobjeto de la clase base**; es decir, los miembros de la clase `Base` en este ejemplo.

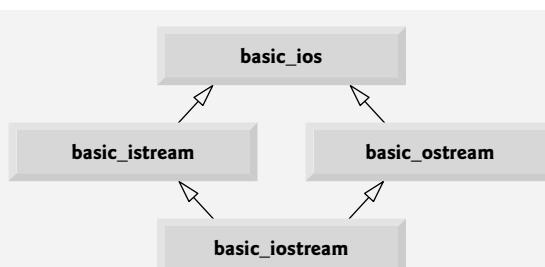


Figura 25.12 | Herencia múltiple para formar la clase `basic_iostream`.

```

1 // Fig. 25.13: fig24_13.cpp
2 // Intento de llamar mediante polimorfismo a una función que
3 // se hereda de dos clases base (herencia múltiple).
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 // definición de la clase Base
9 class Base
10 {
11 public:
12     virtual void imprimir() const = 0; // virtual pura
13 }; // fin de la clase Base
14
15 // definición de la clase DerivadaUno
16 class DerivadaUno : public Base
17 {
18 public:
19     // sobrescribe a la función imprimir
20     void imprimir() const
21     {
22         cout << "DerivadaUno\n";
23     } // fin de la función imprimir
24 }; // fin de la clase DerivadaUno
25
26 // definición de la clase DerivadaDos
27 class DerivadaDos : public Base
28 {
29 public:
30     // sobrescribe a la función imprimir
31     void imprimir() const
32     {
33         cout << "DerivadaDos\n";
34     } // fin de la función imprimir
35 }; // fin de la clase DerivadaDos
36
37 // definición de la clase Multiple
38 class Multiple : public DerivadaUno, public DerivadaDos
39 {
40 public:
41     // califica la versión correspondiente de la función imprimir
42     void imprimir() const
43     {
44         DerivadaDos::imprimir();
45     } // fin de la función imprimir
46 }; // fin de la clase Multiple
47
48 int main()
49 {
50     Multiple ambas; // crea instancia de objeto Multiple
51     DerivadaUno uno; // crea instancia de objeto DerivadaUno
52     DerivadaDos dos; // crea instancia de objeto DerivadaDos
53     Base *arreglo[ 3 ]; // crea arreglo de apunadores a la clase base
54
55     arreglo[ 0 ] = &ambas; // ERROR--ambigüedad
56     arreglo[ 1 ] = &uno;
57     arreglo[ 2 ] = &dos;
58
59     // invoca a imprimir mediante el polimorfismo
60     for ( int i = 0; i < 3; i++ )
61         arreglo[ i ] -> imprimir();

```

Figura 25.13 | Intento de llamar a una función con herencia múltiple mediante el polimorfismo. (Parte I de 2).

```

62
63     return 0;
64 } // fin de main

```

```
C:\ejemplos2\cap25\fig25_13\fig25_13.cpp(55) : error C2594: '=' : conversiones ambiguas de
'Multiple *__w64 ' a 'Base *'
```

Figura 25.13 | Intento de llamar a una función con herencia múltiple mediante el polimorfismo. (Parte 2 de 2).

La clase `Multiple` (líneas 38 a 46) hereda de las clases `DerivadaUno` y `DerivadaDos`. En la clase `Multiple`, la función `imprimir` se sobrescribe para llamar a la función `imprimir` de `DerivadaDos` (línea 44). Observe que debemos calificar la llamada a `imprimir` con el nombre de la clase `DerivadaDos` para especificar qué versión de `imprimir` se debe llamar.

La función `main` (líneas 48 a 64) declara objetos de las clases `Multiple` (línea 50), `DerivadaUno` (línea 51) y `DerivadaDos` (línea 52). En la línea 53 se declara un arreglo de apuntadores `Base *`. Cada elemento del arreglo se inicializa con la dirección de un objeto (líneas 55 a 57). Se produce un error cuando la dirección de `ambas` (un objeto de la clase `Multiple`) se asigna a `arreglo[0]`. El objeto `ambas` contiene dos subobjetos de tipo `Base`, por lo que el compilador no sabe a cuál subobjeto debe apuntar el apuntador `arreglo[0]`, y genera un error de compilación que indica una conversión ambigua.

Eliminación de subobjetos duplicados con la herencia virtual de la clase base

El problema de los subobjetos duplicados se resuelve con la herencia `virtual`. Cuando se hereda una clase base como `virtual`, sólo aparece un subobjeto en la clase derivada; a este proceso se le conoce como **herencia virtual de la clase base**. En la figura 25.14 se modifica el programa de la figura 25.13 para utilizar una clase base `virtual`.

El cambio clave en el programa es que las clases `DerivadaUno` (línea 15) y `DerivadaDos` (línea 26) heredan de la clase `Base` al especificar `virtual public Base`. Como ambas clases heredan de `Base`, cada una contiene un subobjeto `Base`. El beneficio de la herencia `virtual` no queda claro sino hasta que la clase `Multiple` hereda de `DerivadaUno` y `DerivadaDos` (línea 37). Como cada una de las clases base usó la herencia `virtual` para heredar los miembros de la clase `Base`, el compilador asegura que sólo se herede un subobjeto de tipo `Base` en la clase `Multiple`. Esto elimina el error de ambigüedad generado por el compilador en la figura 25.13. Ahora el compilador permite la conversión implícita del apuntador de la clase derivada (`&ambas`) al apuntador de la clase base `arreglo[0]` en la línea 56 de `main`. La instrucción `for` en las líneas 61 y 62 llama mediante el polimorfismo a la función `imprimir` para cada objeto.

```

1 // Fig. 25.14: fig24_14.cpp
2 // Uso de clases base virtuales.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // definición de la clase Base
8 class Base
9 {
10 public:
11     virtual void imprimir() const = 0; // virtual pura
12 }; // fin de la clase Base
13
14 // definición de la clase DerivadaUno
15 class DerivadaUno : virtual public Base
16 {
17 public:
18     // sobrescribe a la función imprimir
19     void imprimir() const
20     {

```

Figura 25.14 | Uso de clases base `virtual`. (Parte 1 de 2).

```

21     cout << "DerivadaUno\n";
22 } // fin de la función imprimir
23 }; // fin de la clase DerivadaUno
24
25 // definición de la clase DerivadaDos
26 class DerivadaDos : virtual public Base
27 {
28 public:
29     // sobrescribe a la función imprimir
30     void imprimir() const
31     {
32         cout << "DerivadaDos\n";
33     } // fin de la función imprimir
34 }; // fin de la clase DerivadaDos
35
36 // definición de la clase Multiple
37 class Multiple : public DerivadaUno, public DerivadaDos
38 {
39 public:
40     // califica la versión correspondiente de la función imprimir
41     void imprimir() const
42     {
43         DerivadaDos::imprimir();
44     } // fin de la función imprimir
45 }; // fin de la clase Multiple
46
47 int main()
48 {
49     Multiple ambas; // crea instancia de objeto Multiple
50     DerivadaUno uno; // crea instancia de objeto DerivadaUno
51     DerivadaDos dos; // crea instancia de objeto DerivadaDos
52
53     // declara arreglo de apunadores a la clase base e inicializa
54     // cada elemento con un tipo de la clase derivada
55     Base *arreglo[ 3 ];
56     arreglo[ 0 ] = &ambas;
57     arreglo[ 1 ] = &uno;
58     arreglo[ 2 ] = &dos;
59
60     // invoca mediante polimorfismo a la función imprimir
61     for ( int i = 0; i < 3; i++ )
62         arreglo[ i ]->imprimir();
63
64     return 0;
65 } // fin de main

```

DerivadaDos
DerivadaUno
DerivadaDos

Figura 25.14 | Uso de clases base `virtual`. (Parte 2 de 2).

Constructores en jerarquías de herencia múltiple con clases base `virtual`

Es más simple implementar jerarquías con clases base si se utilizan constructores predeterminados para las clases base. Los ejemplos en las figuras 25.13 y 25.14 utilizan constructores predeterminados generados por el compilador. Si una clase base `virtual` proporciona un constructor que requiere argumentos, la implementación de la clase derivada se complica más, debido a que la clase más derivada debe invocar en forma explícita al constructor de la clase base para inicializar los miembros heredados de la clase base `virtual`.



Observación de Ingeniería de Software 25.5

Al proporcionar un constructor predeterminado para las clases base `virtual` se simplifica el diseño de las jerarquías.

Información adicional acerca de la herencia múltiple

La herencia múltiple es un tema complejo que se cubre en libros más avanzados sobre C++. Para obtener más información acerca de la herencia múltiple, visite nuestro Centro de recursos de C++ en

www.deitel.com/cplusplus/

En la categoría *C++ Multiple Inheritance* encontrará vínculos a varios artículos y recursos, incluyendo un sitio de preguntas frecuentes (FAQs) sobre la herencia múltiple, y sugerencias para su uso.

25.9 Repaso

En este capítulo aprendió a utilizar el operador `const_cast` para eliminar el calificador `const` de una variable. Después le mostramos cómo utilizar los espacios de nombre (`namespace`) para asegurar que todo identificador en un programa tenga un nombre único, y le explicamos cómo pueden los espacios de nombres ayudar a resolver los conflictos de nombres. Vimos varias palabras clave de operadores para los programadores cuyos teclados no soportan ciertos caracteres que se utilizan en los símbolos de los operadores, como `!`, `&`, `^`, `~` y `|`. Después le mostramos cómo el especificador de clase de almacenamiento `mutable` permite a un programador indicar que un miembro de datos siempre debe ser modificable, aun y cuando aparezca en un objeto que se trate en un momento dado como `const`. También le mostramos la mecánica del uso de apuntadores a miembros de clases, y los operadores `->*` y `.*`. Por último, presentamos la herencia múltiple y hablamos sobre los problemas asociados con el proceso de permitir que una clase derivada herede los miembros de varias clases base. Como parte de esta discusión, demostramos cómo se puede utilizar la herencia `virtual` para resolver estos problemas.

Resumen

Sección 25.2 Operador `const_cast`

- C++ proporciona el operador `const_cast` para eliminar los calificadores `const` o `volatile`.
- Un programa declara a una variable con el calificador `volatile` cuando ese programa espera que la variable sea modificada por otros programas. Al declarar una variable como `volatile`, indicamos que el compilador no debe optimizar el uso de esa variable, ya que hacerlo podría afectar a la habilidad de los otros programas para utilizar y modificar esa variable `volatile`.
- En general, es peligroso utilizar el operador `const_cast`, debido a que permite a un programa modificar una variable que se haya declarado como `const`, y por ende no debe ser modificable.
- Hay casos en los que es conveniente, o incluso necesario, eliminar el calificador `const` de una variable. Por ejemplo, las bibliotecas de C y C++ anteriores podrían proporcionar funciones con parámetros `no const` y que no modifican sus parámetros. Si el lector desea pasar datos `const` a dicha función, tendría que eliminar el calificador `const` de los datos; en caso contrario, el compilador reportaría mensajes de error.
- Si pasamos datos `no const` a una función que trate los datos como si fuera constante, y después devuelve esos datos como constantes, tal vez sea necesario eliminar el calificador `const` de los datos devueltos para utilizar y modificar esos datos.

Sección 25.3 Espacios de nombres

- Un programa incluye muchos identificadores definidos en distintos alcances. Algunas veces una variable de un alcance se “traslapará” con una variable del mismo nombre en un alcance distinto, con lo cual es posible que se cree un conflicto de nombres. El estándar de C++ resuelve este problema con los espacios de nombres.
- Cada espacio de nombres define un alcance en el que se colocan los identificadores. Para utilizar un miembro del espacio de nombres, debe calificarse el nombre del miembro con el nombre del espacio de nombres y el operador binario de resolución de alcance (`::`), o debe aparecer una directiva o declaración `using` antes de utilizar el nombre en el programa.
- Por lo general, las instrucciones `using` se colocan al principio del archivo en el que se utilizan los miembros del espacio de nombres.
- No se garantiza que todos los espacios de nombres serán únicos. Dos distribuidores independientes podrían utilizar en forma inadvertida los mismos identificadores para los nombres de sus espacios de nombres.
- Una directiva `using namespace` especifica que los miembros de un espacio de nombres se utilizarán con frecuencia en un programa. Esto nos permite acceder a todos los miembros del espacio de nombres.
- Una directiva `using namespace` se puede utilizar para los espacios de nombres predefinidos (por ejemplo, `std`) o para los espacios de nombres definidos por el programador.
- Un espacio de nombres puede contener constantes, datos, clases, espacios de nombres anidados, funciones, etcétera. Las definiciones de los espacios de nombres deben ocupar el alcance global, o anidarse dentro de otros espacios de nombres.
- Un espacio de nombres sin nombre tiene una directiva `using` implícita, por lo que sus miembros parecen ocupar el espacio de nombres global, se pueden utilizar en forma directa y no tienen que calificarse con el nombre de un espacio de nombres. Las variables globales también forman parte del espacio de nombres global.

- Al acceder a los miembros de un espacio de nombres anidado, los miembros se deben calificar con el nombre del espacio de nombres (a menos que el miembro se vaya a utilizar dentro del espacio de nombres anidado).
- Los espacios de nombres pueden tener alias.

Sección 25.4 Palabras clave de operadores

- El estándar de C++ proporciona palabras clave de operadores que se pueden utilizar en vez de varios operadores de C++. Las palabras clave de operadores son útiles para los programadores que tienen teclados que no soportan ciertos caracteres, como !, &, ^, ~, |, etc.

Sección 25.5 Miembros de clases mutable

- Si un miembro de datos siempre debe ser modificable, C++ proporciona el especificador de clase de almacenamiento `mutable` como alternativa para `const_cast`. Un miembro de datos `mutable` siempre es modificable, aun en una función miembro `const` o en un objeto `const`. Esto reduce la necesidad de eliminar el calificador `const`.
- `mutable` y `const_cast` se utilizan en diferentes contextos. Para un objeto `const` sin miembros de datos `mutable`, debe utilizarse el operador `const_cast` cada vez que se va a modificar un miembro. Esto reduce en forma considerable la probabilidad de que un miembro se modifique en forma accidental, debido a que no es modificable en forma permanente.
- Las operaciones que involucran a `const_cast` se ocultan generalmente en la implementación de una función miembro. El usuario de una clase podría no estar consciente de que se va a modificar un miembro.

Sección 25.6 Apuntadores a miembros de clases (`.*` y `->*`)

- C++ proporciona los operadores `.*` y `->*` para acceder a los miembros de clases mediante apuntadores. Esta herramienta la utilizan principalmente los programadores avanzados de C++ en raras ocasiones.
- Para declarar un apuntador a una función, el programador tiene que encerrar el nombre del apuntador, al cual se antepone un `*`, entre paréntesis. Un apuntador a una función debe especificar, como parte de su tipo, tanto el tipo de valor de retorno de la función a la que apunta, como la lista de parámetros de esa función.

Sección 25.7 Herencia múltiple

- En C++, una clase se puede derivar de más de una clase base; a esta técnica se le conoce como herencia múltiple, en la cual una clase derivada hereda los miembros de dos o más clases base.
- Un problema común con la herencia múltiple es que cada una de las clases base podría contener miembros de datos o funciones miembro con el mismo nombre. Esto puede producir problemas de ambigüedad al tratar de compilar.
- Las relaciones “*es un*” de la herencia simple también se aplican en las relaciones de herencia múltiple.
- La herencia múltiple se utiliza, por ejemplo, en la Biblioteca estándar de C++ para formar la clase `basic_iostream`. La clase `basic_ios` es la clase base tanto para `basic_istream` como para `basic_ostream`, cada una de las cuales se forman con herencia simple. La clase `basic_iostream` hereda de `basic_istream` y `basic_ostream`. Esto permite a los objetos de la clase `basic_iostream` proporcionar la funcionalidad de los objetos `basic_istream` y `basic_ostream`. En las jerarquías de herencia múltiple, la situación aquí descrita se conoce como herencia de diamante.
- Como las clases `basic_istream` y `basic_ostream` heredan de `basic_ios`, existe un problema potencial para `basic_iostream`. Si no se implementa en forma correcta, la clase `basic_iostream` podría contener dos copias de los miembros de la clase `basic_ios`; una heredada a través de la clase `basic_istream` y la otra a través de `basic_ostream`. Dicha situación sería ambigua y produciría un error de compilación, debido a que el compilador no sabría cuál versión de los miembros de la clase `basic_ios` utilizar.

Sección 25.8 Herencia múltiple y clases base virtual

- La ambigüedad en la herencia de diamante ocurre cuando un objeto de la clase derivada hereda dos o más subobjetos de la clase base. El problema de los subobjetos duplicados se resuelve con la herencia `virtual`. Cuando una clase base se hereda como `virtual`, sólo aparece un subobjeto en la clase derivada; a este proceso se le conoce como herencia de clase base `virtual`.
- Es más simple implementar jerarquías con clases base `virtual` si se utilizan constructores predeterminados para las clases base. Si una clase base `virtual` proporciona un constructor que requiere argumentos, la implementación de las clases derivadas se vuelve más complicada, debido a que la clase más derivada debe invocar en forma explícita al constructor de la clase base `virtual` para inicializar los miembros heredados de la clase base `virtual`.

Terminología

<code>.*</code> , operador	<code>bitor</code> , palabra clave de operador
<code>->*</code> , operador	<code>clase más derivada</code>
alias de espacio de nombres	<code>comp1</code> , palabra clave del operador
<code>and</code> , palabra clave de operador	<code>conflicto de nombres</code>
<code>and_eq</code> , palabra clave de operador	<code>const_cast</code> , operador
<code>bitand</code> , palabra clave de operador	eliminar el calificador <code>const</code>

espacio de nombres	palabra clave de operador
espacio de nombres anidado	palabras clave de operadores a nivel de bits
espacio de nombres global	palabras clave de operadores de asignación a nivel de bits
espacio de nombres sin nombre	palabras clave de operadores de desigualdad
herencia de diamante	palabras clave de operadores lógicos
herencia múltiple	subobjeto de clase base
lista separada por comas de las clases base	using namespace, declaración
mutable, miembro de datos	using, declaración
namespace, palabra clave	virtual, clase base
not, palabra clave de operador	virtual, herencia
not_eq, palabra clave de operador	volatile, calificador
operadores de apuntador a miembro	xor, palabra clave de operador
or, palabra clave de operador	xor_eq, palabra clave de operador
or_eq, palabra clave de operador	

Ejercicios de autoevaluación

25.1 Complete los siguientes enunciados:

- a) El operador _____ califica a un miembro con su espacio de nombres.
- b) El operador _____ permite eliminar el calificador const de un objeto.
- c) Como un espacio de nombres sin nombre tiene una directiva using implícita, sus miembros parecen ocupar el _____, se pueden utilizar directamente y no tienen que calificarse con un nombre de espacio de nombres.
- d) El operador _____ es la palabra clave de operador de desigualdad.
- e) La _____ permite a una clase derivarse de más de una clase base.
- f) Cuando una clase base se hereda como _____, sólo aparecerá un subobjeto de la clase base en la clase derivada.

25.2 Indique cuáles de los siguientes enunciados es *verdadero* y cuál es *falso*. En caso de ser *falso*, explique por qué.

- a) Al pasar un argumento no const a una función const, se debe utilizar el operador const_cast para eliminar el calificador const de la función.
- b) Se garantiza que los espacios de nombres serán únicos.
- c) Al igual que los cuerpos de las clases, los cuerpos de los espacios de nombres también terminan en signos de punto y coma.
- d) Los espacios de nombres no pueden tener otros espacios de nombres como miembros.
- e) Un miembro de datos mutable no se puede modificar en una función miembro const.

Respuestas a los ejercicios de autoevaluación

25.1 a) binario de resolución de ámbito (::). b) const_cast. c) espacio de nombres global. d) not_eq. e) herencia múltiple. f) virtual.

25.2 a) Falso. Es válido pasar un argumento no const a una función const. Sin embargo, al pasar una referencia o apuntador const a una función no const, debemos utilizar el operador const_cast para eliminar el calificador const de la referencia o apuntador.

b) Falso. Los programadores podrían elegir en forma inadvertida el espacio de nombres que ya se encuentre en uso.

c) Falso. Los cuerpos de los espacios de nombres no terminan con signos de punto y coma.

d) Falso. Los espacios de nombres se pueden anidar.

e) Falso. Un miembro de datos mutable siempre es modificable, incluso hasta en una función miembro const.

Ejercicios

25.3 Complete los siguientes enunciados:

- a) La palabra clave _____ especifica que se va a utilizar un espacio de nombres o un miembro del espacio de nombres.
- b) El operador _____ es la palabra clave de operador para el OR lógico.
- c) El especificador de almacenamiento _____ permite modificar un miembro de un objeto const.
- d) El calificador _____ especifica que otros programas pueden modificar a un objeto.
- e) Debemos anteponer a un miembro el nombre de su _____ y el operador de resolución de alcance, si existe la posibilidad de un conflicto de alcances.

1050 Capítulo 25 Otros temas

- f) El cuerpo de un espacio de nombres se delimita mediante _____.
g) Para un objeto `const` sin miembros de datos _____, se debe utilizar el operador _____ cada vez que se va a modificar un miembro.

25.4 Escriba un espacio de nombres llamado `Moneda`, que defina los miembros constantes `UNO`, `DOS`, `CINCO`, `DIEZ`, `VEINTE`, `CINCUENTA` y `CIEN`. Escriba dos programas cortos que utilicen `Moneda`. Un programa debe tener todas las constantes disponibles y el otro sólo debe tener a `CINCO` disponible.

25.5 Dados los espacios de nombres de la figura 25.15, determine si cada una de las siguientes instrucciones es *verdadera o falsa*. Explique las respuestas que sean *falsas*.

- La variable `kilometros` puede verse dentro del espacio de nombres `Datos`.
- El objeto `string1` puede verse dentro del espacio de nombres `Datos`.
- La constante `POLONIA` no puede verse dentro del espacio de nombres `Datos`.
- La constante `ALEMANIA` puede verse dentro del espacio de nombres `Datos`.
- La función `funcion` es visible para el espacio de nombres `Datos`.
- El espacio de nombres `Datos` es visible para el espacio de nombres `InformacionPais`.
- El objeto `map` es visible para el espacio de nombres `InformacionPais`.
- El objeto `string1` puede verse dentro del espacio de nombres `InformacionRegional`.

```
1 namespace InformacionPais
2 {
3     using namespace std;
4     enum Paises { POLONIA, SUIZA, ALEMANIA,
5                  AUSTRIA, REPUBLICA_CHECA };
6     int kilometros;
7     string string1;
8
9     namespace InformacionRegional
10    {
11         short getPoblacion(); // asume que existe la definición
12         DatosMapa mapa; // asume que existe la definición
13     } // fin de InformacionRegional
14 } // fin de InformacionPais
15
16 namespace Datos
17 {
18     using namespace InformacionPais::InformacionRegional;
19     void *funcion( void *, int );
20 } // fin de Datos
```

Figura 25.15 | Espacios de nombres para el ejercicio 25.5.

25.6 Compare y contraste a `mutable` y `const_cast`. Proporcione al menos un ejemplo de cuándo es preferible usar a uno en vez del otro. [Nota: este ejercicio no requiere que se escriba código].

25.7 Escriba un programa que utilice `const_cast` para modificar una variable `const`. [Sugerencia: utilice un apuntador en su solución para apuntar al identificador `const`].

25.8 ¿Qué problema resuelven las clases base `virtual`?

25.9 Escriba un programa que utilice clases base `virtual`. La clase en la parte superior de la jerarquía debe proporcionar un constructor que reciba cuando menos un argumento (es decir, no proporcione un constructor predeterminado). ¿Qué retos presenta esto para la jerarquía de herencia?

25.10 Encuentre el (los) error(es) en cada uno de los siguientes fragmentos de código. Cuando sea posible, explique cómo se puede corregir cada error.

- namespace Nombre {
 int x;
 int y;
 mutable int z;
}
- int entero = const_cast< int >(double);
- namespace PCM(111, "hola"); // construye un espacio de nombres



Tabla de precedencia de operadores y asociatividad

A.1 Precedencia de operadores

Los operadores se muestran en orden decreciente de precedencia, de arriba hacia abajo (figura A.1).

Operador	Tipo	Asociatividad
::	binario de resolución de alcance	izquierda a derecha
::	unario de resolución de alcance	
()	paréntesis	izquierda a derecha
[]	subíndice de arreglo	
.	selección de miembro mediante un objeto	
->	selección de miembro mediante un apuntador	
++	unario de postincremento	
--	unario de postdecremento	
typeid	información de tipos en tiempo de ejecución	
dynamic_cast < tipo >	conversión con comprobación de tipos en tiempo de ejecución	
static_cast< tipo >	conversión con comprobación de tipos en tiempo de compilación	
reinterpret_cast< tipo >	conversión para conversiones no estándar	
const_cast< tipo >	eliminar el calificador const	
++	unario de preincremento	derecha a izquierda
--	unario de predecremento	
+	unario de suma	
-	unario de resta	
!	unario de negación lógica	
~	unario de complemento a nivel de bits	
sizeof	determinar el tamaño en bytes	

Figura A.1 | Tabla de precedencia de operadores y asociatividad. (Parte 1 de 2).

1052 Apéndice A Tabla de precedencia de operadores y asociatividad

Operador	Tipo	Asociatividad
&	dirección	
*	desreferencia	
new	asignación dinámica de memoria	
new[]	asignación dinámica de arreglo	
delete	desasignación dinámica de memoria	
delete[]	desasignación dinámica de arreglo	
(tipo)	unario de conversión estilo C	derecha a izquierda
.*	apuntador a miembro mediante un objeto	izquierda a derecha
->*	apuntador a miembro mediante un apuntador	
*	multiplicación	izquierda a derecha
/	división	
%	módulo	
+	suma	izquierda a derecha
-	resta	
<<	desplazamiento a la izquierda a nivel de bits	izquierda a derecha
>>	desplazamiento a la derecha a nivel de bits	
<	menor que relacional	izquierda a derecha
<=	menor o igual que relacional	
>	mayor que relacional	
>=	mayor o igual que relacional	
==	es igual a relacional	izquierda a derecha
!=	no es igual a relacional	
&	AND a nivel de bits	izquierda a derecha
^	OR exclusivo a nivel de bits	izquierda a derecha
	OR inclusivo a nivel de bits	izquierda a derecha
&&	AND lógico	izquierda a derecha
	OR lógico	izquierda a derecha
? :	ternario condicional	derecha a izquierda
=	asignación	derecha a izquierda
+=	asignación de suma	
-=	asignación de resta	
*=	asignación de multiplicación	
/=	asignación de división	
%=	asignación de módulo	
&=	AND de asignación a nivel de bits	
^=	OR de asignación exclusivo a nivel de bits	
=	OR de asignación inclusivo a nivel de bits	
<<=	desplazamiento a la izquierda de asignación a nivel de bits	
>>=	desplazamiento a la derecha de asignación a nivel de bits	
,	coma	izquierda a derecha

Figura A.1 | Tabla de precedencia de operadores y asociatividad. (Parte 2 de 2).



Conjunto de caracteres ASCII

Conjunto de caracteres ASCII											
	0	1	2	3	4	5	6	7	8	9	
0	nu1	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	sp	!	"	#	\$	%	&	'	
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			

Figura B.1 | El conjunto de caracteres ASCII.

Los dígitos a la izquierda de la tabla son los dígitos izquierdos del equivalente decimal (0-127) del código de caracteres, y los dígitos en la parte superior de la tabla son los dígitos derechos del código de caracteres. Por ejemplo, el código de carácter para la "F" es 70, mientras que para el "&" es 38.



Tipos fundamentales

En la figura C.1 se listan los tipos fundamentales de C++. El Documento del estándar de C++ no proporciona el número exacto de bytes requeridos para almacenar variables de estos tipos en memoria. Sin embargo, el Documento del estándar de C++ sí indica cómo se relacionan los requerimientos de memoria para los tipos fundamentales entre sí. Por orden creciente de requerimientos de memoria, los tipos enteros con signo son `signed char`, `short int`, `int` y `long int`. Esto significa que un `short int` debe proporcionar cuando menos tanto espacio de almacenamiento como un `signed char`; un `int` debe proporcionar cuando menos tanto espacio de almacenamiento como un `short int`; y un `long int` debe proporcionar cuando menos tanto espacio de almacenamiento como un `int`. Cada tipo entero con signo tiene su correspondiente tipo entero sin signo con los mismos requerimientos de memoria. Los tipos sin signo no pueden representar valores negativos, pero pueden representar hasta el doble de valores positivos que sus tipos con signo asociados. Por orden creciente de requerimientos de memoria, los tipos de punto flotante son `float`, `double` y `long double`. Al igual que los tipos enteros, un `double` debe proporcionar cuando menos tanto espacio de almacenamiento como un `float`, y un `long double` debe proporcionar cuando menos tanto espacio de almacenamiento como un `double`.

Los tamaños y rangos exactos de valores para los tipos fundamentales dependen de la implementación. Los archivos de encabezado `<climits>` (para los tipos integrales) y `<cfloat>` (para los tipos de punto flotante) especifican los rangos de valores soportados en el sistema del programador.

Tipos integrales	Tipos de punto flotante
<code>bool</code>	<code>float</code>
<code>char</code>	<code>double</code>
<code>signed char</code>	<code>long double</code>
<code>unsigned char</code>	
<code>short int</code>	
<code>unsigned short int</code>	
<code>int</code>	
<code>unsigned int</code>	
<code>long int</code>	
<code>unsigned long int</code>	
<code>wchar_t</code>	

Figura C.1 | Tipos fundamentales de C++.

El rango de valores que soporta un tipo depende del número de bytes que se utilizan para representar a ese tipo. Por ejemplo, considere un sistema con valores `int` de 4 bytes (32 bits). Para el tipo `int` con signo, los valores no negativos están en el rango de 0 a 2,147,483,647 ($2^{31} - 1$). Los valores negativos están en el rango de -1 a -2,147,483,648 (-2^{31}). Esto da un total de 2^{32} valores posibles. Un valor `unsigned int` en el mismo sistema utilizaría el mismo número de bits para representar datos, pero no representaría ningún valor negativo. Esto produce valores en el rango de 0 a 4,294,967,295 ($2^{32} - 1$). En el mismo sistema, un `short int` no podría utilizar más de 32 bits para representar sus datos, y un `long int` debe utilizar cuando menos 32 bits.

C++ proporciona el tipo de datos `bool` para las variables que sólo pueden contener los valores `true` y `false`.

D



He aquí sólo los números ratificados.

—William Shakespeare

La naturaleza tiene un cierto tipo de sistema de coordenadas aritméticas-geométricas, ya que cuenta con todo tipo de modelos. Lo que experimentamos de la naturaleza está en los modelos, y todos los modelos de la naturaleza son tan bellos.

Se me ocurrió que el sistema de la naturaleza debe ser una verdadera belleza, porque en la química encontramos que las asociaciones se encuentran siempre en hermosos números enteros; no hay fracciones.

—Richard Buckminster Fuller

Sistemas numéricos

OBJETIVOS

En este apéndice aprenderá a:

- Comprender los conceptos acerca de los sistemas numéricos como base, valor posicional y valor simbólico.
- Trabajar con los números representados en los sistemas numéricos binario, octal y hexadecimal.
- Abreviar los números binarios como octales o hexadecimales.
- Convertir los números octales y hexadecimales en binarios.
- Realizar conversiones hacia y desde números decimales y sus equivalentes en binario, octal y hexadecimal.
- Comprender el funcionamiento de la aritmética binaria y la manera en que se representan los números binarios negativos, utilizando la notación de complemento a dos.

- D.1** Introducción
- D.2** Abreviatura de los números binarios como números octales y hexadecimales
- D.3** Conversión de números octales y hexadecimales a binarios
- D.4** Conversión de un número binario, octal o hexadecimal a decimal
- D.5** Conversión de un número decimal a binario, octal o hexadecimal
- D.6** Números binarios negativos: notación de complemento a dos

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)

D.I Introducción

En este apéndice presentaremos los sistemas numéricos clave que utilizan los programadores de C++, especialmente cuando trabajan en proyectos de software que requieren de una estrecha interacción con el hardware a nivel de máquina. Entre los proyectos de este tipo están los sistemas operativos, el software de redes computacionales, los compiladores, sistemas de bases de datos y aplicaciones que requieren de un alto rendimiento.

Cuando escribimos un entero, como 227 o -63, en un programa de C++, se asume que el número está en el sistema numérico decimal (base 10). Los dígitos en el sistema numérico decimal son 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9. El dígito más bajo es el 0 y el más alto es el 9 (uno menos que la base, 10). En su interior, las computadoras utilizan el sistema numérico binario (base 2). Este sistema numérico sólo tiene dos dígitos: 0 y 1. El dígito más bajo es el 0 y el más alto es el 1 (uno menos que la base, 2).

Como veremos, los números binarios tienden a ser mucho más extensos que sus equivalentes decimales. Los programadores que trabajan con lenguajes ensambladores y en lenguajes de alto nivel como C++, que les permiten llegar hasta el nivel de máquina, encuentran que es complicado trabajar con números binarios. Por eso existen otros dos sistemas numéricos, el sistema numérico octal (base 8) y el sistema numérico hexadecimal (base 16), que son populares debido a que permiten abreviar los números binarios de una manera conveniente.

En el sistema numérico octal, los dígitos utilizados son del 0 al 7. Debido a que tanto el sistema numérico binario como el octal tienen menos dígitos que el sistema numérico decimal, sus dígitos son los mismos que sus correspondientes en decimal.

El sistema numérico hexadecimal presenta un problema, ya que requiere de 16 dígitos: el dígito más bajo es 0 y el más alto tiene un valor equivalente al 15 decimal (uno menos que la base, 16). Por convención utilizamos las letras de la A a la F para representar los dígitos hexadecimales que corresponden a los valores decimales del 10 al 15. Por lo tanto, en hexadecimal podemos tener números como el 876, que consisten solamente de dígitos similares a los decimales; números como 8A55F que consisten de dígitos y letras; y números como FFE que consisten solamente de letras. En ocasiones un número hexadecimal puede coincidir con una palabra común como FACE o FEED (en inglés); esto puede parecer extraño para los programadores acostumbrados a trabajar con números. Los dígitos de los sistemas numéricos binario, octal, decimal y hexadecimal se sintetizan en las figuras D.1 y D.2.

Cada uno de estos sistemas numéricos utilizan la notación posicional: cada posición en la que se escribe un dígito tiene un valor posicional distinto. Por ejemplo, en el número decimal 937 (el 9, el 3 y el 7 se conocen como valores simbólicos) decimos que el 7 se escribe en la posición de las unidades; el 3, en la de las decenas; y el 9, en la de las centenas. Observe que cada una de estas posiciones es una potencia de la base (10) y que estas potencias empiezan en 0 y aumentan de 1 en 1 a medida que nos desplazamos hacia la izquierda por el número (figura D.3).

Dígito binario	Dígito octal	Dígito decimal	Dígito hexadecimal
0	0	0	0
1	1	1	1
	2	2	2
	3	3	3
	4	4	4
	5	5	5

Figura D.1 | Dígitos de los sistemas numéricos binario, octal, decimal y hexadecimal. (Parte I de 2).

Dígito binario	Dígito octal	Dígito decimal	Dígito hexadecimal
6	6	6	
7	7	7	
		8	8
		9	9
			A (valor de 10 en decimal)
			B (valor de 11 en decimal)
			C (valor de 12 en decimal)
			D (valor de 13 en decimal)
			E (valor de 14 en decimal)
			F (valor de 15 en decimal)

Figura D.1 | Dígitos de los sistemas numéricos binario, octal, decimal y hexadecimal. (Parte 2 de 2).

Atributo	Binario	Octal	Decimal	Hexadecimal
Base	2	8	10	16
Dígito más bajo	0	0	0	0
Dígito más alto	1	7	9	F

Figura D.2 | Comparación de los sistemas binario, octal, decimal y hexadecimal.

Valores posicionales en el sistema numérico decimal			
Dígito decimal	9	3	7
Nombre de la posición	Centenas	Decenas	Unidades
Valor posicional	100	10	1
Valor posicional como potencia de la base (10)	10^2	10^1	10^0

Figura D.3 | Valores posicionales en el sistema numérico decimal.

Para números decimales más extensos, las siguientes posiciones a la izquierda serían: de millares (10 a la tercera potencia), de decenas de millares (10 a la cuarta potencia), de centenas de millares (10 a la quinta potencia), de los millones (10 a la sexta potencia), de decenas de millones (10 a la séptima potencia), y así sucesivamente.

En el número binario 101 decimos que el 1 más a la derecha se escribe en la posición de los unos, el 0 se escribe en la posición de los dos y el 1 de más a la izquierda se escribe en la posición de los cuatros. Observe que cada una de estas posiciones es una potencia de la base (2) y que estas potencias empiezan en 0 y aumentan de 1 en 1 a medida que nos desplazamos hacia la izquierda por el número (figura D.4). Por lo tanto, $101 = 2^2 + 2^0 + 4 + 1 = 5$.

Para números binarios más extensos, las siguientes posiciones a la izquierda serían la posición de los ochos (2 a la tercera potencia), la posición de los dieciséis (2 a la cuarta potencia), la posición de los treinta y dos (2 a la quinta potencia), la posición de los sesenta y cuatros (2 a la sexta potencia), y así sucesivamente.

En el número octal 425, decimos que el 5 se escribe en la posición de los unos, el 2 se escribe en la posición de los ocho y el 4 se escribe en la posición de los sesenta y cuatro. Observe que cada una de estas posiciones es una potencia de la base (8) y que estas potencias empiezan en 0 y aumentan de 1 en 1 a medida que nos desplazamos hacia la izquierda por el número (figura D.5).

Para números octales más extensos, las siguientes posiciones a la izquierda sería la posición de los quinientos doce (8 a la tercera potencia), la posición de los cuatro mil noventa y seis (8 a la cuarta potencia), la posición de los treinta y dos mil setecientos sesenta y ocho (8 a la quinta potencia), y así sucesivamente.

En el número hexadecimal 3DA, decimos que la A se escribe en la posición de los unos, la D se escribe en la posición de los dieciséis y el 3 se escribe en la posición de los doscientos cincuenta y seis. Observe que cada una de estas posiciones es una potencia de la base (16) y que estas potencias empiezan en 0 y aumentan de 1 en 1 a medida que nos desplazamos hacia la izquierda por el número (figura D.6).

Para números hexadecimales más extensos, las siguientes posiciones a la izquierda serían la posición de los cuatro mil noventa y seis (16 a la tercera potencia), la posición de los sesenta y cinco mil quinientos treinta y seis (16 a la cuarta potencia), y así sucesivamente.

Valores posicionales en el sistema numérico binario			
Dígito binario	1	0	1
Nombre de la posición	Cuatros	Dos	Unos
Valor posicional	4	2	1
Valor posicional como potencia de la base (2)	2^2	2^1	2^0

Figura D.4 | Valores posicionales en el sistema numérico binario.

Valores posicionales en el sistema numérico octal			
Dígito octal	4	2	5
Nombre de la posición	Sesenta y cuatros	Ochos	Unos
Valor posicional	64	8	1
Valor posicional como potencia de la base (8)	8^2	8^1	8^0

Figura D.5 | Valores posicionales en el sistema numérico octal.

Valores posicionales en el sistema numérico hexadecimal			
Dígito hexadecimal	3	D	A
Nombre de la posición	Doscientos cincuenta y seis	Dieciséis	Unos
Valor posicional	256	16	1
Valor posicional como potencia de la base (16)	16^2	16^1	16^0

Figura D.6 | Valores posicionales en el sistema numérico hexadecimal.

D.2 Abreviatura de los números binarios como números octales y hexadecimales

En computación, el uso principal de los números octales y hexadecimales es para abreviar representaciones binarias demasiado extensas. La figura D.7 muestra que los números binarios extensos pueden expresarse más concisamente en sistemas numéricos con bases mayores que en el sistema numérico binario.

Una relación especialmente importante que tienen tanto el sistema numérico octal como el hexadecimal con el sistema binario es que las bases de los sistemas octal y hexadecimal (8 y 16, respectivamente) son potencias de la base del sistema numérico binario (base 2). Considere el siguiente número binario de 12 dígitos y sus equivalentes en octal y

Número decimal	Representación binaria	Representación octal	Representación hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Figura D.7 | Equivalentes en decimal, binario, octal y hexadecimal.

hexadecimal. Vea si puede determinar cómo esta relación hace que sea conveniente abreviar los números binarios en octal o hexadecimal. La respuesta sigue después de los números.

Número binario	Equivalente en octal	Equivalente en hexadecimal
100011010001	4321	8D1

Para ver cómo el número binario se convierte fácilmente en octal, sólo divida el número binario de 12 dígitos en grupos de tres bits consecutivos, empezando desde la derecha, y escriba esos grupos por encima de los dígitos correspondientes del número octal, como se muestra a continuación:

100	011	010	001
4	3	2	1

Observe que el dígito octal que escribió debajo de cada grupo de tres bits corresponde precisamente al equivalente octal de ese número binario de 3 dígitos que se muestra en la figura D.7.

El mismo tipo de relación puede observarse al convertir números de binario a hexadecimal. Divida el número binario de 12 dígitos en grupos de cuatro bits consecutivos, empezando desde la derecha, y escriba esos grupos por encima de los dígitos correspondientes del número hexadecimal, como se muestra a continuación:

1000	1101	0001	
8	D	1	

Observe que el dígito hexadecimal que escribió debajo de cada grupo de cuatro bits corresponde precisamente al equivalente hexadecimal de ese número binario de 4 dígitos que se muestra en la figura D.7.

D.3 Conversión de números octales y hexadecimales a binarios

En la sección anterior vimos cómo convertir números binarios a sus equivalentes en octal y hexadecimal, formando grupos de dígitos binarios y simplemente volviéndolos a escribir como sus valores equivalentes en dígitos octales o hexadecimales. Este proceso puede utilizarse en forma inversa para producir el equivalente en binario de un número octal o hexadecimal.

Por ejemplo, el número octal 653 se convierte en binario simplemente escribiendo el 6 como su equivalente binario de 3 dígitos 110, el 5 como su equivalente binario de 3 dígitos 101 y el 3 como su equivalente binario de 3 dígitos 011 para formar el número binario de 9 dígitos 110101011.

El número hexadecimal FAD5 se convierte en binario simplemente escribiendo la F como su equivalente binario de 4 dígitos 1111, la A como su equivalente binario de 4 dígitos 1010, la D como su equivalente binario de 4 dígitos 1101 y el 5 como su equivalente binario de 4 dígitos 0101, para formar el número binario de 16 dígitos 1111101011010101.

D.4 Conversión de un número binario, octal o hexadecimal a decimal

Como estamos acostumbrados a trabajar con el sistema decimal, a menudo es conveniente convertir un número binario, octal o hexadecimal en decimal para tener una idea de lo que “realmente” vale el número. Nuestros diagramas en la sección D.1 expresan los valores posicionales en decimal. Para convertir un número en decimal desde otra base, multiplique el equivalente en decimal de cada dígito por su valor posicional y sume estos productos. Por ejemplo, el número binario 110101 se convierte en el número 53 decimal, como se muestra en la figura D.8.

Para convertir el número 7614 octal en el número 3980 decimal utilizamos la misma técnica, esta vez utilizando los valores posicionales apropiados para el sistema octal, como se muestra en la figura D.9.

Para convertir el número AD3B hexadecimal en el número 44347 decimal utilizamos la misma técnica, esta vez empleando los valores posicionales apropiados para el sistema hexadecimal, como se muestra en la figura D.10.

D.5 Conversión de un número decimal a binario, octal o hexadecimal

Las conversiones de la sección D.4 siguen naturalmente las convenciones de la notación posicional. Las conversiones de decimal a binario, octal o hexadecimal también siguen estas convenciones.

Conversión de un número binario en decimal

Valores posicionales:	32	16	8	4	2	1
Valores simbólicos:	1	1	0	1	0	1
Productos:	1*32=32	1*16=16	0*8=0	1*4=4	0*2=0	1*1=1
Suma:	$= 32 + 16 + 0 + 4 + 0 + 1 = 53$					

Figura D.8 | Conversión de un número binario en decimal.

Conversión de un número octal en decimal

Valores posicionales:	512	64	8	1
Valores simbólicos:	7	6	1	4
Productos:	7*512=3584	6*64=384	1*8=8	4*1=4
Suma:	$= 3584 + 384 + 8 + 4 = 3980$			

Figura D.9 | Conversión de un número octal en decimal.

Conversión de un número hexadecimal en decimal

Valores posicionales:	4096	256	16	1
Valores simbólicos:	A	D	3	B
Productos:	A*4096=40960	D*256=3328	3*16=48	B*1=11
Suma:	$= 40960 + 3328 + 48 + 11 = 44347$			

Figura D.10 | Conversión de un número hexadecimal en decimal.

Suponga que queremos convertir el número 57 decimal en binario. Empezamos escribiendo los valores posicionales de las columnas de derecha a izquierda, hasta llegar a una columna cuyo valor posicional sea mayor que el número decimal. Como no necesitamos esa columna, podemos descartarla. Por lo tanto, primero escribimos:

Valores posicionales: 64 32 16 8 4 2 1

Luego descartamos la columna con el valor posicional de 64, dejando:

Valores posicionales: 32 16 8 4 2 1

A continuación, empezamos a trabajar desde la columna más a la izquierda y nos vamos desplazando hacia la derecha. Dividimos 57 entre 32 y observamos que hay un 1 en 57, con un residuo de 25, por lo que escribimos 1 en la columna de los 32. Dividimos 25 entre 16 y observamos que hay un 1 en 25, con un residuo de 9, por lo que escribimos 1 en la columna de los 16. Dividimos 9 entre 8 y observamos que hay un 1 en 9 con un residuo de 1. Las siguientes dos columnas producen el cociente de 0 cuando se divide 1 entre sus valores posicionales, por lo que escribimos 0 en las columnas de los 4 y de los 2. Por último, 1 entre 1 es 1, por lo que escribimos 1 en la columna de los 1. Esto nos da:

Valores posicionales: 32 16 8 4 2 1

Valores simbólicos: 1 1 1 0 0 1

y, por lo tanto, el 57 decimal es equivalente al 111001 binario.

Para convertir el número decimal 103 en octal, empezamos por escribir los valores posicionales de las columnas hasta llegar a una columna cuyo valor posicional sea mayor que el número decimal. Como no necesitamos esa columna, podemos descartarla. Por lo tanto, primero escribimos:

Valores posicionales: 512 64 8 1

Luego descartamos la columna con el valor posicional de 512, lo que nos da:

Valores posicionales: 64 8 1

A continuación, empezamos a trabajar desde la columna más a la izquierda y nos vamos desplazando hacia la derecha. Dividimos 103 entre 64 y observamos que hay un 1 en 103 con un residuo de 39, por lo que escribimos 1 en la columna de los 64. Dividimos 39 entre 8 y observamos que el 8 cabe cuatro veces en 39 con un residuo de 7, por lo que escribimos 4 en la columna de los 8. Por último, dividimos 7 entre 1 y observamos que el 1 cabe siete veces en 7 y no hay residuo, por lo que escribimos 7 en la columna de los 1. Esto nos da:

Valores posicionales: 64 8 1

Valores simbólicos: 1 4 7

y por lo tanto, el 103 decimal es equivalente al 147 octal.

Para convertir el número decimal 375 en hexadecimal, empezamos por escribir los valores posicionales de las columnas hasta llegar a una columna cuyo valor posicional sea mayor que el número decimal. Como no necesitamos esa columna, podemos descartarla. Por consecuencia, primero escribimos:

Valores posicionales: 4096 256 16 1

Luego descartamos la columna con el valor posicional de 4096, lo que nos da:

Valores posicionales: 256 16 1

A continuación, empezamos a trabajar desde la columna más a la izquierda y nos vamos desplazando hacia la derecha. Dividimos 375 entre 256 y observamos que 256 cabe una vez en 375 con un residuo de 119, por lo que escribimos 1 en la columna de los 256. Dividimos 119 entre 16 y observamos que el 16 cabe siete veces en 119 con un residuo de 7, por lo que escribimos 7 en la columna de los 16. Por último, dividimos 7 entre 1 y observamos que el 1 cabe siete veces en 7 y no hay residuo, por lo que escribimos 7 en la columna de los 1. Esto produce:

Valores posicionales: 256 16 1

Valores simbólicos: 1 7 7

y por lo tanto, el 375 decimal es equivalente al 177 hexadecimal.

D.6 Números binarios negativos: notación de complemento a dos

La discusión en este apéndice se ha enfocado hasta ahora en números positivos. En esta sección explicaremos cómo las computadoras representan números negativos mediante el uso de la notación de *complementos a dos*. Primero explica-

remos cómo se forma el complemento a dos de un número binario y después mostraremos por qué representa el valor negativo de dicho número binario.

Considere una máquina con enteros de 32 bits. Suponga que se ejecuta la siguiente instrucción:

```
int valor = 13;
```

La representación en 32 bits de valor es:

```
00000000 00000000 00000000 00001101
```

Para formar el negativo de valor, primero formamos su *complemento a uno* aplicando el operador de complemento a nivel de bits de C++ (~):

```
complementoALUnoDeValor = ~valor;
```

Internamente, `~valor` es ahora valor con cada uno de sus bits invertidos; los unos se convierten en ceros y los ceros en unos, como se muestra a continuación:

```
valor:  
00000000 00000000 00000000 00001101
```

`~valor` (es decir, el complemento a uno de valor):
`11111111 11111111 11111111 11110010`

Para formar el complemento a dos de valor, simplemente sumamos 1 al complemento a uno de valor. Por lo tanto:

El complemento a dos de valor es:

```
11111111 11111111 11111111 11110011
```

Ahora, si esto de hecho es igual a -13, deberíamos poder sumarlo al 13 binario y obtener como resultado 0. Comprobemos esto:

```
00000000 00000000 00000000 00001101  
+11111111 11111111 11111111 11110011  
-----  
00000000 00000000 00000000 00000000
```

El bit de acarreo que sale de la columna que está más a la izquierda se descarta y evidentemente obtenemos 0 como resultado. Si sumamos el complemento a uno de un número a ese mismo número, todos los dígitos del resultado serían iguales a 1. La clave para obtener un resultado en el que todos los dígitos sean cero es que el complemento a dos es 1 más que el complemento a 1. La suma de 1 hace que el resultado de cada columna sea 0 y se acarree un 1. El acarreo sigue desplazándose hacia la izquierda hasta que se descarta en el bit que está más a la izquierda, con lo que todos los dígitos del número resultante son iguales a cero.

