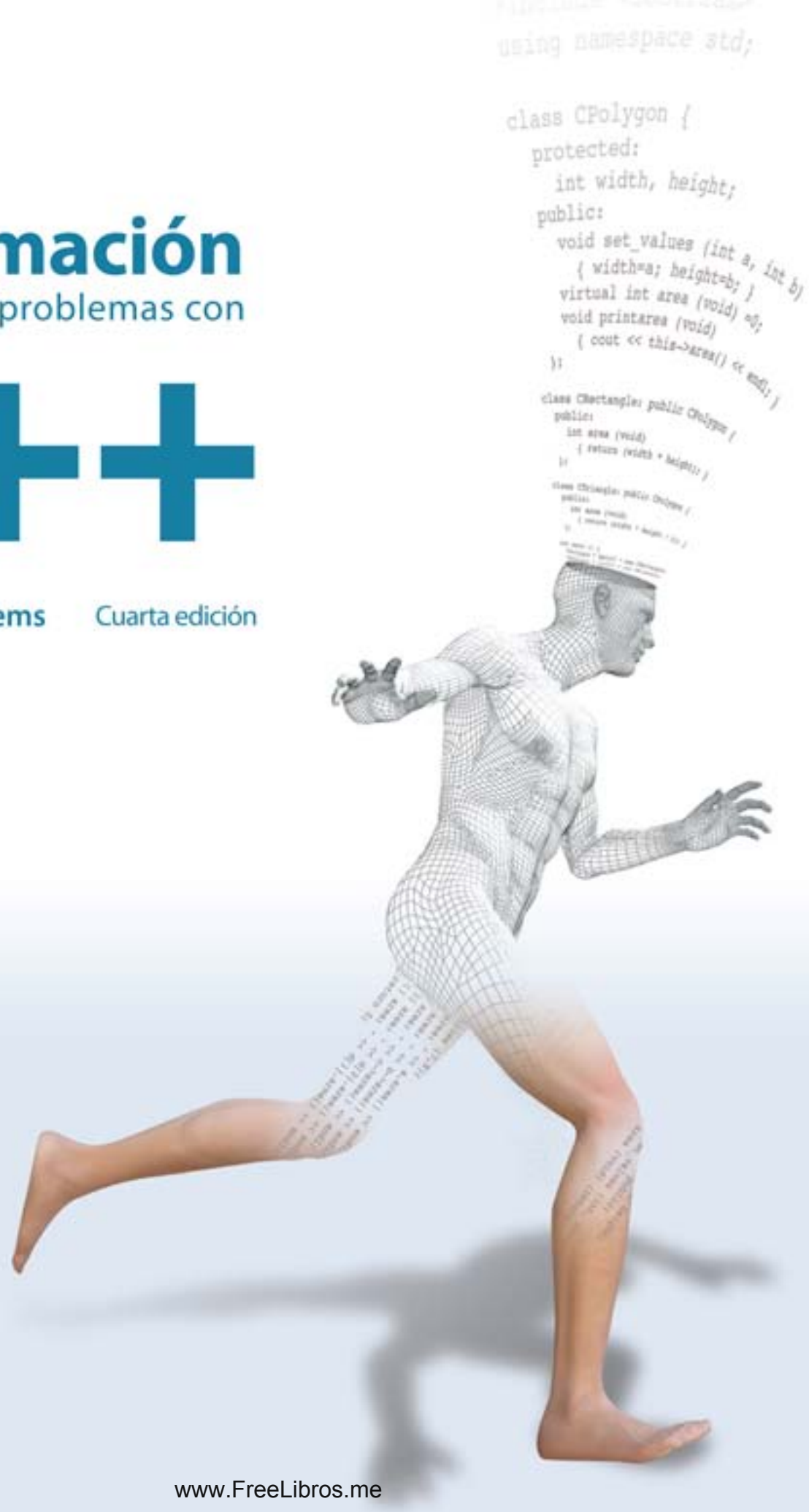
Programación
y resolución de problemas con
C++