En realidad, las computadoras realizan una suma como:

```
x = a - valor;
```

mediante la suma del complemento a dos de valor con a, como se muestra a continuación:

```
x = a + (~valor + 1);
```

Suponga que a es 27 y que `valor` es 13 como en el ejemplo anterior. Si el complemento a dos de `valor` es en realidad el negativo de éste, entonces al sumar el complemento de dos de `valor` con a se produciría el resultado de 14. Comprobemos esto:

```
a (es decir, 27)      00000000 00000000 00000000 00011011  
+(~valor + 1)    +11111111 11111111 11111111 11110011  
-----  
00000000 00000000 00000000 00001110
```

lo que ciertamente da como resultado 14.

Resumen

- Cuando escribimos un entero como 19, 227 o -63 en un programa de C++, suponemos que el número se encuentra en el sistema numérico decimal (base 10). Los dígitos en el sistema numérico decimal son 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9. El dígito más bajo es el 0 y el más alto es el 9 (uno menos que la base, 10).

- En su interior, las computadoras utilizan el sistema numérico binario (base 2). Este sistema numérico sólo tiene dos dígitos: 0 y 1. El dígito más bajo es el 0 y el más alto es el 1 (uno menos que la base, 2).
- El sistema numérico octal (base 8) y el sistema numérico hexadecimal (base 16) son populares debido a que permiten abreviar los números binarios de una manera conveniente.
- Los dígitos que se utilizan en el sistema numérico octal son del 0 al 7.
- El sistema numérico hexadecimal presenta un problema, ya que requiere de dieciséis dígitos: el dígito más bajo es 0 y el más alto tiene un valor equivalente al 15 decimal (uno menos que la base, 16). Por convención utilizamos las letras de la A a la F para representar los dígitos hexadecimales que corresponden a los valores decimales del 10 al 15.
- Cada uno de estos sistemas numéricos utilizan la notación posicional: cada posición en la que se escribe un dígito tiene un distinto valor posicional.
- Una relación especialmente importante que tienen tanto el sistema numérico octal como el hexadecimal con el sistema binario es que las bases de los sistemas octal y hexadecimal (8 y 16, respectivamente) son potencias de la base del sistema numérico binario (base 2).
- Para convertir un número octal en binario, sustituya cada dígito octal con su equivalente binario de tres dígitos.
- Para convertir un número hexadecimal en binario, simplemente sustituya cada dígito hexadecimal con su equivalente binario de cuatro dígitos.
- Como estamos acostumbrados a trabajar con el sistema decimal, es conveniente convertir un número binario, octal o hexadecimal en decimal para tener una idea de lo que “realmente” vale el número.
- Para convertir un número en decimal desde otra base, multiplique el equivalente en decimal de cada dígito por su valor posicional y sume estos productos.
- Las computadoras representan números negativos mediante el uso de la notación de complementos a dos.
- Para formar el negativo de un valor en binario, primero formamos su complemento a uno aplicando el operador de complemento a nivel de bits de C++ (~): esto invierte los bits del valor. Para formar el complemento a dos de un valor, simplemente sumamos uno al complemento a uno de ese valor.

Terminología

base	sistema numérico de base 8
conversiones	sistema numérico de base 10
dígito	sistema numérico de base 16
notación de complemento a uno	sistema numérico decimal
notación de complementos a dos	sistema numérico hexadecimal
notación posicional	sistema numérico octal
operador de complemento a nivel de bits (~)	valor negativo
sistema numérico binario	valor posicional
sistema numérico de base 2	valor simbólico

Ejercicios de autoevaluación

- D.1** Las bases de los sistemas numéricos decimal, binario, octal y hexadecimal son _____, _____, _____ y _____, respectivamente.
- D.2** En general, las representaciones en decimal, octal y hexadecimal de un número binario dado contienen (más/menos) dígitos de los que contiene el número binario.
- D.3** (*Verdadero/falso*) Una de las razones populares de utilizar el sistema numérico decimal es que forma una notación conveniente para abreviar números binarios, en la que simplemente se sustituye un dígito decimal por cada grupo de cuatro dígitos binarios.
- D.4** La representación (octal/hexadecimal/decimal) de un valor binario grande es la más concisa (de las alternativas dadas).
- D.5** (*Verdadero/falso*) El dígito de mayor valor en cualquier base es uno más que la base.
- D.6** (*Verdadero/falso*) El dígito de menor valor en cualquier base es uno menos que la base.
- D.7** El valor posicional del dígito que se encuentra más a la derecha en cualquier número, ya sea binario, octal, decimal o hexadecimal es siempre _____.
- D.8** El valor posicional del dígito que está a la izquierda del dígito que se encuentra más a la derecha en cualquier número, ya sea binario, octal, decimal o hexadecimal es siempre igual a _____.

D.9 Complete los valores que faltan en esta tabla de valores posicionales para las cuatro posiciones que están más a la derecha en cada uno de los sistemas numéricos indicados:

decimal	1000	100	10	1
hexadecimal	...	256
binario
octal	512	...	8	...

- D.10** Convierta el número binario 110101011000 en octal y en hexadecimal.
- D.11** Convierta el número hexadecimal FACE en binario.
- D.12** Convierta el número octal 7316 en binario.
- D.13** Convierta el número hexadecimal 4FEC en octal. (*Sugerencia:* primero convierta el número 4FEC en binario y después convierta el número resultante en octal).
- D.14** Convierta el número binario 1101110 en decimal.
- D.15** Convierta el número octal 317 en decimal.
- D.16** Convierta el número hexadecimal EFD4 en decimal.
- D.17** Convierta el número decimal 177 en binario, en octal y en hexadecimal.
- D.18** Muestre la representación binaria del número decimal 417. Después muestre el complemento a uno de 417 y el complemento a dos del mismo número.
- D.19** ¿Cuál es el resultado cuando se suma el complemento a dos de un número con ese mismo número?

Respuestas a los ejercicios de autoevaluación

D.1 10, 2, 8, 16.

D.2 Menos.

D.3 Falso. El hexadecimal hace esto.

D.4 Hexadecimal.

D.5 Falso. El dígito de mayor valor en cualquier base es uno menos que la base.

D.6 Falso. El dígito de menor valor en cualquier base es cero.

D.7 1 (la base elevada a la potencia de cero).

D.8 La base del sistema numérico.

D.9 Complete la tabla que se muestra a continuación:

decimal	1000	100	10	1
hexadecimal	4096	256	16	1
binario	8	4	2	1
octal	512	64	8	1

D.10 6530 octal; D58 hexadecimal.

D.11 1111 1010 1100 1110 binario.

D.12 111 011 001 110 binario.

D.13 0 100 111 111 101 100 binario; 47754 octal.

D.14 $2 + 4 + 8 + 32 + 64 = 110$ decimal.

D.15 $7 + 1 * 8 + 3 * 64 = 7 + 8 + 192 = 207$ decimal.

D.16 $4 + 13 * 16 + 15 * 256 + 14 * 4096 = 61396$ decimal.

D.17 177 decimal
en binario:

```
256 128 64 32 16 8 4 2 1
128 64 32 16 8 4 2 1
(1*128)+(0*64)+(1*32)+(1*16)+(0*8)+(0*4)+(0*2)+(1*1)
10110001
```

en octal:

```
512 64 8 1
64 8 1
(2*64)+(6*8)+(1*1)
261
```

en hexadecimal:

```
256 16 1
16 1
(11*16)+(1*1)
(B*16)+(1*1)
B1
```

D.18 Binario:

```
512 256 128 64 32 16 8 4 2 1
256 128 64 32 16 8 4 2 1
(1*256)+(1*128)+(0*64)+(1*32)+(0*16)+(0*8)+(0*4)+(0*2)+(1*1)
110100001
```

Complemento a uno: 001011110

Complemento a dos: 001011111

Comprobación: número binario original + su complemento a dos:

```
110100001
001011111
-----
000000000
```

D.19 Cero.

Ejercicios

D.20 Algunas personas argumentan que muchos de nuestros cálculos se realizarían más fácilmente en el sistema numérico de base 12 que en el sistema numérico de base 10 (decimal), ya que el 12 puede dividirse por muchos más números que el 10 (por la base 10). ¿Cuál es el dígito de menor valor en la base 12? ¿Cuál podría ser el símbolo con mayor valor para un dígito en la base 12? ¿Cuáles son los valores posicionales de las cuatro posiciones más a la derecha de cualquier número en el sistema numérico de base 12?

D.21 Complete la siguiente tabla de valores posicionales para las cuatro posiciones más a la derecha en cada uno de los sistemas numéricos indicados:

decimal	1000	100	10	1
base 6	6	...
base 13	...	169
base 3	27

D.22 Convierta el número binario 100101111010 en octal y en hexadecimal.

D.23 Convierta el número hexadecimal 3A7D en binario.

D.24 Convierta el número hexadecimal 765F en octal. (*Sugerencia:* Primero conviértalo en binario y después convierta el número resultante en octal).

D.25 Convierta el número binario 1011110 en decimal.

D.26 Convierta el número octal 426 en decimal.

D.27 Convierta el número hexadecimal FFFF en decimal.

D.28 Convierta el número decimal 299 en binario, en octal y en hexadecimal.

D.29 Muestre la representación binaria del número decimal 779. Después muestre el complemento a uno de 779 y el complemento a dos del mismo número.

D.30 Muestre el complemento a dos del valor entero –1 en una máquina con enteros de 32 bits.

E



Utilizaremos una señal que he probado y descubrí que tiene largo alcance, y es fácil de gritar. ¡Waa-juu!

—Zane Grey

Es sin duda un problema de tres canales.

—Sir Arthur Conan Doyle

Aun así hay unión en la partición.

—William Shakespeare

Temas sobre código heredado de C

OBJETIVOS

En este capítulo aprenderá a:

- Redirigir la entrada del teclado para que provenga de un archivo, y redirigir la salida a la pantalla para que vaya a un archivo.
- Escribir funciones que utilicen listas de argumentos de longitud variable.
- Procesar argumentos de la línea de comandos.
- Procesar eventos inesperados dentro de un programa.
- Asignar memoria en forma dinámica para los arreglos, usando la asignación dinámica de memoria estilo C.
- Cambiar el tamaño de la memoria asignada en forma dinámica, usando la asignación dinámica de memoria estilo C.

- E.1** Introducción
- E.2** Redirección de la entrada/salida en sistemas UNIX/LINUX/Mac OS X y Windows
- E.3** Listas de argumentos de longitud variable
- E.4** Uso de argumentos de línea de comandos
- E.5** Observaciones acerca de la compilación de programas con varios archivos de código fuente
- E.6** Terminación de los programas con `exit` y `atexit`
- E.7** Calificador de tipo `volatile`
- E.8** Sufijos para constantes enteras y de punto flotante
- E.9** Manejo de señales
- E.10** Asignación dinámica de memoria con `calloc` y `realloc`
- E.11** Bifurcación incondicional: `goto`
- E.12** Uniones
- E.13** Especificaciones de vinculación
- E.14** Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)

E.1 Introducción

En este capítulo veremos varios temas que no se cubren comúnmente en los cursos introductorios. Muchas de las herramientas de las que hablaremos aquí son específicas de ciertos sistemas operativos, en especial de UNIX/LINUX/Mac OS X y/o Windows. La mayor parte del material es para beneficio de los programadores de C++ que necesitan trabajar con código antiguo heredado de C.

E.2 Redirección de la entrada/salida en sistemas UNIX/LINUX/ Mac OS X y Windows

Por lo general, la entrada para un programa proviene del teclado (entrada estándar) y la salida de un programa se muestra en la pantalla (salida estándar). En la mayoría de los sistemas computacionales (en especial UNIX, LINUX, Mac OS X y Windows) es posible redirigir las entradas para que provengan de un archivo, y redirigir las salidas para que se coloquen en un archivo. Ambas formas de redirección se pueden lograr sin necesidad de utilizar las herramientas para procesar archivos de la biblioteca estándar.

Hay varias formas de redirigir la entrada y la salida de la línea de comandos de UNIX. Considere el archivo ejecutable `suma` que recibe un valor entero a la vez, mantiene un total de la suma de los valores hasta que se establece el indicador de fin de archivo y después imprime el resultado. Por lo general, el usuario introduce los enteros mediante el teclado e introduce la combinación de teclas de fin de archivo para indicar que no se introducirán más valores. Con la redirección de la entrada, los datos de entrada se pueden almacenar en un archivo. Por ejemplo, si los datos se almacenan en el archivo `entrada`, la línea de comandos

```
$ suma < entrada
```

hace que se ejecute el programa `suma`; el **símbolo de redirección de entrada** (`<`) indica que se van a utilizar los datos en el archivo `entrada` (en vez del teclado) como entrada para el programa. Para redirigir la entrada en una ventana de **Símbolo del sistema** de Windows se realiza la misma acción.

Observe que `$` representa el indicador de línea de comandos de UNIX. (Los indicadores de UNIX varían de un sistema a otro, y de un intérprete de comandos a otro en un solo sistema). La redirección es una función del sistema operativo, no una característica más de C++.

El segundo método para redirigir la entrada es el uso de **canalizaciones**. Una **canalización** (`|`) hace que la salida de un programa se redirija como la entrada para otro programa. Suponga que el programa `aleatorio` imprime una serie de enteros aleatorios; la salida de `aleatorio` puede “canalizarse” directamente al programa `suma`, usando la línea de comandos de UNIX

```
$ aleatorio | suma
```

Esto hace que se calcule la suma de los enteros producidos por `aleatorio`. La canalización se puede realizar en UNIX, LINUX, Mac OS X y Windows.

La salida de un programa se puede redirigir a un archivo mediante el uso del **símbolo de redirección de salida** (`>`). (El mismo símbolo se utiliza para UNIX, LINUX, Mac OS X y Windows). Por ejemplo, para redirigir la salida del programa `aleatorio` a un nuevo archivo llamado `salida`, utilizamos

```
$ aleatorio > salida
```

Por último, la salida de un programa se puede agregar al final de un archivo existente mediante el uso del **símbolo para agregar salida** (`>>`). (El mismo símbolo se utiliza para UNIX, LINUX, Mac OS X y Windows). Por ejemplo, para agregar la salida del programa `aleatorio` al archivo `salida` que se creó en la línea de comandos anterior, utilizamos la línea de comandos

```
$ aleatorio >> salida
```

E.3 Listas de argumentos de longitud variable¹

Es posible crear funciones que reciban un número de argumentos no especificado. Los tres puntos suspensivos (...) en el prototipo de una función indican que esa función recibe un número variable de argumentos de cualquier tipo. Observe que siempre deben colocarse los tres puntos suspensivos al final de la lista de parámetros, y debe haber por lo menos un argumento antes de los tres puntos suspensivos. Las macros y definiciones del **encabezado de argumentos variables** `<cstdarg>` (figura E.1) proporcionan las herramientas necesarias para crear funciones con listas de argumentos de longitud variable.

En la figura E.2 se demuestra la función `promedio` que recibe un número variable de argumentos. El primer argumento de `promedio` siempre es el número de valores a promediar, y el resto de los argumentos deben ser de tipo `double`.

La función `promedio` utiliza todas las definiciones y macros del encabezado `<cstdarg>`. El objeto `lista`, de tipo `va_list`, es utilizado por las macros `va_start`, `va_arg` y `va_end` para procesar la lista de argumentos de longitud variable de la función `promedio`. La función invoca a `va_start` para inicializar el objeto `lista` y utilizarlo en `va_arg` y `va_end`. La macro recibe dos argumentos: el objeto `lista` y el identificador del argumento de más a la derecha en la lista de argumentos antes de los tres puntos suspensivos: en este caso, `cuenta` (`va_start` usa a `cuenta` aquí para determinar en dónde empieza la lista de argumentos de longitud variable).

A continuación, la función `promedio` suma en forma repetida los argumentos en la lista de argumentos de longitud variable al `total`. El valor a sumar a `total` se obtiene de la lista de argumentos mediante una invocación a la macro `va_arg`. La macro `va_arg` recibe dos argumentos: el objeto `lista` y el tipo del valor que se espera en la lista de argumentos (`double` en este caso), y devuelve el valor del argumento. La función `promedio` invoca a la macro `va_end` con el objeto `lista` como argumento antes de regresar. Por último, se calcula el promedio y se devuelve a `main`. Observe que sólo utilizamos argumentos `double` para la porción de longitud variable de la lista de argumentos.

Identificador	Descripción
<code>va_list</code>	Un tipo adecuado para contener la información que necesitan las macros <code>va_start</code> , <code>va_arg</code> y <code>va_end</code> . Para acceder a los argumentos en una lista de argumentos de longitud variable, debe declararse un objeto de tipo <code>va_list</code> .
<code>va_start</code>	Una macro que se invoca antes de que se pueda acceder a los argumentos de una lista de argumentos de longitud variable. La macro inicializa el objeto declarado con <code>va_list</code> para que lo utilicen las macros <code>va_arg</code> y <code>va_end</code> .
<code>va_arg</code>	Una macro que se expande a una expresión del valor y tipo del siguiente argumento en la lista de argumentos de longitud variable. Cada invocación de <code>va_arg</code> modifica el objeto declarado con <code>va_list</code> , de manera que el objeto apunte al siguiente argumento en la lista.
<code>va_end</code>	Una macro que realiza las labores de mantenimiento de terminación en una función cuya lista de argumentos de longitud variable fue referenciada por la macro <code>va_start</code> .

Figura E.1 | El tipo y las macros definidas en el encabezado `<cstdarg>`.

1. En C++, los programadores utilizan la sobrecarga de funciones para realizar gran parte de lo que realizan los programadores de C con las listas de argumentos de longitud variable.

```

1 // Fig. E.2: figE_02.cpp
2 // Uso de listas de argumentos de longitud variable.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::ios;
7
8 #include <iomanip>
9 using std::setw;
10 using std::setprecision;
11 using std::setiosflags;
12 using std::fixed;
13
14 #include <cstdarg>
15 using std::va_list;
16
17 double promedio( int, ... );
18
19 int main()
20 {
21     double double1 = 37.5;
22     double double2 = 22.5;
23     double double3 = 1.7;
24     double double4 = 10.2;
25
26     cout << fixed << setprecision( 1 ) << "double1 = "
27         << double1 << "\n";
28     cout << "double2 = " << double2 << "\n";
29     cout << "double3 = " << double3 << "\n";
30     cout << "double4 = " << double4 << endl
31     << setprecision( 3 )
32     << "\nEl promedio de double1 y double2 es "
33     << promedio( 2, double1, double2 );
34     << "\nEl promedio de double1, double2 y double3 es "
35     << promedio( 3, double1, double2, double3 );
36     << "\nEl promedio de double1, double2, double3 y "
37     << "double4 es "
38     << promedio( 4, double1, double2, double3, double4 )
39     << endl;
40
41     return 0;
42 } // fin de main
43
44 // calcula el promedio
45 double promedio( int cuenta, ... )
46 {
47     double total = 0;
48     va_list lista; // para almacenar la información que necesita va_start
49
50     va_start( lista, cuenta );
51
52     // procesa la lista de argumentos de longitud variable
53     for ( int i = 1; i <= cuenta; i++ )
54         total += va_arg( lista, double );
55
56     va_end( lista ); // termina va_start
57     return total / cuenta;
58 } // fin de la función promedio

```

```

double1 = 37.5
double2 = 22.5
double3 = 1.7
double4 = 10.2

El promedio de double1 y double2 es 30.000
El promedio de double1, double2 y double3 es 20.567
El promedio de double1, double2, double3 y double4 es 17.975

```

Figura E.2 | Uso de listas de argumentos de longitud variable.

Las listas de argumentos de longitud variable promueven las variables del tipo `float` al tipo `double`. Estas listas de argumentos también promueven las variables integrales menores que el tipo `int` al tipo `int` (las variables de tipo `int`, `unsigned`, `long` y `unsigned long` se dejan como están).



Observación de Ingeniería de Software E.I

Las listas de argumentos de longitud variable se pueden utilizar sólo con argumentos de tipos fundamentales y con argumentos de tipos struct estilo C, que no contienen características específicas de C++, como las funciones virtual, los constructores, destructores, referencias, miembros de datos const y clases base virtual.



Error común de programación E.I

Colocar tres puntos suspensivos en medio de una lista de parámetros de una función es un error de sintaxis. Sólo se pueden colocar tres puntos suspensivos al final de la lista de parámetros.

E.4 Uso de argumentos de línea de comandos

En muchos sistemas (Windows, UNIX, LINUX y Mac OS X en especial) es posible pasar argumentos a `main` desde una línea de comandos, para lo cual se incluyen los parámetros `int argc` y `char *argv[]` en la lista de parámetros de `main`. El parámetro `argc` recibe el número de argumentos de la línea de comandos. El parámetro `argv` es un arreglo de apuntadores `char *` que apuntan a cadenas en las que se almacenan los argumentos de la línea de comandos. Los usos comunes de los argumentos de la línea de comandos incluyen imprimir los argumentos, pasar opciones a un programa y pasar nombres de archivos al programa.

En la figura E.3 se copia un archivo a otro archivo, un carácter a la vez. El archivo ejecutable para el programa se llama `copiarArchivo` (es decir, el nombre ejecutable para el archivo). Una línea de comandos típica para el programa `copiarArchivo` en un sistema UNIX es

```
$ copiarArchivo entrada salida
```

Esta línea de comandos indica que el archivo `entrada` se va a copiar al archivo `salida`. Cuando se ejecuta el programa, si `argc` no es 3 (`copiarArchivo` cuenta como uno de los argumentos), el programa imprime un mensaje de error (línea 16).

```

1 // Fig. E.3: figE_03.cpp
2 // Uso de los argumentos de la línea de comandos
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::ios;
7
8 #include <fstream>
9 using std::ifstream;
10 using std::ofstream;
11
12 int main( int argc, char *argv[] )
13 {
14     // comprueba el número de argumentos de la línea de comandos
15     if ( argc != 3 )
16         cout << "Uso: copiarArchivo nombre_archent nombre_archsal" << endl;
17     else
18     {
19         ifstream archEnt( argv[ 1 ], ios::in );
20
21         // no se pudo abrir el archivo de entrada
22         if ( !archEnt )
23         {
24             cout << argv[ 1 ] << " no se pudo abrir" << endl;
25             return -1;
26         } // fin de if
27 }
```

Figura E.3 | Uso de los argumentos de la línea de comandos. (Parte I de 2).

```

28     ifstream archSal( argv[ 2 ], ios::out );
29
30     // no se pudo abrir el archivo de salida
31     if ( !archSal )
32     {
33         cout << argv[ 2 ] << " no se pudo abrir" << endl;
34         archEnt.close();
35         return -2;
36     } // fin de if
37
38     char c = archEnt.get(); // lee el primer carácter
39
40     while ( archEnt )
41     {
42         archSal.put( c ); // escribe el carácter
43         c = archEnt.get(); // lee el siguiente carácter
44     } // fin de while
45 } // fin de else
46
47 return 0;
48 } // fin de main

```

Figura E.3 | Uso de los argumentos de la línea de comandos. (Parte 2 de 2).

En cualquier otro caso, el arreglo `argv` contiene las cadenas "copiarArchivo", "entrada" y "salida". El programa utiliza los argumentos segundo y tercero en la línea de comandos como nombres de archivos. Para abrir los archivos, se crea el objeto `ifstream` llamado `archEnt` y el objeto `ofstream` llamado `archSal` (líneas 19 y 28). Si ambos archivos se abren con éxito, se leen caracteres del archivo `entrada` mediante la función miembro `get` y se escriben en el archivo `salida` mediante la función miembro `put`, hasta que se establece el indicador de fin de archivo para el archivo `entrada` (líneas 40 a 44). Despues el programa termina. El resultado es una copia exacta del archivo `entrada`. Observe que no todos los sistemas de computadoras soportan los argumentos de línea de comandos con tanta facilidad como UNIX, LINUX, Mac OS X y Windows. Por ejemplo, algunos sistemas VMS y Macintosh antiguos requieren ajustes especiales para procesar los argumentos de la línea de comandos. Consulte los manuales de su sistema para obtener más información acerca de los argumentos de la línea de comandos.

E.5 Observaciones acerca de la compilación de programas con varios archivos de código fuente

Como se dijo antes en el libro, es normal crear programas que consisten en varios archivos de código fuente (vea el capítulo 9, Clases: un análisis más detallado, parte 1). Hay varias consideraciones a tener en cuenta al crear programas distribuidos en varios archivos. Por ejemplo, la definición de una función debe estar completamente contenida en un solo archivo, no puede abarcar dos o más archivos.

En el capítulo 6 presentamos los conceptos de clase de almacenamiento y alcance. Aprendimos que las variables que se declaran fuera de cualquier definición de función son de la clase de almacenamiento `static` de manera pre-determinada, y se conocen como variables globales. Las variables globales se pueden utilizar en cualquier función que esté definida en el mismo archivo, después de la declaración de la variable. Las variables globales también se pueden utilizar en funciones definidas en otros archivos; sin embargo, deben declararse en cada archivo en el que se utilicen. Por ejemplo, si definimos la variable entera global `bandera` en un archivo, y hacemos referencia a esta variable en otro archivo, entonces ese otro archivo debe contener la declaración

```
extern int bandera;
```

antes de que se utilice la variable en ese archivo. En la declaración anterior, el especificador de clase de almacenamiento `extern` indica al compilador que la variable `bandera` está definida más adelante en ese mismo archivo, o en un archivo distinto. El compilador informa al vinculador que aparecen referencias a la variable `bandera` sin resolver en el archivo. (El compilador no sabe en dónde está definida `bandera`, por lo que deja que el vinculador trate de encontrarla). Si el vinculador no puede localizar una definición de `bandera`, se reporta un error de vinculación. Si se localiza una definición global apropiada, el vinculador resuelve las referencias al indicar en dónde se encuentra `bandera`.



Tip de rendimiento E.1

Las variables globales incrementan el rendimiento, ya que pueden ser utilizadas directamente por cualquier función; se elimina la sobrecarga de pasar datos a las funciones.



Observación de Ingeniería de Software E.2

Las variables globales deben evitarse, a menos que el rendimiento de la aplicación sea crítico o que la variable represente un recurso global compartido como cin, ya que violan el principio del menor privilegio y dificultan el mantenimiento del software.

Así como se pueden utilizar declaraciones `extern` para declarar variables globales para otros archivos de un programa, también se pueden utilizar los prototipos de las funciones para declarar funciones en otros archivos del programa. (El especificador `extern` no es obligatorio en un prototipo de función). Para ello hay que incluir el prototipo de la función en cada archivo en el que se va a invocar, y después se compila cada archivo de código fuente y se vinculan los archivos de código objeto resultantes. Los prototipos de las funciones indican al compilador que la función especificada está definida más adelante en el mismo archivo, o en un archivo distinto. El compilador no trata de resolver las referencias a dicha función; esa tarea se deja al vinculador. Si el vinculador no puede localizar la definición de una función, se genera un error.

Como ejemplo del uso de prototipos de funciones para extender el alcance de una función, considere cualquier programa que contenga la directiva del preprocesador `#include <cstring>`. Esta directiva incluye en un archivo los prototipos de las funciones tales como `strcmp` y `strcat`. Otras funciones en el archivo pueden utilizar a `strcmp` y `strcat` para realizar sus tareas. Las funciones `strcmp` y `strcat` se definen por separado para nosotros. No necesitamos saber en dónde están definidas. Simplemente reutilizamos el código en nuestros programas. El vinculador resuelve nuestras referencias a esas funciones. Este proceso nos permite utilizar las funciones en la biblioteca estándar.



Observación de Ingeniería de Software E.3

Al crear programas en varios archivos de código fuente se facilita la reutilización de software y la buena ingeniería de software. Las funciones pueden ser comunes para muchas aplicaciones. En dichas casos, esas funciones deben almacenarse en sus propios archivos de código fuente, y cada archivo de código fuente debe tener su correspondiente archivo de encabezado que contenga los prototipos de las funciones. Esto permite que los programadores de distintas aplicaciones reutilicen el mismo código, al incluir el archivo de encabezado apropiado y compilar su aplicación con el archivo de código fuente correspondiente.



Tip de portabilidad E.1

Algunos sistemas no soportan los nombres de variables globales, o los nombres de funciones con más de seis caracteres. Esto debe considerarse al escribir programas que se vayan a portar a varias plataformas.

Es posible restringir el alcance de una variable o función global al archivo en el que está definida. Cuando el especificador de clase de almacenamiento `static` se aplica a una variable o función con alcance de archivo, evita que esa variable sea utilizada por cualquier función que no esté definida en el mismo archivo. Esto se conoce como **vinculación interna**. Las variables y funciones globales (excepto las que son `const`) a las que no se antepone la palabra clave `static` en sus definiciones tienen **vinculación externa**: se pueden utilizar en otros archivos, si esos archivos contienen declaraciones y/o prototipos de funciones apropiados.

La declaración de la variable global

```
static double pi = 3.14159;
```

crea la variable `pi` de tipo `double`, la inicializa con `3.14159` e indica que `pi` sólo es conocida para las funciones en el archivo en el que está definida.

El especificador `static` se utiliza comúnmente con las funciones utilitarias que sólo son llamadas por las funciones en un archivo específico. Si una función no es requerida fuera de un archivo específico, debe hacerse valer el principio del menor privilegio mediante el uso de `static`. Si una función se define antes de utilizarla en un archivo, se debe aplicar `static` a la definición de la función. En caso contrario, se debe aplicar `static` al prototipo de la función. Los identificadores definidos en el espacio de nombres sin nombre también tienen vinculación interna. El estándar de C++ recomienda utilizar el espacio de nombres sin nombre en vez de `static`.

Al crear programas extensos a partir de varios archivos de código fuente, la compilación del programa se vuelve tediosa si para realizar pequeñas modificaciones en un archivo tenemos que volver a compilar todo el programa. Muchos

sistemas proporcionan herramientas especiales que recompilan sólo los archivos de código fuente que dependen del archivo modificado del programa. En los sistemas UNIX, esta herramienta se conoce como `make`. La herramienta `make` lee un archivo llamado `Makefile` que contiene instrucciones para compilar y vincular el programa. Los sistemas como Borland C++ y Microsoft Visual C++ para las PCs proporcionan herramientas `make` y “proyectos”. Para obtener más información acerca de las herramientas `make`, consulte el manual de su sistema específico.

E.6 Terminación de los programas con `exit` y `atexit`

La biblioteca de herramientas generales (`<cstdlib>`) proporciona métodos para terminar la ejecución de un programa, aparte de un retorno convencional de `main`. La función `exit` obliga a un programa a terminar como si se hubiera ejecutado en forma normal. La función se utiliza comúnmente para terminar un programa cuando se detecta un error, o si no se puede abrir un archivo que el programa debe procesar.

La función `atexit` registra una función en el programa, la cual se debe llamar cuando éste termine al llegar al final de `main`, o al invocar a `exit`. La función `atexit` recibe un apuntador a una función (es decir, el nombre de la función) como argumento. Las funciones que se llaman al momento de terminar el programa no pueden tener argumentos y no pueden devolver un valor.

La función `exit` recibe un argumento. Por lo general, ese argumento es la constante simbólica `EXIT_SUCCESS` o `EXIT_FAILURE`. Si se hace una llamada a `exit` con `EXIT_SUCCESS`, se devuelve el valor definido por la implementación para la terminación exitosa al entorno que hizo la llamada. Si se llama a `exit` con `EXIT_FAILURE`, se devuelve el valor definido por la implementación para la terminación no exitosa. Cuando se invoca a la función `exit`, cualquier función anteriormente registrada con `atexit` se invoca en el orden inverso de su registro, se vacían y cierran todos los flujos asociados con el programa y el control regresa al entorno anfitrión. En la figura E.4 se evalúan las funciones `exit` y `atexit`. El programa pide al usuario que determine si el programa se debe terminar con `exit`, o si debe llegar al final de `main`. Observe que la función `imprimir` se ejecuta al momento de terminar el programa en cada caso.

```

1 // Fig. E.4: figE_04.cpp
2 // Uso de las funciones exit y atexit
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::cin;
7
8 #include <cstdlib>
9 using std::atexit;
10 using std::exit;
11
12 void imprimir();
13
14 int main()
15 {
16     atexit( imprimir ); // registra la función imprimir
17
18     cout << "Escriba 1 para terminar el programa con la función exit"
19         << "\nEscriba 2 para terminar el programa en forma normal\n";
20
21     int respuesta;
22     cin >> respuesta;
23
24     // termina con exit si la respuesta es 1
25     if ( respuesta == 1 )
26     {
27         cout << "\nTerminando el programa con la función exit\n";
28         exit( EXIT_SUCCESS );
29     } // fin de if
30
31     cout << "\nTerminando el programa al llegar al final de main"
32         << endl;

```

Figura E.4 | Uso de las funciones `exit` y `atexit`. (Parte I de 2).

```

33
34     return 0;
35 } // fin de main
36
37 // muestra un mensaje antes de terminar
38 void imprimir()
39 {
40     cout << "Ejecutando la funcion imprimir al terminar el programa\n"
41         << "Programa terminado" << endl;
42 } // fin de la función imprimir

```

Escriba 1 para terminar el programa con la función exit
 Escriba 2 para terminar el programa en forma normal
 2

Terminando el programa al llegar al final de main
 Ejecutando la función imprimir al terminar el programa
 Programa terminado

Escriba 1 para terminar el programa con la función exit
 Escriba 2 para terminar el programa en forma normal
 1

Terminando el programa con la función exit
 Ejecutando la función imprimir al terminar el programa
 Programa terminado

Figura E.4 | Uso de las funciones `exit` y `atexit`. (Parte 2 de 2).

Al terminar un programa con la función `exit`, sólo se ejecutan los destructores para los objetos estáticos y globales en el programa. Al terminar con la función `abort`, se termina el programa sin ejecutar destructores.

E.7 Calificador de tipo `volatile`

El calificador de tipo `volatile` se aplica a una definición de una variable que se puede alterar desde el exterior del programa (es decir, la variable no está completamente bajo el control del programa). Por ende, el compilador no puede realizar optimizaciones (como agilizar la ejecución del programa, o reducir el consumo de memoria, por ejemplo) que dependan en “saber que el comportamiento de una variable está influenciado sólo por las actividades del programa que el compilador puede observar”.

E.8 Sufijos para constantes enteras y de punto flotante

C++ proporciona sufijos enteros y de punto flotante para especificar los tipos de las constantes enteras y de punto flotante. Los sufijos enteros son: `u` o `U` para un entero `unsigned`, `l` o `L` para un entero `long`, y `ul` o `UL` para un entero `unsigned long`. Las siguientes constantes son de tipo `unsigned`, `long` y `unsigned long`, respectivamente:

```

174u
8358L
28373ul

```

Si no se coloca un sufijo en una constante entera, su tipo es `int`; si la constante no se puede almacenar en un `int`, se almacena en un `long`.

Los sufijos de punto flotante son `f` o `F` para un `float`, y `l` o `L` para un `long double`. Las siguientes constantes son de tipo `long double` y `float`, respectivamente:

```

3.14159L
1.28f

```

Una constante de punto flotante que no tiene sufijo es de tipo `double`. Una constante con un sufijo inapropiado produce una advertencia o un error del compilador.

E.9 Manejo de señales

Un evento inesperado, o **señal**, puede terminar un programa en forma prematura. Dichos eventos incluyen las **interrupciones** (oprimir **<Ctrl> C** en un sistema UNIX, LINUX, Mac OS X o Windows), **instrucciones ilegales**, **violaciones de segmentación**, **órdenes de terminación del sistema operativo** y **excepciones de punto flotante** (división entre cero o multiplicación de valores grandes de punto flotante). La **biblioteca de manejo de señales** proporciona la función **signal** para **atrapar eventos inesperados**. La función **signal** recibe dos argumentos: un número de señal entero y un apuntador a una función de manejo de señales. Las señales se pueden generar mediante la función **raise**, que recibe un número de señal entero como argumento. En la figura E.5 se sintetizan las señales estándar definidas en el archivo de encabezado **<csignal>**. El siguiente ejemplo demuestra las funciones **signal** y **raise**.

En la figura E.6 se atrapa una señal interactiva (SIGINT) con la función **signal**. El programa llama a **signal** con **SIGINT** y un apuntador a la función **manejadorSeniales**. (Recuerde que el nombre de una función es un apuntador a esa función). Cuando ocurre una señal de tipo SIGINT, se hace una llamada a la función **manejadorSeniales**, se imprime un mensaje y el usuario recibe la opción de continuar la ejecución normal del programa. Si el usuario desea continuar la ejecución, el manejador de señales se reinicializa mediante una llamada a **signal** de nuevo (algunos sistemas requieren que el manejador de señales se reinicialice) y el control regresa al punto en el programa en el que se detectó la señal. En este programa, la función **raise** se utiliza para simular una señal interactiva. Se elige un número aleatorio entre 1 y 50. Si el número es 25, entonces se hace una llamada a **raise** para generar la señal. Por lo general, las señales interactivas se inician fuera del programa. Por ejemplo, al oprimir **<Ctrl> C** durante la ejecución de un programa en un sistema UNIX, LINUX, Mac OS X o Windows, se genera una señal interactiva que termina la ejecución del programa. El manejo de señales se puede utilizar para atrapar la señal interactiva y evitar que el programa termine.

Señal	Explicación
SIGABRT	Terminación anormal del programa (como una llamada a abort).
SIGFPE	Una operación aritmética errónea, como una división entre cero o una operación que resulta en un desbordamiento.
SIGILL	Detección de una instrucción ilegal.
SIGINT	Recepción de una señal de atención interactiva.
SIGSEGV	Un acceso inválido al espacio de almacenamiento.
SIGTERM	Una petición de terminación se envió al programa.

Figura E.5 | Señales definidas en el encabezado **<csignal>**.

```

1 // Fig. E.6: figE_06.cpp
2 // Uso del manejo de señales
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 #include <csignal>
12 using std::raise;
13 using std::signal;
14
15 #include <cstdlib>
16 using std::exit;
17 using std::rand;
18 using std::srand;
```

Figura E.6 | Uso del manejo de señales. (Parte 1 de 3).

```
19 #include <ctime>
20 using std::time;
21
22 void manejadorSeniales( int );
23
24 int main()
25 {
26     signal( SIGINT, manejadorSeniales );
27     srand( time( 0 ) );
28
29     // crea e imprime números aleatorios
30     for ( int i = 1; i <= 100; i++ )
31     {
32         int x = 1 + rand() % 50;
33
34         if ( x == 25 )
35             raise( SIGINT ); // genera SIGINT cuando x es 25
36
37         cout << setw( 4 ) << i;
38
39         if ( i % 10 == 0 )
40             cout << endl; // imprime endl cuando i es un múltiplo de 10
41     } // fin de for
42
43
44     return 0;
45 } // fin de main
46
47 // maneja la señal
48 void manejadorSeniales( int valorSenial )
49 {
50     cout << "\nSe recibio una senial (" << valorSenial
51     << ") de interrupcion.\n"
52     << "Desea continuar (1 = si o 2 = no)? ";
53
54     int respuesta;
55
56     cin >> respuesta;
57
58     // comprueba las respuestas inválidas
59     while ( respuesta != 1 && respuesta != 2 )
60     {
61         cout << "(1 = si o 2 = no)? ";
62         cin >> respuesta;
63     } // fin de while
64
65     // determina si es tiempo de salir
66     if ( respuesta != 1 )
67         exit( EXIT_SUCCESS );
68
69     // llama a signal y le pasa SIGINT y la dirección de manejadorSeniales
70     signal( SIGINT, manejadorSeniales );
71 } // fin de la función manejadorSeniales
```

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
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70

Figura E.6 | Uso del manejo de señales. (Parte 2 de 3).

```

71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99
Se recibio una senial (2) de interrupcion.
Desea continuar (1 = si o 2 = no)? 1
100

```

```

1 2 3 4
Se recibio una senial (2) de interrupcion.
Desea continuar (1 = si o 2 = no)? 2

```

Figura E.6 | Uso del manejo de señales. (Parte 3 de 3).

E.10 Asignación dinámica de memoria con `calloc` y `realloc`

En el capítulo 10 hablamos sobre la asignación dinámica de memoria estilo C++ con `new` y `delete`. Los programadores de C++ deben usar `new` y `delete` en vez de las funciones `malloc` y `free` de C (encabezado `<cstdlib>`). Sin embargo, la mayoría de los programadores de C++ terminarán leyendo en algún momento dado una gran cantidad de código heredado de C, y por lo tanto incluimos esta discusión adicional acerca de la asignación dinámica de memoria estilo C.

La biblioteca de herramientas generales (`<cstdlib>`) proporciona otras dos funciones para la asignación dinámica de memoria: `calloc` y `realloc`. Estas funciones se pueden utilizar para crear y modificar **arreglos dinámicos**. Como se muestra en el capítulo 8, Apunadores y cadenas basadas en apunadores, un apuntador a un arreglo puede utilizar subíndices de igual forma que un arreglo. Por ende, un apuntador a una porción contigua de memoria creada por `calloc` se puede manipular como un arreglo. La función `calloc` asigna en forma dinámica la memoria para un arreglo e inicializa esa memoria con ceros. El prototipo para `calloc` es

```
void *calloc( size_t nmiemb, size_t tamanio );
```

Esta función recibe dos argumentos: el número de elementos (`nmiemb`) y el tamaño de cada elemento (`tamanio`), e inicializa los elementos del arreglo con cero. La función devuelve un apuntador a la memoria asignada o un apuntador nulo (0) si la memoria no se asigna.

La función `realloc` modifica el tamaño de un objeto asignado por una llamada anterior a `malloc`, `calloc` o `realloc`. El contenido del objeto original no se modifica, siempre y cuando la memoria asignada sea mayor que la cantidad asignada con anterioridad. En caso contrario, el contenido queda sin modificar, hasta llegar al tamaño del nuevo objeto. El prototipo para `realloc` es

```
void *realloc( void *ptr, size_t tamanio );
```

La función `realloc` recibe dos argumentos: un apuntador al objeto original (`ptr`) y el nuevo tamaño del objeto (`tamanio`). Si `ptr` es 0, `realloc` funciona de manera idéntica a `malloc`. Si `tamanio` es 0 y `ptr` no es 0, se libera la memoria para el objeto. En cualquier otro caso, si `ptr` no es 0 y `tamanio` es mayor que cero, `realloc` trata de asignar un nuevo bloque de memoria. Si el nuevo espacio no se puede asignar, el objeto al que apunta `ptr` queda sin modificar. La función `realloc` devuelve un apuntador a la memoria reasignada o un apuntador nulo.

Error común de programación E.2



Se pueden producir errores en tiempo de ejecución si utilizamos el operador `delete` en un apuntador que resulte de `malloc`, `calloc` o `realloc`, o si utilizamos `realloc` o `free` en un apuntador que resulte del nuevo operador.

E.11 Bifurcación incondicional: `goto`

A lo largo de este libro hemos hecho énfasis en la importancia de utilizar técnicas de programación estructurada para crear software confiable que sea fácil de depurar, mantener y modificar. En algunos casos, el rendimiento es más importante que adherirse estrictamente a las técnicas de programación estructurada. En estos casos, pueden utilizarse algunas técnicas de programación no estructurada. Por ejemplo, podemos usar `break` para terminar la ejecución de una estructura de repetición antes de que la condición de continuación de ciclo se vuelva falsa. Esto ahorra repeticiones innecesarias del ciclo, si la tarea se completa antes de que termine.

Otra instancia de la programación estructurada es la **instrucción `goto`**; una bifurcación incondicional. El resultado de la instrucción `goto` es una modificación en el flujo de control del programa a la primera instrucción después de la

etiqueta especificada en la instrucción `goto`. Una etiqueta es un identificador seguido de dos puntos. Debe aparecer una etiqueta en la misma función que la instrucción `goto` que hace referencia a ella. En la figura E.7 se utilizan instrucciones `goto` para iterar 10 veces e imprimir el valor del contador en cada iteración. Después de inicializar `cuenta` con 1, el programa evalúa `cuenta` para determinar si es mayor que 10. (Se omite la etiqueta `inicio`, ya que las etiquetas no realizan ninguna acción). De ser así, el control se transfiere de la instrucción `goto` a la primera instrucción después de la etiqueta `fin`. En caso contrario, se imprime `cuenta` y se incrementa, y el control se transfiere de la instrucción `goto` a la primera instrucción después de la etiqueta `inicio`.

```

1 // Fig. E.7: figE_07.cpp
2 // Uso de goto.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::left;
9 using std::setw;
10
11 int main()
12 {
13     int cuenta = 1;
14
15     inicio: // etiqueta
16         // va hasta fin cuando cuenta excede a 10
17         if ( cuenta > 10 )
18             goto fin;
19
20         cout << setw( 2 ) << left << cuenta;
21         ++cuenta;
22
23         // va hasta inicio en la línea 17
24         goto inicio;
25
26     fin: // etiqueta
27         cout << endl;
28
29     return 0;
30 } // fin de main

```

1 2 3 4 5 6 7 8 9 10

Figura E.7 | Uso de `goto`.

En los capítulos 4 y 5 vimos que sólo se requieren tres estructuras de control para escribir cualquier programa: secuencia, selección y repetición. Cuando se siguen las reglas de la programación estructurada, es posible crear estructuras de control con muchos niveles de anidamiento, de las cuales es difícil escapar con eficiencia. Algunos programadores utilizan instrucciones `goto` en tales situaciones, para salir con rapidez de una estructura con muchos niveles de anidamiento. Esto elimina la necesidad de probar las múltiples condiciones para escapar de una estructura de control.



Tip de rendimiento E.2

La instrucción `goto` se puede utilizar para salir de las estructuras de control anidadas con eficiencia, pero puede hacer que el código sea más difícil de leer y de mantener. Su uso no se recomienda en definitiva.



Tip para prevenir errores E.1

La instrucción `goto` se debe utilizar sólo en aplicaciones orientadas al rendimiento. La instrucción `goto` no es estructurada y puede conllevar a programas que sean más difíciles de depurar, mantener, modificar y comprender.

E.12 Uniones

Una **unión** (que se define mediante la palabra clave **union**) es una región de memoria que, con el tiempo, puede contener objetos de una variedad de tipos. Sin embargo, en cualquier momento una **union** puede contener un máximo de un objeto, debido a que los miembros de una **union** comparten el mismo espacio de almacenamiento. Es responsabilidad del programador asegurar que se haga referencia a los datos de una **union** con el nombre de un miembro del tipo de datos apropiado.



Error común de programación E.3

El resultado de hacer referencia al miembro de una union distinto al que se almacenó por última vez es indefinido. Trata los datos almacenados como un tipo distinto.



Tip de portabilidad E.2

Si los datos se almacenan en una union como de un tipo y se referencian como de otro tipo, los resultados son dependientes de la implementación.

En distintos momentos durante la ejecución de un programa, algunos objetos podrían ser irrelevantes, mientras que otro objeto podría ser relevante; por lo tanto, una **union** comparte el espacio en vez de desperdiciar almacenamiento en objetos que no se están utilizando. El número de bytes utilizados para almacenar una **union** será por lo menos suficiente como para contener al miembro más grande.



Tip de rendimiento E.3

El uso de uniones ayuda a conservar espacio de almacenamiento.



Tip de portabilidad E.3

La cantidad de espacio requerido para almacenar una union es dependiente de la implementación.

Una **union** se declara en el mismo formato que una estructura (**struct**) o una clase (**class**). Por ejemplo,

```
union Numero
{
    int x;
    double y;
};
```

indica que **Numero** es un tipo de **union** con los miembros **int x** y **double y**. La definición de la **union** debe ir antes de todas las funciones en las que se vaya a utilizar.



Observación de Ingeniería de Software E.4

Al igual que una declaración struct o class, una declaración union simplemente crea un nuevo tipo. Al colocar una declaración union o struct fuera de cualquier función no se crea realmente una variable global.

Las únicas operaciones integradas válidas que se pueden realizar en una **union** son: asignar una **union** a otra **union** del mismo tipo, tomar la dirección (&) de una **union** y acceder a los miembros de la misma usando el operador miembro de estructura (.) y el operador apuntador a estructura (->). Las uniones no se pueden comparar.



Error común de programación E.4

Al comparar uniones se produce un error de compilación, ya que el compilador no sabe qué miembro de cada union está activo, y por ende cuál miembro de una union debe comparar con cuál miembro de la otra.

Una **union** es similar a una clase, en cuanto a que puede tener un constructor para inicializar cualquiera de sus miembros. Una **union** que no tiene constructor se puede inicializar con otra **union** del mismo tipo, con una expresión del tipo del primer miembro de la **union**, o con un inicializador (encerrado entre llaves) del tipo del primer miembro de la **union**. Las uniones pueden tener otras funciones miembro (como los destructores), pero las funciones miembro de una **union** no pueden declararse como **virtual**. Los miembros de una **union** son **public** de manera predeterminada.



Error común de programación E.5

Es un error de compilación inicializar una union en una declaración con un valor o una expresión cuyo tipo sea distinto del tipo del primer miembro de la union.

Una **union** no se puede utilizar como clase base en la herencia (es decir, las clases no se pueden derivar de uniones). Las uniones pueden tener objetos como miembros, sólo si estos objetos no tienen un constructor, un destructor o un operador de asignación sobrecargado. Ninguno de los miembros de datos de una **union** se pueden declarar como **static**.

En la figura E.8 se utiliza la variable **valor** de tipo **union Numero** para mostrar el valor almacenado en la **union**, como **int** y como **double**. La salida del programa es dependiente de la implementación, y muestra que la representación interna de un valor **double** puede ser bastante distinta de la representación de un **int**.

Una **union anónima** es una **union** sin un nombre de tipo que no intenta definir objetos o apuntadores antes del signo de punto y coma con que termina. Dicha **union** no crea un tipo, pero crea un objeto sin nombre. Los miembros de una **union anónima** se pueden utilizar directamente en el alcance en el que se declara esa **union anónima**, de igual forma que cualquier otra variable local; no hay necesidad de utilizar los operadores punto (.) o flecha (->).

Las uniones anónimas tienen ciertas restricciones. Sólo pueden contener miembros de datos. Todos los miembros de una **union anónima** deben ser **public**. Y una **union anónima** que se declara como **global** (es decir, en alcance de archivo) debe declararse explícitamente como **static**. En la figura E.9 se ilustra el uso de una **union anónima**.

```

1 // Fig. E.8: figE_08.cpp
2 // Un ejemplo de una unión.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // define la unión Numero
8 union Numero
9 {
10     int entero1;
11     double double1;
12 }; // fin de la unión Numero
13
14 int main()
15 {
16     Numero valor; // variable tipo unión
17
18     valor.entero1 = 100; // asigna 100 al miembro entero1
19
20     cout << "Coloca un valor en el miembro entero\n"
21         << "e imprime ambos miembros.\nint: "
22         << valor.entero1 << "\ndouble: " << valor.double1
23         << endl;
24
25     valor.double1 = 100.0; // asigna 100.0 al miembro double1
26
27     cout << "Coloca un valor en el miembro de punto flotante\n"
28         << "e imprime ambos miembros.\nint: "
29         << valor.entero1 << "\ndouble: " << valor.double1
30         << endl;
31
32     return 0;
33 } // fin de main

```

```

Coloca un valor en el miembro entero
e imprime ambos miembros.
int: 100
double: -9.25596e+061
Coloca un valor en el miembro de punto flotante
e imprime ambos miembros.
int: 0
double: 100

```

Figura E.8 | Cómo imprimir el valor de una unión en ambos tipos de datos miembro.

```

1 // Fig. E.9: figE_09.cpp
2 // Uso de una unión anónima.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     // declara una unión anónima
10    // los miembros entero1, double1 y charPtr comparten el mismo espacio
11    union
12    {
13        int entero1;
14        double double1;
15        char *charPtr;
16    }; // fin de la unión anónima
17
18    // declara las variables locales
19    int entero2 = 1;
20    double double2 = 3.3;
21    char *char2Ptr = "Union";
22
23    // asigna un valor a cada miembro de la unión
24    // en forma sucesiva, e imprime cada uno
25    cout << entero2 << ' ';
26    entero1 = 2;
27    cout << entero1 << endl;
28
29    cout << double2 << ' ';
30    double1 = 4.4;
31    cout << double1 << endl;
32
33    cout << char2Ptr << ' ';
34    charPtr = "anónima";
35    cout << charPtr << endl;
36
37    return 0;
38 } // fin de main

```

```

1 2
3.3 4.4
Union anónima

```

Figura E.9 Uso de una unión anónima.

E.13 Especificaciones de vinculación

Es posible llamar desde un programa de C++ a las funciones escritas y compiladas con un compilador de C. Como vimos en la sección 6.17, C++ codifica de forma especial los nombres de las funciones para la vinculación con seguridad de tipos. Sin embargo, C no codifica los nombres de sus funciones. Por ende, una función compilada en C no se reconocerá cuando se haga un intento por vincular código de C con código de C++, debido a que el código de C++ espera un nombre de función codificado en forma especial. C++ nos permite proporcionar **especificaciones de vinculación** para informar al compilador que una función se compiló en un compilador de C, y para evitar que el compilador de C++ codifique el nombre de esa función. Las especificaciones de vinculación son útiles cuando se han desarrollado bibliotecas extensas de funciones especializadas, y el usuario no tiene acceso al código fuente para volver a compilarlo en C++, o no tiene tiempo de convertir las funciones de biblioteca de C a C++.

Para informar al compilador que se han compilado una o varias funciones en C, debemos escribir los prototipos de las funciones de la siguiente manera:

```

extern "C" prototipo de función // una sola función
extern "C" // varias funciones

```

```
{
    prototipos de funciones
}
```

Estas declaraciones informan al compilador que las funciones especificadas no están compiladas en C++, por lo que no se debe realizar la codificación de los nombres en las funciones listadas en la especificación de vinculación. Así, estas funciones se pueden vincular en forma apropiada con el programa. Por lo general, los entornos de C++ incluyen las bibliotecas C estándar y no requieren que el programador utilice especificaciones de vinculación para esas funciones.

E.14 Repaso

En este apéndice presentamos varios temas acerca del código heredado de C. Hablamos sobre cómo redirigir la entrada del teclado para que provenga de un archivo, y cómo redirigir la salida a la pantalla para que vaya a un archivo. También presentamos las listas de argumentos de longitud variable, los argumentos de la línea de comandos y el procesamiento de eventos inesperados. El lector aprendió también acerca de cómo asignar y cambiar el tamaño de la memoria en forma dinámica. En el siguiente apéndice aprenderá acerca del preprocesador para incluir otros archivos, definir constantes simbólicas y macros.

Resumen

Sección E.2 Redirección de la entrada/salida en sistemas UNIX/LINUX/Mac OS X y Windows

- En muchos sistemas (en especial, en los sistemas UNIX, LINUX, Mac OS X o Windows) es posible redirigir la entrada a un programa y la salida de un programa. La entrada se redirige desde las líneas de comandos de UNIX, LINUX, Mac OS X o Windows mediante el símbolo de redirección de entrada (<) o una canalización (|). La salida se redirige desde las líneas de comandos de UNIX, LINUX, Mac OS X o Windows mediante el símbolo de redirección de salida (>) o el símbolo para agregar salida (>>). El símbolo de redirección de salida simplemente almacena la salida del programa en un archivo, y el símbolo para agregar salida la agrega al final de un archivo.

Sección E.3 Listas de argumentos de longitud variable

- Las macros y definiciones del encabezado de argumentos variables <cstdarg> proporcionan las herramientas necesarias para crear funciones con listas de argumentos de longitud variable.
- Los tres puntos suspensivos (...) en un prototipo de función indican que esa función recibe un número variable de argumentos.
- El tipo va_list es adecuado para contener la información que necesitan las macros va_start, va_arg y va_end. Para acceder a los argumentos en una lista de argumentos de longitud variable, se debe declarar un objeto de tipo va_list.
- La macro va_start se invoca antes de que se pueda acceder a los argumentos de una lista de argumentos de longitud variable. La macro inicializa el objeto declarado con va_list para que sea utilizado por las macros va_arg y va_end.
- La macro va_arg se expande en una expresión del valor y tipo del siguiente argumento en la lista de argumentos de longitud variable. Cada invocación de va_arg modifica al objeto va_list, de manera que éste apunte al siguiente argumento en la lista.
- La macro va_end facilita un retorno normal de una función cuya lista de argumentos variable haya sido referenciada por la macro va_start.

Sección E.4 Uso de los argumentos de la línea de comandos

- En muchos sistemas (en especial, en UNIX, LINUX, Mac OS X y Windows) es posible pasar argumentos de línea de comandos a main, para lo cual se incluyen en la lista de parámetros de main los parámetros int argc y char *argv[]. El parámetro argc es el número de argumentos de la línea de comandos. El parámetro argv es un arreglo de apuntadores char * que contienen los argumentos de la línea de comandos.

Sección E.5 Observaciones acerca de la compilación de programas con varios archivos de código fuente

- La definición de una función debe estar contenida completamente en un solo archivo; no puede abarcar dos o más archivos.
- Las variables globales se deben declarar en cada archivo en el que se vayan a utilizar.
- Los prototipos de funciones se pueden utilizar para declarar funciones en otros archivos del programa. (El especificador extern no es obligatorio en un prototipo de función). Para ello hay que incluir el prototipo de la función en cada archivo en el que se vaya a invocar, y después se vinculan los archivos en conjunto.

- Cuando se aplica el especificador de clase de almacenamiento `static` a una variable o función con alcance de archivo, evita que sea utilizada por cualquier función que no esté definida en el mismo archivo. Esto se conoce como vinculación interna. Las variables y funciones globales a las que no se les antepone la palabra clave `static` en sus definiciones tienen vinculación externa; se pueden utilizar en otros archivos, si éstos contienen declaraciones y/o prototipos de funciones apropiados.
- El especificador `static` se utiliza comúnmente con las funciones utilitarias que son llamadas sólo por las funciones en un archivo específico. Si una función no es requerida fuera de un archivo específico, debe hacerse valer el principio de menor privilegio mediante el uso de `static`.
- Al crear programas extensos a partir de varios archivos de código fuente, el proceso de compilar el programa se vuelve tedioso si al realizar pequeñas modificaciones en un archivo tenemos que volver a compilar todo el programa. Muchos sistemas proporcionan herramientas especiales que recompilan sólo el archivo del programa que se modificó. En los sistemas UNIX, esta herramienta se conoce como `make`. Esta herramienta lee un archivo llamado `Makefile` que contiene instrucciones para compilar y vincular el programa.

Sección E.6 Terminación de los programas con `exit` y `atexit`

- La función `exit` obliga a que un programa termine como si se hubiera ejecutado en forma normal.
- La función `atexit` registra una función que debe llamarse al momento en que el programa termine en forma normal; es decir, cuando el programa termine al llegar al final de `main`, o cuando se invoque a `exit`.
- La función `atexit` recibe un apuntador a una función (por ejemplo, un nombre de función) como argumento. Las funciones que se llaman al terminar el programa no pueden tener argumentos y no pueden devolver un valor.
- La función `exit` recibe un argumento, por lo general, la constante simbólica `EXIT_SUCCESS` o `EXIT_FAILURE`. Si se llama a `exit` con `EXIT_SUCCESS`, el valor definido por la implementación para la terminación exitosa se devuelve al entorno que hizo la llamada. Si se llama `exit` con `EXIT_FAILURE`, se devuelve el valor definido por la implementación para la terminación no exitosa.
- Cuando se invoca a la función `exit`, cualquier función registrada con `atexit` se invoca en el orden inverso en que se registró, se vacían y cierran todos los flujos asociados con el programa, y el control regresa al entorno anfitrión.

Sección E.7 Calificador de tipo `volatile`

- El calificador `volatile` se utiliza para evitar las optimizaciones de una variable, cuando ésta puede llegar a ser modificada desde el exterior del alcance del programa.

Sección E.8 Sufijos para las constantes enteras y de punto flotante

- C++ proporciona sufijos enteros y de punto flotante para especificar los tipos de constantes enteras y de punto flotante. Los sufijos enteros son `u` o `U` para un entero `unsigned`, `l` o `L` para un entero `long` y `ul` o `UL` para un entero `unsigned long`. Si una constante entera no tiene sufijo, su tipo se determina con base en el primer tipo capaz de almacenar un valor de ese tamaño (primero `int`, después `long int`). Los sufijos de punto flotante son `f` o `F` para un `float` y `l` o `L` para un `long double`. Una constante de punto flotante que no tiene sufijo es de tipo `double`.

Sección E.9 Manejo de señales

- La biblioteca de manejo de señales proporciona la capacidad de registrar una función para atrapar eventos inesperados con la función `signal`. Esta función recibe dos argumentos: un número de señal entero y un apuntador a la función de manejo de señales.
- Las señales también se pueden generar mediante la función `raise` y un argumento entero.

Sección E.10 Asignación dinámica de memoria con `calloc` y `realloc`

- La biblioteca de herramientas generales (`cstdlib`) proporciona las funciones `calloc` y `realloc` para la asignación dinámica de memoria. Estas funciones se pueden utilizar para crear arreglos dinámicos.
- La función `calloc` recibe dos argumentos: el número de elementos (`nmembr`) y el tamaño de cada elemento (`tamano`); además inicializa los elementos del arreglo con cero. La función devuelve un apuntador a la memoria asignada o, si la memoria no se asigna, devuelve un apuntador nulo.
- La función `realloc` modifica el tamaño de un objeto asignado por una llamada anterior a `malloc`, `calloc` o `realloc`. El contenido del objeto original no se modifica, siempre y cuando la cantidad de memoria asignada sea mayor que la cantidad previamente asignada.
- La función `realloc` recibe dos argumentos: un apuntador al objeto original (`ptr`) y el nuevo tamaño del objeto (`tamano`). Si `ptr` es nulo, `realloc` funciona en forma idéntica a `malloc`. Si `tamano` es 0 y el apuntador recibido no es nulo, se libera la memoria para el objeto. En caso contrario, si `ptr` no es nulo y `tamano` es mayor que cero, `realloc` trata de asignar un nuevo bloque de memoria para el objeto. Si el nuevo espacio no se puede asignar, el objeto al que apunta `ptr` queda sin modificación. La función `realloc` devuelve un apuntador a la memoria reasignada o a un apuntador nulo.

Sección E.11 Bifurcación incondicional: **goto**

- El resultado de la instrucción **goto** es un cambio en el control del flujo del programa. La ejecución del programa continúa en el primer elemento después de la etiqueta en la instrucción **goto**.
- Una etiqueta es un identificador seguido por un signo de dos puntos. Una etiqueta debe aparecer en la misma función que la instrucción **goto** a la que hace referencia.

Sección E.12 Uniones

- Una **union** es un tipo de datos cuyos miembros comparten el mismo espacio de almacenamiento. Los miembros pueden ser de casi cualquier tipo. El almacenamiento reservado para una **union** es lo bastante grande como para almacenar su miembro más grande. En la mayoría de los casos, las uniones contienen dos o más tipos de datos. Sólo un miembro (y por ende, un tipo de datos) se puede referenciar a la vez.
- Una **union** se declara en el mismo formato que una estructura.
- Una **union** se puede inicializar sólo con un valor del tipo de su primer miembro u otro objeto del mismo tipo de unión.

Sección E.13 Especificaciones de vinculación

- C++ nos permite proporcionar especificaciones de vinculación para informar al compilador que una función se compiló en un compilador de C, y para evitar que el compilador de C++ codifique el nombre de la función.
- Para informar al compilador que se han compilado una o varias funciones en C, debemos escribir los prototipos de la función como se muestra a continuación:

```
extern "C" prototipo de función // una sola función
extern "C"      // varias funciones
{
    prototipos de funciones
}
```

Estas declaraciones informan al compilador que las funciones especificadas no están compiladas en C++, por lo que la codificación de nombres no se debe realizar en las funciones listadas en la especificación de vinculación. Así, estas funciones pueden vincularse con el programa.

- Por lo general, los entornos de C++ incluyen las bibliotecas estándar de C y el programador no tiene que utilizar especificaciones de vinculación para esas funciones.

Terminología

>>, símbolo para agregar entrada	instrucción ilegal
, canalización	interrupción
<, símbolo de redirección de entrada	lista de argumentos de longitud variable
<csignal>	long double, sufijo (l o L)
<cstdarg>	long integer, sufijo (l o L)
>, símbolo de redirección de salida	make
argumentos de la línea de comandos	Makefile
argv	malloc
arreglos dinámicos	raise
atexit	realloc
atrapar	redirección de E/S
biblioteca de manejo de señales	signal
calloc	static, especificador de clase de almacenamiento
canalizaciones	union
const	unsigned long, sufijo entero (ul o UL)
evento	unsigned, sufijo entero (u o U)
excepción de punto flotante	va_arg
exit	va_end
EXIT_FAILURE	va_list
EXIT_SUCCESS	va_start
extern "C"	vinculación externa
extern, especificador de clase de almacenamiento	vinculación interna
float, sufijo (f o F)	violación de segmentación
free	volatile
goto, instrucción	

Ejercicios de autoevaluación

E.1 Complete los siguientes enunciados:

- a) El símbolo _____ redirige los datos de entrada del teclado para que provengan de un archivo.
- b) El símbolo _____ se utiliza para redirigir la salida a la pantalla para que vaya a un archivo.
- c) El símbolo _____ se utiliza para agregar la salida de un programa al final de un archivo.
- d) Un(a) _____ se utiliza para dirigir la salida de un programa como entrada para otro programa.
- e) Los _____ en la lista de parámetros de una función indican que esa función puede recibir un número variable de argumentos.
- f) La macro _____ se debe invocar antes de que se pueda acceder a los argumentos en una lista de argumentos de longitud variable.
- g) La macro _____ se utiliza para acceder a los argumentos individuales de una lista de argumentos de longitud variable.
- h) La macro _____ realiza tareas de mantenimiento de terminación en una función cuya lista de argumentos variables fue referenciada por la macro `va_start`.
- i) El argumento _____ de `main` recibe el número de argumentos en una línea de comandos.
- j) El argumento _____ de `main` almacena los argumentos de la línea de comandos como cadenas de caracteres.
- k) La herramienta _____ de UNIX lee un archivo llamado _____, el cual contiene instrucciones para compilar y vincular un programa que consiste de varios archivos de código fuente. Esta herramienta recompila un archivo sólo si este archivo (o un encabezado que utilice) se ha modificado desde la última vez que se compiló.
- l) La función _____ obliga a un programa a terminar su ejecución.
- m) La función _____ registra una función que debe llamarse al momento en que el programa termine en forma normal.
- n) Un _____ entero o de punto flotante se puede agregar a una constante entera o de punto flotante para especificar el tipo exacto de esa constante.
- o) La función _____ se puede utilizar para registrar una función y atrapar eventos inesperados.
- p) La función _____ genera una señal desde el interior de un programa.
- q) La función _____ asigna memoria en forma dinámica para un arreglo, e inicializa los elementos con cero.
- r) La función _____ modifica el tamaño de un bloque de memoria asignada en forma dinámica.
- s) Un(a) _____ es una entidad que contiene una colección de variables que ocupan la misma memoria, pero en distintos momentos.
- t) La palabra clave _____ se utiliza para introducir la definición de una unión.

Respuestas a los ejercicios de autoevaluación

E.1 a) de redirección de entrada (<). b) de redirección de salida (>). c) para agregar salida (>>). d) canalización (|). e) tres puntos suspensivos (...). f) `va_start`. g) `va_arg`. h) `va_end`. i) `argc`. j) `argv`. k) `make`, `Makefile`. l) `exit`. m) `atexit`. n) sufijo. o) `signal`. p) `raise`. q) `calloc`. r) `realloc`. s) unión. t) `union`.

Ejercicios

E.2 Escriba un programa que calcule el producto de una serie de enteros que se pasen a la función `producto`, usando una lista de argumentos de longitud variable. Pruebe su función con varias llamadas, cada una con un número distinto de argumentos.

E.3 Escriba un programa que imprima los argumentos de la línea de comandos del programa.

E.4 Escriba un programa que ordene un arreglo entero en forma ascendente o descendente. El programa debe utilizar argumentos de la línea de comandos para pasar el argumento -a para indicar un orden ascendente, o -d para indicar un orden descendente. [Nota: éste es el formato estándar para pasar opciones a un programa en UNIX].

E.5 Lea los manuales de su sistema para determinar qué señales soporta la biblioteca de manejo de señales (<csignal>). Escriba un programa con manejadores de señales para las señales `SIGABRT` y `SIGINT`. El programa debe evaluar si se atrapan o no estas señales, llamando a la función `abort` para generar una señal de tipo `SIGABRT` y oprimiendo `<Ctrl-C>` para generar una señal de tipo `SIGINT`.

E.6 Escriba un programa que asigne en forma dinámica un arreglo de enteros, usando una función de <cstdlib>, no el operador `new`. El tamaño del arreglo debe introducirse mediante el teclado. A los elementos del arreglo se les deben asignar valores introducidos mediante el teclado. Imprima los valores del arreglo. A continuación, reasigne la memoria para el arreglo

a la mitad del número actual de elementos. Imprima los valores restantes en el arreglo para confirmar que coincidan con la primera mitad de los valores en el arreglo original.

E.7 Escriba un programa que reciba dos nombres de archivos como argumentos de línea de comandos, que lea los caracteres del primer archivo, uno a la vez, y escriba los caracteres en orden inverso en el segundo archivo.

E.8 Escriba un programa que utilice instrucciones `goto` para simular una estructura de iteración anidada que imprima un cuadro de asteriscos, como se muestra en la figura E.10. El programa deberá utilizar sólo las siguientes tres instrucciones de salida:

```
cout << '*';
cout << ' ';
cout << endl;
```

E.9 Proporcione la definición de `union Datos` que contenga `char caracter1`, `short short1`, `long long1`, `float float1` y `double double1`.

E.10 Cree la `union Entero` con los miembros `char c`, `short s`, `int i` y `long l`. Escriba un programa que reciba como entrada valores de tipo `char`, `short`, `int` y `long`, y que almacene los valores en variables `union` de tipo `union Entero`. Cada variable `union` debe imprimirse como un valor `char`, un `short`, un `int` y un `long`. ¿Se imprimen los valores siempre en forma correcta?

E.11 Cree la `union PuntoFlotante` con los miembros `float float1`, `double double1` y `long double longDouble`. Escriba un programa que reciba como entrada valores de tipo `float`, `double` y `long double` y que almacene esos valores en variables `union` de tipo `union PuntoFlotante`. Cada variable `union` debe imprimirse como un valor `float`, un `double` y un `long double`. ¿Se imprimen los valores siempre en forma correcta?

E.12 Dada la `union`

```
union A
{
    double y;
    char *zPtr;
};
```

¿cuáles de las siguientes instrucciones son correctas para inicializar la `union`?

- `A p = b;` // `b` es de tipo `A`
- `A q = x;` // `x` es `double`
- `A r = 3.14159;`
- `A s = { 79.63 };`
- `A t = { "Hola a todos" };`
- `A u = { 3.14159, "Pi" };`
- `A v = { y = -7.843, zPtr = &x };`

```
*****
*   *
*   *
*   *
*****
```

Figura E.10 | Ejemplo para el ejercicio E.8.

F



Preprocesador

Sostén el bien; defínelo en forma adecuada.

—Alfred, Lord Tennyson

Le he encontrado un argumento; pero no estoy obligado a encontrarle una comprensión.

—Samuel Johnson

Un buen símbolo es el mejor argumento, y un misionero para persuadir a miles.

—Ralph Waldo Emerson

Las condiciones son fundamentalmente sólidas.

—Herbert Hoover [Diciembre 1929]

El partisano, cuando está involucrado en una disputa, no se preocupa por los derechos de la cuestión, sino que está ansioso sólo por convencer a sus oyentes de sus propias aserciones.

—Platón

OBJETIVOS

En este apéndice aprenderá a:

- Utilizar `#include` para desarrollar programas extensos.
- Utilizar `#define` para crear macros y macros con argumentos.
- Comprender la compilación condicional.
- Mostrar mensajes de error durante la compilación condicional.
- Utilizar aserciones para probar si los valores de las expresiones son correctos.

- F.1 Introducción
- F.2 La directiva del preprocesador #include
- F.3 La directiva del preprocesador #define: constantes simbólicas
- F.4 La directiva del preprocesador #define: macros
- F.5 Compilación condicional
- F.6 Las directivas del preprocesador #error y #pragma
- F.7 Los operadores # y ##
- F.8 Constantes simbólicas predefinidas
- F.9 Aserciones
- F.10 Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#) | [Ejercicios](#)

F.1 Introducción

En este capítulo presentamos el **preprocesador**. El preprocesamiento ocurre antes de compilar un programa. Algunas acciones posibles son la inclusión de otros archivos en el archivo que se va a compilar, la definición de **constantes simbólicas** y **macros**, la **compilación condicional** de código del programa y la **ejecución condicional de directivas del preprocesador**. Todas las directivas del preprocesador empiezan con **#**, y sólo pueden aparecer espacios en blanco antes de una directiva del preprocesador en una línea. Las directivas del preprocesador no son instrucciones de C++, por lo que no terminan con punto y coma (**;**). Las directivas del preprocesador se procesan por completo antes de que empiece la compilación.

Error común de programación F.1



Colocar un punto y coma al final de una directiva del preprocesador puede producir una variedad de errores, dependiendo del tipo de directiva del preprocesador.

Observación de Ingeniería de Software F.1



Muchas características del preprocesador (en especial las macros) son más apropiadas para los programadores de C que para los programadores de C++. Los programadores de C++ deben estar familiarizados con el preprocesador, ya que podrían tener la necesidad de trabajar con código heredado de C.

F.2 La directiva del preprocesador #include

Hemos utilizado la **directiva del preprocesador #include** a lo largo de este libro. Esta directiva hace que se incluya una copia de un archivo especificado en lugar de la directiva. Las dos formas de la directiva **#include** son:

```
#include <nombreachivo>
#include "nombreachivo"
```

La diferencia entre ellas es la ubicación en la que el preprocesador busca el archivo a incluir. Si el nombre del archivo está encerrado entre los signos **< y >** (la forma que se utiliza para los archivos de encabezado de la biblioteca estándar), el preprocesador busca el archivo especificado de una manera dependiente de la implementación, por lo general a través de directorios previamente designados. Si el nombre de archivo está encerrado entre comillas, el preprocesador busca primero en el mismo directorio en el que se va a compilar el archivo, y después en la misma forma dependiente de la implementación que para un nombre de archivo encerrado entre los signos **< y >**. Este método se utiliza comúnmente para incluir archivos de encabezado definidos por el programador.

La directiva **#include** se utiliza para incluir archivos de encabezado estándar, como **<iostream>** y **<iomanip>**. La directiva **#include** también se utiliza con los programas que consisten de varios archivos de código fuente que se van a compilar en conjunto. Por lo general, se crea y se incluye un archivo de encabezado que contiene declaraciones y definiciones comunes para los archivos separados del programa. Algunos ejemplos de dichas declaraciones y definiciones son: clases, estructuras, uniones, enumeraciones y prototipos de funciones, constantes y objetos de flujo (por ejemplo, **cin**).

F.3 La directiva del preprocesador #define: constantes simbólicas

La directiva del preprocesador `#define` crea constantes simbólicas (constantes representadas como símbolos) y macros (operaciones definidas como símbolos). El formato de esta directiva es:

```
#define identificador texto-de-reemplazo
```

Cuando aparece esta línea en un archivo, todas las ocurrencias subsiguientes (excepto las que están dentro de una cadena) de *identificador* en ese archivo se reemplazarán por *texto-de-reemplazo* antes de que se compile el programa. Por ejemplo,

```
#define PI 3.14159
```

reemplaza todas las ocurrencias subsiguientes de la constante simbólica PI con la constante numérica 3.14159. Las constantes simbólicas nos permiten crear un nombre para una constante, y utilizar el nombre a lo largo del programa. Más adelante, si la constante necesita modificarse en el programa, se puede modificar una sola vez en la directiva del preprocesador `#define`; y cuando el programa se vuelva a compilar, todas las ocurrencias de la constante en el programa se modificarán. [Nota: todo lo que está a la derecha del nombre de la constante simbólica reemplaza a esta constante simbólica. Por ejemplo, `#define PI = 3.14159` hace que el preprocesador reemplace cada ocurrencia de PI con = 3.14159. Dicho reemplazo es la causa de muchos ligeros errores lógicos y de sintaxis]. También es un error redefinir una constante simbólica con un nuevo valor sin primero eliminar la primera definición. Observe que en C++ se prefieren las variables `const` en C++ en vez de las constantes simbólicas. Una vez que se reemplaza una constante simbólica con su texto de reemplazo, sólo el texto de reemplazo está visible para el depurador. Una desventaja de las variables `const` es que podrían requerir una ubicación de memoria del tamaño de su tipo de datos; las constantes simbólicas no requieren memoria adicional.



Error común de programación F.2

Si se utilizan constantes simbólicas en un archivo que no sea el archivo en el que están definidas, se produce un error de compilación (a menos que se incluyan de un archivo de encabezado, mediante `#include`).



Buena práctica de programación F.1

Al utilizar nombres significativos para las constantes simbólicas, los programas se vuelven más auto-documentados.

F.4 La directiva del preprocesador #define: macros

[Nota: esta sección se incluye para beneficio de los programadores de C++ que necesiten trabajar con código heredado de C. En C++, las macros se pueden reemplazar comúnmente por las plantillas y funciones en línea]. Una macro es una operación definida en una directiva del preprocesador `#define`. Al igual que con las constantes simbólicas, el **identificador de macros** se reemplaza con el **texto de reemplazo** antes de compilar el programa. Las macros se pueden definir con o sin **argumentos**. Una macro sin argumentos se procesa como una constante simbólica. En una macro con argumentos, éstos se reemplazan en el *texto-de-reemplazo*, y después la macro se expande (es decir, el *texto-de-reemplazo* reemplaza el identificador de la macro y la lista de argumentos en el programa). No hay comprobación de tipos de datos para los argumentos de una macro. Ésta se utiliza simplemente para sustituir el texto.

Considere la siguiente definición de una macro con un argumento para el área de un círculo:

```
#define AREA_CIRCULO( x ) ( PI * ( x ) * ( x ) )
```

En cualquier parte del archivo en donde aparezca `AREA_CIRCULO(y)`, el valor de y se sustituye por x en el texto de reemplazo, la constante simbólica PI se reemplaza por su valor (definido con anterioridad) y la macro se expande en el programa. Por ejemplo, la instrucción

```
area = AREA_CIRCULO( 4 );
```

se expande a

```
area = ( 3.14159 * ( 4 ) * ( 4 ) );
```

Como la expresión sólo contiene constantes, en tiempo de compilación el valor de la expresión se puede evaluar, y el resultado se asigna a area en tiempo de ejecución. Los paréntesis alrededor de cada x en el texto de reemplazo y alrededor de toda la expresión obligan a que se utilice el orden de evaluación apropiado cuando el argumento de la macro es una expresión. Por ejemplo, la instrucción

```
area = AREA_CIRCULO( c + 2 );
```

se expande a

```
area = ( 3.14159 * ( c + 2 ) * ( c + 2 ) );
```

lo cual se evalúa en forma correcta, ya que los paréntesis obligan a que se utilice el orden de evaluación apropiado. Si se omiten los paréntesis, la expansión de la macro sería

```
area = 3.14159 * c + 2 * c + 2;
```

lo cual se evalúa de manera incorrecta como

```
area = ( 3.14159 * c ) + ( 2 * c ) + 2;
```

debido a las reglas de precedencia de los operadores.

Error común de programación F.3

 Olvidar encerrar los argumentos de una macro entre paréntesis en el texto de reemplazo es un error.

La macro AREA_CIRCULO se podría definir como una función. La función areaCirculo, como en

```
double areaCirculo(double x) { return 3.14159 * x * x; }
```

realiza el mismo cálculo que AREA_CIRCULO, pero la función areaCirculo tiene asociada la sobrecarga de la llamada a una función. Las ventajas de AREA_CIRCULO son que las macros insertan código directamente en el programa (evitando la sobrecarga de las funciones) y el programa mantiene su legibilidad, debido a que AREA_CIRCULO se define por separado y tiene un nombre significativo. Una desventaja es que el argumento se evalúa dos veces. Además, cada vez que aparece una macro en un programa, se expande. Si la macro es extensa, esto produce un aumento en el tamaño del programa. Por ende, hay una concesión entre la velocidad de ejecución y el tamaño del programa (si el espacio en disco es poco). Hay que tener en cuenta que se prefieren las funciones `inline` (vea el capítulo 3) para obtener el rendimiento de las macros y los beneficios de las funciones relacionados con la ingeniería de software.

Tip de rendimiento F.1

 Algunas veces las macros se pueden utilizar para reemplazar la llamada a una función con el código `inline` antes del tiempo de ejecución. Esto elimina la sobrecarga de una llamada a una función. Las funciones en línea son preferibles a las macros, ya que ofrecen los servicios de comprobación de tipos de las funciones.

A continuación se muestra la definición de una macro con dos argumentos para el área de un rectángulo:

```
#define AREA_RECTANGULO( x, y ) ( ( x ) * ( y ) )
```

En cualquier parte del programa en donde aparezca AREA_RECTANGULO(a, b), los valores de a y b se sustituyen en el texto de reemplazo de la macro, y la macro se expande en lugar de su nombre. Por ejemplo, la instrucción

```
areaRect = AREA_RECTANGULO( a + 4, b + 7 );
```

se expande a

```
areaRect = ( ( a + 4 ) * ( b + 7 ) );
```

El valor de la expresión se evalúa y se asigna a la variable areaRect.

El texto de reemplazo para una macro o constante simbólica es por lo general cualquier texto en la línea después del identificador en la directiva `#define`. Si el texto de reemplazo para una macro o constante simbólica es más extenso que el resto de la línea, debemos colocar una barra diagonal inversa (\) al final de cada línea de la macro (excepto la última línea), con lo cual indicamos que el texto de reemplazo continúa en la siguiente línea.

Las constantes simbólicas y las macros se pueden descartar utilizando la directiva del preprocesador `#undef`. La directiva `#undef` elimina la definición de una constante simbólica o nombre de macro. El alcance de una constante simbólica o macro es desde su definición, hasta que quede indefinida con `#undef` o hasta llegar al final del archivo. Una vez indefinido, un nombre puede redefinirse con `#define`.

Observe que las expresiones con efectos secundarios (por ejemplo, que se modifiquen los valores de las variables) no deben pasarse a una macro, ya que los argumentos de la macro pueden llegar a evaluarse más de una vez.

Error común de programación F.4

 A menudo, las macros reemplazan un nombre que no estaba destinado a ser un uso de la macro, sino que simplemente se escribe igual. Esto puede provocar errores de compilación y de sintaxis excepcionalmente misteriosos.

F.5 Compilación condicional

La compilación condicional nos permite controlar la ejecución de las directivas del preprocesador y la compilación del código del programa. Cada una de las directivas del preprocesador condicionales evalúa una expresión entera constante que determinará si el código se va a compilar o no. Las expresiones de conversión de tipos, las expresiones `sizeof` y las constantes de enumeración no se pueden evaluar en las directivas del preprocesador, ya que todas son determinadas por el compilador y el preprocesamiento ocurre antes de la compilación.

La construcción de preprocesador condicional es muy parecida a la estructura de selección `if`. Considere el siguiente código de preprocesador:

```
#ifndef NULL
#define NULL 0
#endif
```

este código determina si la constante simbólica `NULL` ya se encuentra definida. La expresión `#ifndef NULL` incluye el código hasta `#endif` si `NULL` no está definida, y omite el código si `NULL` está definida. Cada construcción `#if` termina con `#endif`. Las directivas `#ifdef` y `#ifndef` son abreviaciones para `#if defined(nombre)` y `#if !defined(nombre)`. Podemos evaluar una construcción del preprocesador condicional que conste de varias partes mediante el uso de las directivas `#elif` (el equivalente de `else if` en una estructura `if`) y `#else` (el equivalente de `else` en una estructura `if`).

Durante el desarrollo del programa, comúnmente los programadores encuentran que es útil “comentar” porciones extensas de código para evitar que se compile. Si el código contiene comentarios estilo C, no se pueden utilizar los signos `/*` y `*/` para realizar esta tarea, debido a que al encontrar el `/*` se terminaría el comentario. En vez de ello, podemos usar la siguiente construcción del preprocesador:

```
#if 0
    código que no se debe compilar
#endif
```

Para permitir que el código se compile, simplemente reemplazamos el valor `0` en la construcción anterior con el valor `1`.

La compilación condicional se utiliza comúnmente como una ayuda para la depuración. A menudo se utilizan instrucciones de salida para imprimir valores de variables y confirmar el flujo de control. Estas instrucciones de salida se pueden encerrar en directivas del preprocesador condicionales, de manera que las instrucciones se compilen sólo hasta que se complete el proceso de depuración. Por ejemplo,

```
#ifdef DEBUG
    cerr << "Variable x = " << x << endl;
#endif
```

hace que se compile la instrucción `cerr` en el programa, si se ha definido la constante simbólica `DEBUG` antes de la directiva `#ifdef DEBUG`. Esta constante simbólica generalmente se establece mediante un compilador de línea de comandos, o a través de las opciones en el IDE (por ejemplo, Visual Studio) y no por una definición `#define` explícita. Cuando se completa la depuración, la directiva `#define` se elimina del archivo de código fuente y las instrucciones de salida que se insertaron para fines de depuración se ignoran durante la compilación. En programas más extensos, podría ser conveniente definir varias constantes simbólicas distintas que controlen la compilación condicional en secciones separadas del archivo de código fuente.

Error común de programación F.5



Al insertar instrucciones de salida compiladas en forma condicional para fines de depuración, en ubicaciones en donde C++ espere una sola instrucción, se pueden provocar errores lógicos y de sintaxis. En este caso, la instrucción compilada en forma condicional debe encerrarse en una instrucción compuesta. Así, cuando se compile el programa con instrucciones de depuración, el flujo de control del programa no se alterará.

F.6 Las directivas del preprocesador `#error` y `#pragma`

La directiva `#error`

```
#error tokens
```

imprime un mensaje dependiente de la implementación, incluyendo los `tokens` especificados en la directiva. Los `tokens` son secuencias de caracteres separados por espacios. Por ejemplo,

```
#error 1 - Error fuera de rango
```

contiene seis tokens. Por ejemplo, en un compilador de C++ popular, cuando se procesa una directiva `#error`, los tokens en la directiva se muestran como un mensaje de error, el preprocessamiento se detiene y el programa no se compila.

La directiva `#pragma`

```
#pragma tokens
```

provoca una acción definida por la implementación. Una directiva `#pragma` que no sea reconocida por la implementación se ignora. Por ejemplo, un compilador de C++ específico podría reconocer directivas `#pragma` que nos permitan aprovechar las capacidades específicas de ese compilador. Para obtener más información acerca de `#error` y `#pragma`, consulte la documentación de su implementación de C++.

F.7 Los operadores # y

Los operadores del procesador `#` y `##` están disponibles en C++ y en C de ANSI/ISO. El operador `#` hace que un token del texto de reemplazo se convierta en una cadena encerrada entre comillas. Considere la siguiente definición de una macro:

```
#define HOLAC( x ) cout << "Hola, " #x << endl;
```

Cuando aparece `HOLA(Juan)` en un archivo del programa, se expande a

```
cout << "Hola, " " Juan " << endl;
```

La cadena "Juan" reemplaza a `#x` en el texto de reemplazo. Las cadenas separadas por espacios en blanco se concatenan durante el preprocessamiento, por lo que la instrucción anterior es equivalente a

```
cout << "Hola, Juan" << endl;
```

Observe que se debe utilizar el operador `#` en una macro con argumentos, debido a que el operando de `#` hace referencia a un argumento de la macro.

El operador `##` concatena dos tokens. Considere la siguiente definición de una macro:

```
#define CONCATTOKEN( x, y ) x ## y
```

Cuando aparece `CONCATTOKEN` en el programa, sus argumentos se concatenan y se utilizan para reemplazar la macro. Por ejemplo, `CONCATTOKEN(0, K)` se reemplaza por `OK` en el programa. El operador `##` debe tener dos operandos.

F.8 Constantes simbólicas predefinidas

Hay seis **constantes simbólicas predefinidas** (figura F.1). Los identificadores para cada una de estas constantes empiezan y (excepto por `_cplusplus`) terminan con *dos* guiones bajos. Estos identificadores y el operador del preprocessador `defined` (sección F.5) no se pueden utilizar en directivas `#define` o `#undef`.

Constante simbólica	Descripción
<code>_LINE_</code>	El número de línea de la línea actual de código fuente (una constante entera).
<code>_FILE_</code>	El presunto nombre del archivo de código fuente (una cadena).
<code>_DATE_</code>	La fecha de compilación del archivo de código fuente (una cadena de la forma "Mmm dd aaaa", tal como "Ago 19 2002").
<code>_STDC_</code>	Indica si el programa se conforma al estándar ANSI/ISO de C. Contiene el valor 1 si hay una conformidad completa, y está indefinida en caso contrario.
<code>_TIME_</code>	La hora de compilación del archivo de código fuente (una literal de cadena de la forma "hh:mm:ss").
<code>_cplusplus</code>	Contiene el valor 199711L (la fecha en que se aprobó el estándar ISO de C++) si el archivo va a ser compilado por un compilador de C++, y está indefinida en caso contrario. Permite establecer un archivo para que se compile como C o C++.

Figura F.1 | Las constantes simbólicas predefinidas.

F.9 Aserciones

La macro `assert` (definida en el archivo de encabezado `<cassert>`) prueba el valor de una expresión. Si el valor de la expresión es 0 (falso), entonces `assert` imprime un mensaje de error y llama a la función `abort` (de la biblioteca de

herramientas generales: <cstdlib>) para terminar la ejecución del programa. Ésa es una herramienta útil de depuración para evaluar si una variable tiene un valor correcto. Por ejemplo, suponga que la variable `x` nunca debe ser mayor que 10 en un programa. Se puede utilizar una aserción para evaluar el valor de `x` e imprimir un mensaje de error si el valor de `x` es incorrecto. La instrucción sería

```
assert( x <= 10 );
```

Si `x` es mayor que 10 al llegar a la instrucción anterior en un programa, se imprime un mensaje de error que contiene el número de línea y el nombre del archivo, y el programa termina. Así, el programador se puede concentrar en esta área del código para encontrar el error. Si la constante simbólica `NDEBUG` está definida, se ignorarán las aserciones subsiguientes. Por ende, cuando ya no sean necesarias las aserciones (es decir, cuando se complete la depuración), insertamos la línea

```
#define NDEBUG
```

en el archivo del programa, en vez de eliminar cada aserción en forma manual. Al igual que con la constante simbólica `DEBUG`, `NDEBUG` se establece a menudo mediante opciones de la línea de comandos del compilador, o a través de una opción en el IDE.

La mayoría de los compiladores de C++ incluyen ahora el manejo de excepciones. Los programadores de C++ prefieren utilizar excepciones en vez de aserciones. Pero las aserciones aún son valiosas para los programadores de C++ que trabajan con código heredado de C.

F.10 Repaso

En este apéndice vimos la directiva `#include`, que se utiliza para desarrollar programas extensos. También aprendió acerca de la directiva `#define`, que se utiliza para crear macros. Presentamos la compilación condicional, cómo mostrar mensajes de error y utilizar aserciones. En el siguiente apéndice implementaremos el diseño del sistema ATM del Ejemplo práctico de Ingeniería de Software de los capítulos 1 a 7, 9 y 13.

Resumen

Sección F.2 La directiva del preprocesador `#include`

- Todas las directivas del preprocesador empiezan con `#` y se procesan antes de que se compile el programa.
- Sólo pueden aparecer caracteres de espacio en blanco antes de una directiva del preprocesador en una línea.
- La directiva `#include` incluye una copia del archivo especificado. Si el nombre de archivo va encerrado entre comillas, el preprocesador empieza a buscar el archivo a incluir en el mismo directorio en el que se va a compilar el archivo. Si el nombre del archivo va encerrado entre los signos `<` y `>`, la búsqueda se realiza de una manera definida por la implementación.

Sección F.3 La directiva del preprocesador `#define`: constantes simbólicas

- La directiva del preprocesador `#define` se utiliza para crear constantes simbólicas y macros.
- Una constante simbólica es un nombre para una constante.

Sección F.4 La directiva del preprocesador `#define`: macros

- Una macro es una operación definida en una directiva del preprocesador `#define`. Las macros se pueden definir con o sin argumentos.
- El texto de reemplazo para una macro o constante simbólica es cualquier texto restante en la línea después del identificador (`y`, si la hay, la lista de argumentos de la macro) en la directiva `#define`. Si el texto de reemplazo para una macro o constante simbólica es demasiado extenso como para caber en una sola línea, se coloca una barra diagonal inversa (`\`) al final de la línea, indicando que el texto de reemplazo continúa en la siguiente línea.
- Las constantes simbólicas y las macros se pueden descartar mediante el uso de la directiva del preprocesador `#undef`. La directiva `#undef` elimina la definición de la constante simbólica o nombre de la macro.
- El alcance de una constante simbólica o macro es a partir de su definición, hasta que se elimine su definición con `#undef` o al llegar al final del programa.

Sección F.5 Compilación condicional

- La compilación condicional nos permite controlar la ejecución de las directivas del preprocesador y la compilación del código del programa.
- Las directivas del preprocesador condicionales evalúan expresiones enteras constantes. Las expresiones de conversión de tipos, las expresiones `sizeof` y las constantes de enumeración no se pueden evaluar en las directivas del preprocesador.

- Cada construcción `#if` termina con `#endif`.
- Las directivas `#ifdef` y `#ifndef` se proporcionan como abreviaciones para `#if defined(nombre)` y `#if !defined(nombre)`.
- Una construcción del preprocesador condicional que consiste de varias partes se evalúa con las directivas `#elif` y `#else`.

Sección F.6 Las directivas del preprocesador `#error` y `#pragma`

- La directiva `#error` imprime un mensaje dependiente de la implementación que incluye los tokens especificados en la directiva, y termina el preprocesamiento y la compilación.
- La directiva `#pragma` provoca una acción definida por la implementación. Si la implementación no reconoce esta directiva, se ignora.

Sección F.7 Los operadores `#` y `##`

- El operador `#` hace que el siguiente token del texto de reemplazo se convierta en una cadena encerrada entre comillas. El operador `#` debe utilizarse en una macro con argumentos, debido a que el operando de `#` debe ser un argumento de la macro.
- El operador `##` concatena dos tokens. El operador `##` debe tener dos operandos.

Sección F.8 Constantes simbólicas predefinidas

- Hay seis constantes simbólicas predefinidas. La constante `_LINE_` es el número de línea de la línea actual en el código fuente (un entero). La constante `_FILE_` es el presunto nombre del archivo (una cadena). La constante `_DATE_` es la fecha de compilación del archivo de código fuente (una cadena). La constante `_TIME_` es la hora de compilación del archivo de código fuente (una cadena). Observe que cada una de las constantes simbólicas predefinidas empieza (y, con la excepción de `_cplusplus`, termina) con dos guiones bajos.

Sección F.9 Aserciones

- La macro `assert` (definida en el archivo de encabezado `<cassert>`) prueba el valor de una expresión. Si el valor de la expresión es 0 (falso), entonces `assert` imprime un mensaje de error y llama a la función `abort` para terminar la ejecución del programa.

Terminología

<code>##</code> , operador del preprocesador de concatenación	<code><cstdio></code>
<code>#</code> , operador	<code><cstdlib></code>
<code>#define</code>	<code>abort</code>
<code>#elif</code>	alcance de una constante simbólica o macro
<code>#else</code>	archivo de encabezado
<code>#endif</code>	archivos de encabezado de la biblioteca estándar
<code>#error</code>	argumento
<code>#if</code>	<code>assert</code>
<code>#ifdef</code>	compilación condicional
<code>#ifndef</code>	constante simbólica
<code>#include "nombrearchivo"</code>	constantes simbólicas predefinidas
<code>#include <nombreachivo></code>	depurador
<code>#pragma</code>	directiva de preprocesamiento
<code>#undef</code>	directiva del preprocesador para convertir a una cadena
<code>\</code> (diagonal inversa), carácter de continuación	directivas
<code>_cplusplus</code>	ejecución condicional del preprocesador
<code>_DATE_</code>	expandir una macro
<code>_FILE_</code>	macro
<code>_LINE_</code>	macro con argumentos
<code>_TIME_</code>	preprocesador
<code><cassert></code>	texto de reemplazo

Ejercicios de autoevaluación

F.I Complete los siguientes enunciados:

- Cada directiva del preprocesador debe empezar con _____.
- La construcción de compilación condicional puede extenderse para evaluar múltiples casos, utilizando las directivas _____ y _____.
- La directiva _____ crea macros y constantes simbólicas.
- Sólo pueden aparecer caracteres _____ antes de una directiva del preprocesador en una línea.

1096 Apéndice F Preprocesador

- e) La directiva _____ descarta los nombres de las constantes simbólicas y las macros.
- f) Las directivas _____ y _____ se proporcionan como notación abreviada para `#if defined(nombre)` y `#ifndef !defined(nombre)`.
- g) _____ nos permite controlar la ejecución de las directivas del preprocesador y la compilación del código del programa.
- h) La macro _____ imprime un mensaje y termina la ejecución del programa si el valor de la expresión con la que se evalúa la macro es 0.
- i) La directiva _____ inserta un archivo en otro archivo.
- j) El operador _____ concatena sus dos argumentos.
- k) El operador _____ convierte su operando en una cadena.
- l) El carácter _____ indica que el texto de reemplazo para una constante simbólica o macro continúa en la siguiente línea.

F.2 Escriba un programa para imprimir los valores de las constantes simbólicas predefinidas `__LINE__`, `__FILE__`, `__DATE__` y `__TIME__` que se listan en la figura F.1.

F.3 Escriba una directiva del preprocesador para realizar cada una de las siguientes acciones:

- a) Definir la constante simbólica `SI` para que tenga el valor 1.
- b) Definir la constante simbólica `NO` para que tenga el valor 0.
- c) Incluir el archivo de encabezado `comun.h`. Este encabezado se encuentra en el mismo directorio que el archivo que se va a compilar.
- d) Si la constante simbólica `TRUE` está definida, eliminar su definición y volverla a definir como 1. No utilice `#ifdef`.
- e) Si la constante simbólica `TRUE` está definida, eliminar su definición y volverla a definir como 1. Utilice la directiva del preprocesador `#ifdef`.
- f) Si la constante simbólica `ACTIVO` no es igual a 0, definir la constante simbólica `INACTIVO` como 0. En caso contrario, definir `INACTIVO` como 1.
- g) Definir la macro `VOLUMEN_CUBO` que calcula el volumen de un cubo (recibe un argumento).

Respuestas a los ejercicios de autoevaluación

F.1 a). b). #elif, #else. c) #define. d) de espacio en blanco. e) #undef. f) #ifdef, #ifndef. g) La compilación condicional. h) assert. i) #include. j) ##. k) #. l) \.

F.2 (Vea a continuación.)

```
1 // ejF_02.cpp
2 // Solución al Ejercicio de autoevaluación F.2.
3 #include <iostream>
4
5 using std::cout;
6     std::endl;
7
8 int main()
9 {
10     cout << "__LINE__ = " << __LINE__ << endl
11     << "__FILE__ = " << __FILE__ << endl
12     << "__DATE__ = " << __DATE__ << endl
13     << "__TIME__ = " << __TIME__ << endl
14     << "__cplusplus = " << __cplusplus << endl;
15
16 return 0;
17
18 } // fin de main
```

```
__LINE__ = 9
__FILE__ = c:\cpphttp6e\cap19\ej19_02.CPP
__DATE__ = Jul 17 2002
__TIME__ = 09:55:58
__cplusplus = 199711L
```

F.3 a) #define SI 1
b) #define NO 0
c) #include "comun.h"

```

d) #if defined(TRUE)
    #undef TRUE
    #define TRUE 1
#endif
e) #ifdef TRUE
    #undef TRUE
    #define TRUE 1
#endif
f) #if ACTIVO
    #define INACTIVO 0
#else
    #define INACTIVO 1
#endif
g) #define VOLUMEN_CUBO( x ) ( ( x ) * ( x ) * ( x ) )

```

Ejercicios

F.4 Escriba un programa que defina una macro con un argumento para calcular el volumen de una esfera. El programa debe calcular el volumen para las esferas cuyos radios se encuentren en el rango de 1 a 10, y debe imprimir los resultados en formato tabular. La fórmula para el volumen de una esfera es

$$(4.0 / 3) * \pi * r^3$$

en donde π es 3.14159.