Nell Dale • Chip Weems

Cuarta edición



Programación y resolución de problemas

con

C++

Programación y resolución de problemas

con

C++

Nell Dale
University of Texas, Austin

Chip Weems
University of Massachusetts, Amherst

Revisión técnica
Jorge Valeriano Assem
Universidad Nacional Autónoma de México,
Facultad de Ingeniería

MÉXICO • BOGOTÁ • BUENOS AIRES • CARACAS • GUATEMALA
LISBOA • MADRID • NUEVA YORK • SAN JUAN • SANTIAGO
AUCKLAND • LONDRES • MILÁN • MONTREAL • NUEVA DELHI
SAN FRANCISCO • SINGAPUR • SAN LUIS • SIDNEY • TORONTO

Director Higher Education: Miguel Ángel Toledo Castellanos
Director editorial: Ricardo del Bosque Alayón
Editor sponsor: Pablo E. Roig Vázquez
Editora de desarrollo: Ana Laura Delgado Rodríguez
Supervisor de producción: Zeferino García García

Diseño de portada: Utopía Visual

Traductores: Francisco Sánchez Fragoso
Thomas Bartenbach Joest

PROGRAMACIÓN Y RESOLUCIÓN DE PROBLEMAS CON C++

McGraw-Hill Interamericana

DERECHOS RESERVADOS © 2007, respecto a la primera edición en español por
McGRAW-HILL/INTERAMERICANA EDITORES, S.A. DE C.V.
A Subsidiary of ***The McGraw-Hill*** *Companies, Inc.*
Edificio Punta Santa Fe
Prolongación Paseo de la Reforma 1015, Torre A
Piso 17, Colonia Desarrollo Santa Fe
Delegación Álvaro Obregón
C.P. 01376, México, D.F.
Miembro de la Cámara Nacional de la Industria Editorial Mexicana, Reg. Núm. 736

ISBN-13: 978-970-10-6110-7
ISBN-10: 970-10-6110-1

Traducido de la cuarta edición de *Programming and Problem Solving with C++*.
Copyright © MMV by Jones and Bartlett Publishers, Inc. All rights reserved.
ISBN: 0-7637-0798-8

1234567890 09865432107

Impreso en México *Printed in Mexico*

The McGraw-Hill Companies

A Al, mi esposo y mejor amigo, y a nuestros hijos e hijos de nuestros hijos.

N.D.

A Lisa, Charlie y Abby con amor.

C.W.

Por mencionar a Mefistófeles, uno de los demonios principales, y el temperamento de Fausto,

...Mi amigo, seré pedagógico,
Y te digo que debes empezar con lógica...
...Se tendrán que invertir días para que aprendas
Eso es lo que alguna vez hiciste de un golpe,
Como comer y beber tan fácil y libre,
Sólo puede hacerse con uno, dos, tres.
Sin embargo la red del pensamiento no tiene tales pliegues
Y tiene más parecido con las obras maestras de un tejedor;
Un paso, miles de hilos surgen,
Aquí y allá dispara cada lanzadera,
Los hilos fluyen, invisibles y sutiles,
Cada golpe afecta miles de enlaces.
El filósofo viene con el análisis
Y demuestra que tiene que ser como esto;
Lo primero fue así, lo segundo así,
Y por tanto el tercero y cuarto fueron así,
Y el primero y segundo no estuvieron aquí,
Entonces el tercero y cuarto nunca podrían aparecer.
Eso es lo que creen los alumnos,
Pero nunca han aprendido a tejer.

J. W. von Goeth, *Fausto*, fragmento.

Conforme lea este libro, no permita que la lógica de los algoritmos ciegue su imaginación, por el contrario hágala su herramienta para tejer obras maestras del pensamiento.

Prefacio

A través de las ediciones sucesivas de *Programación y resolución de problemas con C++*, una cosa no ha cambiado: nuestro compromiso con el alumno. Como siempre, nuestros esfuerzos están dirigidos a hacer más accesibles a los alumnos los conceptos de computación en ocasiones difíciles.

Esta edición de *Programación y resolución de problemas con C++* continúa con la filosofía de que los temas considerados demasiado avanzados pueden ser enseñados en un primer curso. Por ejemplo, se atienden de modo explícito los metalenguajes como medio formal de especificar la sintaxis del lenguaje de programación. Se introduce la notación *O* mayúscula (Big-O) al principio, y se usa para comparar algoritmos en capítulos posteriores. Se analiza el diseño modular en términos de pasos abstractos, pasos concretos, equivalencia funcional y cohesión funcional. Las precondiciones y poscondiciones se usan en el contexto de repaso del algoritmo, en el desarrollo de estrategias de prueba y como documentación de interfaz para funciones escritas por el usuario. La discusión del diseño de interfaz de función incluye encapsulación, abstracción de control y complejidad de comunicación. La abstracción de datos y los tipos de datos abstractos (TDA) se explican junto con el mecanismo de clase C++, de modo que se crea una guía natural para la programación orientada a objetos.

C++ estándar ISO/ANSI se emplea en todo el libro, inclusive partes importantes de la nueva biblioteca estándar de C++.

La presente edición

En esta edición se han actualizado completamente los objetivos, los casos prácticos y los ejercicios. Además, en el capítulo 13, el lenguaje del material se ha vuelto más orientado a objetos.

Objetivos Los objetivos del capítulo han sido organizados para reflejar dos aspectos del aprendizaje: conocimiento y habilidades. Así, los objetivos se dividen en dos secciones. La primera lista los objetivos de conocimiento, expresados en términos de lo que el alumno debe saber después de leer el capítulo. La segunda reúne lo que el alumno debe poder hacer después de leer el capítulo.

Casos prácticos de resolución de problemas Cada capítulo tiene un caso práctico completamente nuevo. Los casos prácticos que comienzan con un enunciado de problema y terminan con un programa probado han sido la marca distintiva de nuestros libros. En esta sección se han añadido imágenes de pantallas que muestran el resultado para cada uno de los casos.

El caso práctico del capítulo 14 comienza con la construcción de un calendario de citas. El proyecto se completa en el capítulo 16. En el capítulo 17 se cambia la ejecución de una clase, enfatizando que tales cambios no afectan al usuario. El programa también se hace más robusto al añadir y manejar

excepciones. En cada etapa del proyecto se escriben los controladores para probar las clases conforme se crean. Esta organización muestra en acción al diseño y la programación orientados a objetos.

Debido a que algunos de los ejemplos pequeños empleados en un capítulo encuentran su camino en el código de caso práctico, estos ejemplos han sido cambiados para que sean congruentes con los nuevos casos prácticos.

Ejercicios Con excepción del capítulo 17, todos los ejercicios son nuevos. El número de ejercicios ha sido ampliado por entre veinte y treinta por ciento. Todos los problemas de programación son nuevos.

Lenguaje orientado a objetos La lista TDA del capítulo 13 ha sido cambiada eliminando la operación Print e introduciendo un par de iteradores, `Reset` y `GetNestItem`. Este cambio proporciona mejor encapsulación. La lista no necesita saber nada acerca de los ítems que contiene. La lista simplemente devuelve objetos al programa cliente, que debe conocer cuáles son los objetos. La desventaja en este diseño se señala en el capítulo 14. Las operaciones `Delete` y `BinSearch` usan operadores relacionales, lo que limita el tipo de ítem a tipos integrados. En este capítulo, los operadores relacionales se remplazan por operaciones `LessThan` y `Equal`; la documentación establece que `ItemType` debe llevar a cabo estas operaciones. Se analizan también los conceptos de responsabilidades de acción y responsabilidades de conocimiento.

El uso de clases para construir ítems cada vez más complejos se remarca en los casos prácticos. Cada clase se prueba de modo independiente, remarcando la importancia de probar.