F.5 Escriba un programa que produzca los siguientes resultados:

La suma de x y y es 13

El programa debe definir la macro SUMA con dos argumentos, x y y, y debe utilizar SUMA para producir los resultados.

F.6 Escriba un programa que utilice la macro MINIMO2 para determinar el menor de dos valores numéricos. Debe recibir los valores como entrada mediante el teclado.

F.7 Escriba un programa que utilice la macro MINIMO3 para determinar el menor de tres valores numéricos. La macro MINIMO3 debe utilizar la macro MINIMO2 definida en el ejercicio F.6 para determinar el menor número. Debe recibir los valores como entrada mediante el teclado.

F.8 Escriba un programa que utilice la macro IMPRIMIR para imprimir un valor de cadena.

F.9 Escriba un programa que utilice la macro IMPRIMIRARREGLO para imprimir un arreglo de enteros. La macro debe recibir el arreglo y el número de elementos en el arreglo como argumentos.

F.10 Escriba un programa que utilice la macro SUMARARREGLO para sumar los valores en un arreglo numérico. La macro debe recibir el arreglo y el número de elementos en el arreglo como argumentos.

F.11 Vuelva a escribir las soluciones a los ejercicios F.4 a F.10 como funciones *inline*.

F.12 Para cada una de las siguientes macros, identifique los posibles problemas (si los hay) cuando el preprocesador expanda las macros:

- a) #define SQR(x) x * x
- b) #define SQR(x) (x * x)
- c) #define SQR(x) (x) * (x)
- d) #define SQR(x) ((x) * (x))



Código del caso de estudio del ATM

G.1 Implementación del caso de estudio del ATM

Este apéndice contiene la implementación funcional completa del sistema ATM que diseñamos en las secciones tituladas “Ejemplo práctico de Ingeniería de Software” al final de los capítulos 1 a 7, 9 y 13. La implementación comprende 877 líneas de código de C++. Consideramos las clases en el orden en el que las identificamos en la sección 3.11:

- `ATM`
- `Pantalla`
- `Teclado`
- `DispensadorEfectivo`
- `RanuraDeposito`
- `Cuenta`
- `BaseDatosBanco`
- `Transaccion`
- `SolicitudSaldo`
- `Retiro`
- `Deposito`

Aplicamos los lineamientos que vimos en las secciones 9.11 y 13.10 para codificar estas clases, con base en la manera en que las modelamos en los diagramas de clases UML de las figuras 13.28 y 13.29. Para desarrollar las definiciones de las funciones miembro de las clases, nos referimos a los diagramas de actividad presentados en la sección 5.11 y los diagramas de comunicaciones y secuencia presentados en la sección 7.12. Observe que nuestro diseño del ATM no especifica toda la lógica de programación, por lo que tal vez no especifique todos los atributos y operaciones requeridos para completar la implementación del ATM. Ésta es una parte normal del proceso de diseño orientado a objetos. A medida que implementamos el sistema, completamos la lógica del programa, agregando atributos y comportamientos según sea necesario para construir el sistema ATM especificado por la especificación de requerimientos de la sección 2.8.

Concluimos la discusión presentando un programa de C++ (`EjemploPracticoATM.cpp`) que inicia el ATM y pone en uso las demás clases del sistema. Recuerde que estamos desarrollando la primera versión del sistema ATM que se ejecuta en una computadora personal, y que utiliza el teclado y el monitor para lograr la mayor semejanza posible con el teclado y la pantalla de un ATM. Además, sólo simulamos las acciones del dispensador de efectivo y la ranura de depósito del ATM. Sin embargo, tratamos de implementar el sistema de manera tal que las versiones reales de hardware de esos dispositivos pudieran integrarse sin necesidad de cambios considerables en el código.

G.2 La clase ATM

La clase ATM (figuras G.1 y G.2) representa al ATM como un todo. La figura G.1 contiene la definición de la clase ATM, encerrada entre directivas del preprocesador `#ifndef`, `#define` y `#endif` para asegurar que esta definición se incluya sólo una vez en un programa. En un momento hablaremos sobre las líneas 6 a 11. Las líneas 16 y 17 contienen los prototipos de las funciones miembro `public` de la clase. El diagrama de clases de la figura 13.29 no lista operaciones para la clase ATM, pero ahora declaramos una función miembro `public` llamada `ejecutar` (línea 17) en la clase ATM, que permite que un cliente externo de la clase (como `EjemploPracticoATM.cpp`) indique a ATM que debe ejecutarse. También incluimos un prototipo de función para un constructor predeterminado (línea 16), del cual hablaremos en breve.

En las líneas 19 a 25 de la figura G.1 se implementan los atributos de la clase como miembros de datos `private`. Determinamos todos estos atributos (excepto uno) de los diagramas de clases de UML de las figuras 13.28 y 13.29. Observa que en la figura 13.29 implementamos el atributo `Boolean usuarioAutenticado` de UML como un miembro de datos `bool` en C++ (línea 19). En la línea 20 se declara un miembro de datos que no se incluye en nuestro diseño UML: el miembro de datos `int` llamado `numeroCuentaActual`, que lleva el registro del número de cuenta del usuario autenticado actual. Pronto veremos cómo es que la clase utiliza este atributo.

En las líneas 21 a 24 se crean objetos para representar las partes del ATM. En el diagrama de clases de la figura 13.28 vimos que la clase ATM tiene relaciones de composición con las clases `Pantalla`, `Teclado`, `DispensadorEfectivo` y `RanuraDeposito`, por lo que la clase ATM es responsable de su creación. En la línea 25 se crea un objeto `BaseDatosBanco`, con el que el ATM interactúa para acceder a la información de las cuentas del banco y manipularla. [Nota: si éste fuera un sistema ATM real, la clase ATM recibiría una referencia a un objeto de base de datos existente, creado por el banco. No obstante, en esta implementación sólo estamos simulando la base de datos del banco, por lo que la clase ATM crea el objeto `BaseDatosBanco` con el que interactúa]. Observa que en las líneas 6 a 10 se incluyen (mediante

```

1 // ATM.h
2 // Definición de la clase ATM. Representa a un cajero automático.
3 #ifndef ATM_H
4 #define ATM_H
5
6 #include "Pantalla.h" // Definición de la clase Pantalla
7 #include "Teclado.h" // Definición de la clase Teclado
8 #include "DispensadorEfectivo.h" // Definición de la clase DispensadorEfectivo
9 #include "RanuraDeposito.h" // Definición de la clase RanuraDeposito
10 #include "BaseDatosBanco.h" // Definición de la clase BaseDatosBanco
11 class Transaccion; // declaración anticipada de la clase Transaccion
12
13 class ATM
14 {
15 public:
16     ATM(); // el constructor inicializa los miembros de datos
17     void ejecutar(); // inicia el ATM
18 private:
19     bool usuarioAutenticado; // indica si el usuario está autenticado
20     int numeroCuentaActual; // número de cuenta del usuario actual
21     Pantalla pantalla; // pantalla del ATM
22     Teclado teclado; // teclado del ATM
23     DispensadorEfectivo dispensadorEfectivo; // dispensador de efectivo del ATM
24     RanuraDeposito ranuraDeposito; // ranura de depósito del ATM
25     BaseDatosBanco baseDatosBanco; // base de datos de información de las cuentas
26
27     // funciones utilitarias privadas
28     void autenticarUsuario(); // trata de autenticar al usuario
29     void realizarTransacciones(); // realiza transacciones
30     int mostrarMenuPrincipal() const; // muestra el menú principal
31
32     // devuelve un objeto de la clase derivada de Transaccion que se especifica
33     Transaccion *crearTransaccion( int );
34 }; // fin de la clase ATM
35
36 #endif // ATM_H

```

Figura G.1 | Definición de la clase ATM, que representa al ATM.

#include) las definiciones de Pantalla, Teclado, DispensadorEfectivo, RanuraDeposito y BaseDatosBanco, de manera que el ATM pueda almacenar objetos de estas clases.

Las líneas 28 a 30 y 33 contienen los prototipos de las funciones utilitarias `private` que la clase utiliza para realizar sus tareas. En breve veremos cómo estas funciones dan servicio a la clase. Observe que la función miembro `crearTransaccion` (línea 33) devuelve un apuntador `Transaccion`. Para incluir el nombre de la clase `Transaccion` en este archivo, debemos por lo menos incluir una declaración anticipada de la clase `Transaccion` (línea 11). Recuerde que una declaración anticipada indica al compilador que una clase existe, pero está definida en otra parte. Una declaración anticipada es suficiente aquí, ya que estamos utilizando un apuntador `Transaccion` como tipo de valor de retorno; si fuéramos a crear o devolver un objeto `Transaccion` actual, tendríamos que incluir (mediante `#include`) el archivo de encabezado de `Transaccion` completo.

Definiciones de las funciones miembro de la clase ATM

La figura G.2 contiene las definiciones de las funciones miembro para la clase ATM. En las líneas 3 a 7 se incluyen (mediante `#include`) los archivos de encabezado requeridos por el archivo de implementación `ATM.cpp`. Observe que al incluir el archivo de encabezado de ATM permitimos al compilador asegurar que las funciones miembro de la clase se definan en forma correcta. Esto también permite a las funciones miembro utilizar los miembros de datos de la clase.

```

1 // ATM.cpp
2 // Definiciones de las funciones miembro para la clase ATM.
3 #include "ATM.h" // Definición de la clase ATM
4 #include "Transaccion.h" // Definición de la clase Transaccion
5 #include "SolicitudSaldo.h" // Definición de la clase SolicitudSaldo
6 #include "Retiro.h" // Definición de la clase Retiro
7 #include "Deposito.h" // Definición de la clase Deposito
8
9 // constantes de enumeración que representan las opciones del menú principal
10 enum OpcionMenu { SOLICITUD_SALDO = 1, RETIRO, DEPOSITO, SALIR };
11
12 // el constructor predeterminado del ATM inicializa los miembros de datos
13 ATM::ATM()
14 {
15     : usuarioAutenticado ( false ), // el usuario no está autenticado para empezar
16     numeroCuentaActual( 0 ) // no hay número de cuenta actual para empezar
17 {
18     // cuerpo vacío
19 } // fin del constructor predeterminado del ATM
20
21 // inicia el ATM
22 void ATM::ejecutar()
23 {
24     // da la bienvenida y autentica al usuario; realiza transacciones
25     while ( true )
26     {
27         // itera mientras el usuario no esté autenticado
28         while ( !usuarioAutenticado )
29         {
30             pantalla.mostrarLineaMensaje( "\nBienvenido!" );
31             autenticarUsuario(); // autentica al usuario
32         } // fin de while
33
34         realizarTransacciones(); // ahora el usuario es autenticado
35         usuarioAutenticado = false; // restablece antes de la siguiente sesión del ATM
36         numeroCuentaActual = 0; // restablece antes de la siguiente sesión del ATM
37         pantalla.mostrarLineaMensaje( "\nGracias! Hasta luego!" );
38     } // fin de while
39 } // fin de la función ejecutar
40
41 // intenta autenticar al usuario con la base de datos

```

Figura G.2 | Definiciones de las funciones miembro de la clase ATM. (Parte I de 3).

```

42 {
43     pantalla.mostrarMensaje( "\nEscriba su numero de cuenta: " );
44     int numeroCuenta = teclado.obtenerEntrada(); // introduce el número de cuenta
45     pantalla.mostrarMensaje( "\nEscriba su NIP: " ); // pide el NIP
46     int nip = teclado.obtenerEntrada(); // introduce el NIP
47
48     // establece usuarioAutenticado con el valor bool devuelto por la base de datos
49     usuarioAutenticado =
50         baseDatosBanco.autenticarUsuario( numeroCuenta, nip );
51
52     // comprueba si la autenticación tuvo éxito
53     if ( usuarioAutenticado )
54     {
55         numeroCuentaActual = numeroCuenta; // guarda el # de cuenta del usuario
56     } // fin de if
57     else
58         pantalla.mostrarLineaMensaje(
59             "Numero de cuenta o NIP invalido. Intenta de nuevo." );
60 } // fin de la función autenticarUsuario
61
62 // muestra el menú principal y realiza las transacciones
63 void ATM::realizarTransacciones()
64 {
65     // apuntador local para almacenar la transacción actual en proceso
66     Transaccion *transaccionActualPtr;
67
68     bool usuarioSalio = false; // el usuario no ha elegido salir
69
70     // itera mientras el usuario no haya elegido la opción para salir del sistema
71     while ( !usuarioSalio )
72     {
73         // muestra el menú principal y obtiene la selección del usuario
74         int seleccionMenuPrincipal = mostrarMenuPrincipal();
75
76         // decide cómo proceder con base en la opción del menú elegida por el usuario
77         switch ( seleccionMenuPrincipal )
78         {
79             // el usuario eligió realizar uno de tres tipos de transacciones
80             case SOLICITUD_SALDO:
81             case RETIRO:
82             case DEPOSITO:
83                 // se inicializa como nuevo objeto del tipo elegido
84                 transaccionActualPtr =
85                     crearTransaccion( seleccionMenuPrincipal );
86
87                 transaccionActualPtr->ejecutar(); // ejecuta la transacción
88
89                 // libera el espacio para la Transaccion asignada en forma dinámica
90                 delete transaccionActualPtr;
91
92                 break;
93             case SALIR: // el usuario eligió terminar la sesión
94                 pantalla.mostrarLineaMensaje( "\nSaliendo del sistema..." );
95                 usuarioSalio = true; // esta sesión del ATM debe terminar
96                 break;
97             default: // el usuario no introdujo un entero del 1 al 4
98                 pantalla.mostrarLineaMensaje(
99                     "\nNo introdujo una selección válida. Intenta de nuevo." );
100                 break;
101         } // fin de switch
102     } // fin de while
103 } // fin de la función realizarTransacciones

```

Figura G.2 | Definiciones de las funciones miembro de la clase ATM. (Parte 2 de 3).

```

104
105 // muestra el menú principal y devuelve una selección de entrada
106 int ATM::mostrarMenuPrincipal() const
107 {
108     pantalla.mostrarLineaMensaje( "\nMenu principal:" );
109     pantalla.mostrarLineaMensaje( "1 - Ver mi saldo" );
110     pantalla.mostrarLineaMensaje( "2 - Retirar efectivo" );
111     pantalla.mostrarLineaMensaje( "3 - Depositar fondos" );
112     pantalla.mostrarLineaMensaje( "4 - Salir\n" );
113     pantalla.mostrarMensaje( "Introduzca una opcion: " );
114     return teclado.obtenerEntrada(); // devuelve la selección del usuario
115 } // fin de la función mostrarMenuPrincipal
116
117 // devuelve el objeto de la clase derivada de Transaccion especificada
118 Transaccion *ATM::crearTransaccion( int tipo )
119 {
120     Transaccion *tempPtr; // apuntador Transaccion temporal
121
122     // determina qué tipo de Transaccion crear
123     switch ( tipo )
124     {
125         case SOLICITUD_SALDO: // crea nueva transacción SolicitudSaldo
126             tempPtr = new SolicitudSaldo(
127                 numeroCuentaActual, pantalla, baseDatosBanco );
128             break;
129         case RETIRO: // crea nueva transacción Retiro
130             tempPtr = new Retiro( numeroCuentaActual, pantalla,
131                 baseDatosBanco, teclado, dispensadorEfectivo );
132             break;
133         case DEPOSITO: // crea nueva transacción Deposito
134             tempPtr = new Deposito( numeroCuentaActual, pantalla,
135                 baseDatosBanco, teclado, ranuraDeposito );
136             break;
137     } // fin de switch
138
139     return tempPtr; // devuelve el objeto recién creado
140 } // fin de la función crearTransaccion

```

Figura G.2 | Definiciones de las funciones miembro de la clase ATM. (Parte 3 de 3).

En la línea 10 se declara una enumeración (`enum`) llamada `OpcionMenu`, la cual contiene las constantes que corresponden a las cuatro opciones en el menú principal del ATM (es decir, solicitud de saldo, retiro, depósito y salir). Observe que al establecer `SOLICITUD_SALDO` en 1, a las constantes de enumeración subsiguientes se les asignan los valores 2, 3 y 4 de manera automática, ya que los valores de las constantes de enumeración se incrementan por 1.

En las líneas 13 a 18 se define el constructor de la clase ATM, el cual inicializa los miembros de datos de la clase. Cuando se crea un objeto ATM por primera vez, ningún usuario está autenticado, por lo que en la línea 14 se utiliza un inicializador de miembros para establecer `usuarioAutenticado` en `false`. De igual forma, en la línea 15 se inicializa `numeroCuentaActual` en 0, debido a que no hay un usuario actual todavía.

La función miembro `ejecutar` de ATM (líneas 21 a 38) utiliza un ciclo infinito (líneas 24 a 37) para dar la bienvenida al usuario en forma repetida, trata de autenticarlo y, si la autenticación tiene éxito, permitir que el usuario realice transacciones. Una vez que un usuario autenticado realiza las transacciones deseadas y opta por salir, el ATM se restablece a sí mismo, muestra un mensaje de despedida al usuario y reinicia el proceso. Aquí utilizamos un ciclo infinito para simular el hecho de que un ATM parece funcionar en forma continua hasta que el banco lo desconecta (una acción que está más allá del control del usuario). El usuario de un ATM tiene la opción de salir del sistema, pero no tiene la habilidad de desconectar el ATM por completo.

Dentro del ciclo infinito de la función miembro `ejecutar`, en las líneas 27 a 31 el ATM da la bienvenida en forma repetida y trata de autenticar al usuario, siempre y cuando éste no haya sido autenticado ya (es decir, que `!usuarioAutenticado` sea `true`). En la línea 29 se invoca a la función miembro `mostrarLineaMensaje` de la `pantalla` del ATM

para mostrar un mensaje de bienvenida. Al igual que la función miembro `mostrarMensaje` de `Pantalla` designada en el ejemplo práctico, la función miembro `mostrarLineaMensaje` (declarada en la línea 13 de la figura G.3 y definida en las líneas 20 a 23 de la figura G.4) muestra un mensaje al usuario, pero esta función miembro también imprime una nueva línea después de mostrar el mensaje. Hemos agregado esta función miembro durante la implementación para dar a los clientes de la clase `Pantalla` mayor control sobre la colocación de los mensajes mostrados. En la línea 30 de la figura G.2 se invoca la función utilitaria `private autenticarUsuario` de la clase ATM (líneas 41 a 60) para tratar de autenticar al usuario.

Consultamos la especificación de requerimientos para determinar los pasos necesarios para autenticar al usuario antes de permitir que ocurran transacciones. En la línea 43 de la función miembro `autenticarUsuario` se invoca a la función miembro `mostrarMensaje` de la `pantalla` del ATM para pedir al usuario que introduzca un número de cuenta. En la línea 44 se invoca a la función miembro `obtenerEntrada` del `teclado` del ATM para obtener la entrada del usuario, y después se almacena el valor entero introducido por el usuario en una variable local llamada `numeroCuenta`. A continuación, la función miembro `autenticarUsuario` pide al usuario que introduzca un NIP (línea 45) y almacena el NIP introducido por el usuario en una variable local llamada `nip` (línea 46). Después, en las líneas 49 a 50 se hace un intento de autenticar al usuario, para lo cual se pasan el `numeroCuenta` y `nip` introducidos por el usuario a la función miembro `autenticarUsuario` de `baseDatosBanco`. La clase ATM establece su miembro de datos `usuarioAutenticado` con el valor `bool` devuelto por esta función; `usuarioAutenticado` se vuelve `true` si la autenticación tiene éxito (es decir, que el `numeroCuenta` y el `nip` coincidan con los de una `Cuenta` existente en la `baseDatosBanco`) y permanece `false` en caso contrario. Si `usuarioAutenticado` es `true`, en la línea 55 se guarda el número de cuenta introducido por el usuario (es decir, `numeroCuenta`) en el miembro de datos `numeroCuentaActual` de ATM. Las demás funciones miembro de la clase ATM utilizan esta variable cada vez que una sesión con el ATM requiere acceso al número de cuenta del usuario. Si `usuarioAutenticado` es `false`, en las líneas 58 y 59 se utiliza la función miembro `mostrarLineaMensaje` de `pantalla` para indicar que se introdujo un número de cuenta y/o NIP inválidos, por lo que el usuario debe intentar otra vez. Observe que establecemos `numeroCuentaUsuario` sólo después de autenticar el número de cuenta del usuario y su NIP asociado; si la base de datos no puede autenticar al usuario, `numeroCuentaUsuario` permanece en 0.

Una vez que la función miembro `ejecutar` trata de autenticar al usuario (línea 30), si `usuarioAutenticado` sigue siendo `false`, el ciclo `while` de las líneas 27 a 31 se ejecuta otra vez. Si `usuarioAutenticado` es ahora `true`, el ciclo termina y el control continúa con la línea 33, en donde se hace una llamada a la función utilitaria `realizarTransacciones` de la clase ATM.

La función miembro `realizarTransacciones` (líneas 63 a 103) lleva a cabo una sesión con el ATM para un usuario autenticado. En la línea 66 se declara un apuntador `Transaccion` local, el cual apuntamos a un objeto `SolicitudSaldo, Retiro o Deposito` que representa la transacción con el ATM que se está procesando en ese momento. Observe que aquí utilizamos un apuntador `Transaccion` para poder aprovechar el polimorfismo. Además, observe que utilizamos el nombre del rol incluido en el diagrama de clases de la figura 3.20 (`transaccionActual`) para nombrar a este apuntador. Siguiendo nuestra convención de nombramiento de apuntadores, agregamos “`Ptr`” al nombre del rol para formar el nombre de la variable `transaccionActualPtr`. En la línea 68 se declara otra variable local: una variable `bool` llamada `usuarioSalio`, la cual contiene información acerca de si el usuario ha elegido salir de su sesión con el ATM. Esta variable controla un ciclo `while` (líneas 71 a 102) que permite al usuario ejecutar un número ilimitado de transacciones, antes de que elija la opción para salir. Dentro de este ciclo, en la línea 74 se muestra el menú principal y se obtiene la opción del menú que eligió el usuario, llamando a una función utilitaria de ATM llamada `mostrarMenuPrincipal` (definida en las líneas 106 a 115). Esta función miembro muestra el menú principal, invocando a las funciones miembro de la `pantalla` del ATM, y devuelve una selección del menú que obtiene del usuario a través del `teclado` del ATM. Observe que esta función miembro es `const` porque no modifica los contenidos del objeto. En la línea 74 se almacena la selección del usuario devuelta por `mostrarMenuPrincipal`, en la variable local `seleccionMenuPrincipal`.

Después de obtener una selección del menú principal, la función miembro `realizarTransacciones` utiliza una instrucción `switch` (líneas 77 a 101) para responder a la selección en forma apropiada. Si `seleccionMenuPrincipal` es igual a cualquiera de las tres constantes de enumeración que representan tipos de transacciones (es decir, si el usuario eligió realizar una transacción), en las líneas 84 y 85 se hace una llamada a la función utilitaria `crearTransaccion` (definida en las líneas 118 a 140) para devolver un apuntador a un objeto recién instanciado del tipo que corresponde a la transacción seleccionada. Al apuntador `transaccionActualPtr` se le asigna el apuntador devuelto por `crearTransaccion`. Después, en la línea 87 se utiliza `transaccionActualPtr` para invocar a la función miembro `ejecutar` del nuevo objeto para ejecutar la transacción. En breve hablaremos sobre la función miembro `ejecutar` de `Transaccion` y las tres clases derivadas de `Transaccion`. Por último, cuando el objeto de la clase derivada de `Transaccion` ya no se necesita, en la línea 90 se libera la memoria que tiene asignada en forma dinámica.

Observe que orientamos el apuntador `Transaccion` llamado `transaccionActualPtr` a un objeto de una de las tres clases derivadas de `Transaccion`, para poder ejecutar transacciones mediante el polimorfismo. Por ejemplo, si el usuario elige realizar una solicitud de saldo, `seleccionMenuPrincipal` coincide con `SOLICITUD_SALDO`, lo cual hace que `crearTransaccion` devuelva un apuntador a un objeto `SolicitudSaldo`. Por ende, `transaccionActualPtr` apunta a un objeto `SolicitudSaldo` y la invocación `transaccionActual->ejecutar()` produce una llamada a la versión de `ejecutar` correspondiente a `SolicitudSaldo`.

La función miembro `crearTransaccion` (líneas 118 a 140) utiliza una instrucción `switch` (líneas 123 a 137) para instanciar un nuevo objeto de la clase derivada `Transaccion`, del tipo indicado por el parámetro `tipo`. Recuerde que la función miembro `realizarTransacciones` pasa el valor de `seleccionMenuPrincipal` a esta función miembro sólo cuando `seleccionMenuPrincipal` contiene un valor que corresponde a uno de los tres tipos de transacciones. Por lo tanto, `tipo` es igual a `SOLICITUD_SALDO`, `RETIRO` o `DEPOSITO`. Cada `case` en la instrucción `switch` orienta el apuntador temporal `tempPtr` a un objeto recién creado de la clase derivada de `Transaccion` que sea apropiada. Observe que cada constructor tiene una lista de parámetros única, basada en los datos específicos que se requieren para inicializar el objeto de la clase derivada. Un objeto `SolicitudSaldo` requiere sólo el número de cuenta del usuario actual, y hace referencia a la `pantalla` y la `baseDatosBanco` del ATM. Además de estos parámetros, un objeto `Retiro` requiere referencias al `teclado` y `dispensadorEfectivo` del ATM, y un objeto `Deposito` requiere referencias al `teclado` y la `ranuraDeposito` del ATM. Como pronto veremos, cada uno de los constructores de `SolicitudSaldo`, `Retiro` y `Deposito` especifican parámetros de referencia para recibir los objetos que representan las partes requeridas del ATM. Así, cuando la función miembro `crearTransaccion` pasa objetos en el ATM (por ejemplo, `pantalla` y `teclado`) al inicializador para cada objeto recién creado de una clase derivada de `Transaccion`, el nuevo objeto en realidad recibe *referencias* a los objetos compuestos del ATM. En las secciones G.9 a G.12 hablaremos sobre las clases de transacciones con más detalle.

Después de ejecutar una transacción (línea 87 en `realizarTransacciones`), `usuarioSalio` sigue siendo `false` y se repite el ciclo `while` en las líneas 71 a 102, con lo cual el usuario regresa al menú principal. No obstante, si un usuario no realiza una transacción y selecciona la opción del menú principal para salir, en la línea 95 se establece `usuarioSalio` a `true`, haciendo que la condición del ciclo `while` (`!usuarioSalio`) sea `false`. Esta instrucción `while` es la instrucción final de la función miembro `realizarTransacciones`, por lo que el control regresa a la función `ejecutar` que hizo la llamada. Si el usuario introduce una selección inválida del menú principal (por decir, un número que no sea entero en el rango de 1 a 4), en las líneas 98 y 99 se muestra un mensaje de error apropiado, `usuarioSalio` sigue siendo `false` y el usuario regresa al menú principal para intentar de nuevo.

Cuando `realizarTransacciones` devuelve el control a la función miembro `ejecutar`, el usuario ha elegido salir del sistema, por lo que en las líneas 34 y 35 se restablecen los miembros de datos `usuarioAutenticado` y `numeroCuentaActual` del ATM, como parte del proceso de preparación para el siguiente usuario del ATM. En la línea 36 se muestra un mensaje de despedida antes de que el ATM empiece otra vez y dé la bienvenida al siguiente usuario.

G.3 La clase Pantalla

La clase `Pantalla` (figuras G.3 y G.4) representa la pantalla del ATM y encapsula todos los aspectos relacionados con el proceso de mostrar los resultados al usuario. La clase `Pantalla` simula a la pantalla de un ATM real con el monitor de la computadora, e imprime en pantalla mensajes de texto mediante el uso de `cout` y el operador inserción de flujo (`<<`). En este ejemplo práctico diseñamos la clase `Pantalla` con una operación: `mostrarMensaje`. Para obtener una mayor flexibilidad al mostrar mensajes en la `Pantalla`, ahora declaramos tres funciones miembro de `Pantalla`: `mostrarMensaje`, `mostrarLineaMensaje` y `mostrarMontoEnDolares`. Los prototipos para estas funciones miembro aparecen en las líneas 12 a 14 de la figura G.3.

Definiciones de las funciones miembro de la clase `Pantalla`

La figura G.4 contiene las definiciones de las funciones miembro de la clase `Pantalla`. En la línea 11 se incluye (mediante `#include`) la definición de la clase `Pantalla`. La función miembro `mostrarMensaje` (líneas 14 a 17) recibe un objeto `string` como argumento y lo imprime en la consola mediante `cout` y el operador inserción de flujo (`<<`). El cursor permanece en la misma línea, por lo cual esta función miembro es apropiada para mostrar indicadores al usuario. La función miembro `mostrarLineaMensaje` (líneas 20 a 23) también imprime un objeto `string`, pero imprime una nueva línea para desplazar el cursor a la siguiente línea. Por último, la función miembro `mostrarMontoEnDolares` (líneas 26 a 29) imprime en pantalla un monto en dólares con el formato apropiado (por ejemplo, `$123.45`). En la línea 28 se utilizan los manipuladores de flujo `fixed` y `setprecision` para imprimir un valor con formato y dos posiciones decimales. En el capítulo 15, Entrada y salida de flujos, podrá obtener más información acerca de cómo aplicar formato a la salida.

```

1 // Pantalla.h
2 // Definición de la clase Pantalla. Representa la pantalla del ATM.
3 #ifndef PANTALLA_H
4 #define PANTALLA_H
5
6 #include <string>
7 using std::string;
8
9 class Pantalla
10 {
11 public:
12     void mostrarMensaje( string ) const; // imprime un mensaje en pantalla
13     void mostrarLineaMensaje( string ) const; // imprime un mensaje con nueva línea
14     void mostrarMontoDolares( double ) const; // imprime un monto en dólares
15 }; // fin de la clase Pantalla
16
17 #endif // PANTALLA_H

```

Figura G.3 | Definición de la clase Pantalla.

```

1 // Pantalla.cpp
2 // Definiciones de las funciones miembro para la clase Pantalla.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 #include "Pantalla.h" // Definición de la clase Pantalla
12
13 // imprime un mensaje sin nueva línea
14 void Pantalla::mostrarMensaje( string mensaje ) const
15 {
16     cout << mensaje;
17 } // fin de la función mostrarMensaje
18
19 // imprime un mensaje con una nueva línea
20 void Pantalla::mostrarLineaMensaje( string mensaje ) const
21 {
22     cout << mensaje << endl;
23 } // fin de la función mostrarLineaMensaje
24
25 // imprime un monto en dólares
26 void Pantalla::mostrarMontoDolares( double monto ) const
27 {
28     cout << fixed << setprecision( 2 ) << "$" << monto;
29 } // fin de la función mostrarMontoDolares

```

Figura G.4 | Definiciones de las funciones miembro de la clase Pantalla.

G.4 La clase Teclado

La clase (figuras G.5 y G.6) representa el teclado del ATM, y es responsable de recibir toda la entrada por parte del usuario. Recuerde que estamos simulando este hardware, por lo que utilizamos el teclado de la computadora para simular el teclado del ATM. Un teclado de computadora contiene muchas teclas que no se encuentran en el teclado del ATM. Sin embargo, vamos a suponer que el usuario sólo opriime las teclas en el teclado de computadora que aparecen también en el teclado del ATM: los números del 0 al 9 y la tecla *Intro*. La línea 9 de la figura G.5 contiene el prototipo de la única función miembro de la clase Teclado llamada *obtenerEntrada*. Esta función miembro se declara como *const*, debido a que no modifica el objeto.

```

1 // Teclado.h
2 // Definición de la clase Teclado. Representa el teclado del ATM.
3 #ifndef TECLADO_H
4 #define TECLADO_H
5
6 class Teclado
7 {
8 public:
9     int obtenerEntrada() const; // devuelve un valor entero introducido por el usuario
10}; // fin de la clase Teclado
11
12#endif // TECLADO_H

```

Figura G.5 | Definición de la clase Teclado.

```

1 // Teclado.cpp
2 // Definición de las funciones miembro de la clase Teclado (el teclado del ATM).
3 #include <iostream>
4 using std::cin;
5
6 #include "Teclado.h" // Definición de la clase Teclado
7
8 // devuelve un valor entero introducido por el usuario
9 int Teclado::obtenerEntrada() const
10{
11    int entrada; // variable para almacenar la entrada
12    cin >> entrada; // asumimos que el usuario introduce un entero
13    return entrada; // devuelve el valor introducido por el usuario
14} // fin de la función function obtenerEntrada

```

Figura G.6 | Definición de la función miembro de la clase Teclado.

Definición de la función miembro de la clase Teclado

En el archivo de implementación de Teclado (figura G.6), la función miembro `obtenerEntrada` (definida en las líneas 9 a 14) utiliza el flujo de entrada estándar `cin` y el operador extracción de flujo (`>>`) para obtener la entrada del usuario. En la línea 11 se declara una variable local para almacenar la entrada del usuario. En la línea 12 se lee la entrada y se coloca en la variable local `entrada`, y después en la línea 13 se devuelve este valor. Recuerde que la función `obtenerEntrada` obtiene toda la entrada utilizada por el ATM. La función miembro `obtenerEntrada` de `Teclado` simplemente devuelve el entero introducido por el usuario. Si un cliente de la clase `Teclado` requiere entrada que cumpla con ciertos criterios específicos (es decir, un número que corresponda con una opción válida del menú), el cliente debe realizar la comprobación de errores apropiada. [Nota: el uso del flujo de entrada estándar `cin` y del operador extracción de flujo (`>>`) permite que se lean datos de entrada del usuario que no sean enteros. Como el teclado real del ATM sólo permite introducir enteros, vamos a suponer que el usuario introducirá un entero y no intentaremos corregir los problemas ocasionados por la introducción de valores que no sean de este tipo].

G.5 La clase DispensadorEfectivo

La clase `DispensadorEfectivo` (figura G.7 y G.8) representa el dispensador de efectivo del ATM. La definición de la clase (figura G.7) contiene el prototipo de función para un constructor predeterminado (línea 9). La clase `DispensadorEfectivo` declara dos funciones miembro `public` adicionales: `dispensarEfectivo` (línea 12) y `haySuficienteEfectivoDisponible` (línea 15). La clase confía en que un cliente (es decir, `Retiro`) llama a `dispensarEfectivo` sólo después de establecer que hay suficiente efectivo disponible, para lo cual se hace una llamada a `haySuficienteEfectivoDisponible`. Por lo tanto, `dispensarEfectivo` simplemente simula el proceso de dispensar el monto solicitado sin comprobar que haya suficiente efectivo disponible. En la línea 17 se declara la constante `private` llamada `CUENTA_INICIAL`, la cual indica la cuenta inicial de billetes en el dispensador de efectivo cuando el ATM inicia sus operaciones (es decir, 500). En la línea 18 se implementa el atributo `cuenta` (modelado en la figura 13.29), que lleva la cuenta del número de billetes restantes en el `DispensadorEfectivo`, en cualquier momento dado.

Definiciones de las funciones miembro de la clase **DispensadorEfectivo**

La figura G.8 contiene las definiciones de las funciones miembro de la clase **DispensadorEfectivo**. El constructor (líneas 6 a 9) establece **cuenta** con la cuenta inicial (es decir, 500). La función miembro **dispensarEfectivo** (líneas 13 a 17) simula el proceso de dispensar el efectivo. Si nuestro sistema estuviera conectado a un dispensador de efectivo de hardware real, esta función miembro interactuaría con el dispositivo de hardware para dispensar físicamente el efectivo. Nuestra versión simulada de la función miembro simplemente resta a la cuenta de billetes restantes el número requerido para dispensar el **monto** especificado (línea 16). Hay que tener en cuenta que en la línea 15 se calcula el número de billetes de \$20 requeridos para dispensar el **monto** especificado. El ATM permite al usuario elegir sólo montos de retiro que sean múltiplos de \$20, por lo que dividimos **monto** entre 20 para obtener el número de **billetesRequeridos**. También hay que tener en cuenta que es responsabilidad del cliente de la clase (es decir, **Retiro**) informar al usuario que se ha dispensado el efectivo; **DispensadorEfectivo** no puede interactuar directamente con la Pantalla.

```

1 // DispensadorEfectivo.h
2 // Definición de la clase DispensadorEfectivo. Representa el dispensador de efectivo del ATM.
3 #ifndef DISPENSADOR_EFECTIVO_H
4 #define DISPENSADOR_EFECTIVO_H
5
6 class DispensadorEfectivo
7 {
8 public:
9     DispensadorEfectivo(); // el constructor inicializa la cuenta de billetes con 500
10
11    // simula el proceso de dispensar el monto especificado de efectivo
12    void dispensarEfectivo( int );
13
14    // indica si el dispensador de efectivo puede dispensar el monto deseado
15    bool haySuficienteEfectivoDisponible( int ) const;
16 private:
17    const static int CUENTA_INICIAL = 500;
18    int cuenta; // número de billetes de $20 restantes
19 }; // fin de la clase DispensadorEfectivo
20
21 #endif // DISPENSADOR_EFECTIVO_H

```

Figura G.7 | Definición de la clase **DispensadorEfectivo**.

```

1 // DispensadorEfectivo.cpp
2 // Definiciones de las funciones miembro para la clase DispensadorEfectivo.
3 #include "DispensadorEfectivo.h" // Definición de la clase DispensadorEfectivo
4
5 // el constructor predeterminado de DispensadorEfectivo inicializa la cuenta al valor
// predeterminado
6 DispensadorEfectivo::DispensadorEfectivo()
7 {
8     cuenta = CUENTA_INICIAL; // establece el atributo cuenta con el valor predeterminado
9 } // fin del constructor predeterminado de DispensadorEfectivo
10
11 // simula el proceso de dispensar el monto especificado de efectivo; asume que hay suficiente
// efectivo
12 // disponible (la llamada anterior a haySuficienteEfectivoDisponible devolvió true)
13 void DispensadorEfectivo::dispensarEfectivo( int monto )
14 {
15     int billetesRequeridos = monto / 20; // número de billetes de $20 requeridos
16     cuenta -= billetesRequeridos; // actualiza la cuenta de billetes
17 } // fin de la función dispensarEfectivo
18

```

Figura G.8 | Definiciones de las funciones miembro de la clase **DispensadorEfectivo**. (Parte I de 2).

```

19 // indica si el dispensador de efectivo puede dispensar el monto deseado
20 bool DispensadorEfectivo::haySuficienteEfectivoDisponible( int monto ) const
21 {
22     int billetesRequeridos = monto / 20; // número de billetes de $20 requeridos
23
24     if ( cuenta >= billetesRequeridos )
25         return true; // hay suficientes billetes disponibles
26     else
27         return false; // no hay suficientes billetes disponibles
28 } // fin de la función haySuficienteEfectivoDisponible

```

Figura G.8 | Definiciones de las funciones miembro de la clase `DispensadorEfectivo`. (Parte 2 de 2).

La función miembro `haySuficienteEfectivoDisponible` (líneas 20 a 28) tiene un parámetro llamado `monto` que especifica el monto de efectivo en cuestión. En las líneas 24 a 27 se devuelve `true` si la cuenta del `DispensadorEfectivo` es mayor o igual que `billetesRequeridos` (es decir, que haya suficientes billetes disponibles) y devuelve `false` en caso contrario (es decir, que no haya suficientes billetes). Por ejemplo, si un usuario desea retirar \$80 (es decir, `billetesRequeridos` es 4) pero sólo quedan tres billetes (`cuenta` = 3), la función miembro devuelve `false`.

G.6 La clase RanuraDeposito

La clase `RanuraDeposito` (figuras G.9 y G.10) representa la ranura de depósito del ATM. Al igual que la versión de la clase `DispensadorEfectivo` que se presenta en este apéndice, esta versión de la clase `RanuraDeposito` simplemente simula la funcionalidad de una ranura de depósito de hardware real. `RanuraDeposito` no tiene miembros de datos y sólo tiene una función miembro: `seRecibioSobreDeposito` (declarada en la línea 9 de la figura G.9 y definida en las líneas 7 a 10 de la figura G.10), la cual indica si se recibió o no un sobre de depósito.

En la especificación de requerimientos vimos que el ATM permite al usuario hasta dos minutos para insertar un sobre. La versión actual de la función miembro `seRecibioSobreDeposito` simplemente devuelve `true` de inmediato (línea 9 de la figura G.10), ya que ésta es sólo una simulación de software, por lo que se asume que el usuario inserta un sobre dentro del rango de tiempo requerido. Si se conectara una ranura de depósito real en nuestro sistema, la función miembro `seRecibioSobreDeposito` podría implementarse de manera que esperara un máximo de dos minutos para

```

1 // RanuraDeposito.h
2 // Definición de la clase RanuraDeposito. Representa la ranura de depósito del ATM.
3 #ifndef RANURA_DEPOSITO_H
4 #define RANURA_DEPOSITO_H
5
6 class RanuraDeposito
7 {
8 public:
9     bool seRecibioSobre() const; // indica si se recibió el sobre o no
10 }; // fin de la clase RanuraDeposito
11
12 #endif // RANURA_DEPOSITO_H

```

Figura G.9 | Definición de la clase `RanuraDeposito`.

```

1 // RanuraDeposito.cpp
2 // Definición de las funciones miembro de la clase RanuraDeposito.
3 #include "RanuraDeposito.h" // definición de la clase RanuraDeposito
4
5 // indica si se recibió el sobre (siempre devuelve true, ya que ésta
6 // sólo es una simulación de software de una ranura de depósito real)
7 bool RanuraDeposito::seRecibioSobre() const
8 {
9     return true; // se recibió el sobre de depósito
10 } // fin de la función seRecibioSobre

```

Figura G.10 | Definición de la función miembro de la clase `RanuraDeposito`.

recibir una señal de la ranura de depósito de hardware, indicando que el usuario verdaderamente ha insertado un sobre de depósito. Si `seRecibioSobreDeposito` fuera a recibir dicha señal dentro de un plazo no mayor a dos minutos, devolvería `true`. Si pasaran dos minutos y el método no hubiera recibido una señal, entonces devolvería `false`.

G.7 La clase Cuenta

La clase `Cuenta` (figuras G.11 y G.12) representa una cuenta de banco. En las líneas 9 a 15 de la definición de la clase (figura G.11) hay prototipos de función para el constructor de la clase y seis funciones miembro, que describiremos en breve. Cada `Cuenta` tiene cuatro atributos (que se modelan en la figura 13.29): `numeroCuenta`, `nip`, `saldoDisponible` y `saldoTotal`. En las líneas 17 a 20 se implementan estos atributos como miembros de datos `private`. El miembro de datos `saldoDisponible` representa el monto de los fondos disponibles para retiro. El miembro de datos `saldoTotal` representa el monto de fondos disponibles, más el monto de los fondos depositados cuya verificación o confirmación está pendiente.

Definiciones de las funciones miembro de la clase `Cuenta`

La figura G.12 presenta las definiciones de las funciones miembro de la clase `Cuenta`. El constructor de la clase (líneas 6 a 14) recibe un número de cuenta, el NIP establecido para la cuenta, el saldo inicial disponible y el saldo total inicial como argumentos. En las líneas 8 a 11 se asignan estos valores a los miembros de datos de la clase, usando inicializadores de miembros.

La función miembro `validarNIP` (líneas 17 a 23) determina si un NIP especificado por el usuario (es decir, el parámetro `nipUsuario`) coincide con el NIP asociado con la cuenta (es decir, el miembro de datos `nip`). Recuerde que modelamos el parámetro `nipUsuario` de esta función miembro en el diagrama de clases de UML de la figura 6.37. Si los dos NIPs coinciden, la función miembro devuelve `true` (línea 20); en caso contrario devuelve `false` (línea 22).

Las funciones miembro `obtenerSaldoDisponible` (líneas 26 a 29) y `obtenerSaldoTotal` (líneas 32 a 35) son funciones *obtener* que devuelven los valores de los miembros de datos `double` llamados `saldoDisponible` y `saldoTotal`, respectivamente.

```

1 // Cuenta.h
2 // Definición de la clase Cuenta. Representa una cuenta de banco.
3 #ifndef CUENTA_H
4 #define CUENTA_H
5
6 class Cuenta
7 {
8 public:
9     Cuenta( int, int, double, double ); // el constructor establece los atributos
10    bool validarNIP( int ) const; // ¿es correcto el NIP especificado por el usuario?
11    double obtenerSaldoDisponible() const; // devuelve el saldo disponible
12    double obtenerSaldoTotal() const; // devuelve el saldo total
13    void abonar( double ); // suma un monto al saldo de la Cuenta
14    void cargar( double ); // resta un monto del saldo de la Cuenta
15    int obtenerNumeroCuenta() const; // devuelve el número de cuenta
16 private:
17     int numeroCuenta; // número de cuenta
18     int nip; // NIP para autenticación
19     double saldoDisponible; // fondos disponibles para retirar
20     double saldoTotal; // fondos disponibles + fondos esperando ser verificados
21 }; // fin de la clase Cuenta
22
23 #endif // CUENTA_H

```

Figura G.11 | Definición de la clase `Cuenta`.

```

1 // Cuenta.cpp
2 // Definiciones de las funciones miembro de la clase Cuenta.
3 #include "Cuenta.h" // definición de la clase Cuenta
4
5 // el constructor de Cuenta inicializa los atributos
6 Cuenta::Cuenta( int elNumeroDeCuenta, int elNIP,

```

Figura G.12 | Definiciones de las funciones miembro de la clase `Cuenta`. (Parte I de 2).

```

7     double elSaldoDisponible, double elSaldoTotal )
8     : numeroCuenta( elNumeroDeCuenta ),
9     nip( elNIP ),
10    saldoDisponible( elSaldoDisponible ),
11    saldoTotal( elSaldoTotal )
12 {
13     // cuerpo vacío
14 } // fin del constructor de Cuenta
15
16 // determina si un NIP especificado por el usuario coincide con el NIP en la Cuenta
17 bool Cuenta::validarNIP( int nipUsuario ) const
18 {
19     if ( nipUsuario == nip )
20         return true;
21     else
22         return false;
23 } // fin de la función validarNIP
24
25 // devuelve el saldo disponible
26 double Cuenta::obtenerSaldoDisponible() const
27 {
28     return saldoDisponible;
29 } // fin de la función obtenerSaldoDisponible
30
31 // devuelve el saldo total
32 double Cuenta::obtenerSaldoTotal() const
33 {
34     return saldoTotal;
35 } // fin de la función obtenerSaldoTotal
36
37 // abona un monto a la cuenta
38 void Cuenta::abonar( double monto )
39 {
40     saldoTotal += monto; // lo suma al saldo total
41 } // fin de la función abonar
42
43 // carga un monto a la cuenta
44 void Cuenta::cargar( double monto )
45 {
46     saldoDisponible -= monto; // resta del saldo disponible
47     saldoTotal -= monto; // resta del saldo total
48 } // fin de la función cargar
49
50 // devuelve el número de cuenta
51 int Cuenta::obtenerNumeroCuenta() const
52 {
53     return numeroCuenta;
54 } // fin de la función obtenerNumeroCuenta

```

Figura G.12 | Definiciones de las funciones miembro de la clase Cuenta. (Parte 2 de 2).

La función miembro `abonar` (líneas 38 a 41) suma un monto de dinero (es decir, el parámetro `monto`) a una `Cuenta` como parte de una transacción de depósito. Observe que esta función miembro suma el `monto` sólo al miembro de datos `saldoTotal` (línea 40). El dinero abonado a una cuenta durante un depósito no se hace disponible de inmediato, por lo que sólo modificamos el saldo total. Estamos suponiendo que el banco actualiza el saldo disponible en forma apropiada más adelante. Nuestra implementación de la clase `Cuenta` sólo incluye las funciones miembro requeridas para llevar a cabo transacciones con el ATM. Por lo tanto, omitimos las funciones miembro que invocaría algún otro sistema bancario para sumar un monto al miembro de datos `saldoDisponible` (para confirmar un depósito), o restar un monto al miembro de datos `saldoTotal` (para rechazar un depósito).

La función miembro `cargar` (líneas 44 a 48) resta un monto de dinero (es decir, el parámetro `monto`) de una `Cuenta`, como parte de una transacción de retiro. Este método resta el `monto` tanto del miembro de datos `saldoDisponible` (línea 46), como del miembro de datos `saldoTotal` (línea 47), ya que un retiro afecta en ambos saldos de una cuenta.

La función miembro `obtenerNumeroCuenta` (líneas 51 a 54) proporciona acceso al `numeroCuenta` de una `Cuenta`. Incluimos esta función miembro en nuestra implementación para que un cliente de la clase (es decir, `BaseDatosBanco`) pueda identificar una `Cuenta` específica. Por ejemplo, `BaseDatosBanco` contiene muchos objetos `Cuenta`, y puede invocar a esta función miembro en cada uno de sus objetos `Cuenta` para localizar el objeto que tenga un número de cuenta específico.

G.8 La clase BaseDatosBanco

La clase `BaseDatosBanco` (figuras G.13 y G.14) modela la base de datos del banco con la que el ATM interactúa para acceder a la información de la cuenta de un usuario y modificarla. La definición de la clase (figura G.13) declara los prototipos de función para el constructor de la clase y varias funciones miembro. Hablaremos sobre éstas en unos momentos. La definición de la clase también declara los miembros de datos de `BaseDatosBanco`. Determinamos un miembro de datos para la clase `BaseDatosBanco` con base en su relación de composición con la clase `Cuenta`. En la figura 13.28 vimos que `BaseDatosBanco` está compuesta de cero o más objetos de la clase `Cuenta`. En la línea 24 de la figura G.13 se implementa el miembro de datos `cuentas` (un vector de objetos `Cuenta`) para implementar esta relación de composición. En las líneas 6 y 7 podemos utilizar a `vector` en este archivo. La línea 27 contiene el prototipo para una función utilitaria `private` llamada `obtenerCuenta`, que permite a las funciones miembro de la clase obtener un apuntador a un objeto `Cuenta` específico en el vector `cuentas`.

```

1 // BaseDatosBanco.h
2 // Definición de la clase BaseDatosBanco. Representa la base de datos del banco.
3 #ifndef BASEDATOS_BANCO_H
4 #define BASEDATOS_BANCO_H
5
6 #include <vector> // la clase utiliza un vector para almacenar objetos Cuenta
7 using std::vector;
8
9 #include "Cuenta.h" // definición de la clase Cuenta
10
11 class BaseDatosBanco
12 {
13 public:
14     BaseDatosBanco(); // el constructor inicializa las cuentas
15
16     // determina si el número de cuenta y el NIP coinciden con los de una Cuenta
17     bool autenticarUsuario( int, int ); // devuelve true si la Cuenta es auténtica
18
19     double obtenerSaldoDisponible( int ); // obtiene el saldo disponible
20     double obtenerSaldoTotal( int ); // obtiene el saldo total de una Cuenta
21     void abonar( int, double ); // suma el monto al saldo de la Cuenta
22     void cargar( int, double ); // resta el monto del saldo de la Cuenta
23 private:
24     vector< Cuenta > cuentas; // vector de las Cuentas del banco
25
26     // función utilitaria privada
27     Cuenta * obtenerCuenta( int ); // obtiene el apuntador al objeto Cuenta
28 }; // fin de la clase BaseDatosBanco
29
30 #endif // BASEDATOS_BANCO_H

```

Figura G.13 | Definición de la clase `BaseDatosBanco`.

Definiciones de las funciones miembro de la clase `BaseDatosBanco`

La figura G.14 contiene las definiciones de las funciones miembro de la clase `BaseDatosBanco`. Implementamos la clase con un constructor predeterminado (líneas 6 a 15) que agrega objetos `Cuenta` al miembro de datos `cuentas`. Para los fines de evaluar el sistema, creamos dos nuevos objetos `Cuenta` con datos de prueba (líneas 9 y 10) y después los agregamos al final del vector (líneas 13 y 14). Observe que el constructor de `Cuenta` tiene cuatro parámetros: el número de cuenta, el NIP asignado a esa cuenta, el saldo inicial disponible y el saldo inicial total.

1112 Apéndice G Código del caso de estudio del ATM

```
1 // BaseDatosBanco.cpp
2 // Definiciones de las funciones miembro de la clase BaseDatosBanco.
3 #include "BaseDatosBanco.h" // Definición de la clase BaseDatosBanco
4
5 // el constructor predeterminado de BaseDatosBanco inicializa las cuentas
6 BaseDatosBanco::BaseDatosBanco()
7 {
8     // crea dos objetos Cuenta para evaluar
9     Cuenta cuenta1( 12345, 54321, 1000.0, 1200.0 );
10    Cuenta cuenta2( 98765, 56789, 200.0, 200.0 );
11
12    // agrega los objetos Cuenta al vector cuentas
13    cuentas.push_back( cuenta1 ); // agrega cuenta1 al final del vector
14    cuentas.push_back( cuenta2 ); // agrega cuenta2 al final del vector
15 } // fin del constructor predeterminado de BaseDatosBanco
16
17 // obtiene el objeto Cuenta que contiene el número de cuenta especificado
18 Cuenta * BaseDatosBanco::obtenerCuenta( int numeroCuenta )
19 {
20     // itera a través de cuentas en busca de un número de cuentas que coincida
21     for ( size_t i = 0; i < cuentas.size(); i++ )
22     {
23         // devuelve la cuenta actual si se encontró una coincidencia
24         if ( cuentas[ i ].obtenerNumeroCuenta() == numeroCuenta )
25             return &cuentas[ i ];
26     } // fin de for
27
28     return NULL; // si no encontró una cuenta que coincida, devuelve NULL
29 } // fin de la función obtenerCuenta
30
31 // determina si el número de cuenta y NIP especificados por el usuario coinciden
32 // con los de una cuenta en la base de datos
33 bool BaseDatosBanco::autenticarUsuario( int numeroCuentaUsuario,
34                                         int nipUsuario )
35 {
36     // intenta obtener la cuenta con el número de cuentas
37     Cuenta * const cuentaUsuarioPtr = obtenerCuenta( numeroCuentaUsuario );
38
39     // si la cuenta existe, devuelve el resultado de la función validarNIP de Cuenta
40     if ( cuentaUsuarioPtr != NULL )
41         return cuentaUsuarioPtr->validarNIP( nipUsuario );
42     else
43         return false; // no se encontró el número de cuenta, por lo que devuelve false
44 } // fin de la función autenticarUsuario
45
46 // devuelve el saldo disponible de la Cuenta con el número de cuenta especificado
47 double BaseDatosBanco::obtenerSaldoDisponible( int numeroCuentaUsuario )
48 {
49     Cuenta * const cuentaUsuarioPtr = obtenerCuenta( numeroCuentaUsuario );
50     return cuentaUsuarioPtr->obtenerSaldoDisponible();
51 } // fin de la función obtenerSaldoDisponible
52
53 // devuelve el saldo total de la Cuenta con el número de cuenta especificado
54 double BaseDatosBanco::obtenerSaldoTotal( int numeroCuentaUsuario )
55 {
56     Cuenta * const cuentaUsuarioPtr = obtenerCuenta( numeroCuentaUsuario );
57     return cuentaUsuarioPtr->obtenerSaldoTotal();
58 } // fin de la función obtenerSaldoTotal
59
60 // abona un monto a la Cuenta con el número de cuenta especificado
61 void BaseDatosBanco::abonar( int numeroCuentaUsuario, double monto )
```

Figura G.14 | Definiciones de las funciones miembro de la clase BaseDatosBanco. (Parte I de 2).

```

62  {
63      Cuenta * const cuentaUsuarioPtr = obtenerCuenta( numeroCuentaUsuario );
64      cuentaUsuarioPtr->abonar( monto );
65  } // fin de la función abonar
66
67  // carga un monto a la Cuenta con el número de cuenta especificado
68  void BaseDatosBanco::cargar( int numeroCuentaUsuario, double monto )
69  {
70      Cuenta * const cuentaUsuarioPtr = obtenerCuenta( numeroCuentaUsuario );
71      cuentaUsuarioPtr->cargar( monto );
72  } // fin de la función cargar

```

Figura G.14 | Definiciones de las funciones miembro de la clase `BaseDatosBanco`. (Parte 2 de 2).

Recuerde que la clase `BaseDatosBanco` sirve como intermediario entre la clase `ATM` y los objetos `Cuenta` actuales que contienen la información de las cuentas de los usuarios. Por ende, las funciones miembro de la clase `BaseDatosBanco` no hacen nada más que invocar a las funciones miembro correspondientes del objeto `Cuenta` que pertenece al usuario actual del `ATM`.

Incluimos la función utilitaria `private` llamada `obtenerCuenta` (líneas 18 a 29) para permitir que la `BaseDatosBanco` obtenga un apuntador a una `Cuenta` específica dentro del vector `cuentas`. Para localizar la `Cuenta` del usuario, la `BaseDatosBanco` compara el valor devuelto por la función miembro `obtenerNumeroCuenta` para cada elemento de `cuentas` con un número de cuenta especificado, hasta que encuentra una coincidencia. En las líneas 21 a 26 se recorre el vector `cuentas`. Si el número de la `Cuenta` actual (es decir, `cuentas[i]`) es igual al valor del parámetro `numeroCuenta`, la función miembro devuelve de inmediato la dirección de la `Cuenta` actual (es decir, un apuntador a la `Cuenta` actual). Si ninguna cuenta tiene el número de cuenta dado, entonces en la línea 28 se devuelve `NULL`. Observe que esta función miembro debe devolver un apuntador en vez de una referencia, ya que existe la posibilidad de que el valor de retorno sea `NULL`; una referencia no puede ser `NULL`, pero un apuntador sí.

Observe que la función `size` de `vector` (que se invoca en la condición de continuación de ciclo de la línea 21) devuelve el número de elementos en un `vector` como un valor de tipo `size_t` (que por lo general es `unsigned int`). Como resultado, declaramos la variable de control `i` para que sea del tipo `size_t` también. En algunos compiladores, si declaramos `i` como `int` el compilador emitirá un mensaje de advertencia, debido a que la condición de continuación de ciclo compararía un valor `signed` (es decir, un `int`) con un valor `unsigned` (es decir, un valor de tipo `size_t`).

La función miembro `autenticarUsuario` (líneas 33 a 44) aprueba o desaprueba la identidad de un usuario del `ATM`. Esta función recibe un número de cuenta y un NIP especificados por el usuario como argumentos, e indica si coinciden con el número de cuenta y el NIP de una `Cuenta` en la base de datos. En la línea 37 se hace una llamada a la función utilitaria `obtenerCuenta`, la cual devuelve un apuntador a una `Cuenta` con `numeroCuentaUsuario` como su número de cuenta, o `NULL` para indicar que el `numeroCuentaUsuario` es inválido. Declaramos `cuentaUsuarioPtr` como un apuntador `const` debido a que, una vez que la función miembro oriente este apuntador a la `Cuenta` del usuario, el apuntador no debe cambiar. Si `obtenerCuenta` devuelve un apuntador a un objeto `Cuenta`, en la línea 41 se devuelve el valor `bool` devuelto por la función miembro `validarNIP` de ese objeto. Observe que la función miembro `autenticarUsuario` de `BaseDatosBanco` no realiza la comparación de NIPs por sí sola; en vez de ello, envía `nipUsuario` a la función miembro `validarNIP` del objeto `Cuenta` para hacerlo. El valor devuelto por la función miembro `validarNIP` de `Cuenta` indica si el NIP especificado por el usuario coincide con el NIP de la `Cuenta` del usuario, por lo que la función miembro `autenticarUsuario` simplemente devuelve este valor al cliente de la clase (es decir, `ATM`).

`BaseDatosBanco` confía en que el `ATM` invoque a la función miembro `autenticarUsuario` y reciba un valor de retorno de `true` antes de permitir que el usuario realice transacciones. `BaseDatosBanco` también confía en que cada objeto `Transaccion` creado por el `ATM` contiene el número de cuenta válido del usuario actual autenticado, y que éste es el número de cuenta que se pasa a las funciones miembro restantes de `BaseDatosBanco` como el argumento `numeroCuentaUsuario`. Por lo tanto, las funciones miembro `obtenerSaldoDisponible` (líneas 47 a 51), `obtenerSaldoTotal` (líneas 54 a 58), `abonar` (líneas 61 a 65) y `cargar` (líneas 68 a 72) obtienen un apuntador al objeto `Cuenta` del usuario con la función utilitaria `obtenerCuenta`, y después utilizan este apuntador para invocar a la función miembro apropiada de `Cuenta` en el objeto `Cuenta` del usuario. Sabemos que las llamadas a `obtenerCuenta` dentro de estas funciones miembro nunca devolverán `NULL`, debido a que `numeroCuentaUsuario` debe hacer referencia a una `Cuenta` existente. Observe que `obtenerSaldoDisponible` y `obtenerSaldoTotal` devuelven los valores devueltos por las funciones miembro correspondientes de `Cuenta`. Observe además que `abonar` y `cargar` simplemente redirigen el parámetro `monto` a las funciones miembro de `Cuenta` que invocan.

G.9 La clase Transaccion

La clase `Transaccion` (figuras G.15 y G.16) es una clase base abstracta que representa la noción de una transacción del ATM. Contiene las características comunes de las clases derivadas `SolicitudSaldo`, `Retiro` y `Deposito`. La figura G.15 se expande con base en el archivo de encabezado de `Transaccion` que se desarrolló por primera vez en la sección 13.10. Las líneas 13, 17 a 19 y 22 contienen los prototipos de función para el constructor de la clase y cuatro funciones miembro, que describiremos en breve. En la línea 15 se define un destructor `virtual` con un cuerpo vacío; esto hace a todos los destructores de las clases derivadas `virtual` (incluso aquellos definidos implícitamente por el compilador) y asegura que los objetos de la clase derivada asignados en forma dinámica se destruyan en forma apropiada cuando se eliminén a través de un apuntador de la clase base. En las líneas 24 a 26 se declaran los miembros de datos `private` de la clase. En el diagrama de clases de la figura 13.29 vimos que la clase `Transaccion` contiene un atributo llamado `numeroCuenta` (implementado en la línea 24) que indica la cuenta involucrada en la `Transaccion`. Derivamos los miembros de datos `pantalla` (línea 25) y `baseDatosBanco` (línea 26) de las asociaciones de la clase `Transaccion` modeladas en la figura 13.28; todas las transacciones requieren acceso a la pantalla del ATM y a la base de datos del banco, por lo que incluimos referencias a `Pantalla` y `BaseDatosBanco` como miembros de datos de la clase `Transaccion`. Como veremos pronto, el constructor de `Transaccion` inicializa estas referencias. Observe que las declaraciones anticipadas en las líneas 6 y 7 indican que el archivo de encabezado contiene referencias a objetos de las clases `Pantalla` y `BaseDatosBanco`, pero que las definiciones de estas clases están fuera del archivo de encabezado.

```

1 // Transaccion.h
2 // Definición de la clase base abstracta Transaccion.
3 #ifndef TRANSACCION_H
4 #define TRANSACCION_H
5
6 class Pantalla; // declaración anticipada de la clase Pantalla
7 class BaseDatosBanco; // declaración anticipada de la clase BaseDatosBanco
8
9 class Transaccion
10 {
11 public:
12     // el constructor inicializa las características comunes de todas las Transacciones
13     Transaccion( int, Pantalla &, BaseDatosBanco & );
14
15     virtual ~Transaccion() { } // destructor virtual con cuerpo vacío
16
17     int obtenerNumeroCuenta() const; // devuelve el número de cuenta
18     Pantalla &obtenerPantalla() const; // devuelve una referencia a la pantalla
19     BaseDatosBanco &obtenerBaseDatosBanco() const; // devuelve una referencia a la base de datos
20
21     // función virtual pura para realizar la transacción
22     virtual void ejecutar() = 0; // se sobrescribe en las clases derivadas
23 private:
24     int numeroCuenta; // indica la cuenta involucrada
25     Pantalla &pantalla; // referencia a la pantalla del ATM
26     BaseDatosBanco &baseDatosBanco; // referencia a la base de datos de información de las
27     cuentas
28 }; // fin de la clase Transaccion
29
#endif // TRANSACCION_H

```

Figura G.15 | Definición de la clase `Transaccion`.

```

1 // Transaccion.cpp
2 // Definiciones de las funciones miembro para la clase Transaccion.
3 #include "Transaccion.h" // Definición de la clase Transaccion
4 #include "Pantalla.h" // Definición de la clase Pantalla
5 #include "BaseDatosBanco.h" // Definición de la clase BaseDatosBanco
6

```

Figura G.16 | Definiciones de las funciones miembro de la clase `Transaccion`. (Parte I de 2)

```

7 // el constructor inicializa las características comunes de todas las Transacciones
8 Transaccion::Transaccion( int numeroCuentaUsuario, Pantalla &pantallaATM,
9     BaseDatosBanco &baseDatosBancoATM )
10    : numeroCuenta( numeroCuentaUsuario ),
11        pantalla( pantallaATM ),
12        baseDatosBanco( baseDatosBancoATM )
13 {
14     // cuerpo vacío
15 } // fin del constructor de Transaccion
16
17 // devuelve el número de cuenta
18 int Transaccion::obtenerNumeroCuenta() const
19 {
20     return numeroCuenta;
21 } // fin de la función obtenerNumeroCuenta
22
23 // devuelve referencia a la pantalla
24 Pantalla &Transaccion::obtenerPantalla() const
25 {
26     return pantalla;
27 } // fin de la función obtenerPantalla
28
29 // devuelve referencia a la base de datos del banco
30 BaseDatosBanco &Transaccion::obtenerBaseDatosBanco() const
31 {
32     return baseDatosBanco;
33 } // fin de la función obtenerBaseDatosBanco

```

Figura G.16 | Definiciones de las funciones miembro de la clase `Transaccion`. (Parte 2 de 2)

La clase `Transaccion` tiene un constructor (declarado en la línea 13 de la figura G.15 y definido en las líneas 8 a 15 de la figura G.16) que recibe el número de cuenta del usuario actual y referencias a la pantalla y la base de datos del ATM como argumentos. Como `Transaccion` es una clase abstracta, este constructor nunca se llamará directamente para instanciar objetos `Transaccion`. En vez de ello, los constructores de las clases derivadas de `Transaccion` utilizarán la sintaxis del inicializador de la clase base para invocar a este constructor.

La clase `Transaccion` tiene tres funciones `obtener public: obtenerNumeroCuenta` (declarada en la línea 17 de la figura G.15 y definida en las líneas 18 a 21 de la figura G.16), `obtenerPantalla` (declarada en la línea 18 de la figura G.15 y definida en las líneas 24 a 27 de la figura G.16) y `obtenerBaseDatosBanco` (declarada en la línea 19 de la figura G.15 y definida en las líneas 30 a 33 de la figura G.16). Las clases derivadas de `Transaccion` heredan estas funciones miembro de `Transaccion` y las utilizan para obtener acceso a los miembros de datos `private` de la clase `Transaccion`.

La clase `Transaccion` también declara una función `virtual` pura llamada `ejecutar` (línea 22 de la figura G.15). No tiene sentido proporcionar una implementación para esta función miembro, debido a que no se puede ejecutar una transacción genérica. Por ende, declaramos esta función miembro como una función `virtual` pura y obligamos a cada clase derivada de `Transaccion` a proporcionar su propia implementación concreta que ejecuta ese tipo específico de transacción.

G.10 La clase SolicitudSaldo

La clase `SolicitudSaldo` (figuras G.17 y G.18) se deriva de la clase base abstracta `Transaccion` y representa una transacción de solicitud de saldo del ATM. `SolicitudSaldo` no tiene miembros de datos propios, pero hereda los miembros de datos `numeroCuenta`, `pantalla` y `baseDatosBanco` de `Transaccion`, a los cuales se puede acceder a través de las funciones `obtener public` de `Transaccion`. Observe que en la línea 6 se incluye (mediante `#include`) la definición de la clase base `Transaccion`. El constructor de `SolicitudSaldo` (declarado en la línea 11 de la figura G.17 y definido en las líneas 8 a 13 de la figura G.18) recibe los argumentos que corresponden a los miembros de datos de `Transaccion`, y simplemente los pasa al constructor de `Transaccion`, utilizando la sintaxis del inicializador de la clase base (línea 10 de la figura G.18). La línea 12 de la figura G.17 contiene el prototipo para la función miembro `ejecutar`, que se requiere para indicar la intención de sobrescribir la función `virtual` pura de la clase base que tiene el mismo nombre.

La clase `SolicitudSaldo` sobrescribe a la función `virtual` pura `ejecutar` de `Transaccion` para proporcionar una implementación concreta (líneas 16 a 37 de la figura G.18) que realice los pasos involucrados en una solicitud de saldo.

1116 Apéndice G Código del caso de estudio del ATM

En las líneas 19 y 20 se obtienen referencias a la base de datos del banco y a la pantalla del ATM, para lo cual se invoca a las funciones miembro heredadas de la clase base `Transaccion`. En las líneas 23 y 24 se obtiene el saldo disponible de la cuenta involucrada, para lo cual se invoca a la función miembro `obtenerSaldoDisponible` de `baseDatosBanco`. Observe que en la línea 24 se utiliza la función miembro heredada `obtenerNumeroCuenta` para obtener el número de cuenta del usuario actual, que a su vez lo pasa a `obtenerSaldoDisponible`. En las líneas 27 y 28 se obtiene el saldo total de la cuenta del usuario actual. En las líneas 31 a 36 se muestra la información del saldo en la pantalla del ATM. Recuerde que `mostrarMontoDolares` recibe un argumento `double` y lo imprime en la pantalla con formato de monto en dólares. Por ejemplo, si el `saldoDisponible` de un usuario es de 700.5, en la línea 33 se imprime \$700.50. Observe que en la línea 36 se inserta una línea en blanco de salida para separar la información del saldo de la salida subsiguiente (es decir, el menú principal repetido por la clase ATM después de ejecutar la `SolicitudSaldo`).

```
1 // SolicitudSaldo.h
2 // Definición de la clase SolicitudSaldo. Representa una solicitud de saldo.
3 #ifndef SOLICITUD_SALDO_H
4 #define SOLICITUD_SALDO_H
5
6 #include "Transaccion.h" // Definición de la clase Transaccion
7
8 class SolicitudSaldo : public Transaccion
9 {
10 public:
11     SolicitudSaldo( int, Pantalla &, BaseDatosBanco & ); // constructor
12     virtual void ejecutar(); // realiza la transacción
13 }; // fin de la clase SolicitudSaldo
14
15 #endif // SOLICITUD_SALDO_H
```

Figura G.17 | Definición de la clase `SolicitudSaldo`.

```
1 // SolicitudSaldo.cpp
2 // Definiciones de las funciones miembro para la clase SolicitudSaldo.
3 #include "SolicitudSaldo.h" // Definición de la clase SolicitudSaldo
4 #include "Pantalla.h" // Definición de la clase Pantalla
5 #include "BaseDatosBanco.h" // Definición de la clase BaseDatosBanco
6
7 // el constructor de SolicitudSaldo inicializa los miembros de datos de la clase base
8 SolicitudSaldo:: SolicitudSaldo( int numeroCuentaUsuario, Pantalla &pantallaATM,
9     BaseDatosBanco &baseDatosBancoATM )
10    : Transaccion( numeroCuentaUsuario, pantallaATM, baseDatosBancoATM )
11 {
12     // cuerpo vacío
13 } // fin del constructor de SolicitudSaldo
14
15 // realiza una transacción; sobrescribe la función virtual pura de Transaccion
16 void SolicitudSaldo::ejecutar()
17 {
18     // obtiene referencias a la base de datos del banco y la pantalla
19     BaseDatosBanco &baseDatosBanco = obtenerBaseDatosBanco();
20     Pantalla &pantalla = obtenerPantalla();
21
22     // obtiene el saldo disponible para la Cuenta del usuario actual
23     double saldoDisponible =
24         baseDatosBanco.obtenerSaldoDisponible( obtenerNumeroCuenta() );
25
26     // obtiene el saldo total para la Cuenta del usuario actual
27     double saldoTotal =
28         baseDatosBanco.obtenerSaldoTotal( obtenerNumeroCuenta() );
29
30     // muestra la información del saldo en la pantalla
```

Figura G.18 | Definiciones de las funciones miembro de la clase `SolicitudSaldo`. (Parte I de 2).

```

31     pantalla.mostrarLineaMensaje( "\nInformación de saldo:" );
32     pantalla.mostrarMensaje( " - Saldo disponible: " );
33     pantalla.mostrarMontoDolares( saldoDisponible );
34     pantalla.mostrarMensaje( "\n - Saldo total: " );
35     pantalla.mostrarMontoDolares( saldoTotal );
36     pantalla.mostrarLineaMensaje( "" );
37 } // fin de la función ejecutar

```

Figura G.18 | Definiciones de las funciones miembro de la clase `SolicitudSaldo`. (Parte 2 de 2).

G.11 La clase Retiro

La clase `Retiro` (figuras G.19 y G.20) se deriva de `Transaccion` y representa una transacción de retiro del ATM. La figura G.19 se expande con base en el archivo de encabezado para esta clase que se desarrolló en la figura 13.31. La clase `Retiro` tiene un constructor y una función miembro `ejecutar`, que describiremos en breve. En el diagrama de clases de la figura 13.29 vimos que la clase `Retiro` tiene un atributo llamado `monto`, que en la línea 16 se implementa como un miembro de datos `int`. En la figura 13.28 se modelan las asociaciones entre la clase `Retiro` y las clases `Teclado` y `DispensadorEfectivo`, para las cuales en las líneas 17 y 18 se implementan las referencias `teclado` y `dispensadorEfectivo`, respectivamente. En la línea 19 se encuentra el prototipo de una función utilitaria `private` que describiremos pronto.

```

1 // Retiro.h
2 // Definición de la clase Retiro. Representa una transacción de retiro.
3 #ifndef RETIRO_H
4 #define RETIRO_H
5
6 #include "Transaccion.h" // Definición de la clase Transaccion
7 class Teclado; // declaración anticipada de la clase Teclado
8 class DispensadorEfectivo; // declaración anticipada de la clase DispensadorEfectivo
9
10 class Retiro : public Transaccion
11 {
12 public:
13     Retiro( int, Pantalla &, BaseDatosBanco &, Teclado &, DispensadorEfectivo & );
14     virtual void ejecutar(); // realiza la transacción
15 private:
16     int monto; // monto a retirar
17     Teclado &teclado; // referencia al teclado del ATM
18     DispensadorEfectivo &dispensadorEfectivo; // referencia al dispensador de efectivo del ATM
19     int mostrarMenuDeMontos() const; // muestra el menú de retiro
20 }; // fin de la clase Retiro
21
22 #endif // RETIRO_H

```

Figura G.19 | Definición de la clase `Retiro`.

```

1 // Retiro.cpp
2 // Definiciones de las funciones miembro para la clase Retiro.
3 #include "Retiro.h" // Definición de la clase Retiro
4 #include "Pantalla.h" // Definición de la clase Pantalla
5 #include "BaseDatosBanco.h" // Definición de la clase BaseDatosBanco
6 #include "Teclado.h" // Definición de la clase Teclado
7 #include "DispensadorEfectivo.h" // Definición de la clase DispensadorEfectivo
8
9 // constante global que corresponde a la opción del menú para cancelar
10 const static int CANCEL0 = 6;
11
12 // el constructor de Retiro inicializa los miembros de datos de la clase
13 Retiro::Retiro( int numeroCuentaUsuario, Pantalla &pantallaATM,

```

Figura G.20 | Definiciones de las funciones miembro de la clase `Retiro`. (Parte 1 de 3).

```

14     BaseDatosBanco &baseDatosBancoATM, Teclado &tecladoATM,
15     DispensadorEfectivo &dispensadorEfectivoATM )
16     : Transaccion( numeroCuentaUsuario, pantallaATM, baseDatosBancoATM ),
17     teclado( tecladoATM ), dispensadorEfectivo( dispensadorEfectivoATM )
18 {
19     // cuerpo vacío
20 } // fin del constructor de Retiro
21
22 // realiza una transacción; sobrescribe la función virtual pura de Transaccion
23 void Retiro::ejecutar()
24 {
25     bool efectivoDispensado = false; // no se ha dispensado todavía el efectivo
26     bool transaccionCancelada = false; // no se ha cancelado todavía la transacción
27
28     // obtiene referencias a la base de datos del banco y la pantalla
29     BaseDatosBanco &baseDatosBanco = obtenerBaseDatosBanco();
30     Pantalla &pantalla = obtenerPantalla();
31
32     // itera hasta que se dispensa el efectivo o hasta que el usuario cancela
33     do
34     {
35         // obtiene el monto de retiro elegido del usuario
36         int seleccion = mostrarMenuDeMontos();
37
38         // comprueba si el usuario eligió un monto de retiro o canceló
39         if ( seleccion != CANCEL0 )
40         {
41             monto = seleccion; // establece monto con el monto en dólares seleccionado
42
43             // obtiene el saldo disponible de la cuenta involucrada
44             double saldoDisponible =
45                 baseDatosBanco.obtenerSaldoDisponible( obtenerNumeroCuenta() );
46
47             // comprueba si el usuario tiene suficiente dinero en la cuenta
48             if ( monto <= saldoDisponible )
49             {
50                 // comprueba si el dispensador de efectivo tiene suficiente dinero
51                 if ( dispensadorEfectivo.haySuficienteEfectivoDisponible( monto ) )
52                 {
53                     // actualiza la cuenta involucrada para reflejar el retiro
54                     baseDatosBanco.cargar( obtenerNumeroCuenta(), monto );
55
56                     dispensadorEfectivo.dispensarEfectivo( monto ); // dispensa el efectivo
57                     efectivoDispensado = true; // se dispuso el efectivo
58
59                     // instruye al usuario para que tome el efectivo
60                     pantalla.mostrarLineaMensaje(
61                         "\nPor favor tome su efectivo del dispensador de efectivo." );
62                 } // fin de if
63                 else // el dispensador de efectivo no tiene suficiente efectivo
64                 pantalla.mostrarLineaMensaje(
65                     "\nNo hay suficiente efectivo disponible en el ATM."
66                     "\n\nElija un monto menor." );
67             } // fin de if
68             else // no hay suficiente dinero en la cuenta del usuario
69             {
70                 pantalla.mostrarLineaMensaje(
71                     "\nNo hay suficientes fondos en su cuenta."
72                     "\n\nElija un monto menor." );
73             } // fin de else
74         } // fin de if

```

Figura G.20 | Definiciones de las funciones miembro de la clase Retiro. (Parte 2 de 3).

```

75     else // el usuario eligió la opción del menú para cancelar
76     {
77         pantalla.mostrarLineaMensaje( "\nCancelando la transacción..." );
78         transaccionCancelada = true; // el usuario canceló la transacción
79     } // fin de else
80 } while ( !efectivoDispensado && !transaccionCancelada ); // fin de do...while
81 } // fin de la función ejecutar
82
83 // muestra un menú de montos de retiro y la opción para cancelar;
84 // devuelve el monto elegido o 0 si el usuario optó por cancelar
85 int Retiro::mostrarMenuDeMontos() const
86 {
87     int opcionUsuario = 0; // variable local para almacenar el valor de retorno
88
89     Pantalla &pantalla = obtenerPantalla(); // obtiene la referencia a la pantalla
90
91     // arreglo de montos que corresponden a los números del menú
92     int montos[] = { 0, 20, 40, 60, 100, 200 };
93
94     // itera mientras no se haya seleccionado una opción válida
95     while ( opcionUsuario == 0 )
96     {
97         // muestra el menú
98         pantalla.mostrarLineaMensaje( "\nOpciones de retiro:" );
99         pantalla.mostrarLineaMensaje( "1 - $20" );
100        pantalla.mostrarLineaMensaje( "2 - $40" );
101        pantalla.mostrarLineaMensaje( "3 - $60" );
102        pantalla.mostrarLineaMensaje( "4 - $100" );
103        pantalla.mostrarLineaMensaje( "5 - $200" );
104        pantalla.mostrarLineaMensaje( "6 - Cancelar transacción" );
105        pantalla.mostrarMensaje( "\nElige una opción de retiro (1-6): " );
106
107        int entrada = teclado.obtenerEntrada(); // obtiene la entrada del usuario a través del
108        // teclado
109
110        // determina cómo proceder con base en el valor de la entrada
111        switch ( entrada )
112        {
113            case 1: // si el usuario eligió un monto de retiro
114            case 2: // (es decir, si eligió la opción 1, 2, 3, 4 o 5), devuelve
115            case 3: // el monto correspondiente del arreglo montos
116            case 4:
117            case 5:
118                opcionUsuario = montos[ entrada ]; // guarda la opción del usuario
119                break;
120                case CANCELAR: // el usuario eligió cancelar
121                    opcionUsuario = CANCELAR; // guarda la opción del usuario
122                    break;
123                default: // el usuario no introdujo un valor entre 1 y 6
124                    pantalla.mostrarLineaMensaje(
125                        "\nSelección invalida. Intente de nuevo." );
126                } // fin de switch
127        } // fin de while
128
129        return opcionUsuario; // devuelve monto de retiro o CANCELAR
130    } // fin de la función mostrarMenuDeMontos

```

Figura G.20 | Definiciones de las funciones miembro de la clase **Retiro**. (Parte 3 de 3).

*Definiciones de las funciones miembro de la clase **Retiro***

La figura G.20 contiene las definiciones de las funciones miembro para la clase **Retiro**. En la línea 3 se incluye (mediante `#include`) la definición de la clase, y en las líneas 4 a 7 se incluyen las definiciones de las otras clases utilizadas en las

funciones miembro de `Retiro`. En la línea 11 se declara una constante global que corresponde a la opción cancelar en el menú de retiro. Pronto veremos cómo es que la clase utiliza esta constante.

El constructor de la clase `Retiro` (definido en las líneas 13 a 20 de la figura G.20) tiene cinco parámetros. Utiliza un inicializador de la clase base en la línea 16 para pasar los parámetros `numeroCuentaUsuario`, `pantallaATM` y `base-DatosATM` al constructor de la clase base `Transaccion` para establecer los miembros de datos que `Retiro` hereda de `Transaccion`. El constructor también recibe las referencias `tecladoATM` y `dispensadorEfectivoATM` como parámetros, y los asigna a los miembros de datos de referencia `teclado` y `dispensadorEfectivo` mediante el uso de inicializadores de miembros (línea 17).

La clase `Retiro` sobrescribe la función `virtual` pura `ejecutar` de `Transaccion` con una implementación concreta (líneas 23 a 81) que lleva a cabo los pasos involucrados en un retiro. En la línea 25 se declara e inicializa una variable `bool` local llamada `efectivoDispensado`. Esta variable indica si se ha dispensado efectivo (es decir, si la transacción se completó con éxito) y en un principio es `false`. En la línea 26 se declara e inicializa con `false` una variable `bool` llamada `transaccionCancelada`, que indica si el usuario canceló la transacción. En las líneas 29 y 30 se obtienen referencias a la base de datos del banco y la pantalla del ATM, para lo cual se invocan las funciones miembro heredadas de la clase base `Transaccion`.

Las líneas 33 a 80 contienen una instrucción `do...while` que ejecuta su cuerpo hasta que se dispensa efectivo (es decir, hasta que `efectivoDispensado` se vuelve `true`) o hasta que el usuario seleccione cancelar (es decir, hasta que `transaccionCancelada` se vuelve `true`). Este ciclo regresa en forma continua al usuario al inicio de la transacción si ocurre un error (es decir, el monto de retiro solicitado es mayor que el saldo disponible del usuario, o mayor que el monto de efectivo en el dispensador). En la línea 36 se muestra un menú de montos de retiro y se obtiene la selección del usuario, para lo cual se llama a la función utilitaria `private mostrarMenuDeMontos` (definida en las líneas 85 a 129). Esta función muestra el menú de montos y devuelve un monto de retiro `int` o la constante `int CANCELO` para indicar que el usuario ha elegido cancelar la transacción.

La función miembro `mostrarMenuDeMontos` (líneas 85 a 129) declara primero la variable local `opcionUsuario` (al principio es 0) para almacenar el valor que devolverá la función miembro (línea 87). En la línea 89 se obtiene una referencia a la pantalla, para lo cual se hace una llamada a la función miembro `obtenerPantalla` heredada de la clase base `Transaccion`. En la línea 92 se declara un arreglo entero de montos de retiro que corresponden a los montos mostrados en el menú de retiro. Ignoramos el primer elemento en el arreglo (subíndice 0) debido a que el menú no tiene una opción 0. La instrucción `while` en las líneas 95 a 126 se repite hasta que `opcionUsuario` recibe un valor distinto de 0. En un momento veremos que esto ocurre cuando el usuario ha elegido una opción válida del menú. En las líneas 98 a 105 se muestra el menú de retiro en la pantalla y se pide al usuario que introduzca una opción. En la línea 107 se obtiene el entero `entrada` mediante el teclado. La instrucción `switch` en las líneas 110 a 125 determina cómo proceder con base en la entrada del usuario. Si el usuario selecciona un número entre 1 y 5, en la línea 117 se establece `opcionUsuario` con el valor del elemento en `montos`, en el subíndice `entrada`. Por ejemplo, si el usuario introduce 3 para retirar \$60, en la línea 117 se establece `opcionUsuario` con el valor de `montos[3]` (es decir, 60). En la línea 118 termina la instrucción `switch`. La variable `opcionUsuario` ya no es igual a 0, por lo que la instrucción `while` en las líneas 95 a 126 termina y en la línea 128 se devuelve el valor de `opcionUsuario`. Si el usuario selecciona la opción del menú para cancelar, se ejecutan las líneas 120 y 121, y `usuarioOpcion` se establece con `CANCELO`, lo cual hace que la función miembro devuelva ese valor. Si el usuario no introduce una selección válida del menú, en las líneas 123 y 124 se muestra un mensaje de error y el usuario regresa al menú de retiro.

La instrucción `if` en la línea 39 de la función miembro `ejecutar` determina si el usuario ha seleccionado un monto de retiro, o si optó por cancelar. Si el usuario cancela, se ejecutan las líneas 77 y 78 para mostrar un mensaje apropiado al usuario y establecer `transaccionCancelada` a `true`. Esto hace que la prueba de continuación de ciclo en la línea 80 falle y el control regresa a la función miembro que hizo la llamada (es decir, la función miembro `realizarTransacciones de ATM`). Si el usuario ha elegido un monto de retiro, en la línea 41 se asigna la variable local `seleccion` al miembro de datos `uento`. En las líneas 44 y 45 se obtiene el saldo disponible de la Cuenta del usuario actual y se almacena en una variable `double` local llamada `saldoDisponible`. A continuación, la instrucción `if` en la línea 48 determina si el monto seleccionado es menor o igual que el saldo disponible del usuario. Si no es así, en las líneas 70 y 72 se muestra un mensaje de error apropiado. Después el control continúa hacia el final del ciclo `do...while`, y el ciclo se repite debido a que tanto `efectivoDispensado` como `transaccionCancelada` siguen siendo `false`. Si el saldo del usuario es lo suficientemente alto, la instrucción `if` en la línea 51 determina si el dispensador de efectivo tiene suficiente dinero para satisfacer la solicitud de retiro, para lo cual invoca a la función miembro `haySuficienteEfectivoDisponible de dispensadorEfectivo`. Si esta función miembro devuelve `false`, en las líneas 64 a 66 se muestra un mensaje de error apropiado y se repite el ciclo `do...while`. Si hay suficiente efectivo disponible, entonces se satisfacen los requerimientos para el retiro y en la línea 54 se carga el monto a la cuenta del usuario en la base de datos. Después, en las líneas 56 y 57

se instruye al dispensador de efectivo para que dispense el efectivo al usuario y se establezca la variable `efectivoDispensado` a `true`. Por último, en las líneas 60 y 61 se muestra un mensaje al usuario, indicando que se ha dispensado el efectivo. Como ahora `efectivoDispensado` es `true`, el control continúa después del ciclo `do...while`. No aparecen instrucciones adicionales debajo del ciclo, por lo que la función miembro devuelve el control a la clase ATM.

En las llamadas a las funciones en las líneas 64 a 66, y en las líneas 70 a 72, dividimos el argumento para la función miembro `mostrarLineaMensaje` de `Pantalla` en dos literales de cadena, cada una de las cuales se coloca en una línea separada en el programa. Hicimos esto debido a que cada argumento es demasiado largo como para caber en una sola línea. C++ concatena (es decir, combina) las literales de cadena adyacentes, incluso aunque se encuentren en líneas separadas. Por ejemplo, si escribimos "Feliz" "Cumpleanios" en un programa, C++ verá a estas dos literales de cadena adyacentes como una sola literal de cadena "Feliz Cumpleanios". Como resultado, cuando se ejecutan las líneas 64 a 66, `mostrarLineaMensaje` recibe un solo objeto `string` como parámetro, incluso aunque el argumento en la llamada a la función aparezca como dos literales de cadena.

G.12 La clase Deposito

La clase `Deposito` (figuras G.21 y G.22) se deriva de `Transaccion` y representa una transacción de depósito del ATM. La figura G.21 contiene la definición de la clase `Deposito`. Al igual que las clases derivadas `SolicitudSaldo` y `Retiro`, `Deposito` declara un constructor (línea 13) y la función miembro `ejecutar` (línea 14); en unos instantes hablaremos sobre estas funciones. En el diagrama de clases de la figura 13.29 vimos que la clase `Deposito` tiene un atributo llamado `monto`, el cual se implementa en la línea 16 como un miembro de datos `int`. En las líneas 17 y 18 se crean los miembros de datos de referencia `teclado` y `ranuraDeposito`, que implementan las asociaciones entre la clase `Deposito` y las clases `Teclado` y `RanuraDeposito` modeladas en la figura 13.28. En la línea 19 se encuentra el prototipo para una función utilitaria `private` llamada `pedirMontoADepositar`, que describiremos en breve.

Definiciones de las funciones miembro de la clase Deposito

La figura G.22 presenta la implementación de la clase `Deposito`. En la línea 3 se incluye (mediante `#include`) la definición de la clase `Deposito`, y en las líneas 4 a 7 se incluyen las definiciones de las otras clases utilizadas en las funciones miembro de `Deposito`. En la línea 9 se declara una constante llamada `CANCEL0`, que corresponde al valor que introduce un usuario para cancelar un depósito. Pronto veremos cómo es que la clase utiliza esta constante.

Al igual que la clase `Retiro`, la clase `Deposito` contiene un constructor (líneas 12 a 19) que pasa tres parámetros al constructor de la clase base `Transaccion` mediante el uso de un inicializador de la clase base (línea 15). El constructor también tiene los parámetros `tecladoATM` y `ranuraDepositoATM`, que asigna a sus correspondientes miembros de datos (línea 16).

```

1 // Deposito.h
2 // Definición de la clase Deposito. Representa una transacción de depósito.
3 #ifndef DEPOSITO_H
4 #define DEPOSITO_H
5
6 #include "Transaccion.h" // Definición de la clase Transaccion
7 class Teclado; // declaración anticipada de la clase Teclado
8 class RanuraDeposito; // declaración anticipada de la clase RanuraDeposito
9
10 class Deposito : public Transaccion
11 {
12 public:
13     Deposito( int, Pantalla &, BaseDatosBanco &, Teclado &, RanuraDeposito & );
14     virtual void ejecutar(); // realiza la transacción
15 private:
16     double monto; // monto a depositar
17     Teclado &teclado; // referencia al teclado del ATM
18     RanuraDeposito &ranuraDeposito; // referencia a la ranura de depósito del ATM
19     double pedirMontoADepositar() const; // obtiene el monto a depositar del usuario
20 }; // fin de la clase Deposito
21
22 #endif // DEPOSITO_H

```

Figura G.21 | Definición de la clase `Deposito`.

```

1 // Deposito.cpp
2 // Definiciones de las funciones miembro para la clase Deposito.
3 #include "Deposito.h" // Definición de la clase Deposito
4 #include "Pantalla.h" // Definición de la clase Pantalla
5 #include "BaseDatosBanco.h" // Definición de la clase BaseDatosBanco
6 #include "Teclado.h" // Definición de la clase Teclado
7 #include "RanuraDeposito.h" // Definición de la clase RanuraDeposito
8
9 const static int CANCELO = 0; // constante que representa la opción de cancelar
10
11 // el constructor de Deposito inicializa los miembros de datos de la clase
12 Deposito::Deposito( int numeroCuentaUsuario, Pantalla &pantallaATM,
13     BaseDatosBanco &baseDatosBancoATM, Teclado &tecladoATM,
14     RanuraDeposito &ranuraDepositoATM )
15 : Transaccion( numeroCuentaUsuario, pantallaATM, baseDatosBancoATM ),
16     teclado( tecladoATM ), ranuraDeposito( ranuraDepositoATM )
17 {
18     // cuerpo vacío
19 } // fin del constructor de Deposito
20
21 // realiza una transacción; sobrescribe a la función virtual pura de Transaccion
22 void Deposito::ejecutar()
23 {
24     BaseDatosBanco &baseDatosBanco = obtenerBaseDatosBanco(); // obtiene una referencia
25     Pantalla &pantalla = obtenerPantalla(); // obtiene una referencia
26
27     monto = pedirMontoADepositar(); // obtiene el monto a depositar del usuario
28
29     // comprueba si el usuario introdujo un monto a depositar o canceló
30     if ( monto != CANCELO )
31     {
32         // solicita el sobre de depósito que contiene el monto especificado
33         pantalla.mostrarMensaje(
34             "\nInserte un sobre de deposito que contenga " );
35         pantalla.mostrarMontoDolares( monto );
36         pantalla.mostrarLineaMensaje( " en la ranura de deposito." );
37
38         // recibe el sobre de depósito
39         bool sobreRecibido = ranuraDeposito.seRecibioSobre();
40
41         // comprueba si se recibió el sobre de depósito
42         if ( sobreRecibido )
43         {
44             pantalla.mostrarLineaMensaje( "\nSe recibio su sobre."
45                 "\nNOTA: El dinero depositado no estara disponible sino hasta"
46                 "\nverificar el monto de cualquier efectivo incluido, junto con "
47                 "los cheques." );
48
49             // abona a la cuenta para reflejar el depósito
50             baseDatosBanco.abonar( obtenerNumeroCuenta(), monto );
51         } // fin de if
52     } // no se recibió el sobre de depósito
53     else
54     {
55         pantalla.mostrarLineaMensaje( "\nUsted no inserto un "
56             "sobre, por lo que el ATM cancelo su transaccion." );
57     } // fin de else
58 } // fin de if
59 else // el usuario canceló en vez de introducir el monto
60 {
61     pantalla.mostrarLineaMensaje( "\nCancelando la transaccion..." );
62 } // fin de else

```

Figura G.22 | Definiciones de las funciones miembro de la clase Deposito. (Parte I de 2).

```

62 } // fin de la función ejecutar
63
64 // pide al usuario que introduzca un monto a depositar en centavos
65 double Deposito::pedirMontoADepositar() const
66 {
67     Pantalla &pantalla = obtenerPantalla(); // obtiene referencia a la pantalla
68
69     // muestra el indicador y recibe la entrada
70     pantalla.mostrarMensaje( "\nIntroduzca un monto a depositar en "
71         "CENTAVOS (o 0 para cancelar): " );
72     int entrada = teclado.obtenerEntrada(); // recibe la entrada del monto a depositar
73
74     // comprueba si el usuario canceló o introdujo un monto válido
75     if ( entrada == CANCEL0 )
76         return CANCEL0;
77     else
78     {
79         return static_cast< double >( entrada ) / 100; // devuelve el monto en dólares
80     } // fin de else
81 } // fin de la función pedirMontoADepositar

```

Figura G.22 | Definiciones de las funciones miembro de la clase `Deposito`. (Parte 2 de 2).

La función miembro `ejecutar` (líneas 22 a 62) sobrescribe a la función `virtual` pura `ejecutar` en la clase base `Transaccion` con una implementación concreta que lleva a cabo los pasos requeridos en una transacción de depósito. En las líneas 24 y 25 se obtienen referencias a la base de datos y la pantalla. En la línea 27 se pide al usuario que introduzca un monto de depósito, para lo cual se invoca la función utilitaria llamada `pedirMontoADepositar` (definida en las líneas 65 a 81) y se establece el miembro de datos `monto` con el valor devuelto. La función miembro `pedirMontoADepositar` pide al usuario que introduzca un monto de depósito como un número entero de centavos (debido a que el teclado del ATM no contiene un punto decimal; esto es consistente con muchos ATMs reales) y devuelve el valor `double` que representa el monto en dólares a depositar.

La línea 67 en la función miembro `pedirMontoADepositar` obtiene una referencia a la pantalla del ATM. En las líneas 70 y 71 se muestra un mensaje en la pantalla que pide al usuario que introduzca un monto de depósito como un número de centavos, o “0” para cancelar la transacción. En la línea 72 se recibe la entrada del usuario mediante el teclado. La instrucción `if` en las líneas 75 a 80 determina si el usuario ha introducido un monto de depósito real, o si optó por cancelar. Si el usuario decide cancelar, en la línea 76 se devuelve la constante `CANCEL0`. En caso contrario, en la línea 79 se devuelve el monto de depósito después de convertir el número de centavos a un monto en dólares mediante una conversión de `entrada` a un `double`, y después se divide entre 100. Por ejemplo, si el usuario introduce 125 como el número de centavos, en la línea 179 se devuelve 125.0 dividido entre 100, o 1.25; 124 centavos equivalen a \$1.25.

La instrucción `if` en las líneas 30 a 61 en la función miembro `ejecutar` determina si el usuario eligió cancelar la transacción en vez de introducir un monto de depósito. Si el usuario cancela, en la línea 60 se muestra un mensaje de error apropiado y la función miembro regresa. Si el usuario introduce un monto de depósito, en las líneas 33 a 36 se pide al usuario que introduzca un sobre de depósito con el monto correcto. Recuerde que la función miembro `mostrarMontoDolares` imprime en pantalla un valor `double` con formato de monto en dólares.

En la línea 39 se establece una variable `bool` local con el valor devuelto por la función miembro `seRecibioSobre` de `ranuraDeposito`, indicando si se recibió un sobre de depósito. Recuerde que codificamos `seRecibioSobre` (líneas 7 a 10 de la figura G.10) para que siempre devuelva `true`, ya que estamos simulando la funcionalidad de la ranura de depósito y asumimos que el usuario siempre inserta un sobre. Sin embargo, codificamos la función miembro `ejecutar` de la clase `Deposito` para que evalúe la posibilidad de que el usuario no inserte un sobre; la buena ingeniería de software exige que los programas tomen en cuenta todos los posibles valores de retorno. Así, la clase `Deposito` está preparada para futuras versiones de `seRecibioSobre` que pudieran devolver `false`. Las líneas 44 a 50 se ejecutan si la ranura de depósito devuelve un sobre. En las líneas 44 a 47 se muestra un mensaje apropiado al usuario. Después, en la línea 50 se abona el monto de depósito a la cuenta del usuario en la base de datos. Las líneas 54 y 55 se ejecutan si la ranura de depósito no recibe un sobre de depósito. En este caso, se muestra un mensaje al usuario indicando que el ATM canceló la transacción. Después, la función miembro regresa sin modificar la cuenta del usuario.

G.13 El programa de prueba EjemploPracticoATM.cpp

EjemploPracticoATM.cpp (figura G.23) es un programa simple de C++ que nos permite empezar (o “encender”) el ATM y evaluar la implementación de nuestro modelo del sistema ATM. La función `main` del programa (líneas 6 a 11) no hace más que instanciar un nuevo objeto ATM llamado `atm` (línea 8) e invocar a su función miembro `ejecutar` (línea 9) para iniciar el ATM.

```
1 // EjemploPracticoATM.cpp
2 // Programa controlador para el ejemplo práctico del ATM.
3 #include "ATM.h" // Definición de la clase ATM
4
5 // la función main crea y ejecuta el ATM
6 int main()
7 {
8     ATM atm; // crea un objeto ATM
9     atm.ejecutar(); // indica al ATM que debe empezar
10    return 0;
11 } // fin de main
```

Figura G.23 | EjemploPracticoATM.cpp inicia el sistema del ATM.

G.14 Repaso

Felicidades por haber completado todo el Ejemplo práctico de Ingeniería de Software acerca del ATM! Esperamos que esta experiencia le haya parecido valiosa y que haya reforzado muchos de los conceptos que aprendió en los capítulos del 1 al 13. Apreciamos sinceramente que envíe sus comentarios, críticas y sugerencias. Puede contactarnos en deitel@deitel.com. Le responderemos lo más pronto posible.