C++ y programación orientada a objetos

Algunos profesores rechazan a la familia de lenguajes C (C, C++, Java) por ser demasiado permisiva y conducente a escribir programas no legibles y difíciles de descifrar. Nuestra experiencia no apoya este punto de vista, *siempre que el uso de características de lenguaje se modele de manera apropiada*. El hecho de que la familia C permita un estilo de programación conciso y compacto no se puede etiquetar simplemente como "bueno" o "malo". Casi cualquier lenguaje de programación se puede usar para escribir en un estilo que es demasiado conciso e inteligente para que sea entendido con facilidad. La familia C se puede de hecho de esta manera con más frecuencia que los otros lenguajes, pero se ha encontrado que con instrucción cuidadosa en ingeniería de software y un estilo de programación que sea directo, disciplinado y libre de características de lenguaje intrincadas, los alumnos pueden aprender a usar C++ para producir código claro y legible.

Se debe remarcar que aunque se usa C++ como un vehículo para enseñar conceptos de computación, el libro no es un manual de lenguaje y no intenta hacer una cobertura completa de C++. Ciertas características de lenguaje, sobrecarga del operador, argumentos por omisión, información tipo tiempo de ejecución y mecanismos para formas avanzadas de herencia, por nombrar algunas, se omiten en un esfuerzo por no abrumar con mucho, demasiado rápido, al alumno principiante.

Hay diversas opiniones acerca de cuándo introducir el tema de la programación orientada a objetos (POO). Algunos profesores abogan por una inmersión en la POO desde el principio, mientras que otros (para quienes está dirigido este libro) favorecen un método más heterogéneo, en el que tanto la descomposición funcional como el diseño orientado a objetos se presentan como herramientas de diseño. La organización del capítulo de *Programación y resolución de problemas con C++* refleja un enfoque de transición a la POO. Aunque se provee una presentación anticipada del diseño orientado a objetos en el capítulo 4, se retrasa una discusión enfocada hasta el capítulo 14, después que los alumnos han adquirido bases firmes en el diseño de algoritmos, abstracción de control y abstracción de datos con clases.

Sinopsis

El capítulo 1 está diseñado para crear un entendimiento confortable entre los alumnos y el tema. Se presentan los conceptos básicos de hardware y software, se plantean cuestiones de ética en computación y se introducen y refuerzan técnicas de resolución de problemas en un caso práctico de resolución de problemas.

En lugar de abrumar inmediatamente al alumno con los distintos tipos numéricos disponibles en C++, el capítulo 2 se concentra en sólo dos tipos: char y string. (Para el último, se usa la clase de

cadena ISO/ANSI proporcionada por la biblioteca estándar.) Con menos tipos de datos que seguir, los alumnos pueden centrar su atención en la estructura general del programa y lograr un comienzo temprano en la creación y ejecución de un programa simple. En el capítulo 3 se continúa con el análisis de los tipos numéricos de C++ y se procede con material sobre expresiones aritméticas, llamadas de función y salida. A diferencia de muchos libros que detallan *todos* los tipos de datos de C++ y *todos* los operadores a la vez, estos dos capítulos se enfocan en los tipos de cadena int, float, char y string, y los operadores aritméticos básicos. Los detalles de los otros tipos de datos y los operadores de C++ más elaborados se posponen hasta el capítulo 10.

Las metodologías de descomposición funcional y de diseño orientado a objetos son un objetivo principal en el capítulo 4, y el análisis se escribe con un saludable grado de formalismo. El tratamiento anticipado del diseño orientado a objetos en el libro es más superficial que la descomposición funcional. Sin embargo, los alumnos ganan la perspectiva oportuna de que hay dos, no sólo una, metodologías de diseño de uso extendido y que cada una sirve para un propósito específico. El diseño orientado a objetos se cubre a profundidad en el capítulo 14. En el capítulo 4 se cubre también la entrada y la entrada y salida de archivos. La introducción temprana de archivos permite la asignación de problemas de programación que requiere el uso de archivos de datos muestrales.