UML 2: tipos de diagramas adicionales

H.1 Introducción

Si leyó las secciones opcionales del Ejemplo práctico de Ingeniería de Software en los capítulos del 2 al 7, 9 y 13, ahora debe tener una idea de los tipos de diagramas de UML que utilizamos para modelar nuestro sistema ATM. El ejemplo práctico está pensado para usarse en cursos de primer o segundo semestre, por lo que limitamos nuestra discusión a un subconjunto conciso del UML. UML 2 cuenta con un total de 13 tipos de diagramas. El final de la sección 2.8 resume los seis tipos de diagramas que utilizamos en el ejemplo práctico. Este apéndice lista y define brevemente los siete tipos de diagramas restantes.

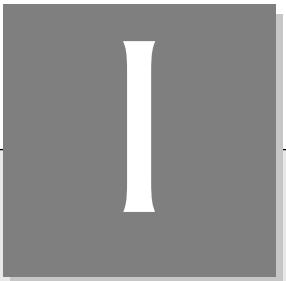
H.2 Tipos de diagramas adicionales

A continuación se describen los siete tipos de diagramas que optamos por no utilizar en nuestro Ejemplo práctico de Ingeniería de Software.

- **Diagramas de objetos:** modelan una “instantánea” del sistema; para ello modelan los objetos de un sistema y sus relaciones en un punto específico en el tiempo. Cada objeto representa una instancia de una clase de un diagrama de clases, y pueden crearse varios objetos a partir de una clase. Para nuestro sistema ATM, un diagrama de objetos podría mostrar varios objetos **Cuenta** distintos, uno al lado del otro, ilustrando que todos forman parte de la base de datos de cuentas del banco.
- **Diagramas de componentes:** modelan los **artefactos y componentes**: recursos (incluyen archivos de código fuente) que conforman el sistema.
- **Diagramas de despliegue:** modelan los requerimientos del sistema en tiempo de ejecución (como la computadora o computadoras en donde residirá el sistema), los requerimientos de memoria, o los demás dispositivos que requiera el sistema durante la ejecución.
- **Diagramas de paquetes:** modelan la estructura jerárquica de los **paquetes** (que son grupos de clases) en el sistema en tiempo de compilación, así como las relaciones que existen entre los paquetes
- **Diagramas de estructuras compuestas:** modelan la estructura interna de un objeto complejo en tiempo de ejecución. Estos diagramas son nuevos en UML 2, y permiten a los diseñadores de sistemas descomponer en forma jerárquica un objeto complejo en partes más pequeñas. Los diagramas de estructuras compuestas están más allá del alcance de nuestro caso de estudio. Son más apropiados para aplicaciones industriales más grandes, las cuales exhiben agrupamientos complejos de objetos en tiempo de ejecución.

- **Diagramas de vista de interacción:** estos diagramas son nuevos en UML 2 y proporcionan un resumen del flujo de control en el sistema, mediante una combinación de elementos de varios tipos de diagramas de comportamiento (por ejemplo, diagramas de actividad, diagramas de secuencia).
- **Diagramas de sincronización:** también nuevos en UML 2, modelan las restricciones de sincronización impuestas en los cambios de etapas y las interacciones entre los objetos en un sistema.

Para aprender más acerca de estos diagramas y de los temas avanzados de UML, visite el sitio Web www.uml.org y los recursos Web listados al final de las secciones 1.21 y 2.8.



Por lo tanto, debo atrapar la mosca.

—William Shakespeare

Estamos creados para cometer equivocaciones, codificados para el error.

—Lewis Thomas

Lo que anticipamos raras veces ocurre; lo que menos esperamos es lo que generalmente pasa.

—Benjamin Disraeli

Puede correr, pero no puede ocultarse.

—Joe Louis

Una cosa es mostrar a un hombre que está equivocado, y otra es darle posesión de la verdad.

—John Locke

Uso del depurador de Visual Studio

OBJETIVOS

En este apéndice aprenderá a:

- Establecer puntos de interrupción para depurar programas.
- Ejecutar un programa a través del depurador.
- Establecer, deshabilitar y eliminar un punto de interrupción.
- Utilizar el comando **Continuar** para continuar la ejecución.
- Utilizar la ventana **Variables locales** para ver y modificar los valores de las variables.
- Utilizar la ventana **Inspección** para evaluar expresiones.
- Control de la ejecución mediante los comandos **Paso a paso por instrucciones**, **Paso a paso por procedimientos**, **Paso a paso para salir** y **Continuar**.
- Utilizar la ventana **Automático** para ver las variables que se utilizan en las instrucciones circundantes.

- I.1 Introducción
- I.2 Los puntos de interrupción y el comando Continuar
- I.3 Las ventanas Variables locales e Inspección
- I.4 Control de la ejecución mediante los comandos Paso a paso por instrucciones, Paso a paso por procedimientos, Paso a paso para salir y Continuar
- I.5 La ventana Automático
- I.6 Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#)

I.1 Introducción

En el capítulo 2 vimos que hay dos tipos de errores (errores de compilación y errores lógicos) y el lector aprendió a eliminar los errores de compilación de su código. Los errores lógicos (también llamados “bugs” en inglés) no evitan que un programa se compile con éxito, pero pueden hacer que el programa produzca resultados erróneos al ejecutarse. La mayoría de los distribuidores de compiladores de C++ proporcionan un programa de software conocido como **depurador**, el cual nos permite supervisar la ejecución de los programas para localizar y eliminar errores lógicos. El depurador será una de sus herramientas de desarrollo de programas más importantes. En este apéndice demostramos las características clave del depurador de Visual Studio. En el apéndice J hablamos sobre las características y herramientas del depurador de GNU C++. Nuestro Centro de recursos de C++ (www.deitel.com/cplusplus/) proporciona vínculos a tutoriales que pueden ayudar a los estudiantes e instructores a familiarizarse con los depuradores incluidos en otras herramientas de desarrollo.

I.2 Los puntos de interrupción y el comando Continuar

Para empezar con nuestro estudio del depurador, vamos a investigar los **puntos de interrupción**, que son marcadores que pueden establecerse en cualquier línea de código ejecutable. Cuando la ejecución del programa llega a un punto de interrupción, la ejecución se detiene, lo cual nos permite examinar los valores de las variables para ayudarnos a determinar si existe un error lógico. Por ejemplo, podemos examinar el valor de una variable que almacena el resultado de un cálculo para determinar si el cálculo se realizó en forma correcta. Observe que al tratar de establecer un punto de interrupción en una línea de código que no es ejecutable (como un comentario), el punto de interrupción se establecerá en la siguiente línea de código ejecutable en esa función.

Para ilustrar las características del depurador, vamos a usar el programa que se lista en la figura I.3, el cual crea y manipula un objeto de la clase **Cuenta** (figuras I.1 y I.2). La ejecución empieza en **main** (líneas 12 a 30 de la figura I.3). En la línea 14 se crea un objeto **Cuenta** con un saldo inicial de \$50.00. El constructor de **Cuenta** (líneas 10 a 22 de la figura I.2) acepta un argumento, el cual especifica el saldo inicial de la **Cuenta**. En la línea 17 de la figura I.3 se imprime el saldo inicial de la cuenta mediante el uso de la función miembro **obtenerSaldo** de **Cuenta**. En la línea 19 se declara una variable local llamada **montoRetiro**, la cual almacena un monto de retiro que se lee del usuario. En la línea 21 se pide al usuario el monto de retiro, y en la línea 22 se recibe como entrada el monto en **montoRetiro**. En la línea 25 se resta el monto de retiro del saldo de la **Cuenta** mediante el uso de su función miembro **cargar**. Por último, en la línea 28 se muestra el nuevo saldo.

```

1 // Fig. I.1: Cuenta.h
2 // Definición de la clase Cuenta.
3
4 class Cuenta
5 {
6 public:
7     Cuenta( int ); // el constructor inicializa el saldo
8     void abonar( int ); // suma un monto al saldo de la cuenta
9     void cargar( int ); // resta un monto del saldo de la cuenta
10    int obtenerSaldo(); // devuelve el saldo de la cuenta
11 private:
12    int saldo; // miembro de datos que almacena el saldo
13 };// fin de la clase Cuenta

```

Figura I.1 | Archivo de encabezado para la clase **Cuenta**.

```

1 // Fig. I. 2: Cuenta.cpp
2 // Definiciones de las funciones miembro de la clase Cuenta.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Cuenta.h" // incluye la definición de la clase Cuenta
8
9 // el constructor de Cuenta inicializa el miembro de datos saldo
10 Cuenta::Cuenta( int saldoInicial )
11 {
12     saldo = 0; // asume que el saldo empieza en 0
13
14     // si saldoInicial es mayor que 0, establece este valor como el
15     // saldo de la cuenta; en caso contrario, saldo sigue siendo 0
16     if ( saldoInicial > 0 )
17         saldo = saldoInicial;
18
19     // si saldoInicial es negativo, imprime un mensaje de error
20     if ( saldoInicial < 0 )
21         cout << "Error: El saldo inicial no puede ser negativo.\n" << endl;
22 } // fin del constructor de Cuenta
23
24 // abona (suma) un monto al saldo de la cuenta
25 void Cuenta::abonar( int monto )
26 {
27     saldo = saldo + monto; // suma el monto al saldo
28 } // fin de la función abonar
29
30 // carga (resta) un monto al saldo de la cuenta
31 void Cuenta::cargar( int monto )
32 {
33     if ( monto <= saldo ) // monto a cargar no excede al saldo
34         saldo = saldo - monto;
35
36     else // monto a cargar excede al saldo
37         cout << "El monto a cargar excede al saldo de la cuenta.\n" << endl;
38 } // fin de la función cargar
39
40 // devuelve el saldo de la cuenta
41 int Cuenta::obtenerSaldo()
42 {
43     return saldo; // proporciona el valor de saldo a la función que hizo la llamada
44 } // fin de la función obtenerSaldo

```

Figura I.2 | Definición de la clase Cuenta.

```

1 // Fig. I.3: figI_03.cpp
2 // Crea y manipula objetos Cuenta.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 // incluye la definición de la clase Cuenta de Cuenta.h
9 #include "Cuenta.h"
10
11 // la función main empieza la ejecución del programa
12 int main()
13 {

```

Figura I.3 | Clase de prueba para la depuración. (Parte I de 2).

```

14 Cuenta cuenta1( 50 ); // crea un objeto Cuenta
15
16 // muestra el saldo inicial de cada objeto
17 cout << "Saldo de cuenta1: $" << cuenta1.obtenerSaldo() << endl;
18
19 int montoRetiro; // almacena el monto de retiro que se lee del usuario
20
21 cout << "\nEscriba el monto de retiro para cuenta1: " // indicador
22 cin >> montoRetiro; // obtiene la entrada del usuario
23 cout << "\n\ntratando de restar " << montoRetiro
24     << " del saldo de cuenta1\n\n";
25 cuenta1.cargar( montoRetiro ); // trata de restar de cuenta1
26
27 // muestra los saldos
28 cout << "cuenta1 saldo: $" << cuenta1.obtenerSaldo() << endl;
29 return 0; // indica que terminó con éxito
30 } // fin de main

```

Figura I.3 | Clase de prueba para la depuración. (Parte 2 de 2).

En los siguientes pasos, utilizaremos puntos de interrupción y varios comandos del depurador para examinar el valor de la variable `montoRetiro` declarada en la figura I.3.

- Habilitar el depurador.** Por lo general, el depurador está habilitado de manera predeterminada. Si no está habilitado, debe modificar las opciones del cuadro combinado *Configuraciones de soluciones* (figura I.4) en la barra de herramientas. Para ello, haga clic en la flecha desplegable del cuadro combinado y después seleccione **Debug**.
- Insertar puntos de interrupción en Visual Studio 2005.** Para insertar un punto de interrupción en Visual Studio 2005, haga clic dentro de la barra indicadora de margen (el margen gris en la parte izquierda de la ventana de código de la figura I.5) enseguida de la línea de código en la que desea interrumpir, o haga clic con el botón derecho del ratón en esa línea de código y seleccione **Punto de interrupción > Insertar punto de interrupción**. Puede establecer todos los puntos de interrupción que sean necesarios. Establezca puntos de interrupción en las líneas 21 y 25 de su código. Un círculo rojo aparecerá en la barra indicadora de margen en donde usted hizo clic, indicando que se ha establecido un punto de interrupción (figura I.5). Cuando el programa se ejecute, el depurador interrumpirá la ejecución en cualquier línea que contenga un punto de interrupción. Se dice que el programa está en **modo de interrupción** cuando el depurador suspende el programa. Los puntos de interrupción se pueden establecer antes de ejecutar un programa, en modo de interrupción y mientras un programa se ejecuta.
- Empezar a depurar.** Después de establecer puntos de interrupción en el editor de código, seleccione **Generar > Generar solución** para compilar el programa, y después seleccione **Depurar > Iniciar depuración** para comenzar el proceso de depuración. [Nota: si no compila primero el programa, de todas formas se compilará cuando seleccione **Depurar > Iniciar depuración**]. Al depurar una aplicación de consola, aparece una ventana **Símbolo del sistema** (figura I.6) en la que podemos especificar la entrada del programa y ver sus resultados. El depurador entra al modo de interrupción cuando la ejecución llega al punto de interrupción en la línea 21.

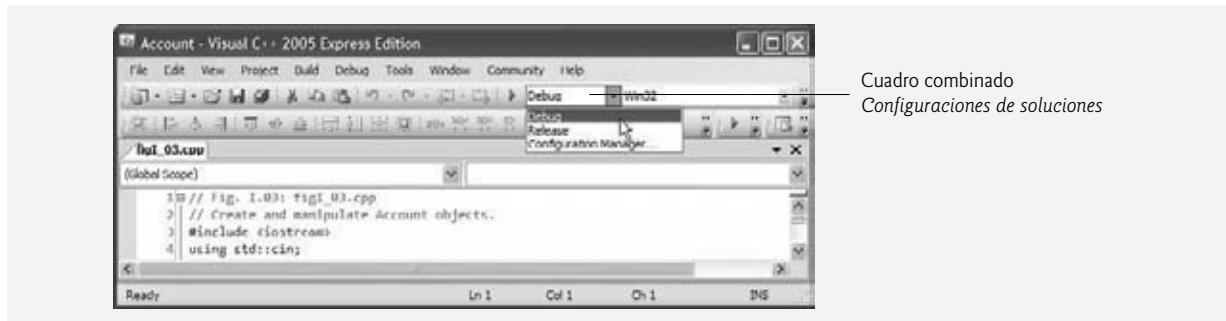


Figura I.4 | Habilitar el depurador.

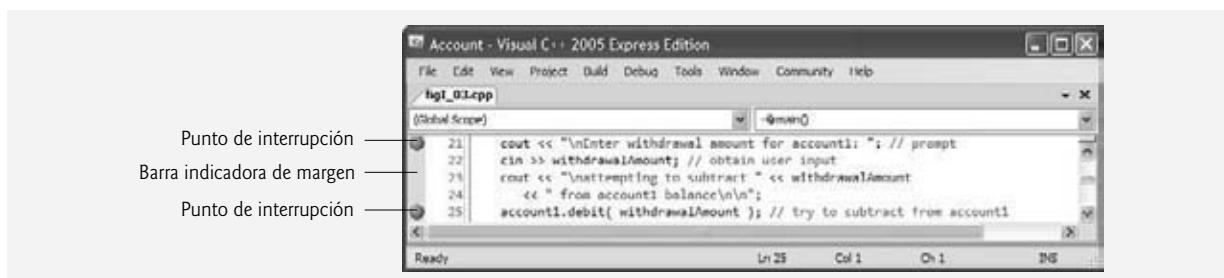


Figura I.5 | Establecer dos puntos de interrupción.



Figura I.6 | Programa Cuenta en ejecución.

4. **Examinar la ejecución del programa.** Al entrar al modo de interrupción en el primer punto de interrupción (línea 21), el IDE se convierte en la ventana activa (figura I.7). La flecha amarilla a la izquierda de la línea 21 indica que esta línea contiene la siguiente instrucción que se debe ejecutar.
5. **Utilizar el comando Continuar para reanudar la ejecución.** Para reanudar la ejecución, seleccione **Depurar > Continuar**. El comando Continuar reanuda la ejecución del programa hasta encontrar el siguiente punto de interrupción o el final de main, lo que ocurra primero. El programa continúa su ejecución y se detiene para recibir datos de entrada en la línea 22. Escriba 13 como el monto de retiro. El programa se ejecutará hasta detenerse en el siguiente punto de interrupción (línea 25). Observe que al colocar el puntero del ratón sobre el nombre de la variable `montoRetiro`, el valor almacenado en la variable se muestra en un cuadro de *Información rápida* (figura I.8). Como puede ver, esto puede ayudar al programador a detectar los errores lógicos en los programas.

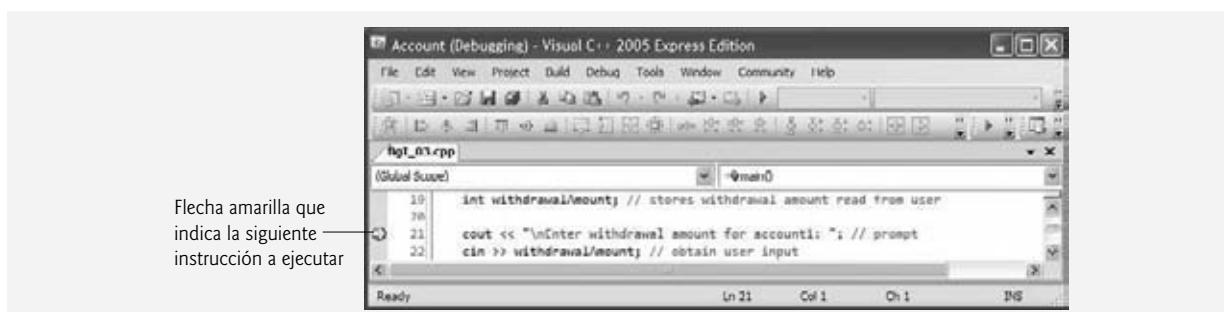


Figura I.7 | La ejecución del programa está suspendida en el primer punto de interrupción.

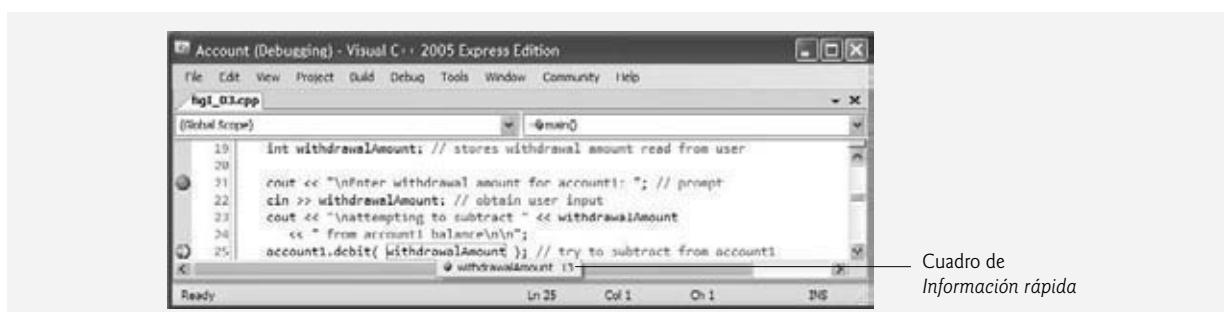


Figura I.8 | El cuadro de *Información rápida* muestra el valor de una variable.

6. **Establecer un punto de interrupción en la instrucción return.** Establezca un punto de interrupción en la línea 29 del código fuente; para ello, haga clic en la barra indicadora de margen a la izquierda de la línea 29. Esto evitará que el programa se cierre de inmediato después de mostrar su resultado. Cuando no hay más puntos de interrupción en los que se pueda suspender la ejecución, el programa se ejecutará hasta completarse y la ventana **Símbolo del sistema** se cerrará. Si no establece este punto de interrupción, no podrá ver los resultados del programa antes de que se cierre la consola.
7. **Continuar la ejecución del programa.** Use el comando **Depurar > Continuar** para ejecutar el código hasta el siguiente punto de interrupción. El programa muestra el resultado de su cálculo (figura I.9).
8. **Deshabilitar un punto de interrupción.** Para deshabilitar un punto de interrupción, haga clic con el botón derecho del ratón en una línea de código en la que se haya establecido un punto de interrupción (o en el mismo punto de interrupción) y seleccione **Deshabilitar punto de interrupción**. El punto de interrupción deshabilitado se indica mediante un círculo sin relleno (figura I.10). Al deshabilitar un punto de interrupción en vez de eliminarlo, podemos volver a habilitarlo más adelante, haciendo clic en el círculo sin relleno o haciendo clic con el botón derecho del ratón en el círculo relleno y seleccionando **Habilitar punto de interrupción**.
9. **Eliminar un punto de interrupción.** Para eliminar un punto de interrupción que ya no necesita, haga clic con el botón derecho del ratón en una línea de código en la que se haya establecido un punto de interrupción y seleccione **Punto de interrupción > Eliminar punto de interrupción**. También puede eliminar un punto de interrupción haciendo clic en el punto de interrupción en la barra indicadora de margen.
10. **Terminar la ejecución del programa.** Seleccione **Depurar > Continuar** para ejecutar el programa hasta completarse.

En esta sección aprendió a habilitar el depurador y establecer puntos de interrupción, de manera que se puedan examinar los resultados del código mientras el programa se ejecuta. También aprendió a continuar la ejecución después de que un programa la suspende en un punto de interrupción, y cómo deshabilitar y eliminar los puntos de interrupción.



Figura I.9 | Resultados del programa.

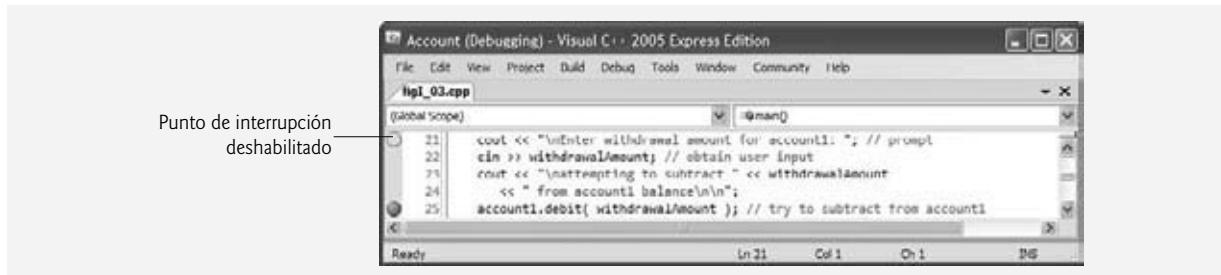


Figura I.10 | Punto de interrupción deshabilitado.

I.3 Las ventanas Variables locales e Inspección

En la sección anterior vimos que la característica *Información rápida* nos permite examinar el valor de una variable. En esta sección aprenderá a utilizar la **ventana Variables locales** para asignar nuevos valores a las variables mientras el programa se ejecuta. También utilizará la **ventana Inspección** para examinar el valor de expresiones más complejas.

1. **Insertar puntos de interrupción.** Borre los puntos de interrupción existentes. Despues establezca un punto de interrupción en la línea 25 del código fuente, haciendo clic en la barra indicadora de margen a la izquierda de la línea 25 (figura I.11). Establezca otro punto de interrupción en la línea 28, haciendo clic en la barra indicadora de margen a la izquierda de la línea 28.

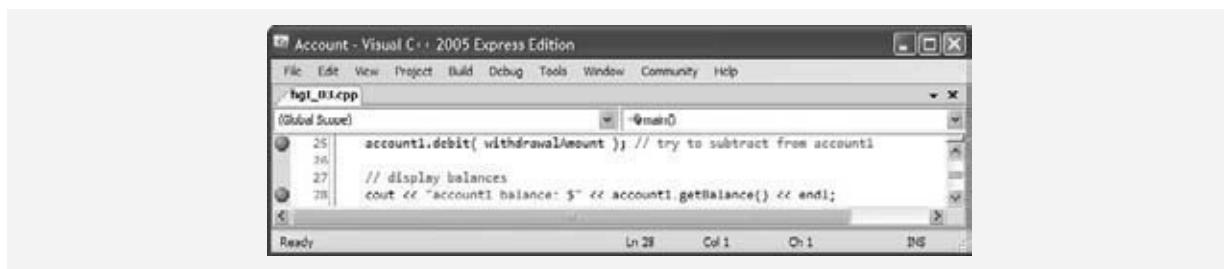


Figura I.11 | Establecer puntos de interrupción en las líneas 25 y 28.

- Empezar a depurar.** Seleccione **Depurar > Iniciar depuración**. Escriba 13 en el indicador **Escriba el monto de retiro para cuenta1:** y oprima *Intro* de manera que su programa lea el valor que acaba de introducir. El programa se ejecutará hasta el punto de interrupción en la línea 25.
- Suspender la ejecución del programa.** El depurador entra al modo de interrupción en la línea 25 (figura I.12). En este punto, en la línea 22 se ha introducido el `montoRetiro` que usted escribió (13), las líneas 23 y 24 han mostrado en pantalla que el programa tratará de retirar dinero y en la línea 25 se encuentra la siguiente instrucción a ejecutar.
- Examinar los datos.** En modo de interrupción podemos explorar los valores de las variables locales, usando la ventana **Variables locales** del depurador. Para ver esta ventana, seleccione **Depurar > Ventanas > Variables locales**. La figura I.13 muestra los valores de las variables locales de `main` llamadas `cuenta1` y `montoRetiro` (13).
- Evaluar las expresiones aritméticas y booleanas.** Podemos evaluar las expresiones aritméticas y booleanas mediante la ventana **Inspección**. Podemos mostrar hasta cuatro ventanas **Inspección**. Seleccione **Depurar | Ventanas > Inspección > Inspección 1**. En la primera fila de la columna **Nombre**, escriba `(montoRetiro + 3) * 5` y oprima *Intro*. El valor de esta expresión (en este caso, 80) se muestra en la columna **Valor** (figura I.14). En la siguiente fila de la columna **Nombre**, escriba `montoRetiro == 3` y oprima *Intro*. Esta expresión determina si el valor de `montoRetiro` es 3. Las expresiones que contienen el operador `==` (o cualquier otro operador relacional o de igualdad) se consideran como expresiones `bool`. El valor de la expresión en este caso es `false` (figura I.14), debido a que `montoRetiro` contiene actualmente 13, no 3.

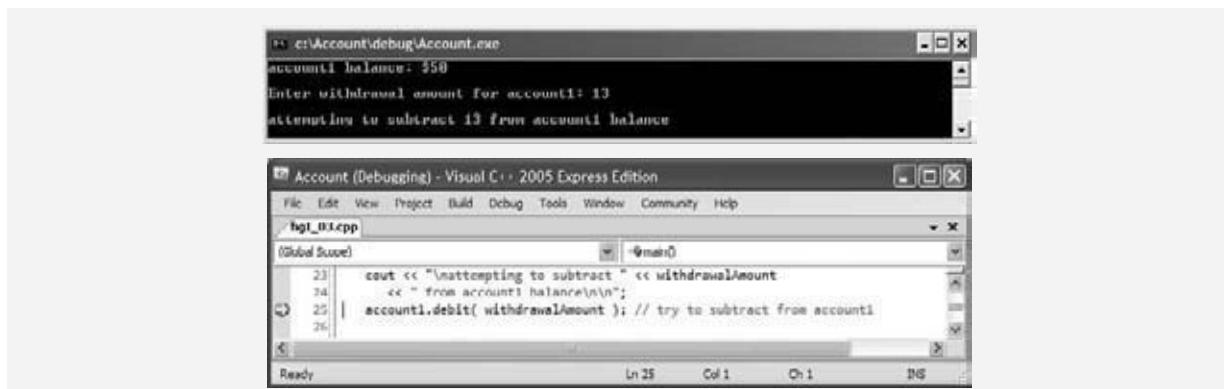


Figura I.12 | La ejecución del programa se suspende cuando el depurador llega al punto de interrupción en la línea 25.

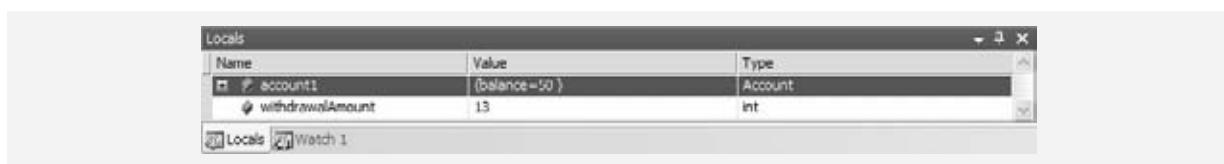


Figura I.13 | Examinar la variable `montoRetiro`.

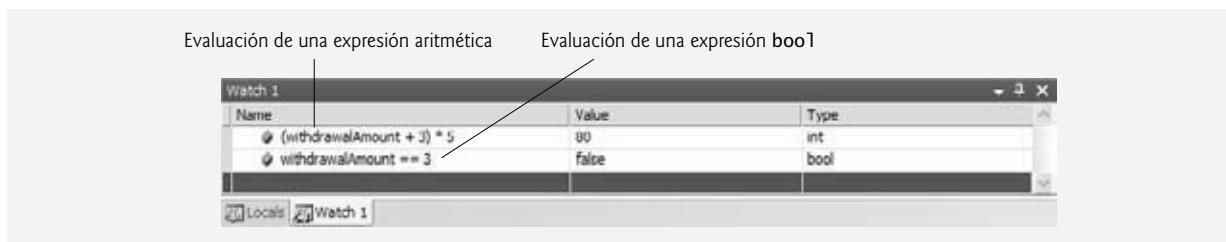


Figura I.14 | Examinar los valores de las expresiones.

6. **Reanudar la ejecución.** Seleccione Depurar > Continuar para reanudar la ejecución. En la línea 25 se carga a la cuenta el monto de retiro, y el depurador vuelve a entrar al modo de interrupción en la línea 28. Seleccione Depurar > Ventanas > Variables locales o haga clic en la ficha Variables locales en la parte inferior de Visual Studio para volver a mostrar la ventana Variables locales. El valor actualizado de saldo en cuenta1 se muestra ahora en color rojo (figura I.15) para indicar que se ha modificado desde el último punto de interrupción. Haga clic en el cuadro con el signo positivo a la izquierda de cuenta1 en la columna Nombre de la ventana Variables locales. Esto nos permite ver cada uno de los valores de los miembros de datos de cuenta1 en forma individual; esto es especialmente útil para los objetos que tienen varios miembros de datos.
7. **Modificar los valores.** Con base en el valor introducido por el usuario (13), el saldo de la cuenta que muestra el programa debe ser \$37. Sin embargo, podemos usar la ventana Variables locales para modificar los valores de las variables durante la ejecución del programa. Esto puede ser valioso para experimentar con distintos valores y localizar errores lógicos. En la ventana Variables locales, haga clic en el campo Valor en la fila saldo para seleccionar el valor 37. Escriba 33 y oprima Intro. El depurador cambia el valor de saldo y muestra su nuevo valor en color rojo (en la pantalla de su computadora, figura I.16).
8. **Establecer un punto de interrupción en la instrucción return.** Establezca un punto de interrupción en la línea 29 en el código fuente para evitar que el programa se cierre de inmediato, después de mostrar su resultado. Si no establece este punto de interrupción, no podrá ver los resultados del programa antes de que se cierre la ventana de consola.
9. **Ver el resultado del programa.** Seleccione Depurar > Continuar para reanudar la ejecución del programa. La función main se ejecutará hasta la instrucción return en la línea 29 y mostrará el resultado. Observe que este resultado es \$33 (figura I.17). Esto muestra que el *paso 7* modificó el valor de saldo, del valor calculado (37) a 33.
10. **Detener la sesión de depuración.** Seleccione Depurar > Detener depuración. Esto cerrará la ventana Símbolo del sistema. Elimine el resto de los puntos de interrupción.



Figura I.15 | Mostrar el valor de las variables locales.

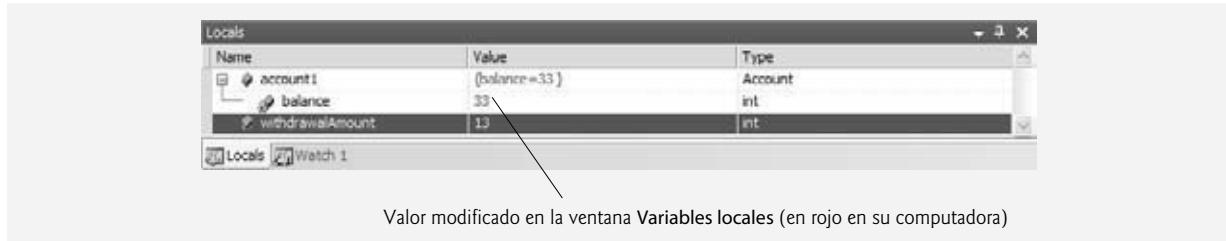


Figura I.16 | Modificar el valor de una variable.



I.17 | Los resultados que se muestran después de modificar la variable cuenta1.

En esta sección aprendió a utilizar las ventanas **Inspección** y **Variables locales** del depurador para evaluar expresiones aritméticas y booleanas. También aprendió a modificar el valor de una variable durante la ejecución de un programa.

I.4 Control de la ejecución mediante los comandos Paso a paso por instrucciones, Paso a paso por procedimientos, Paso a paso para salir y Continuar

Algunas veces, al ejecutar un programa línea por línea podemos verificar que el código de una función se ejecute en forma correcta, lo cual nos puede ayudar a encontrar y corregir errores lógicos. Los comandos que aprenderá en esta sección le permitirán ejecutar una función línea por línea, ejecutar todas las instrucciones de una función a la vez, o ejecutar sólo el resto de las instrucciones de una función (si ya hemos ejecutado algunas instrucciones dentro de la función).

- 1. Establecer un punto de interrupción.** Establezca un punto de interrupción en la línea 25, haciendo clic en la barra indicadora de margen a la izquierda de la línea.
- 2. Empezar el depurador.** Seleccione **Depurar > Iniciar depuración**. Escriba el valor 13 en el indicador **Escriba el monto de retiro para cuenta1:**. La ejecución se detendrá cuando el programa llegue al punto de interrupción en la línea 25.
- 3. Uso del comando Paso a paso por instrucciones.** El comando **Paso a paso por instrucciones** ejecuta la siguiente instrucción en el programa (a la que apunta la flecha amarilla en la figura I.18), y después se detiene de inmediato. Si esa instrucción es una llamada a una función (como es el caso aquí), el control se transfiere a la función que se llamó. Esto nos permite ejecutar cada instrucción dentro de la función de manera individual, para confirmar su ejecución. Seleccione **Depurar > Paso a paso por instrucciones** para entrar a la función cargar. Después seleccione **Depurar > Paso a paso por instrucciones** de nuevo, de manera que la flecha amarilla esté posicionada en la línea 33, como se muestra en la figura I.19.

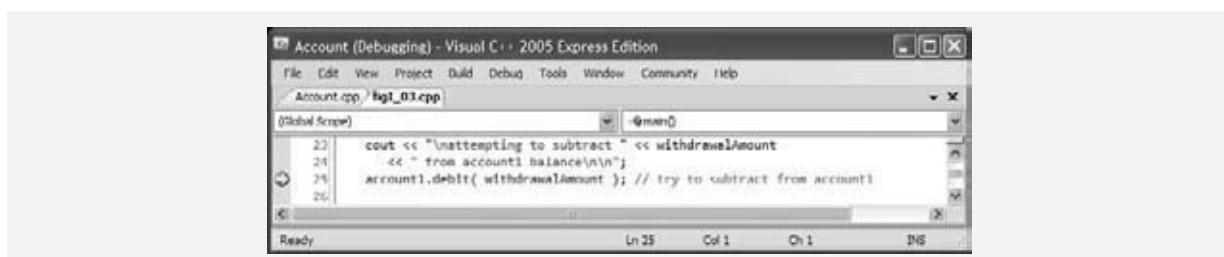


Figura I.18 | Uso del comando **Paso a paso por instrucciones** para ejecutar una instrucción.

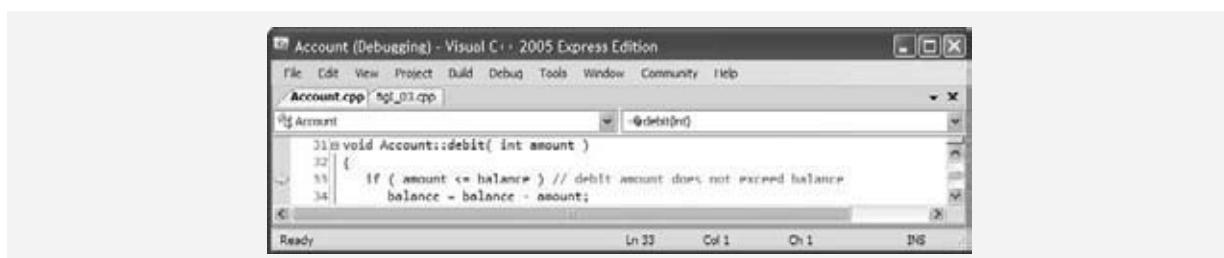


Figura I.19 | Ejecución de la función **cargar** paso a paso por cada instrucción.

4. **Utilizar el comando Paso a paso por procedimientos.** Seleccione Depurar > Paso a paso por procedimientos para ejecutar la instrucción actual (línea 33 en la figura I.19) y transferir el control a la línea 34 (figura I.20). El comando **Paso a paso por procedimientos** se comporta como el comando **Paso a paso por instrucciones** cuando la siguiente instrucción a ejecutar no contiene una llamada a una función. En el *paso 10* veremos cuál es la diferencia entre los comandos **Paso a paso por procedimientos** y **Paso a paso por instrucciones**.
5. **Utilizar el comando Paso a paso para salir.** Seleccione Depurar > Paso a paso para salir para ejecutar el resto de las instrucciones en la función y regresar el control a la siguiente instrucción ejecutable (línea 28 en la figura I.3). A menudo, en las funciones extensas es conveniente analizar unas cuantas líneas clave de código, y después seguir depurando el código de la función que hizo la llamada. El comando **Paso a paso para salir** nos permite continuar la ejecución del programa en la función que hizo la llamada, sin tener que avanzar por toda la función que se llamó completa, línea por línea.
6. **Establecer un punto de interrupción.** Establezca un punto de interrupción en la instrucción `return` de `main` en la línea 29 de la figura I.3. En el siguiente paso utilizará este punto de interrupción.
7. **Utilizar el comando Continuar.** Seleccione Depurar > Continuar para ejecutar el programa hasta llegar al siguiente punto de interrupción en la línea 29. El comando **Continuar** es útil cuando deseamos ejecutar todo el código hasta el siguiente punto de interrupción.
8. **Detener el depurador.** Seleccione Depurar > Detener depuración para terminar la sesión de depuración. Esto cerrará la ventana **Símbolo del sistema**.
9. **Iniciar el depurador.** Antes de poder demostrar la siguiente característica del depurador, debemos iniciararlo de nuevo. Inicie el depurador, como hizo en el *paso 2*, y escriba 13 en respuesta al indicador. El depurador entrará al modo de interrupción en la línea 25.
10. **Utilizar el comando Paso a paso por procedimientos.** Seleccione Depurar > Paso a paso por procedimientos (figura I.21). Recuerde que este comando se comporta como el comando **Paso a paso por instrucciones** cuando la siguiente instrucción a ejecutar no contiene una llamada a una función. Si la siguiente instrucción a ejecutar contiene una llamada a una función, la función a la que se llama se ejecuta en su totalidad (sin detener la ejecución en ninguna instrucción dentro de la función), y la flecha amarilla avanza hasta la siguiente línea ejecutable (después de la llamada a la función) en la función actual. En este caso, el depurador ejecuta la línea 25, ubicada

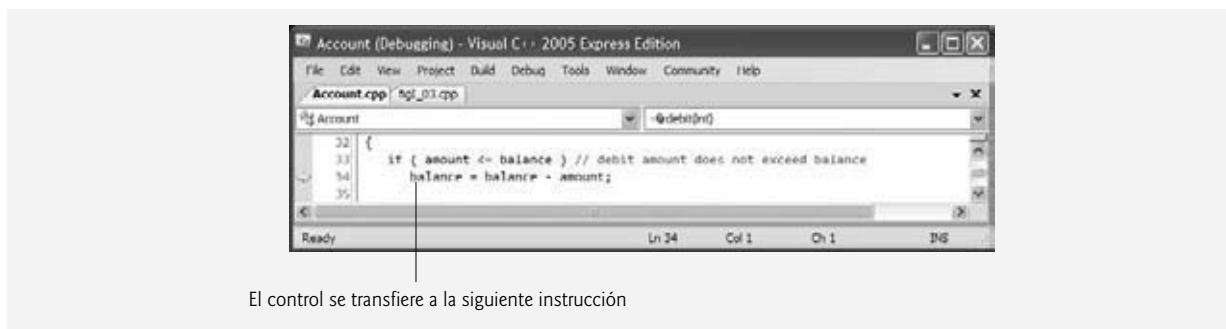


Figura I.20 | Ejecución de una instrucción en la función **cargar** mediante el comando **Paso a paso por procedimientos**.

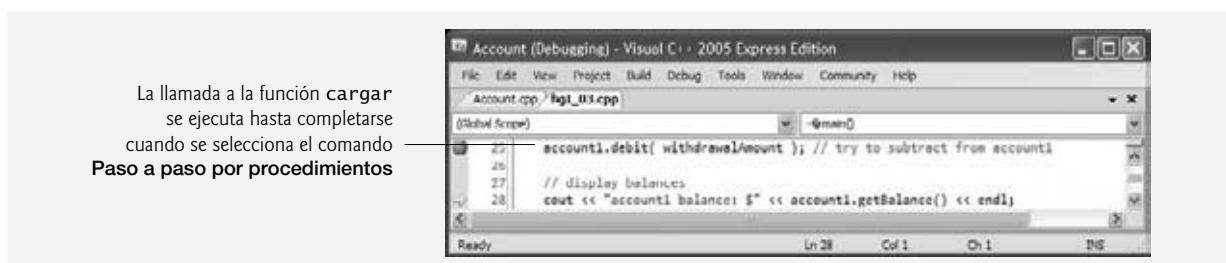


Figura I.21 | Uso del comando **Paso a paso por procedimientos** del depurador.

en `main` (figura I.3). En la línea 25 se hace una llamada a la función `cargar`. Después, el depurador detiene la ejecución en la línea 28, la siguiente línea ejecutable en la función actual, `main`.

11. **Detener el depurador.** Seleccione **Depurar > Detener depuración**. Esto cerrará la ventana **Símbolo del sistema**. Elimine el resto de los puntos de interrupción.

En esta sección aprendió a utilizar el comando **Paso a paso por instrucciones** del depurador para depurar las funciones que son llamadas durante la ejecución del programa. Vimos cómo se puede usar el comando **Paso a paso por procedimientos** para avanzar por la llamada a una función. Utilizamos el comando **Paso a paso para salir** para continuar la ejecución hasta el final de la función actual. También aprendió que el comando **Continuar** reanuda la ejecución hasta encontrar otro punto de interrupción, o hasta que termina el programa.

I.5 La ventana Automático

La ventana **Automático** muestra las variables utilizadas en la instrucción antes ejecutada (incluyendo el valor de retorno de una función, si la hay) y las variables en la siguiente instrucción a ejecutar.

1. **Establecer puntos de interrupción.** Establezca puntos de interrupción en las líneas 14 y 22 en `main`, haciendo clic en la barra indicadora de margen.
2. **Utilizar la ventana Automático.** Inicie el depurador seleccionando **Depurar > Iniciar depuración**. Cuando el depurador entre en el modo de interrupción en la línea 14, abra la ventana **Automático** (figura I.22), seleccionando **Depurar > Ventanas > Automático**. Como acaba de empezar la ejecución del programa, la ventana **Automático** sólo lista las variables en la siguiente instrucción que se va a ejecutar; en este caso, el objeto `cuenta1`, su valor y su tipo. Al ver los valores almacenados en un objeto, podemos verificar que el programa esté manipulando esas variables en forma correcta. Observe que `cuenta1` contiene un valor negativo extenso. Este valor, que puede ser distinto cada vez que el programa se ejecuta, es el valor de `cuenta1` sin inicializar. Este valor impredecible (y a menudo, indeseable) demuestra por qué es importante inicializar todas las variables de C++ antes de utilizarlas.
3. **Utilizar el comando Paso a paso por procedimientos.** Seleccione **Depurar > Paso a paso por procedimientos** para ejecutar la línea 14. La ventana **Automático** actualiza el valor del miembro de datos `saldo` de `cuenta1` (figura I.23) después de inicializarlo. El valor de `saldo` se muestra en color rojo (en la computadora) para indicar que acaba de cambiar.
4. **Continuar la ejecución.** Seleccione **Depurar > Continuar** para ejecutar el programa hasta el segundo punto de interrupción en la línea 22. La ventana **Automático** muestra la variable local `montoRetiro` sin inicializar (figura I.24), que tiene un valor negativo extenso.



Figura I.22 | La ventana **Automático** muestra el estado del objeto `cuenta1`.



Figura I.23 | La ventana **Automático** muestra el estado del objeto `cuenta1` después de inicializarlo.



Figura I.24 | La ventana **Automático** muestra la variable local `montoRetiro`.

5. **Introducir datos.** Seleccione **Depurar > Paso a paso por procedimientos** para ejecutar la línea 22. En el indicador de entrada del programa, escriba un valor para el monto de retiro. La ventana **Automático** actualizará el valor de la variable local `montoRetiro` con el valor que usted escribió (figura I.25).
6. **Detener el depurador.** Seleccione **Depurar > Detener depuración** para terminar la sesión de depuración. Elimine el resto de los puntos de interrupción.

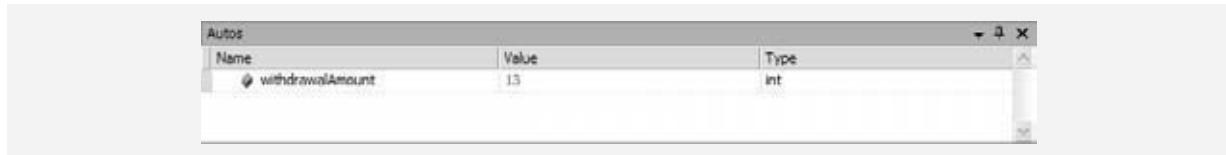


Figura I.25 | La ventana **Automático** muestra la variable local actualizada `montoRetiro`.

I.6 Repaso

En este apéndice aprendió a insertar, deshabilitar y eliminar puntos de interrupción en el depurador de Visual Studio. Los puntos de interrupción nos permiten detener la ejecución del programa para poder examinar los valores de las variables. Esta herramienta nos ayuda a localizar y corregir errores lógicos en los programas. También vio cómo utilizar las ventanas **Variables locales** y **Automático** para examinar el valor de una expresión y cómo modificar el valor de una variable. Además aprendió a usar los comandos **Paso a paso por instrucciones**, **Paso a paso por procedimientos**, **Paso a paso para salir** y **Continuar** del depurador, que se pueden utilizar para determinar si una función se está ejecutando en forma correcta. Por último, aprendió a usar la ventana **Automático** para examinar las variables que se utilizan específicamente en los comandos anteriores.

Resumen

Sección I.1 Introducción

- La mayoría de los distribuidores de compiladores de C++ proporcionan un programa de software conocido como depurador, el cual nos permite supervisar la ejecución de los programas para localizar y eliminar errores lógicos.
- Los puntos de interrupción son marcadores que pueden establecerse en cualquier línea de código ejecutable. Cuando la ejecución del programa llega a un punto de interrupción, la ejecución se detiene.
- El depurador está habilitado de manera predeterminada. Si no está habilitado, debe modificar las opciones del cuadro combinado *Configuraciones de soluciones*.

Sección I.2 Los puntos de interrupción y el comando Continuar

- Para insertar un punto de interrupción, haga clic dentro de la barra indicadora de margen enseguida de la línea de código, o haga clic con el botón derecho del ratón en esa línea de código y seleccione **Punto de interrupción | Insertar punto de interrupción**. A continuación aparecerá un círculo de color rojo en donde hizo clic, indicando que se ha establecido un punto de interrupción.
- Cuando el programa se ejecuta, suspende la ejecución en cualquier línea que contiene un punto de interrupción. Entonces se dice que está en modo de interrupción.
- Una flecha amarilla indica que esta línea contiene la siguiente instrucción que se debe ejecutar.
- Al colocar el puntero del ratón sobre el nombre de una variable, el valor que almacena esa variable se muestra en un cuadro de *Información rápida*.
- Para deshabilitar un punto de interrupción, haga clic con el botón derecho en una línea de código en la que se haya establecido un punto de interrupción y seleccione **Punto de interrupción | Deshabilitar punto de interrupción**. El punto de interrupción deshabilitado se indica mediante un círculo sin relleno.
- Para eliminar un punto de interrupción que ya no sea necesario, haga clic con el botón derecho del ratón en una línea de código en la que se haya establecido un punto de interrupción y seleccione **Punto de interrupción | Eliminar punto de interrupción**. También puede eliminar un punto de interrupción haciendo clic en el círculo en la barra indicadora de margen.

Sección I.3 Las ventanas Variables locales e Inspección

- Una vez que el programa entra al modo de interrupción, podemos explorar los valores de las variables usando la ventana **Variables locales** del depurador. Para ver la ventana **Variables locales**, seleccione **Depurar | Ventanas | Variables locales**.
- Podemos evaluar expresiones aritméticas y booleanas mediante el uso de la ventana **Inspección**.
- Las variables actualizadas se muestran en color rojo para indicar que fueron modificadas a partir del último punto de interrupción.
- Al hacer clic en el signo de suma enseguida de un objeto en la columna **Nombre** de la ventana **Variables locales** podemos ver los valores de los miembros de datos de cada objeto en forma individual.
- Podemos hacer clic en el campo **Valor** de una variable para modificar su valor en la ventana **Variables locales**.

Sección I.4 Control de la ejecución mediante los comandos Paso a paso por instrucciones, Paso a paso por procedimientos, Paso a paso para salir y Continuar

- El comando **Paso a paso por instrucciones** ejecuta la siguiente instrucción (la línea resaltada en amarillo) en el programa. Si la siguiente instrucción va a ejecutar una llamada a una función y seleccionamos **Paso a paso por instrucciones**, el control se transfiere a la función que se va a llamar.
- El comando **Paso a paso por procedimientos** se comporta de igual forma que el comando **Paso a paso por instrucciones** cuando la siguiente instrucción a ejecutar no contiene una llamada a una función. Si la siguiente instrucción a ejecutar contiene una llamada a una función, la función a la que se llama se ejecuta en su totalidad, y la flecha amarilla avanza a la siguiente línea ejecutable en la función actual.
- El comando **Paso a paso por procedimientos** ejecuta el resto de las instrucciones en la función y devuelve el control a la llamada a la función.
- El comando **Continuar** ejecuta todas las instrucciones entre la siguiente instrucción ejecutable y el siguiente punto de interrupción o el final de `main`, lo que ocurra primero.

Sección I.5 La ventana Automático

- La ventana **Automático** nos permite ver el contenido de las variables utilizadas en la última instrucción que se ejecutó. La ventana **Auto** también lista los valores en la siguiente instrucción a ejecutar.

Terminología

Automático , ventana	<i>Información rápida</i> , cuadro
barra indicadora de margen	Inspección , ventana
bug (error)	modo de interrupción
<i>Configuraciones de soluciones</i> , cuadro combinado	Paso a paso para salir , comando
Continuar , comando	Paso a paso por instrucciones , comando
Depurador	Paso a paso por procedimientos , comando
deshabilitar un punto de interrupción	punto de interrupción
flecha amarilla en modo de interrupción	Variables locales , ventana

Ejercicios de autoevaluación

I.1 Complete cada uno de los siguientes enunciados:

- Cuando el depurador suspende la ejecución del programa en un punto de interrupción, se dice que el programa está en modo _____.
- La característica _____ en Visual Studio 2005 nos permite analizar el valor de una variable, para lo cual se posiciona el ratón sobre el nombre de la variable en el código.
- Podemos examinar el valor de una expresión mediante el uso de la ventana _____ del depurador.
- El comando _____ se comporta como el comando **Paso a paso por instrucciones** cuando la siguiente instrucción a ejecutar no contiene una llamada a una función.

I.2 Indique si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.

- Cuando la ejecución del programa se suspende en un punto de interrupción, la siguiente instrucción a ejecutar es la instrucción que está después del punto de interrupción.
- Cuando el valor de una variable se modifica, se pone en color amarillo en las ventanas **Automático** y **Variables locales**.
- Durante la depuración, el comando **Paso a paso para salir** ejecuta el resto de las instrucciones en la función actual y devuelve el control del programa al lugar en donde se llamó a la función.

Respuestas a los ejercicios de autoevaluación

I.1 a) de interrupción. b) cuadro de *Información rápida*. c) **Inspección**. d) **Paso a paso por procedimientos**.

I.2 a) Falso. Cuando la ejecución de un programa se suspende en un punto de interrupción, la siguiente instrucción a ejecutarse es la instrucción que está en el punto de interrupción. b) Falso. Una variable se pone en color rojo cuando se modifica su valor. c) Verdadero.

J



Por lo tanto, debo atrapar la mosca.

—William Shakespeare

Estamos creados para cometer equivocaciones, codificados para el error.

—Lewis Thomas

Lo que anticipamos raras veces ocurre; lo que menos esperamos es lo que generalmente pasa.

—Benjamin Disraeli

Puede correr, pero no puede ocultarse.

—Joe Louis

Una cosa es mostrar a un hombre que está equivocado, y otra es darle posesión de la verdad.

—John Locke

Uso del depurador de GNU C++

OBJETIVOS

En este apéndice aprenderá a:

- Utilizar el comando `run` para ejecutar un programa en el depurador.
- Utilizar el comando `break` para establecer un punto de interrupción.
- Utilizar el comando `continue` para reanudar la ejecución.
- Utilizar el comando `print` para evaluar las expresiones.
- Utilizar el comando `set` para modificar los valores de las variables durante la ejecución de un programa.
- Utilizar los comandos `step`, `finish` y `next` para controlar la ejecución.
- Utilizar el comando `watch` para ver cómo se modifica un miembro de datos durante la ejecución de un programa.
- Utilizar el comando `delete` para eliminar un punto de interrupción o un punto de inspección.

- J.1 Introducción
- J.2 Los puntos de interrupción y los comandos `run`, `stop`, `continue` y `print`
- J.3 Los comandos `print` y `set`
- J.4 Control de la ejecución mediante los comandos `step`, `finish` y `next`
- J.5 El comando `watch`
- J.6 Repaso

[Resumen](#) | [Terminología](#) | [Ejercicios de autoevaluación](#) | [Respuestas a los ejercicios de autoevaluación](#)

J.1 Introducción

En el capítulo 2 vimos que hay dos tipos de errores (errores de compilación y errores lógicos) y el lector aprendió a eliminar los errores de compilación de su código. Los errores lógicos no evitan que un programa se compile con éxito, pero pueden hacer que el programa produzca resultados erróneos al ejecutarse. GNU incluye un programa de software conocido como **depurador**, el cual nos permite supervisar la ejecución de los programas para localizar y eliminar errores lógicos.

El depurador es una de las herramientas de desarrollo de programas más importantes. Muchos IDEs proporcionan sus propios depuradores, similares al que se incluye en GNU, o proporcionan una interfaz gráfica de usuario para el depurador de GNU. En este apéndice demostramos las características clave del depurador de GNU. En el apéndice I hablamos sobre las características y herramientas del depurador de Visual Studio. Nuestro Centro de recursos de C++ (www.deitel.com/cplusplus/) proporciona vínculos a tutoriales que pueden ayudar a los estudiantes e instructores a familiarizarse con los depuradores incluidos en otras herramientas de desarrollo.

J.2 Los puntos de interrupción y los comandos `run`, `stop`, `continue` y `print`

Para empezar con nuestro estudio del depurador, vamos a investigar los **puntos de interrupción**, que son marcadores que pueden establecerse en cualquier línea de código ejecutable. Cuando la ejecución del programa llega a un punto de interrupción, la ejecución se detiene, lo cual nos permite examinar los valores de las variables para ayudarnos a determinar si existe un error lógico. Por ejemplo, podemos examinar el valor de una variable que almacena el resultado de un cálculo para determinar si el cálculo se realizó en forma correcta. Observe que al tratar de establecer un punto de interrupción en una línea de código que no es ejecutable (como un comentario), el punto de interrupción se establecerá en la siguiente línea de código ejecutable en esa función.

Para ilustrar las características del depurador, vamos a usar la clase `Cuenta` (figuras J.1 y J.2) y el programa que se lista en la figura J.3, el cual crea y manipula un objeto de la clase `Cuenta`. La ejecución empieza en `main` (líneas 12 a 30 de la figura J.3). En la línea 14 se crea un objeto `Cuenta` con un saldo inicial de \$50.00. El constructor de `Cuenta` (líneas 10 a 22 de la figura J.2) acepta un argumento, el cual especifica el saldo inicial de la `Cuenta`. En la línea 17 de la figura J.3 se imprime el saldo inicial de la cuenta mediante el uso de la función miembro `obtenerSaldo` de `Cuenta`. En la línea 19 se declara una variable local llamada `montoRetiro`, la cual almacena un monto de retiro que se lee del usuario. En la línea 21 se pide al usuario el monto de retiro; en la línea 22 se recibe como entrada el `montoRetiro`. En la línea 25 se utiliza la función miembro `cargar` de `Cuenta` para restar el `montoRetiro` del saldo de la `Cuenta`. Por último, en la línea 28 se muestra el nuevo saldo.

```

1 // Fig. J.1: Cuenta.h
2 // Definición de la clase Cuenta.
3
4 class Cuenta
5 {
6 public:
7     Cuenta( int ); // el constructor inicializa el saldo
8     void abonar( int ); // suma un monto al saldo de la cuenta
9     void cargar( int ); // resta un monto del saldo de la cuenta

```

Figura J.1 | Archivo de encabezado para la clase `Cuenta`. (Parte 1 de 2).

```

10     int obtenerSaldo(); // devuelve el saldo de la cuenta
11 private:
12     int saldo; // miembro de datos que almacena el saldo
13 };// fin de la clase Cuenta

```

Figura J.1 | Archivo de encabezado para la clase Cuenta. (Parte 2 de 2).

```

1 // Fig. J.2: Cuenta.cpp
2 // Definiciones de las funciones miembro de la clase Cuenta.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Cuenta.h" // incluye la definición de la clase Cuenta
8
9 // el constructor de Cuenta inicializa el miembro de datos saldo
10 Cuenta::Cuenta( int saldoInicial )
11 {
12     saldo = 0; // asume que el saldo empieza en 0
13
14     // si saldoInicial es mayor que 0, establece este valor como el
15     // saldo de la cuenta; en caso contrario, saldo sigue siendo 0
16     if ( saldoInicial > 0 )
17         saldo = saldoInicial;
18
19     // si saldoInicial es negativo, imprime un mensaje de error
20     if ( saldoInicial < 0 )
21         cout << "Error: El saldo inicial no puede ser negativo.\n" << endl;
22 } // fin del constructor de Cuenta
23
24 // abona (suma) un monto al saldo de la cuenta
25 void Cuenta::abonar( int monto )
26 {
27     saldo = saldo + monto; // suma el monto al saldo
28 } // fin de la función abonar
29
30 // carga (resta) un monto al saldo de la cuenta
31 void Cuenta::cargar( int monto )
32 {
33     if ( monto <= saldo ) // monto a cargar no excede al saldo
34         saldo = saldo - monto;
35
36     else // monto a cargar excede al saldo
37         cout << "El monto a cargar excede al saldo de la cuenta.\n" << endl;
38 } // fin de la función cargar
39
40 // devuelve el saldo de la cuenta
41 int Cuenta::obtenerSaldo()
42 {
43     return saldo; // proporciona el valor de saldo a la función que hizo la llamada
44 } // fin de la función obtenerSaldo

```

Figura J.2 | Definición de la clase Cuenta.

```

1 // Fig. J.3: figJ_03.cpp
2 // Crea y manipula objetos Cuenta.
3 #include <iostream>
4 using std::cin;

```

Figura J.3 | Clase de prueba para la depuración. (Parte 1 de 2).

```

5  using std::cout;
6  using std::endl;
7
8 // incluye la definición de la clase Cuenta de Cuenta.h
9 #include "Cuenta.h"
10
11 // la función main empieza la ejecución del programa
12 int main()
13 {
14     Cuenta cuenta1( 50 ); // crea un objeto Cuenta
15
16     // muestra el saldo inicial de cada objeto
17     cout << "Saldo de cuenta1: $" << cuenta1.obtenerSaldo() << endl;
18
19     int montoRetiro; // almacena el monto de retiro que se lee del usuario
20
21     cout << "\nEscriba el monto de retiro para cuenta1: "; // indicador
22     cin >> montoRetiro; // obtiene la entrada del usuario
23     cout << "\nTratando de restar " << montoRetiro
24         << " del saldo de cuenta1\n\n";
25     cuenta1.cargar( montoRetiro ); // trata de restar de cuenta1
26
27     // muestra los saldos
28     cout << "cuenta1 saldo: $" << cuenta1.obtenerSaldo() << endl;
29
30 } // fin de main

```

Figura J.3 | Clase de prueba para la depuración. (Parte 2 de 2).

En los siguientes pasos, utilizaremos puntos de interrupción y varios comandos del depurador para examinar el valor de la variable `montoRetiro` declarada en la línea 19 de la figura J.3.

1. **Compilar el programa para depurarlo.** Para utilizar el depurador, debe compilar su programa con la opción `-g`, la cual genera información adicional que el depurador necesita para ayudar al programador a depurar sus programas. Para ello, escriba

```
g++ -g -o figJ_03 figJ_03.cpp Cuenta.cpp
```

2. **Iniciar el depurador.** Escriba `gdb figJ_03` (figura J.4). El comando `gdb` inicia el depurador y muestra el indicador (`gdb`) en el que se pueden escribir comandos.
3. **Ejecutar un programa en el depurador.** Ejecute el programa a través del depurador, escribiendo `run` (figura J.5). Si no establece puntos de interrupción antes de ejecutar su programa en el depurador, el programa se ejecutará hasta completarse.
4. **Insertar puntos de interrupción mediante el depurador de GNU.** Establezca un punto de interrupción en la línea 17 del archivo `FigJ_03.cpp`, escribiendo `break 17`. El comando `break` inserta un punto de interrupción

```

$ gdb FigJ_03
GNU gdb 6.3-debian
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...Using host libthread_db
library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb)

```

Figura J.4 | Iniciar el depurador para ejecutar el programa.

```
(gdb)run
Starting program: /home/nuke/ApJ/FigJ_03
Saldo de cuenta1: $50

Escriba el monto de retiro para cuenta1: 13
tratando de restar 13 del saldo de cuenta1
Saldo de cuenta1: $37

Program exited normally.
(gdb)
```

Figura J.5 | Ejecutar el programa sin establecer puntos de interrupción.

en el número de línea especificado como su argumento (es decir, 17). Puede establecer todos los puntos de interrupción que sean necesarios. Cada punto de interrupción se identifica con base en el orden en el que se creó. El primer punto de interrupción se conoce como **Breakpoint 1**. Establezca otro punto de interrupción en la línea 25, escribiendo `break 25` (figura J.6). Este nuevo punto de interrupción se conoce como **Breakpoint 2**. Cuando el programa se ejecuta, suspende la ejecución en cualquier línea que contiene un punto de interrupción y el depurador entra en **modo de interrupción**. Pueden establecerse puntos de interrupción, incluso hasta después de que haya empezado el proceso de depuración. [Nota: si no tiene un listado enumerado para su código, puede utilizar el comando `list` para imprimirlo con números de línea. Para obtener más información acerca del comando `list`, escriba `help list` del indicador `gdb`].

5. **Ejecutar el programa e iniciar el proceso de depuración.** Escriba `run` para ejecutar su programa y empezar el proceso de depuración (figura J.7). El depurador entra al modo de interrupción cuando la ejecución llega al punto de interrupción en la línea 17. En este punto, el depurador notifica que ha llegado a un punto de interrupción y muestra el código fuente en esa línea (17), que será la siguiente instrucción en ejecutarse.
6. **Utilizar el comando `continue` para reanudar la ejecución.** Escriba `continue`. El comando `continue` hace que el programa continúe ejecutándose hasta llegar al siguiente punto de interrupción (línea 25). Escriba `13` en el indicador. El depurador le notificará cuando la ejecución llegue al segundo punto de interrupción (figura J.8). Observe que la salida normal de `FigJ_03` aparece entre los mensajes del depurador.
7. **Examinar el valor de una variable.** Escriba `print montoRetiro` para mostrar el valor actual almacenado en la variable `montoRetiro` (figura J.9). El comando `print` nos permite espiar dentro de la computadora el valor de una de las variables. Esto se puede utilizar para ayudarnos a buscar y eliminar los errores lógicos en el código. En este caso, el valor de la variable es `13`; el valor que escribió se asignó a la variable `montoRetiro` en la línea 22 de la figura J.3. A continuación, use `print` para mostrar el contenido del objeto `cuenta1`. Cuando se muestra un objeto con `print`, se colocan llaves alrededor de los miembros de datos del objeto. En este caso, hay un solo miembro de datos (`saldo`) que tiene un valor de `50`.

```
(gdb) break 17
Breakpoint 1 at 0x80486f6: file FigJ_03.cpp, line 17.
(gdb) break 25
Breakpoint 2 at 0x8048799: file FigJ_03.cpp, line 25.
(gdb)
```

Figura J.6 | Establecer dos puntos de interrupción en el programa.

```
(gdb) run
Starting program: /home/nuke/ApJ/FigJ_03
Breakpoint 1, main () at FigJ_03.cpp:17
17          cout << "Saldo de cuenta1: $" << cuenta1.obtenerSaldo() << endl;
(gdb)
```

Figura J.7 | Ejecutar el programa hasta que llegue al primer punto de interrupción.

```
(gdb) continue
Continuing.
Saldo de cuenta1: $50
Escriba el monto de retiro para cuenta1: 13
tratando de restar 13 del saldo de cuenta1

Breakpoint 2, main () at FigJ_03.cpp:25
25      cuenta1.cargar( montoRetiro ); // trata de restar de cuenta1
(gdb)
```

Figura J.8 | La ejecución continúa hasta llegar al segundo punto de interrupción.

```
(gdb) print montoRetiro
$2 = 13
(gdb) print cuenta1
$3 = {saldo = 50}
(gdb)
```

Figura J.9 | Imprimir los valores de las variables.

8. **Utilizar variables de conveniencia.** Cuando utilizamos `print`, el resultado se almacena en una variable de conveniencia tal como `$1`. Las variables de conveniencia son variables temporales creadas por el depurador, que se nombran mediante el uso de un signo de dólares seguido de un entero. Las variables de conveniencia se pueden utilizar para realizar operaciones aritméticas y evaluar expresiones booleanas. Escriba `print $1`. El depurador muestra el valor de `$1` (figura J.10), que contiene el valor de `montoRetiro`. Observe que al imprimir el valor de `$1` se crea una nueva variable de conveniencia: `$3`.
9. **Continuar la ejecución del programa.** Escriba `continue` para reanudar la ejecución del programa. El depurador no encuentra puntos de interrupción adicionales, por lo que continúa su ejecución y termina en forma normal (figura J.11).
10. **Eliminar un punto de interrupción.** Puede mostrar una lista de todos los puntos de interrupción en el programa, escribiendo `info break`. Para eliminar un punto de interrupción, escriba `delete` seguido de un espacio y del número del punto de interrupción a eliminar. Para eliminar el primer punto de interrupción, escriba `delete 1`. Elimine el segundo punto de interrupción también. Ahora escriba `info break` para listar el resto de los puntos de interrupción en el programa. El depurador debe indicar que no hay puntos de interrupción establecidos (figura J.12).
11. **Ejecutar el programa sin puntos de interrupción.** Escriba `run` para ejecutar el programa. Escriba el valor `13` en el indicador. Como eliminó correctamente los dos puntos de interrupción, los resultados del programa se muestran sin que el depurador entre en el modo de interrupción (figura J.13).

```
(gdb) print $1
$3 = 13
(gdb)
```

Figura J.10 | Imprimir una variable de conveniencia.

```
(gdb) continue
Continuing.
Saldo de cuenta1: $37.
Program exited normally.
(gdb)
```

Figura J.11 | Terminar la ejecución del programa.

```
(gdb) info break
Num Type Disp Enb Address What
1 breakpoint keep y 0x080486f6 in main at FigJ_03.cpp:17
breakpoint already hit 1 time
2 breakpoint keep y 0x08048799 in main at FigJ_03.cpp:25
breakpoint already hit 1 time
(gdb) delete 1
(gdb) delete 2
(gdb) info break
No breakpoints or watchpoints.
(gdb)
```

Figura J.12 | Ver y eliminar puntos de interrupción.

```
(gdb) run
Starting program: /home/nuke/ApJ/FigJ_03
Saldo de cuenta1: $50
Escriba el monto de retiro para cuenta1: 13
tratando de restar 13 del saldo de cuenta1
Saldo de cuenta1: $37
Program exited normally.
(gdb)
```

Figura J.13 | Ejecución del programa sin puntos de interrupción establecidos.

12. **Utilizar el comando quit.** Use el comando `quit` para terminar la sesión de depuración (figura J.14). Este comando hace que el depurador termine.

En esta sección, utilizó el comando `gdb` para iniciar el depurador y el comando `run` para empezar a depurar un programa. Estableció un punto de interrupción en un número de línea específico en la función `main`. El comando `break` también se puede utilizar para establecer un punto de interrupción en un número de línea en otro archivo, o en una función específica. Al escribir `break`, después el nombre de archivo, un signo de dos puntos y el número de línea, se establecerá un punto de interrupción en una línea de otro archivo. Al escribir `break` y después el nombre de una función, el depurador entrará al modo de interrupción cada vez que se haga una llamada a esa función.

Además, en esta sección vimos cómo el comando `help list` proporciona más información acerca del comando `list`. Si tiene dudas acerca del depurador o cualquiera de sus comandos, escriba `help` o `help` seguido del nombre del comando para obtener más información.

Por último, examinamos las variables con el comando `print` y eliminamos puntos de interrupción con el comando `delete`. Aprendió a utilizar el comando `continue` para reanudar la ejecución después de llegar a un punto de interrupción, y el comando `quit` para finalizar el depurador.

```
(gdb) quit
$
```

Figura J.14 | Salir del depurador usando el comando `quit`.

J.3 Los comandos `print` y `set`

En la sección anterior aprendió a utilizar el comando `print` del depurador para examinar el valor de una variable durante la ejecución del programa. En esta sección aprenderá a utilizar el comando `print` para examinar el valor de expresiones más complejas. También aprenderá acerca del comando `set`, que nos permite asignar nuevos valores a las variables. Asumimos que está trabajando en el directorio que contiene los ejemplos de este apéndice y que los ha compilado para depuración con la opción `-g` del compilador.

1. **Iniciar la depuración.** Escriba `gdb figJ_03` para iniciar el depurador de GNU.
2. **Insertar un punto de interrupción.** Establezca un punto de interrupción en la línea 25 del código fuente, escribiendo `break 25` (figura J.15).
3. **Ejecutar el programa y llegar a un punto de interrupción.** Escriba `run` para iniciar el proceso de depuración (figura J.16). Esto hará que `main` se ejecute hasta llegar al punto de interrupción de la línea 25. Se suspenderá la ejecución del programa, y éste entrará al modo de interrupción. La instrucción en la línea 25 es la siguiente instrucción que se va a ejecutar.
4. **Evaluar expresiones aritméticas y booleanas.** En la sección J.2 vimos que, una vez que el depurador entra al modo de interrupción, podemos explorar los valores de las variables del programa mediante el comando `print`. También podemos usar `a print` para evaluar expresiones aritméticas y booleanas. Escriba `print montoRetiro - 2`. Esta expresión devuelve el valor 11 (figura J.17), pero en realidad no modifica el valor de `montoRetiro`. Escriba `print montoRetiro == 11`. Las expresiones que contienen el símbolo `==` devuelven valores `bool`. El valor devuelto es `false` (figura J.17), debido a que `montoRetiro` aún contiene 13.
5. **Modificar valores.** Puede modificar los valores de las variables durante la ejecución del programa en el depurador. Esto puede ser útil para experimentar con distintos valores y para localizar errores lógicos. Puede utilizar el comando `set` del depurador para modificar el valor de una variable. Escriba `set montoRetiro = 42` para modificar el valor de `montoRetiro`, y después escriba `print montoRetiro` para mostrar su nuevo valor (figura J.18).
6. **Ver el resultado del programa.** Escriba `continue` para reanudar la ejecución del programa. A continuación se ejecutará la línea 25 de la figura J.3, pasando `montoRetiro` a la función miembro `cargar` de `Cuenta`. Después la función `main` muestra el nuevo saldo. Observe que el resultado es \$8 (figura J.19). Esto muestra que el paso anterior modificó el valor de `montoRetiro`, del valor 13 que introdujo el usuario a 42.
7. **Utilizar el comando `quit`.** Use el comando `quit` para terminar la sesión de depuración (figura J.20). Este comando hace que el depurador termine.

```
(gdb) break 25
Breakpoint 1 at 0x8048799: file FigJ_03.cpp, line 25.
(gdb)
```

Figura J.15 | Establecer un punto de interrupción en el programa.

```
(gdb) run
Starting program: /home/nuke/ApJ/FigJ_03
Saldo de cuenta1: $50

Escriba el monto de retiro para cuenta1: 13
tratando de restar 13 del saldo de cuenta1

Breakpoint 1, main () at FigJ_03.cpp:25
25      cuenta1.cargar( montoRetiro ); // trata de restar de cuenta1
(gdb)
```

Figura J.16 | Ejecutar el programa hasta llegar al punto de interrupción en la línea 25.

```
(gdb) print montoRetiro - 2
$1 = 11
(gdb) print montoRetiro == 11
$2 = false
(gdb)
```

Figura J.17 | Imprimir expresiones con el depurador.

```
(gdb) set montoRetiro = 42
(gdb) print montoRetiro
$3 = 42
(gdb)
```

Figura J.18 | Establecer el valor de una variable mientras el programa está en modo de interrupción.

```
(gdb) continue
Continuing.
Saldo de cuenta: $8
Program exited normally.
(gdb)
```

Figura J.19 | Utilizar una variable modificada en la ejecución de un programa.

```
(gdb) quit
$
```

Figura J.20 | Salir del depurador mediante el comando `quit`.

En esta sección utilizamos el comando `print` del depurador para evaluar expresiones aritméticas y booleanas. También aprendió a utilizar el comando `set` para modificar el valor de una variable durante la ejecución de un programa.

J.4 Control de la ejecución mediante los comandos step, finish y next

Algunas veces es necesario ejecutar un programa línea por línea para encontrar y corregir los errores. Puede ser útil avanzar a través de una porción del programa de esta forma, para verificar que el código de una función se ejecute correctamente. Los comandos en esta sección nos permiten ejecutar una función línea por línea, ejecutar todas las instrucciones de una función a la vez o ejecutar sólo el resto de las instrucciones de una función (si ya hemos ejecutado algunas instrucciones dentro de la función).

1. **Iniciar el depurador.** Para iniciar el depurador, escriba `gdb figJ_03`.
2. **Establecer un punto de interrupción.** Escriba `break 25` para establecer un punto de interrupción en la línea 25.
3. **Ejecutar el programa.** Para ejecutar el programa escriba `run`, y después escriba 13 en el indicador. Una vez que el programa muestra sus dos mensajes de salida, el depurador indica que se ha llegado al punto de interrupción y muestra el código en la línea 25. Después, el depurador se detiene y espera a que se introduzca el siguiente comando.
4. **Uso del comando step.** El comando `step` ejecuta la siguiente instrucción en el programa. Si la siguiente instrucción a ejecutar es una llamada a una función, el control se transfiere a la función que se llamó. El comando `step` nos permite entrar a una función y estudiar sus instrucciones individuales. Por ejemplo, podemos utilizar los comandos `print` y `set` para ver y modificar las variables dentro de la función. Escriba `step` para entrar a la función miembro `debit` de la clase `Cuenta` (figura J.2). El depurador indica que se ha completado el paso y muestra la siguiente instrucción ejecutable (figura J.21); en este caso, la línea 33 de la clase `Cuenta` (figura J.2).

```
(gdb) step
Cuenta::cargar (this=0xbff81700, monto=13) at Cuenta.cpp:33
33 if ( monto <= saldo ) // monto a cargar no excede al saldo
(gdb)
```

Figura J.21 | Utilizar el comando `step` para entrar a una función.

5. **Uso del comando `finish`.** Después de haber avanzado paso a paso en la función miembro `cargar`, escriba `finish`. Este comando ejecuta el resto de las instrucciones en la función y devuelve el control al lugar desde donde se hizo la llamada a la función. El comando `finish` ejecuta el resto de las instrucciones en la función miembro `cargar`, y después se detiene en la línea 28 de `main` (figura J.22). En las funciones extensas, tal vez sea conveniente ver unas cuantas líneas clave de código, y después continuar depurando el código de la función que hizo la llamada. El comando `finish` es útil para situaciones en las que no queremos avanzar a través del resto de una función línea por línea.
6. **Utilizar el comando `continue` para reanudar la ejecución.** Escriba el comando `continue` para reanudar la ejecución hasta que termine el programa.
7. **Ejecutar el programa de nuevo.** Los puntos de interrupción persisten hasta el final de la sesión de depuración en la que se establecen. Por lo tanto, el punto de interrupción que estableció en el *paso 2* sigue ahí. Escriba `run` para ejecutar el programa y escriba `13` en el indicador. Como en el *paso 3*, el programa se ejecuta hasta llegar al punto de interrupción en la línea 25, y después el depurador se detiene y espera al siguiente comando (figura J.23).
8. **Utilizar el comando `next`.** Escriba `next`. Este comando se comporta como el comando `step`, excepto cuando la siguiente instrucción a ejecutar contiene una llamada a una función. En ese caso, la función que se llamó se ejecuta en su totalidad y el programa avanza a la siguiente línea ejecutable después de la llamada a la función (figura J.24). En el *paso 4*, el comando `step` entra a la función que se llamó. En este ejemplo, el comando `next` ejecuta la función miembro `cargar` de `Cuenta`, y después el depurador se detiene en la línea 28.
9. **Utilizar el comando `quit`.** Use el comando `quit` para terminar la sesión de depuración (figura J.25). Mientras el programa está en ejecución, este comando hace que el programa termine de inmediato en vez de ejecutar el resto de las instrucciones en `main`.

```
(gdb) finish
Run till exit from #0 Cuenta::cargar (this=0xbff81700, monto=13) at
Cuenta.cpp:33
0x080487a9 in main () at FigJ_03.cpp:25
25 cuenta1.cargar( montoRetiro ); // trata de restar de cuenta1
(gdb)
```

Figura J.22 | Utilizar el comando `terminar` para completar la ejecución de una función y regresar a la función que hizo la llamada.

```
(gdb) run
Starting program: /home/nuke/ApJ/FigJ_03
Saldo de cuenta1: $50

Escriba el monto de retiro para cuenta1: 13
tratando de restar 13 del saldo de cuenta1

Breakpoint 1, main () at FigJ_03.cpp:25
25     cuenta1.cargar( montoRetiro ); // tratando de restar de cuenta1
(gdb)
```

Figura J.23 | Reiniciar el programa.

```
(gdb) next
28     cout << "Saldo de cuenta1: $" << cuenta1.obtenerSaldo() << endl;
(gdb)
```

Figura J.24 | Utilizar el comando `next` para ejecutar una función en su totalidad.

```
(gdb) quit
The program is running. Exit anyway? (y or n) y
$
```

Figura J.25 | Salir del depurador usando el comando `quit`.

En esta sección utilizamos los comandos `step` y `finish` del depurador para depurar las funciones que se llaman durante la ejecución de un programa. Vimos cómo el comando `next` puede avanzar paso a paso por una llamada a una función. También aprendió que el comando `quit` termina una sesión de depuración.

J.5 El comando `watch`

El comando `watch` indica al depurador que debe inspeccionar un miembro de datos. Cuando ese miembro de datos esté a punto de cambiar, el depurador se lo notificará al programador. En esta sección utilizaremos el comando `watch` para ver cómo se modifica el miembro de datos `saldo` del objeto `Cuenta` durante la ejecución.

1. **Iniciar el depurador.** Inicie el depurador escribiendo `gdb figJ_03`.
2. **Establecer un punto de interrupción y ejecutar el programa.** Escriba `break 14` para establecer un punto de interrupción en la línea 14. Después ejecute el programa con el comando `run`. El depurador y el programa se detendrán en el punto de interrupción en la línea 14 (figura J.26).
3. **Inspeccionar un miembro de datos de una clase.** Establezca un punto de inspección en el miembro de datos `saldo` de `cuenta1`, escribiendo `watch cuenta1.saldo` (figura J.27). Este punto de inspección se etiqueta como `watchpoint 2`, debido a que los puntos de inspección se etiquetan con la misma secuencia de números que los puntos de interrupción. Puede establecer un punto de inspección en cualquier variable o miembro de datos de un objeto que se encuentre actualmente dentro del alcance. Cada vez que cambia el valor de una variable que se está inspeccionando, el depurador entra al modo de interrupción y notifica al programador que el valor se ha modificado.
4. **Ejecutar el constructor.** Use el comando `next` para ejecutar el constructor e inicializar el miembro de datos `saldo` del objeto `cuenta1`. El depurador indica que el valor del miembro de datos `saldo` se modificó, muestra los valores anterior y nuevo, y entra al modo de interrupción en la línea 20 (figura J.28).
5. **Salir del constructor.** Escriba `finish` para completar la ejecución del constructor y regresar a `main`.
6. **Retirar dinero de la cuenta.** Escriba `continue` para continuar la ejecución y escribir un valor de retiro en el indicador. El programa se ejecutará en forma normal. En la línea 25 de la figura J.3 se hace una llamada a la función miembro `cargar` de `Cuenta` para reducir el `saldo` del objeto `Cuenta` en base a un `monto` especificado. En la línea 34 de la figura J.2 dentro de la función `cargar` se modifica el valor de `saldo`. El depurador notifica al programador acerca de este cambio y entra al modo de interrupción (figura J.29).

```
(gdb) break 14
Breakpoint 1 at 0x80486e5: file FigJ_03.cpp, line 14.
(gdb) run
Starting program: /home/nuke/ApJ/FigJ_03

Breakpoint 1, main () at FigJ_03.cpp:14
14      Cuenta cuenta1( 50 ); // crea un objeto Cuenta
(gdb)
```

Figura J.26 | Ejecutar el programa hasta el primer punto de interrupción.

```
(gdb) watch cuenta1.saldo
Hardware watchpoint 2: cuenta1.saldo
(gdb)
```

Figura J.27 | Establecer un punto de inspección en un miembro de datos.

```
(gdb) next
Hardware watchpoint 2: cuenta1.saldo

Old value = 0
New value = 50
Cuenta (this=0xbfcfd6b90, saldoInicial=50) at Cuenta.cpp:20
20      if ( saldoInicial < 0 )
(gdb)
```

Figura J.28 | Avanzar paso a paso por el constructor.

```
(gdb) continue
Continuing.
Saldo de cuenta1: $50

Escriba el monto de retiro para cuenta1: 13
tratando de restar 13 del saldo de cuenta1

Hardware watchpoint 2: cuenta1.saldo

Old value = 50
New value = 37
0x0804893b in Cuenta::cargar (this=0xbfcfd6b90, monto=13) at Cuenta.cpp:34
34      saldo = saldo - monto;
(gdb)
```

Figura J.29 | Entrar al modo de interrupción cuando se modifica una variable.

7. **Continuar la ejecución.** Escriba `continue`; el programa terminará de ejecutar la función `main`, ya que no intenta realizar modificaciones adicionales al `saldo`. El depurador elimina el punto de inspección en el miembro de datos `saldo` de `cuenta1`, debido a que el objeto `cuenta1` queda fuera de alcance cuando termina la función `main`. Al eliminar el punto de inspección, el depurador entra al modo de interrupción. Escriba `continue` otra vez para terminar la ejecución del programa (figura J.30).
8. **Reiniciar el depurador y restablecer el punto de inspección en la variable.** Escriba `run` para iniciar el depurador. Una vez más, establezca un punto de inspección en el miembro de datos `saldo` de `cuenta1`, escribiendo `watch cuenta1.saldo`. Este punto de inspección se etiqueta como `watchpoint 3`. Escriba `continue` para reanudar la ejecución (figura J.31).
9. **Eliminar el punto de inspección en el miembro de datos.** Suponga que desea inspeccionar un miembro de datos sólo durante parte de la ejecución de un programa. Puede eliminar el punto de inspección del depurador en la variable `saldo` si escribe `delete 3` (figura J.32). Escriba `continue`; el programa terminará su ejecución sin volver a entrar al modo de interrupción.

```
(gdb) continue
Continuing.
Saldo de cuenta1: $37

Punto de inspección 2 eliminado porque el programa ha abandonado el bloque en
el cual esta expresión es válida.
0xb7da0595 in exit () from /lib/tls/i686/cmov/libc.so.6
(gdb) continue
Continuing.

Program exited normally.
(gdb)
```

Figura J.30 | Continuación hasta el final del programa.

```
(gdb) run
Starting program: /home/nuke/ApJ/FigJ_03

Breakpoint 1, main () at FigJ_03.cpp:14
14     Cuenta cuenta1( 50 ); // crea un objeto Cuenta
(gdb) watch cuenta1.saldo
Hardware watchpoint 3: cuenta1.saldo
(gdb) continue
Continuing.
Hardware watchpoint 3: cuenta1.saldo

Old value = 0
New value = 50
Cuenta (this=0xbfd8eb90, saldoInicial=50) at Cuenta.cpp:20
20         if ( saldoInicial < 0 )
(gdb)
```

Figura J.31 | Restablecer el valor en un miembro de datos.

```
(gdb) delete 3
(gdb) continue
Continuing.
Saldo de cuenta1: $50

Escriba el monto de retiro para cuenta1: 13
tratando de restar 13 del saldo de cuenta1

Saldo de cuenta1: $37

Program exited normally.
(gdb)
```

Figura J.32 | Eliminar un punto de inspección.

En esta sección utilizamos el comando `watch` para permitir que el depurador notifique al programador cuando se modifica el valor de una variable. Utilizamos el comando `delete` para eliminar un punto de inspección en un miembro de datos antes del final del programa.

J.6 Repaso

En este apéndice aprendió a insertar y eliminar puntos de interrupción en el depurador. Los puntos de interrupción nos permiten detener la ejecución del programa para poder examinar los valores de las variables mediante el comando `print` del depurador, que nos puede ayudar a localizar y corregir errores lógicos en los programas. Utilizó el comando `print` para examinar el valor de una expresión, y utilizó el comando `set` para modificar el valor de una variable. También aprendió acerca de los comandos del depurador (incluyendo a `step`, `finish` y `next`) que se pueden utilizar para determinar si una función se está ejecutando en forma correcta. Aprendió también a utilizar el comando `watch` para llevar el registro de un miembro de datos a lo largo del alcance de ese miembro de datos. Por último, aprendió a utilizar el comando `info break` para listar todos los puntos de interrupción y puntos de inspección establecidos para un programa, y el comando `delete` para eliminar los puntos de interrupción y puntos de inspección individuales.

Resumen

Sección J.1 Introducción

- GNU incluye un programa de software llamado depurador, el cual nos permite supervisar la ejecución de sus programas para localizar y eliminar errores lógicos.

Sección J.2 Los puntos de interrupción y los comandos *run*, *stop*, *continue* y *print*

- El depurador de GNU trabaja sólo con los archivos ejecutables que se compilaron con la opción `-g` del compilador, la cual genera información que el depurador utiliza para ayudar al programador a depurar sus programas.
- El comando `gdb` inicia el depurador de GNU y permite al programador utilizar sus características. El comando `run` ejecuta un programa a través del depurador.
- Los puntos de interrupción son marcadores que se pueden establecer en cualquier línea ejecutable de código. Cuando la ejecución del programa llega a un punto de interrupción, la ejecución se detiene.
- El comando `break` inserta un punto de interrupción en el número de línea especificado después del comando.
- Cuando el programa se ejecuta, suspende la ejecución en cualquier línea que contiene un punto de interrupción y se dice que se encuentra en modo de interrupción.
- El comando `continue` hace que el programa continúe su ejecución hasta llegar al siguiente punto de interrupción.
- El comando `print` nos permite espiar dentro de la computadora, el valor de una de las variables de un programa.
- Cuando se utiliza el comando `print`, el resultado se almacena en una variable de conveniencia tal como `$1`. Las variables de conveniencia son variables temporales que se pueden utilizar en el proceso de depuración para realizar operaciones aritméticas y evaluar expresiones booleanas.
- Para mostrar una lista de todos los puntos de interrupción en el programa, escriba `info break`.
- Para eliminar un punto de interrupción, escriba `delete` seguido de un espacio y el número del punto de interrupción a eliminar.

Sección J.3 Los comandos *print* y *set*

- Utilice el comando `quit` para terminar la sesión de depuración.
- El comando `set` nos permite asignar nuevos valores a las variables.

Sección J.4 Control de la ejecución mediante los comandos *step*, *finish* y *next*

- El comando `step` ejecuta la siguiente instrucción en el programa. Si la siguiente instrucción a ejecutar es una llamada a una función, el control se transfiere a la función que se llamó. El comando `step` nos permite entrar a una función y estudiar las instrucciones individuales de esa función.
- El comando `finish` ejecuta el resto de las instrucciones en la función y devuelve el control al lugar en el que se hizo la llamada a la función.
- El comando `next` se comporta igual que el comando `step`, excepto cuando la siguiente instrucción a ejecutar contiene una llamada a una función. En este caso, la función que se llamó se ejecuta en su totalidad y el programa avanza a la siguiente línea ejecutable después de la llamada a la función.

Sección J.5 El comando *watch*

- El comando `watch` establece un punto de inspección en cualquier variable o miembro de datos de un objeto que se encuentre actualmente dentro del alcance, durante la ejecución del depurador. Cada vez que se modifica el valor de una variable que está bajo inspección, el depurador entra al modo de interrupción y notifica al programador que el valor se ha modificado.

Terminología

<code>break</code> , comando	<code>next</code> , comando
<code>continue</code> , comando	<code>print</code> , comando
<code>delete</code> , comando	punto de interrupción
depurador	<code>quit</code> , comando
<code>finish</code> , comando	<code>run</code> , comando
<code>-g</code> , opción del compilador	<code>set</code> , comando
<code>gdb</code> , comando	<code>step</code> , comando
<code>info break</code> , comando	<code>watch</code> , comando
modo de interrupción	

Ejercicios de autoevaluación**J.1 Complete los siguientes enunciados:**

- a) Un punto de interrupción no se puede establecer en un(a) _____.
- b) Para examinar el valor de una expresión, podemos utilizar el comando _____ del depurador.
- c) Para modificar el valor de una variable, podemos utilizar el comando _____ del depurador.
- d) Durante la depuración, el comando _____ ejecuta el resto de las instrucciones en la función actual y devuelve el control del programa al lugar en donde se hizo la llamada a la función.

- e) El comando _____ del depurador se comporta igual que el comando `step` cuando la siguiente instrucción a ejecutar no contiene una llamada a una función.
- f) El comando `watch` del depurador nos permite ver todas las modificaciones a un(a) _____.
- J.2 Indique si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.
- Cuando la ejecución de un programa se suspende en un punto de interrupción, la siguiente instrucción a ejecutar es la que está después del punto de interrupción.
 - Los puntos de inspección se pueden eliminar mediante el comando `remove` del depurador.
 - Debemos utilizar la opción `-g` del compilador a la hora de compilar programas para depurarlos.

Respuestas a los ejercicios de autoevaluación

J.1 a) línea no ejecutable. b) `print`. c) `set`. d) `finish`. e) `next`. f) miembro de datos.

J.2 a) Falso. Cuando la ejecución del programa se suspende en un punto de interrupción, la siguiente instrucción a ejecutar es la que está en el punto de interrupción. b) Falso. Los puntos de inspección se pueden eliminar mediante el comando `delete` del depurador. c) Verdadero.

Bibliografía

Para más libros, artículos y demás información sobre C++, visite el Centro de Recursos Deitel de C++ en

www.deitel.com/cplusplus

Abrahams, D. y A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools and Techniques from Boost and Beyond.* Boston, MA: Addison-Wesley Professional, 2004.

Alexandrescu, A. *Modern C++ Design: Generic Programming and Design Patterns Applied.* Boston, MA: Addison-Wesley Professional, 2001.

Alhir, S. *UML in a Nutshell.* Cambridge, MA: O'Reilly & Associates, Inc., 1998.

Almarode, J. "Object Security". *Smalltalk Report.* Vol. 5, Núm. 3, noviembre/diciembre 1995, 15-17.

American National Standard, Programming Language C++. (Documento ANSI ISO/IEC 14882), Nueva York, NY: Instituto Estadounidense de Estándares Nacionales, 1998.

Anderson, A. E. y W. J. Heinze. *C++ Programming and Fundamental Concepts.* Englewood Cliffs, NJ: Prentice Hall, 1992.

Arciniegas, F. *C++ XML.* Indianapolis, IN: Sams, 2001.

Arlow, J. e I. Neustadt. *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design, Second Edition.* Boston, MA: Addison-Wesley Professional, 2005.

Astle, D. y K. Hawkins, *Beginning OpenGL Game Programming,* Boston, MA: Course Technology PTR, 2004.

Bar-David, T. *Object-Oriented Design for C++.* Englewood Cliffs, NJ: Prentice Hall, 1993.

Beck, K. "Birds, Bees, and Browsers—Obvious Sources of Objects". *The Smalltalk Report.* Vol. 3, Núm. 8, junio 1994, 13.

Becker, P. "Shrinking the Big Switch Statement". *Windows Tech Journal* Vol. 2, Núm. 5, mayo 1993, 26-33.

Becker, P. "Conversion Confusion". *C++ Report.* Octubre 1993, 26-28.

Berard, E. V. *Essays on Object-Oriented Software Engineering: Volume I.* Englewood Cliffs, NJ: Prentice Hall, 1993.

Binder, R. V. "State-Based Testing". *Object Magazine.* Vol. 5, Núm. 4, agosto 1995, 75-78.

Binder, R. V. "State-Based Testing: Sneak Paths and Conditional Transitions". *Object Magazine.* Vol. 5, Núm. 6, octubre 1995, 87 a 89.

Blum, A. *Neural Networks in C++: An Object-Oriented Framework for Building Connectionist Systems.* Nueva York, NY: John Wiley & Sons, 1992.

Booch, G. *Object Solutions: Managing the Object-Oriented Project.* Reading, MA: Addison-Wesley, 1996.

Booch, G. *Object-Oriented Analysis and Design with Applications, Third Edition.* Reading: MA: Addison-Wesley, 2005.

Booch, G., J. Rumbaugh e I. Jacobson. *The Unified Modeling Language User Guide.* Reading, MA: Addison-Wesley, 1999.

Cargill, T. *C++ Programming Style,* Reading, MA: Addison-Wesley, 1993.

- Carroll, M. D. y M. A. Ellis. *Designing and Coding Reusable C++*. Reading, MA: Addison-Wesley, 1995.
- Chonoles, M. J. y J. A. Schardt. *UML 2 for Dummies*. Nueva York, NY: Wiley Publishing, Inc., 2003.
- Coplien, J. O. y D. C. Schmidt. *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley, 1995.
- Dawson, M. *Beginning C++ Game Programming*. Boston, MA: Course Technology PTR, 2004.
- Deitel, H. M., P. J. Deitel y D. R. Choffnes. *Operating Systems, Third Edition*. Upper Saddle River, NJ: Prentice Hall, 2004.
- Deitel, H. M. y P. J. Deitel. *Java How to Program, Seventh Edition*. Upper Saddle River, NJ: Prentice Hall, 2007.
- Deitel, H. M. y P. J. Deitel. *C How to Program, Fifth Edition*. Upper Saddle River, NJ: Prentice Hall, 2007.
- Dennis, A., B. H. Wixom y D. Tegarden. *Systems Analysis and Design with UML Version 2.0: An Object-Oriented Approach, Second Edition*. Nueva York, NY: Wiley Publishing, Inc., 2004.
- Dewhurst, S. C. *C++ Common Knowledge: Essential Intermediate Programming*. Boston, MA: Addison-Wesley Professional, 2005.
- Donovan, S. *C++ Example*. Indianapolis, IN: Que, 2001.
- Duncan, R. "Inside C++: Friend and Virtual Functions, and Multiple Inheritance". *PC Magazine*, 15 octubre 1991, 417-420.
- Ellis, M. A. y B. Stroustrup. *The Annotated C++ Reference Manual*. Reading, MA: Addison-Wesley, 1990.
- Embley, D. W., B. D. Kurtz y S. N. Woodfield. *Object-Oriented Systems Analysis: A Model-Driven Approach*. Englewood Cliffs, NJ: Yourdon Press, 1992.
- Eriksson, H., D. Fado, B. Lyons y M. Penker. *UML 2 Toolkit*. Nueva York, NY: Wiley Publishing, Inc., 2003.
- Firesmith, D. G. y B. Henderson-Sellers. "Clarifying Specialized Forms of Association in UML and OML". *Journal of Object-Oriented Programming*. Mayo 1998: 47-50.
- Flamig, B. *Practical Data Structures in C++*. Nueva York, NY: John Wiley & Sons, 1993.
- Fowler, M. *UML Distilled: A Brief Guide to the Standard Object Modeling Language, Third Edition*. Reading, MA: Addison-Wesley, 2004.
- Giancola, A. y L. Baker. "Bit Arrays with C++". *The C Users Journal*. Vol. 10, Núm. 7, julio 1992, 21-26.
- Glass, G. y B. Shuchert. *The STL <Primer>*. Upper Saddle River, NJ: Prentice Hall PTR, 1995.
- Gooch, T. "Obscure C++". *Inside Microsoft Visual C++*. Vol. 6, Núm. 11, noviembre 1995, 13 a 15.
- Henricson, M. y E. Nyquist. *Industrial Strength C++: Rules and Recommendations*. Upper Saddle River, NJ: Prentice Hall, 1997.
- International Standard: Programming Languages—C++*. ISO/IEC 14882:1998. Nueva York, NY: Instituto Estadounidense de Estándares Nacionales, 1998.
- Jacobson, I. "Is Object Technology Software's Industrial Platform?" *IEEE Software Magazine*. Vol. 10, Núm. 1, enero 1993, 24-30.
- Jaeschke, R. *Portability and the C Language*. Indianapolis, IN: Sams Publishing, 1989.
- Johnson, L. J. "Model Behavior". *Enterprise Development*. Mayo 2000: 20-28.
- Josuttis, N. *The C++ Standard Library: A Tutorial and Reference*. Boston, MA: Addison-Wesley, 1999.
- Karlsson, B. *Beyond the C++ Standard Library: An Introduction to Boost*. Boston, MA: Addison-Wesley Professional, 2005.
- Koenig, A. "What is C++ Anyway?" *Journal of Object-Oriented Programming*, abril/mayo 1991, 48-52.
- Koenig, A. "Implicit Base Class Conversions". *The C++ Report* Vol. 6, Núm. 5, junio 1994, 18-19.

- Koenig, A. y B. Stroustrup. "Exception Handling for C++ (Revised)", *Proceedings of the USENIX C++ Conference*, San Francisco, CA, abril 1990.
- Koenig, A. y B. E. Moo. *Accelerated C++: Practical Programming Example*. Boston, MA: Addison-Wesley Professional, 2000.
- Koenig, A. y B. E. Moo. *Ruminations on C++: A Decade of Programming Insight and Experience*. Reading, MA: Addison-Wesley, 1997.
- Kruse, R. L. y A. J. Ryba. *Data Structures and Program Design in C++*. Upper Saddle River, NJ: Prentice Hall, 1999.
- Lajoie, J., S. B. Lippman y B. E. Moo. *C++ Primer, Fourth Edition*. Boston, MA: Addison-Wesley Professional, 2005.
- Langer, A. y K. Kreft. *Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference*. Reading, MA: Addison-Wesley, 2000.
- Larman, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Third Edition*. Upper Saddle River, NJ: Prentice Hall PTR, 2004.
- Lee, L., A. Lumsdaine y J. G. Siek. *The Boost Graph Library User Guide and Reference Manual*. Boston, MA: Addison-Wesley Professional, 2001.
- Lippman, S. B. y J. Lajoie. *C++ Primer, Third Edition*, Reading, MA: Addison-Wesley, 1998.
- Lorenz, M. *Object-Oriented Software Development: A Practical Guide*. Englewood Cliffs, NJ: Prentice Hall, 1993.
- Lorenz, M. "A Brief Look at Inheritance Metrics". *The Smalltalk Report* Vol. 3, Núm. 8, junio 1994, 1, 4-5.
- Malik, D. S. *C++ Programming: From Problem Analysis to Program Design, Third Edition*. Boston, MA: Course Technology, 2006.
- Martin, J. *Principles of Object-Oriented Analysis and Design*. Englewood Cliffs, NJ: Prentice Hall, 1993.
- Martin, R. C. *Designing Object-Oriented C++ Applications Using the Booch Method*. Englewood Cliffs, NJ: Prentice Hall, 1995.
- Matsche, J. J. "Object-Oriented Programming in Standard C". *Object Magazine* Vol. 2, Núm. 5, enero/febrero 1993, 71-74.
- McCabe, T. J. y A. H. Watson. "Combining Comprehension and Testing in Object-Oriented Development". *Object Magazine* Vol. 4, No. 1, marzo/abril 1994, 63-66.
- McGrath, M. *C++ Programming in Easy Steps*. Southam, Warwickshire, Reino Unido: Computer Step, 2005.
- McLaughlin, M. y A. Moore. "Real-Time Extensions to the UML". *Dr. Dobb's Journal*, diciembre 1998: 82-93.
- Melewski, D. "UML Gains Ground". *Application Development Trends*, octubre 1998: 34-44.
- Melewski, D. "UML: Ready for Prime Time?" *Application Development Trends*, noviembre 1997: 30-44.
- Melewski, D. "Wherefore and What Now, UML?" *Application Development Trends*, diciembre 1999: 61-68.
- Meyer, B. *Object-Oriented Software Construction, Second Edition*. Englewood Cliffs, NJ: Prentice Hall, 1997.
- Meyer, B. y D. Mandrioli. *Advances in Object-Oriented Software Engineering*. Englewood Cliffs, NJ: Prentice Hall, 1992.
- Meyers, S. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Third Edition*. Boston, MA: Addison-Wesley Professional, 2005.
- Meyers, S. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Reading, MA: Addison-Wesley, 2001.
- Muller, P. *Instant UML*. Birmingham, Reino Unido: Wrox Press Ltd, 1997.
- Murray, R. *C++ Strategies and Tactics*. Reading, MA: Addison-Wesley, 1993.
- Musser, D. R. y A. A. Stepanov. "Algorithm-Oriented Generic Libraries". *Software Practice and Experience*, Vol. 24, Núm. 7, julio 1994.

- Musser D. R., G. J. Derge y A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library, Second Edition*. Reading, MA: Addison-Wesley, 2001.
- Nierstrasz, O., S. Gibbs y D. Tsichritzis. "Component-Oriented Software Development". *Communications of the ACM* Vol. 35, Núm. 9, septiembre 1992, 160-165.
- Pender, T. *UML Bible*. Wiley Publishing, Inc., 2003.
- Perry, P. "UML Steps to the Plate". *Application Development Trends*, mayo 1999: 33-36.
- Pilone, D. y N. Pitman. *UML 2.0 in a Nutshell, Second Edition*. Sebastopol, CA: O'Reilly Media, Inc., 2005.
- Pittman, M. "Lessons Learned in Managing Object-Oriented Development". *IEEE Software Magazine* Vol. 10, Núm. 1, enero 1993, 43-53.
- Plauger, D. "Making C++ Safe for Threads". *The C Users Journal* Vol. 11, Núm. 2, febrero 1993, 58-62.
- Podeswa, H. *UML for the IT Business Analyst: A Practical Guide to Object-Oriented Requirements Gathering*. Boston, MA: Course Technology PTR, 2005.
- Pohl, I. *C++ Distilled: A Concise ANSI/ISO Reference and Style Guide*. Reading, MA: Addison-Wesley, 1997.
- Prata, S. *C++ Primer Plus, Fifth Edition*. Indianapolis, IN: Sams, 2004.
- Prieto-Díaz, R. "Status Report: Software Reusability". *IEEE Software* Vol. 10, Núm. 3, mayo 1993, 61-66.
- Prosise, J. "Wake Up and Smell the MFC: Using the Visual C++ Classes and Applications Framework". *Microsoft Systems Journal* Vol. 10, Núm. 6, junio 1995, 17-34.
- Ritchie, D. M. "The UNIX System: The Evolution of the UNIX Time-Sharing System". *AT&T Bell Laboratories Technical Journal* Vol. 63, Núm. 8, Parte 2, octubre 1984, 1577-1593.
- Rosler, L. "The UNIX System: The Evolution of C—Past and Future". *AT&T Laboratories Technical Journal* Vol. 63, Núm. 8, Parte 2, octubre 1984, 1685-1699.
- Robson, R. *Using the STL: The C++ Standard Template Library*. Nueva York, NY: Springer Verlag, 2000.
- Rubin, K. S. y A. Goldberg. "Object Behavior Analysis". *Communications of the ACM* Vol. 35, Núm. 9, septiembre 1992, 48-62.
- Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy y W. Lorensen. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice Hall, 1991.
- Rumbaugh, J., Jacobson, I. y G. Booch. *The Unified Modeling Language Reference Manual, Second Edition*. Reading, MA: Addison-Wesley, 2005.
- Saks, D. "Inheritance". *The C Users Journal*, mayo 1993, 81-89.
- Schildt, H. *STL Programming from the Ground Up*. Berkeley, CA: Osborne McGraw-Hill, 1999.
- Schildt, H. *The Art of C++*. Berkeley, CA: McGraw-Hill Osborne Media, 2004.
- Schlaer, S. y S. J. Mellor. *Object Lifecycles: Modeling the World in States*. Englewood Cliffs, NJ: Prentice Hall, 1992.
- Sedgwick, R. *Bundle of Algorithms in C++, Parts 1-5: Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms (Third Edition)*. Reading, MA: Addison-Wesley, 2002.
- Skelly, C. "Pointer Power in C and C++". *The C Users Journal* Vol. 11, Núm. 2, febrero 1993, 93-98.
- Snyder, A. "The Essence of Objects: Concepts and Terms". *IEEE Software Magazine* Vol. 10, Núm. 1, enero 1993, 31-42.
- Stepanov, A. y M. Lee. "The Standard Template Library", 31 octubre 1995 <www.cs.rpi.edu/~musser/doc.ps>.
- Stroustrup, B. "The UNIX System: Data Abstraction in C". *AT&T Bell Laboratories Technical Journal* Vol. 63, Núm. 8, Parte 2, Octubre 1984, 1701-1732.
- Stroustrup, B. "What is Object-Oriented Programming?" *IEEE Software* Vol. 5, Núm. 3, mayo 1988, 10-20.
- Stroustrup, B. "Parameterized Types for C++". *Proceedings of the USENIX C++ Conference*, Denver, CO, octubre 1988.

- Stroustrup, B. "Why Consider Language Extensions?: Maintaining a Delicate Balance". *The C++ Report*, septiembre 1993, 44-51.
- Stroustrup, B. "Making a vector Fit for a Standard". *The C++ Report*, octubre 1994.
- Stroustrup, B. *The Design and Evolution of C++*. Reading, MA: Addison-Wesley, 1994.
- Stroustrup, B. *The C++ Programming Language, Special Third Edition*. Reading, MA: Addison-Wesley, 2000.
- Taylor, D. *Object-Oriented Information Systems: Planning and Implementation*. Nueva York, NY: John Wiley & Sons, 1992.
- Urlocker, Z. "Polymorphism Unbounded". *Windows Tech Journal* Vol. 1, Núm. 1, enero 1992, 11-16.
- Van Camp, K. E. "Dynamic Inheritance Using Filter Classes". *The C/C++ Users Journal* Vol. 13, Núm. 6, junio 1995, 69-78.
- Vilot, M. J. "An Introduction to the Standard Template Library". *The C++ Report* Vol. 6, Núm. 8, octubre 1994.
- Voss, G. "Objects and Messages". *Windows Tech Journal*, Febrero 1993, 15-16.
- Wang, B. L. y J. Wang. "Is a Deep Class Hierarchy Considered Harmful?" *Object Magazine* Vol. 4, Núm. 7, noviembre/diciembre 1994, 35-36.
- Weisfeld, M. "An Alternative to Large Switch Statements". *The C Users Journal* Vol. 12, Núm. 4, abril 1994, 67-76.
- Weiskamp, K. y B. Flamig. *The Complete C++ Primer, Second Edition*. Orlando, FL: Academic Press, 1993.
- Wiebel, M. y S. Halladay. "Using OOP Techniques Instead of switch in C++". *The C Users Journal* Vol. 10, Núm. 10, octubre 1993, 105-112.
- Wilde, N. y R. Huitt. "Maintenance Support for Object-Oriented Programs". *IEEE Transactions on Software Engineering* Vol. 18, Núm. 12, diciembre 1992, 1038-1044.
- Wilde, N., P. Matthews y R. Huitt. "Maintaining Object-Oriented Software". *IEEE Software Magazine* Vol. 10, Núm. 1, enero 1993, 75-80.
- Wilson, G. V. y P. Lu. *Parallel Programming Using C++*. Cambridge, MA: MIT Press, 1996.
- Wilt, N. "Templates in C++". *The C Users Journal*, mayo 1993, 33-51.
- Wirfs-Brock, R., B. Wilkerson y L. Wiener. *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice Hall PTR, 1990.
- Wyatt, B. B., K. Kavi y S. Hufnagel. "Parallelism in Object-Oriented Languages: A Survey". *IEEE Software* Vol. 9, Núm. 7, noviembre 1992, 56-66.
- Yamazaki, S., K. Kajihara, M. Ito y R. Yasuhara. "Object-Oriented Design of Telecommunication Software". *IEEE Software Magazine* Vol. 10, Núm. 1, enero 1993, 81-87.
- Yuzwa, E. *Game Programming In C++: Start To Finish*. Boston, MA: Charles River Media, 2006.

Índice

Símbolos

! NOT lógico, 182
% (signo de porcentaje), 45
& (operador dirección), 343
&& (AND condicional), 180
(*), 45
(::), operador de resolución de ámbito binario, 239
*argv[], 1071
--, 139
. (operador punto), 72
| (canalización), 1067
|| (OR lógico), 181
++, 139
+=, operador de asignación de suma, 139
<, 633
<< (desplazamiento a la izquierda), 844
> (símbolo de redirección de salida), 1069
>, 633
>> (desplazamiento a la derecha), 844
>> (símbolo para agregar salida), 1069
>>, operador de extracción de flujo, 40, 43

A

abort, 692, 424, 1093
abstracción de datos, 472
abstractas
 clases, 597
 clases base, 597
acceso
 comprobado, 748
 especificador, 71
accesores, 80
acciones, 2, 110
activación, 325
actividad, 57, 112
actividades, 189
actor, 56
acumulate, 926
Ada, 10

adaptadores, 883
 de contenedores, 884, 911
administración dinámica de memoria, 466
AdSense, 6
Agile Software Development (Desarrollo Ágil de Software), 19
agregación, 100, 413
alcance 163
 de archivo, 225, 413
 de bloque, 225
 de clase, 225
 de espacio de nombres, 225
 de función, 225
 de prototipo de función, 225
<algorithm>, 899
algoritmo, 110
 de ordenamiento heapsort, 937
algoritmos, 883
 de secuencia cambiante, 892
alias de plantilla, 1020
almacenamiento libre, 466
análisis y diseño de sistemas estructurados, 11
analizar, 23
anchura
 de campo, 168
 del campo de bits, 851
AND (&) a nivel de bits, 844
anfitriones, objetos, 453
anidado, 136
anidamiento, 135
 de instrucciones de control, 114
anónima, 1081
ANSI (Instituto nacional estadounidense de estándares), 3
apilamiento de instrucciones de control, 114
aplazamiento indefinido, 367
aplicaciones de acceso instantáneo, 721
append, 749
apuntador
 constante a datos constantes, 354
 constante a datos no constantes, 353
 de posición del archivo, 717
no constante a datos constantes, 351
no constante a datos no constantes, 350
nulo, 343
suspendido, 500
apuntadores
 aritmética, 351
 inteligentes, 20, 1007
árbol de búsqueda binaria, 811
árboles binarios, 789, 810
archivo
 alcance, 225
 de código fuente, 85
 de encabezado, 85
 de flujos de entrada/salida, 37
 secuencial, 711
archivos, 709
 de acceso aleatorio, 721
argc, 1071
argumento predeterminado, 237
argumentos, 72, 1090
 de línea de comandos, 366
 tipo apuntador, paso por referencia, 346
 tipo referencia, paso por referencia, 346
aritmética de apuntadores, 351
arreglo
 asociativo, 910
 de cadenas, 366
 de m por n , 307
arreglos, 278
 bidimensionales, 307
 dinámicos, 1078
 multidimensionales, 307
artefactos, 1125
ASCII (Código estándar estadounidense para el intercambio de información), 174
asignación
 a nivel de miembros, 429
 de uno a uno, 910
asignar, 466
asíncrona, llamada, 322
asociación, 99

asociaciones, 22
 asociatividad, 46
assign, 747, 903
at, 321, 748
atexit registra, 1074
atof, 860
atoi, 860
atol, 860
 atrapar eventos inesperados, 1076
 atributos, 21
auto, 223
 auto-asignación, 500
auto_ptr, 698
autodocumente, 42
Automático, ventana, 1137

B

back, 894
back_inserter, 931
bad, 671
bad_alloc, 694
bad_cast, 700
bad_exception, 700
bad_typeid, 700
badbit, 654
base de datos, 711
BASIC, 10
basic_fstream, 652
basic_ifstream, 652
basic_ofstream, 652
basic_regex, 1000
basic_string, 746
basic_string< char >, 746
begin, 761, 887
biblioteca
 de audio OpenAL, 20, 984
 de manejo de señales, 1076
 de plantillas estándar (STL), 882
 estándar de C++, 8
 general de utilerías, 859
bibliotecas
 Boost de C++, 20, 996
 de flujos clásicos, 650
 de flujos estándar, 651
binary_function, 947
binary_search, 928
bit, 709
 bits de estado, 654
bitset, 941
 blogósfera, 6
 blogs, 6
bloque, 50, 119
 alcance, 225
bombing, 128
boolalpha, manipulador de flujo, 183

Boolean, atributo, 143
Boost Sandbox, 997
Boost.Array, 997
Boost.Bind, 998
Boost.mem_fn, 998
Boost.Random, 998
Boost.Ref, 998
Boost.Regex, 999
Boost.Smart_ptr, 999
Boost.Tuple, 999
Boost.Type_traits, 999
boost::bad_weak_ptr, 1011
boost::format_first_only, 1006
break
 comando, 1144
 instrucción, 179

Buenas prácticas de programación, 9
búfer (buffer)
 con, 652
 sin, 652
 bugs, 1128
 búsqueda, 304, 770
 clave, 770
 lineal, 304
 byte, 843
 bytes, 709

C

C++ Biblioteca estándar, 8
C++0x, 996
C99, 1016
 cabeza, 789
 cadena, 38
 cadena vacía, 79, 746
 de caracteres, 38
 literal, 38
 cadenas, 38
 calificador **const**, 283
 camello, nomenclatura, 71
Camera, 968
 campo de bits, 851
 sin nombre con anchura cero, 853
 campos, 709
 canal alfa, 957
 canalización (|), 1067
 canalizaciones, 1067
 cantidades escalares, 295
 capacidad, 752
capacity, 754, 896
 carácter
 de escape, 38
 de palabra, 1000
 de relleno, 411
 nulo, 291
 tilde (~), 423

caracteres, 709
 conjunto, 709
 de relleno, 660
 delimitadores, 383
 especiales, 376
 carga, 11, 13
 cargador, 13
Case, etiquetas, 175
 casi contenedores, 883
 caso(s) base, 245
 casos de uso, 55
Cerr, 13
char, 174
char16_t, 1020
char32_t, 1020
 ciclo, 120
 de vida del software, 55
infinito, 120
 ciclos, 113
cifrado de clave simétrica, 766
 cima, 127
cin.getline, 377
 círculo relleno, 113, 190
 clase, 22
 alcance, 225, 412
 anticipada, declaración, 476
 autorreferenciada, 789
 base, 536
 directa, 536
 indirecta, 536
 de almacenamiento, 223
 especificadores, 223
 derivada, 53
 más derivada, 1046
 proxy, 475
 clases, 8, 22
 abstractas, 597
 base abstractas, 597
 concretas, 597
 contenedoras, 474
 de caracteres, 1000
 de colecciones, 474
 clave
 de búsqueda, 304, 770
 de registro, 711
simétrica, cifrado, 766
 clave/valor, pares, 884
 claves, 904
 de búsqueda, 904
 de ordenamiento, 770
clear, 672, 900
 cliente, 5
 clientes, 22
 de los objetos de la clase, 69
close, 715

- COBOL (COmmon Business Oriented Language, Lenguaje común orientado a negocios), 10
- código
- activo, método, 2
 - fuente, 12
 - abierto, 6
 - objeto, 7
 - `reentrant`, 413
- códigos de caracteres, 382
- coerción de argumentos, 212
- cola, 474, 807
- de impresión, 808
- colaboración, 321
- colas, 789
- colores, 957
- columnas, 307
- coma, operador, 164
- comando
- `break`, 1144
 - `continue`, 1145
 - Paso a paso para salir, 1136
 - Paso a paso por instrucciones, 1135
 - Paso a paso por procedimientos, 1136
 - `print`, 1145
 - `quit`, 1147
 - `set`, 1147
 - `step`, 1149
- comas, lista separada por, 41
- comentario, 36
- de una sola línea, 36
- compare, 751
- compilación, 11
- compilación condicional, 1089, 1092
- compiladores, 7
- compilar, 13
- complejidad exponencial, 250
- complemento
- a nivel de bit (~), 844
 - a uno, 844
- componentes, 8, 22, 1125
- comportamiento
- del sistema, 56
 - soportado en forma condicional, 1018
- comportamientos, 21
- composición, 100, 413, 452
- comprobación de rango, 748
- computación
- cliente/servidor, 5
 - conversacional, 43
 - distribuida, 5
 - interactiva, 43
 - personal, 5
- computadora, 3
- concatenamiento (encadenamiento), 44
- concretas, 597
- condición, 47
- de continuación de ciclo, 113, 161
- condiciones
- de guardia, 115
 - simples, 180
- conjunto de caracteres, 66, 709
- ASCII, 66
 - Unicode®, 650, 709
- conjuntos de bits, 941
- `const`, calificador, 283
- `const_cast`, 1029
- `const_reverse_iterator`, 761
- constantes
- con nombre, 283
 - de enumeración, 222
 - de punto flotante, 133
 - simbólicas, 1089, 1090
 - predefinidas, 1093
 - tipo carácter, 376
- constructor, 81
- un solo argumento, 509
 - de copia, 431, 498-499
 - predeterminado, 81
- constructores de conversión, 502
- contador, 122
- contenedores, 883
- asociativos, 884
 - de primera clase, 883
 - de secuencia, 884
- contenido
- dinámico, 9
 - generado por la comunidad, 6
 - premium, 6
- `conteo`
- con base cero, 163
 - de referencias, 1007
- `continue`, comando, 1145
- control
- del programa, 110
 - estructuras, 112
 - instrucciones, 110
 - transferencia, 112
- conversión, 66
- descendente, 591
 - explícita, 133
 - implícita, 134
- `copy`, 759, 898
- `copy_backward`, 929
- `count`, 905, 925
- `count_if`, 925
- crashing, 128
- crear una asociación, 910
- `<csignal>`, 1076
- `<cstdarg>`, encabezado de argumentos variables, 1069
- `<cstdlib>`, 859
- cuadro, 958
- cuantificador, 1002
- cubetas, 1016
- cuenta de referencia, 1007
- cuerpo, 38, 71
- culling, 968
- cursor, 38
- D**
- `data`, 759
- datos, 3, 723
- constantes, apuntador constante a, 354
 - constantes, apuntador no constante a, 351
 - no constantes, apuntador constante a, 353
 - referenciales circulares, 1011
 - representación, 473
- DBMS (sistema de administración de bases de datos), 711
- `dec`, 658
- decisiones, 2
- declaración
- anticipada, 435
 - de clase anticipada, 476
 - `static_assert`, 1018
- declaraciones, 40
- using, 49
- decoración de nombres, 241
- decremento, 161
- `Default`, caso, 175
- `#define`, 410
- directiva del preprocesador, 1090
- definición, 161
- de la clase, 70
- definidos por el usuario*, 22
- definir una clase, 70
- delegación, 807
- `delete`, 466, 1007, 1146
- dependientes de la máquina, 7
- depurador, 1128, 1142
- depurar, 9
- `Deque`, 883
- `dequeue`, 474, 807
- derivarse, 413
- desasignar, 466
- desbordamiento de pila, 229
- deserializarse, 736
- desplazamiento, 363, 616, 717
- a la derecha (>>), 844
 - a la izquierda (<<), 844
- desplazamos, 215

desreferenciar un apuntador, 344
destructivo, 44
destructor, 423
destructores no virtuales, 620
detección de colisiones, 958
diagrama
 con elementos omitidos, 98
 de actividad, 112
 de caso-uso, 55
 de clases de UML, 72
 de comunicación, 322
 de secuencia, 322
diagramas
 de actividad, 57
 de caso-uso, 57
 de clases, 57, 98
 de colaboración, 57, 322
 de componentes, 1125
 de comunicación, 57
 de despliegue, 1125
 de estructuras compuestas, 1125
 de interacción, 322
 de máquina de estado, 57
 de objetos, 1125
 de paquetes, 1125
 de secuencia, 57
 de sincronización, 1126
 de vista de interacción, 1126
diamantes sólidos, 100
dígito, 1000
 binario, 709
dígitos decimales, 709
dinámica, 592
dirección de búsqueda, 717
Directional, luces, 968
Direct3D, 956
directiva
 del preprocesador, 37
 del preprocesador **#define**, 1090
 del preprocesador **#include**, 1089
 del preprocesador **#undef**, 1091
#error, 1092
#pragma, 1093
directivas del preprocesador, 13
diseño, 23
 orientado a objetos (OO), 21
dispositivos
 de almacenamiento secundario, 709
 de entrada, 4
 de salida, 4
distribución uniforme, 998
división de enteros, 45
DOO (diseño orientado a objetos), 21
double, 129
dynamic_cast, 619

E
E/S
 con formato, 650
 en memoria, 761
 sin formato, 650
EBCDIC, 382
edición, 11
editor, 12
efectos
 de red, 6
 secundarios, 249
ejecución, 11
 condicional de directivas del preprocesador, 1089
 secuencial, 112
ejecuta, 13
elemento cero, 279
elementos, 279
eliminación
 de duplicados, 817
 del goto, 112
else suelto, problema, 118
empty, 520, 899, 911, 913, 914
encabezado
 de argumentos variables **<cstdarg>**, 1069
 de la función, 71-72
encapsular, 22
end, 761, 887
#endif, 410
#error, directiva, 1092
enlace, 11, 13, 790
 doble, lista, 803
 doble, lista circular, 803
 simple, lista, 803
 simple, lista circular, 803
enlaces de apuntador, 791
enlazador, 13
enqueue, 474, 807
ensambladores, 7
entero extendido, 1019
enteros, 40
Entity, 968
enum, 222
enumeración, 222
eofbit, 671
equal, 919
equal_range, 907, 937
erase, 756, 899
error
 de desbordamiento aritmético, 701
 de desplazamiento en, 1, 126
 de sintaxis, 38
 de subdesbordamiento, 701
 lógico, 48
lógico fatal, 48, 128
lógico no fatal, 48
por desplazamiento en, 1, 162
errores
 comunes de programación, 9
 del compilador, 38
 en tiempo de ejecución, 13
 fatales, 398
 fatales en tiempo de ejecución, 13
 no fatales en tiempo de ejecución, 13
 síncronos, 689
escalables, 285
escalar, 215
escalares, 295
 cantidades, 295
escena, 968
espacio
 de nombres, alcance, 225
 de nombres anidado, 1033
 de nombres global, 1033
 de nombres sin nombre, 1033
 del padre, 978
 en blanco, 37, 1000
 en blanco, caracteres, 654
 local, 978
 mundial, 978
espacio de nombres, miembro de, 1031
especialización explícita, 642
especializaciones, 621
 de plantilla de función, 243
especificación
 de diseño, 56
 de excepciones, 691
 de excepciones vacía, 691
 de requerimientos, 52
especificaciones de vinculación, 1082
especificador
 de acceso, 71
 puro, 598
establecer, 79
estaciones de trabajo, 5
estado, 57
 consistente, 94
 del formato, 670
 diagramas, 189
 final, 113
 inicial, 113, 190
estados de acción, 112
estructura
 autorreferenciada, 839
 de repetición, 112
 de secuencia, 112
 de selección, 112
 de selección múltiple, 113

- del sistema, 56
nombre de la, 838
tipo, 838
estructurada, programación, 112
estructuras, 279
 de control, 112
 de datos, 278, 789
 de datos dinámicas, 789
 de datos lineales, 791
 de datos no lineales, 791
 de datos “último en entrar, primero en salir” (UEPS), 228
etiqueta, 1079
 del especificador de acceso `public`, 71
etiquetas, 225
evaluación en corto circuito, 182
excepción, 683
 objeto, 688
 parámetro, 687
excepciones
 de punto flotante, 1076
 manejadores, 687
exception, 685
Exit, 1074
EXIT_FAILURE, 1074
EXIT_SUCCESS, 1074
explicit, 522
exportar modelos en, 3D, 957
expresión
 condicional, 117
 de acción, 112
 de control, 175
 integral constante, 171
expresiones
 de tipo mixto, 212
 regulares, 20
extensibilidad, 650
extern, 223
- F**
- factor de escala, 215
factorEscala, 220
fail, 671
failbit, 654
filas, 307
fill, 916
 función miembro, 665
fill_n, 916
find, 756, 906, 928
find_first_not_of, 756
find_first_of, 756
find_if, 928
find_last_of, 756
finish, 1150
- firma, 212
 de la función, 212
first, 907
fixed, 134, 667
flags, 670
flechas
 de navegabilidad, 432
 de transición, 112
flip, 942
float, 129
flujo
 de datos, 4
 de trabajo, 112
estándar de entrada, 13
estándar de error, 13
estándar de salida, 13, 38
no parametrizado, manipulador, 134
parametrizado, manipulador, 134
flujos, 38, 650, 688
 de cadena, procesamiento, 761
 de entrada, 888
 de salida, 888
fmtflags, 670
for_each, 926
formato
 de línea recta, 45
 de punto fijo, 134
FORTRAN (FORMula TRANslator, Traductor de fórmulas), 9
frágil, 559
FrameEvent, 978
FrameListener, 978
friend, función, 458
front, 894, 913
front_inserter, 931
fstream, 652
fuerza bruta, 200
fuga
 de memoria, 467
 de recursos, 701
función, 22, 37
 alcance, 225
 ayudante, 416
 binaria, 945
 de hash, 1016
 de operador de conversión, 502
 encabezado, 71-72
 friend, 458
 generadora, 916
 miembro, llamada, 69
 recursiva, 245
 virtual, 592
 virtual pura, 598
- funciones, 8
 de acceso, 416
 de conversión de cadenas, 859
 definidas por el programador, 205
 definidas por el usuario, 205
 en línea, 232
 funciones predicado, 416
 globales, 206
 miembro, 22
 cascada, llamadas, 463
 sobrecarga, 240
<functional>, 945
functores, 945
- G**
- g, 1144**
gcount, 657
gdb, 1144
generador de números seudoaleatorios, 998
generalización, 621
generate, 916
generate_n, 916
get, 654
getline, 74, 747
globales, 260
good, 672
goodbit, 672
Google, 6
goto, eliminación, 112
gráfico de escena, 968
- H**
- hardware, 2
hash, tabla, 1016
heap, 937
heapsort, 914
 algoritmo de ordenamiento, 937
help, 1145
Herede, 536
herencia, 21, 413, 536
 de diamante, 1043
 de implementación, 600
 de interfaz, 600
 múltiple, 536, 1039
 private, 538
 protected, 538
 public, 538
 simple, 536
 virtual de la clase base, 1046
herramientas de modelado en, 3D, 957
heurística, 337
hex, 658
hijo, 810
 derecho, 810
 izquierdo, 810

I

identificador, 41
de macros, 1090
if, instrucción, 47
#ifdef, 1092
#ifndef, 1092
#ifndef, 410
ifstream, 652
ignore, 657
imagen ejecutable, 13
inanición, 367
#include, directiva del preprocesador, 1089
includes, 933
inclusión circular, 435
incremento, 161
indicador, 43
índice, 279
indirección, 342
infijo, notación, 821
info break, 1146
información de tipos en tiempo de ejecución (RTTI), 581
Informe técnico, 1, 20
inizializador, 467
inizializadores, 281
inline, 232
InputManager, 979
inserción de flujo
en cascada, operaciones, 44
operador, 38
insert, 758, 899, 906
inserter, 931
Inspección, ventana, 1132
instanciar, 22
Instituto nacional estadounidense de estándares (ANSI), 3
instrucción, 38
 break, 179
 compuesta, 50, 119
 de ciclo, 120
 de repetición, 120
 de selección doble, 113
 de selección simple, 113
goto, 1078
goto, 112
if, 47
nula, 120
vacía, 120
instrucciones
 de ciclo, 113
 de control, 110
 de control, anidamiento, 114
 de control, apilamiento, 114
 de control de una sola entrada/

una sola salida, 114
ejecutables, 111
if...else anidadas, 117
ilegales, 1076
terminador, 38
int, 40
inteligencia colectiva, 6
interfaces, 88
interfaz de una clase, 88
internal, 397, 665
Internet, 5
 de banda ancha, 6
intérpretes, 7
interrupciones, 1076
invalid_argument, 700
ios::app, 713
ios::beg, 717
ios::binary, 725
ios::cur, 717
ios::end, 717
ios::in, 715
ios::out, 713
iostream, 651
<**iostream**>, 37
isalnum, 855
isalpha, 855
iscntrl, 858
isdigit, 855
isgraph, 858
islower, 350, 856
ISO (Organización internacional para la estandarización), 3, 1017
isprint, 858
ispunct, 858
isspace, 858
istream, 651
istringstream, 761
isupper, 856
isxdigit, 855
iter_swap, 929
iteraciones, 122
iteradores, 475, 883
iterar, 125
iterativa, 245

J

Java, 9
jerarquía
 de clases, 536
 de datos, 709

K

Keyboard, 979
KeyEvent, 979
KeyListener, 979

L

La segunda mejora, 127
LAMP, 19
lanzar
 la excepción, 690
 una excepción, 686
lanzarExcepcion, 690
left, 169, 664
length, 93, 747
 error, 700
lenguaje
 extensible, 72
 máquina, 7
Lenguaje Unificado de Modelado™ (UML™), 21
lenguajes
 de alto nivel, 7
 ensambladores, 7
less<int>, objeto función de comparación, 904
letras, 709
lexicográficamente, 751
Light, 968
limpieza de la pila, 688
línea
 de comandos, argumentos, 366
 de vida, 324
 punteada, 113
Linux, 19
list, 1145
List, 883
lista
 circular con enlace doble, 803
 circular de enlace simple, 803
 de enlace doble, 803
 de enlace simple, 803
 de inizializadores de miembros, 450
 de parámetros, 74
 de parámetros de plantilla, 243
 inizializadora, 281
 separada por comas, 41
listas enlazadas, 789
literal de cadena, 38
literales de cadena, 376
llamada
 a una función miembro, 69
 asíncrona, 322
 recursiva, 245
 síncrona, 322
llamadas a funciones miembro en cascada, 463
llave
 derecha, }, 38
 izquierda, {, 38

- lock**, 1011
LoD (niveles de detalle), 957
logic_error, 700
long long, 1019
lotes, 4
lower_bound, 906, 935
luces
 Direccional, 968
 Point, 968
 Spot, 968
luz
 ambiental, 958
 difusa, 958
 emisiva, 958
 especular, 958
lvalues, 185
- M**
- macro assert**, 1093
macros, 633, 1089
mainframes, 3
make, 1074
make_heap, 939
Makefile, 1074
malla, 968
manejadores
 catch, 687
 de excepciones, 687
manejo de excepciones, 683
 modelo de reanudación, 687
 modelo de terminación, 687
manipulador
 de flujo *boolalpha*, 183
 de flujo parametrizado, 134
 de flujos, 43
 de flujos no parametrizado, 134
manipuladores de flujos, 658
 parametrizados, 651
<map>, 908
map, 904
máquina de estado, diagramas, 189
marcado, 6
marcador de fin de archivo, 711
marcadores de visibilidad, 431
marco de pila, 228
máscara, 845
mashups, 6
match_not_dot_newline, 1002
match_results, 1001
materiales, 957
max, 939
max_element, 925
max_size, 754
maxheap, 937
media dorada, 248
- mejoramiento de arriba a abajo, paso a paso**, 127
mem_fun, 998
mem_fun_ref, 998
memchr, 869
memcmp, 869
memcpy, 868
memmove, 869
memoria, 4
 primaria, 4
memset, 870
mensaje, 38, 321
mensajes, 69
 anidados, 323
 secuencia, 323
merge, 902, 931
mesh, 968
métodos, 22, 205
miembro
 de datos static, 467
 de un espacio de nombres, 1031
miembros, 838
 de datos, 22, 75
min, 939
min_element, 925
modelado de caso-uso, 55
modelo
 de programación acción/decisión, 116
 en, 3D, 957
modelos
 de cascada, 55
 iterativos, 55
modo
 de apertura de archivo, 713
 de interrupción, 1145
 de métrica, 976
 de pixel, 976
 relativo, 976
módulo, 45
montón, 466, 937
 motores de gráficos en 3D, 956
multimap, 904
multiplicidad, 99
multiprocesadores, 4
multiprogramación, 5
multitarea, 10
mutable, 223, 1036
mutadores, 80
MySQL, 19
- N**
- name**, 619
namespace, 1031
navegabilidad bidireccional, 432
negación lógica, 182
- new**, 466
 manejador, 697
next, 1150
niveles de detalle (LoD), 957
no destructivo, 44
noboolalpha, 669
nodo
 hoja, 810
 padre, 811
 raíz, 810, 973
nodos, 791
nombre, 279
nombre de archivo, 713
 de la estructura, 838
 de rol, 99
 de tipo, 222
 de una variable de control, 161
 del atributo, 144
nombres, manipulación, 241
nomenclatura de camello, 71
normal, 973
noshowbase, 667
noshowpoint, 663
noshowpos, 397, 665
notación
 apuntador/desplazamiento, 363
 apuntador/subíndice, 363
Big O, 770
científica, 134
infijo, 821
postfijo, 821
notas, 113
nothrow, 696
nouppercase, 667
nueva línea, 38
NULL, 343
nulo, carácter, 291
<numeric>, 893
número
 de posición, 279
 de punto flotante, 129
números
 aleatorios no determinísticos, 998
 de punto flotante con precisión doble, 133
 de punto flotante con precisión simple, 133
 mágicos, 285
 seudoaleatorios, 218
- O**
- O(1)**, 770
O(log n), 776
O(n log n), 782
O(n), 771

- O(n²), 771*
- Object Management Group™ (OMG™, Grupo de administración de objetos), 23
- objeto
- de función, 944
 - de función binaria, 945
 - excepción, 688
 - flujo de entrada cin, 43
 - flujo estándar de salida, 38
 - función de comparación, 904
 - función de comparación `less<int>`, 904
 - `Root`, 966
 - `SceneManager`, 968
 - serializado, 736
- objetos, 8, 21
- anfitriones*, 453
 - iteradotes, 475
 - serialización, 736
- obtener*, 79
- `oct`, 658
- ocultamiento
- de datos, 78
 - de información, 22
- ocurrencia de definición, 15
- `ofstream`, 652
- Ogre (Motor de visualización de gráficos orientado a objetos), 20, 956
- Engine Rendering Setup, 966
- OgreAL, 20, 984
- OIS (Sistema de entrada orientado a objetos), 979
- opciones pegajosas, 169
- `open`, 713
- OpenAL, biblioteca de audio, 984
- OpenGL, 957
- operaciones, 22
- abstractas, 622
 - de inserción de flujo en cascada, 44
- operador
- `<<`, 38
 - binario de resolución de ámbito (o alcance), 90
 - de asignación `=`, 43
 - de asignación de suma, `+=`, 139
 - de conversión, 129, 502
 - de decremento, `--`, 139
 - de desreferencia, 344
 - de dirección, 343
 - de extracción de flujo, `>>`, 40, 43
 - de incremento, `++`, 139
 - de indirección, 344
 - de inserción de flujo, 38
- de llamada a función sobrecargado, 512
- de postincremento, 139
- de preincremento, 139
- de resolución de ámbito binario, `(::)`, 239
- módulo, 45
- punto `(.)`, 72
- ternario, 117
- unario, 134
- de conversión de tipo, 133
- operadores
- aritméticos, 45
 - binarios, 43
 - de asignación, 139
 - de asignación a nivel de bits, 850
 - de igualdad, 48
 - de multiplicación, 134
 - lógicos, 180
 - palabras clave, 1034
 - relacionales, 48
- operando, 38
- OR
- exclusivo (`\wedge`) a nivel de bits, 844
 - inclusivo (`\mid`) a nivel de bits, 844
- orden, 110
- `log n`, 776
 - `n`, 771
 - `n` al cuadrado, 771
- ordenamiento, 305
- de árbol binario, 817
 - de burbuja, 332
 - de cubeta, 339
 - claves, 770
 - por combinación, 777
 - por inserción, 306
 - por selección, 339, 355
- orden-z, 976
- órdenes de terminación del sistema operativo, 1076
- Organización internacional para la estandarización (ISO), 3, 1017
- orientada a la acción, 22
- orientados a objetos, 22
- `ostream`, 651
- `ostream_iterator`, 888
- `ostringstream`, 761
- `out_of_range`, 700
- `overflow_error`, 701
- `OverlayContainer`, 976
- P**
- palabra clave, 37
- palabras clave de operadores, 1034
- `PanelOverlayElement`, 976
- paquete de parámetros
- de función, 1020
 - de tipo de plantilla, 1020
- paquetes, 1125
- parámetro, 72
- de excepción, 687
 - de tipo formal, 243
 - formal, 210
- parámetros
- de plantilla de tipo, 633
 - de plantilla sin tipo, 641
 - por referencia, 234
 - sin tipo, 641
- paréntesis
- anidados o incrustados, 46
 - redundantes, 47
- pares clave/valor, 884
- parte superior, 789
- Pascal, 10
- paso
- por referencia, 233
 - por valor, 233
 - recursivo, 245
- paso a paso
- para salir*, comando, 1136
 - por instrucciones*, 1135
 - por procedimientos*, comando, 1136
- patrones de diseño, 19
- pequeños círculos, 112
- perezoso, 1002
- persistencia de los datos, 709
- PHP, 19
- pila, 228
- de ejecución del programa, 228
 - de llamadas a funciones, 228
 - desbordamiento, 229
 - limpieza, 688
 - marco, 228
 - meter, 228
 - sacar, 228
- pilas, 789
- plantilla variadic, 1020
- plantillas, 632
- de clases, 632
 - especialización, 632
- de funciones, 243, 632
- definiciones, 633
 - especialización, 632
- plataforma(s), 14
- de hardware, 7
 - .NET, 10

- P**
- Point**, luces, 968
 - polimorfismo, 580
 - POO (programación orientada a objetos), 22
 - pop**, 911, 913, 914
 - pop_back**, 902
 - pop_front**, 902
 - pop_heap**, 939
 - por procedimientos, 22
 - portables, 7
 - postdecrementar, 140
 - postdecremento, 139
 - postfijo**, notación, 821
 - postincrementar, 140
 - postincremento, operador, 139
 - postorden, 811
 - pow**, función de la biblioteca estándar, 167
 - #pragma**, directiva, 1093
 - precisión, 129, 134, 659
 - predeterminada, 134
 - precision**, 659
 - predecrementar, 140
 - predecremento, 139
 - predicado, funciones, 416
 - preincrementar, 140
 - preorden, 811
 - preprocesador, 13, 1089
 - envoltura, 409
 - preprocesamiento, 11
 - primera mejora, 127
 - print**, comando, 1145
 - priority_queue**, 914
 - private:**, 77
 - herencia, 538
 - probabilidad, 215
 - problema del else suelto, 118
 - procedimiento, 110
 - puro*, 413
 - procedimientos, 205
 - procesamiento
 - de flujos de cadena, 761
 - por lotes, 4
 - profundidad de color, 966
 - programa
 - control, 110
 - controlador, 85
 - de edición, 12
 - programación
 - de juegos, 19
 - estructurada, 10, 112
 - genérica, 632
 - Programación Orientada a Objetos (POO), 8, 22
 - programadores de computadoras, 3
- Q**
- quebradizo, 559
 - queue**, 913
 - <queue>, 914
 - quit**, comando, 1147
- R**
- rabo, 789
 - raise**, 1076
 - random_shuffle**, 925
 - randomización, 218
 - rangos, 888
 - rbegin**, 761, 897
 - rdstate**, 672
 - read**, 657
 - realizar una acción, 38
 - recopilación de requerimientos, 55
 - recorridos inorden, 811
 - rectángulo redondeado, 190
 - redes
 - de área local (LANs), 5
 - sociales, 6
 - redirigir, 1067
 - redondear, 134
 - Refactoring (Refabricación), 19
- reference_wrapper**, 998
- referencia**
 - directa a un valor, 342
 - indirecta a un valor, 342
 - rvalue*, 1017
 - parámetros, 234
- referencias sueltas**, 237
- regex**, 1000
- regex_match**, 1000
- regex_replace**, 1005
- regex_search**, 1000
- regex_token_iterator**, 1007
- register**, 223
- registro**, 710
 - de activación, 228
- regla**
 - de anidamiento, 186
 - de apilamiento, 185
- reglas**
 - de precedencia de operadores, 46
 - de promoción, 212
- reinterpret_cast**, 722
- relación “tiene un”**, 100, 452
 - de uno a uno, 101
 - de uno a varios, 101
 - de varios a uno, 101
- relleno**, 660
- remove**, 903, 919
- remove_copy**, 919
- remove_copy_if**, 921
- remove_if**, 921
- rend**, 761, 897
- rendering**, 957
- RenderWindow**, 966
- repetición**
 - controlada por contador, 121
 - definida, 122
 - estructuras, 112
 - indefinida, 127
- replace**, 756, 921
- replace_copy**, 921
- replace_copy_if**, 923
- replace_if**, 923
- reporte**
 - técnico, 1, 996
 - técnico, 2 (TR2), 1016
- requerimientos**, 23
 - del sistema, 55
- reset**, 942, 1011
- resize**, 754
- resolución**, 966
- ResourceGroupManager**, 985
- result_of**, 999
- reutilización de software**, 9
- reutilizar**, 22

r
reverse, 931
reverse_copy, 932
reverse_iterator, 761
right, 169, 664
robustos, 683
rombos (diamantes), 112
Root, objeto, 966
Ruby on Rails, 19
run, 1144
runtime_error, 685
rvalue, referencia, 1017
rvalues, 185

S

SAAS (Software as a Service), 20
salir de una función, 38
SceneManager, objeto, 968
SceneNode, 973
scientific, 667
script, 972
second, 907
secuencia
 de escape, 38
 de mensajes, 323
 estructura, 112
seekg, 717
seekp, 717
selección
 doble, instrucción, 113
 múltiple, estructura, 113
 simple, instrucción, 113
 estructuras, 112
señal, 1076
sensible a mayúsculas y minúsculas, 41
separar la interfaz de la implementación, 87
serialización de objetos, 736
servicios
 públicos, 88
 Web, 6
servidores, 5
set, comando, 1147
set_difference, 934
set_intersection, 934
set_new_handler, 694
set_symmetric_difference, 935
set_terminate, 692
set_union, 935
setbase, 658
setfill, 397, 411
 manipulador, 665
setprecision, 134
setw, 168
seudocódigo, 111
showbase, 667

showpoint, 134
showpos, 397, 665
signal, 1076
signo de porcentaje (%), 45
símbolo
 de decisión, 115
 de fusión, 120
 de redirección de entrada (<), 1067
 de redirección de salida (>), 1069
 del sistema, 1067
 para agregar salida (>>), 1069
símbolos
 de estado de acción, 112
 especiales, 709
síncrona, llamada, 322
síncronos, 689
sintaxis, 38
 de inicializador de clase base, 549
 de inicializador de miembros, 448
sistema, 56
 de administración de bases de datos (DBMS), 711
 de entrada orientado a objetos (OIS), 979
 de visualización, 966
 requerimientos, 55
sistemas
 de procesamiento de transacciones, 721
 operativos, 4
size, 319, 723, 747, 911, 913
smatch, 1001
sobrecarga
 de funciones, 240
 de operadores, 44, 484
sobrescribir, 592
software, 2
 ciclo de vida, 55
 de código fuente abierto, 19
 reutilización, 9
Software as a Service (SAAS), 20
solicitar un servicio de un objeto, 69
sólo lectura, variables, 283
sonido en 3D, 958
sort, 772, 902, 928
sort_heap, 939
Sound, 984
SoundManager, 984
splice, 902
spooler, 808
Spot, luces, 968
srand, 218
<stack>, 912
static, 223
 miembro de datos, 467
static_assert, declaración, 1018
std::cin, 40
std::cout, 38
<stdexcept>, 685
step, comando, 1149
STL (Biblioteca de plantillas estándar), 882
str, 761
strcat, 380
strchr, 864
strcmp, 381
strcpy, 379
strcspn, 865
string, 74
<string>, archivo de encabezado, 74
string::const_iterator, 761
string::npos, 756
strlen, 384
strncmp, 381
strncpy, 379
strupr, 865
strrchr, 866
strspn, 866
strstr, 866
strtod, 861
strtok, 383
strtol, 862
strtoul, 862
struct, 838
subárbol
 derecho, 810
 izquierdo, 810
subclase, 536
subíndice, 279
 0 (cero), 279
subobjeto de la clase base, 1043
subprocesamiento múltiple, 10
substr, 751
suffix, 1002
superclase, 536
supercomputadoras, 3
sustantivos, 22
swap, 752, 902, 928
swap_ranges, 929
switch de selección múltiple, 171

T

tabla
 de funciones virtuales (*viable*), 614
 de hash, 1016
tablas
 de valores, 307
 de verdad, 181
tamaño máximo, 752
tarea, 4
tareas de mantenimiento de terminación, 424

- T**
- `Tellg`, 717
 - `Tellp`, 717
 - `template`, 633
 - temporizadores, 958
 - terminador de instrucciones, 38
 - `terminate`, 690
 - `TextAreaOverlayElement`, 976
 - texto de reemplazo, 1090
 - textura, 957
 - `this`, 461
 - `throw`, 688
 - `throw`, lista, 691
 - `tie`, 673
 - tiempo
 - compartido, 5
 - de ejecución constante, 770
 - de ejecución cuadrático, 771
 - de ejecución lineal, 771
 - de ejecución logarítmico, 776
 - tipo
 - de valor de retorno, 71
 - del atributo, 144
 - estructura, 838
 - tipos, 22
 - de datos abstractos, 472
 - de datos agregados, 838
 - fundamentales, 41
 - integrados, 41
 - primitivos, 41
 - vinculación segura, 241
 - tips
 - de portabilidad, 9
 - de rendimiento, 9
 - para prevenir errores, 9
 - tokens, 383
 - tolerancia a fallas, programas, 683
 - `tolower`, 856
 - `top`, 911, 914
 - Torres de Hanoi, 272
 - total, 122
 - `toupper`, 856
 - trabajo, 4
 - traducción, 6
 - a lenguaje máquina, 6
 - transferencia de control, 112
 - `transform`, 926
 - transiciones, 112, 190
- U**
- `trunca`, 45, 126, 133
 - `tuple`, 999
 - `<typeinfo>`, 619
 - `type_info`, 619
 - `typedef`, 651, 840
 - `typename`, 633
- UML**
- Extensible, 24
 - Partners (Socios de UML), 24
- unario**, operador, 134
- `#undef`, directiva del preprocesador, 1091
- `underflow_error`, 701
- `unexpected`, 691
- Unicode®, conjunto de caracteres, 650, 709
- unidad**
- de almacenamiento secundario, 4
 - de entrada, 4
 - de memoria, 4
 - de salida, 4
- Unidad aritmética y lógica (ALU), 4
- Unidad central de procesamiento (CPU), 4
- unidades lógicas, 4
- `union`, 1080
- `unique`, 902, 931
- `unique_copy`, 932
- uno a uno, relación, 101
- uno a varios, relación, 101
- `unordered_map`, 1016
- `unordered_multimap`, 1016
- `unordered_multiset`, 1016
- `unordered_set`, 1016
- `upper_bound`, 906, 935
- `use_count`, 1010
- V**
- validación, 93
 - valor, 43, 279
 - “basura”, 124
 - centinela, 126
 - clave, 304
 - de bandera, 126
 - de prueba, 126
 - de retorno, tipo, 71
 - de señal, 126
 - final, 161
- indefinido, 124
- inicial, 161
- `valorDesplazamiento`, 220
- valores asignados, 904
- variable constante, 283
- variables, 40
 - de clase, 302
 - de sólo lectura, 283
- globales, 225
- locales, 75
- locales, ventana, 1132
- varios a uno, relación, 101
- `vector`, 316
- ventana
 - Automático, 1137
 - Inspección, 1132
 - Variables locales, 1132
- `verbos`, 22
- verificación de validez, 93
- `Viewport`, 968
- vinculación, 223
 - dinámica, 592
 - en tiempo de ejecución, 592
 - externa, 1073
 - interna, 1073
- violaciones de segmentación, 1076
- `virtual`
- función, 592
 - herencia de la clase base, 1046
- visibilidad, 431
- Visual Basic, 10
- Visual C#, 10
- Visual C++, 10
- visualización, 957
- `void`, 71
 - función que la llama, 71
- `volatile`, 1029, 1075
- W**
- `watch`, comando, 1151
 - `wchar_t`, 651, 746
 - `weak_ptr`, 1011
 - `what`, 685
 - `width`, 660
 - wikis, 6
 - World Wide Web, 5
 - `write`, 657

NOTAS

NOTAS

NOTAS

NOTAS

NOTAS