Los alumnos aprenden a reconocer funciones en los capítulos 1 y 2 y aprenden a usar las funciones de biblioteca estándar en el capítulo 3. El capítulo 4 refuerza los conceptos básicos de llamadas de función, paso de argumentos y bibliotecas de función. El capítulo 4 relaciona también funciones con la ejecución de diseños modulares, y comienza el análisis de diseño de interfaz que es esencial para escribir funciones apropiadas.

El capítulo 5 comienza con datos booleanos, pero su propósito principal es introducir el concepto de flujo de control. La selección, con estructuras If-Then e If-Then-Else, se emplea para demostrar la distinción entre orden físico de declaraciones y orden lógico. Se desarrolla también el concepto de estructuras de control anidadas. El capítulo 5 concluye con una sección larga de Prueba y depuración que se amplía en el análisis de diseño modular al introducir precondiciones y poscondiciones. El repaso de algoritmo y el repaso de código se introducen como medios para evitar errores, y el seguimiento de la ejecución se usa para hallar errores que se pudieron haber cometido en el código. También se cubre de forma extensa la validación de datos y estrategias de prueba en esta sección.

El capítulo 6 se dedica a las estrategias de control de bucles y operaciones iterativas por medio de sintaxis de la declaración While. En vez de introducir estructuras sintácticas múltiples, nuestro método es enseñar los conceptos de iteración usando sólo la declaración While. Sin embargo, debido a que muchos profesores nos han expresado que prefieren mostrar a los alumnos la sintaxis para las declaraciones de iteración de C++ a la vez, el examen de las declaraciones For y Do-While del capítulo 9 se pueden cubrir después del capítulo 6.

Por el capítulo 7 los alumnos ya se sienten cómodos con la descomposición de problemas en módulos y el uso de funciones de biblioteca, y son receptivos a la idea de escribir sus propias funciones. Así, el capítulo 7 se centra en pasar argumentos por valor y cubre el flujo de control en llamadas de función, argumentos o parámetros, variables locales y diseño de interfaz. La cobertura del diseño de interfaz incluye precondiciones y poscondiciones en la documentación de interfaz, abstracción de control, encapsulación y ocultación física contra conceptual de una ejecución. En el capítulo 8 se amplía el análisis para incluir parámetros de referencia, alcance y tiempo de vida, talones y controladores, y más sobre el diseño de interfaz, inclusive efectos secundarios.

En el capítulo 9 se cubren las demás estructuras de control de C++ (Switch, Do-While y For), junto con las declaraciones Break y Continue. Estas estructuras son útiles pero no necesarias. El capítulo 9 es un punto terminal natural para primer trimestre de una serie de cursos introductorios en dos trimestres.

El capítulo 10 comienza la transición entre la orientación de estructuras de control de la primera mitad del libro y la orientación de tipo de datos abstractos de la segunda mitad. Se examinan los tipos de datos simples integrados en términos del conjunto de valores presentados por cada tipo y las operaciones permisibles en esos valores. Se introducen operadores adicionales de C++ y se examinan en detalle los problemas de presentación de punto flotante y precisión. Los tipos simples definidos por el usuario, archivos de encabezado escritos por el usuario y coerción de tipo están entre los otros temas cubiertos en este capítulo.

El capítulo 11 comienza con una explicación de tipos de datos simples contra estructurados. Se introduce el registro (struct en C++) como una estructura de datos heterogénea, se describe la sintaxis para tener acceso a sus componentes y se demuestra cómo combinar tipos de registro en una estructura de registro jerárquica. De esta base, se procede al concepto de abstracción de datos y se da una definición precisa para la noción de un TDA, remarcando la separación de especificación y ejecución. El mecanismo de clase de C++ se introduce como una representación del lenguaje de programación de un TDA. Se remarcan los conceptos de encapsulación, ocultación de información y miembros de clase pública y privada. Se describe la compilación separada de archivos de programa, y los alumnos aprenden la técnica de colocar una declaración y ejecución de clase en dos archivos separados: el archivo de especificación (.h) y el archivo de ejecución (.ccp).

En el capítulo 12 se introduce el arreglo como una estructura de datos homogénea a cuyos componentes se tiene acceso por posición y no por nombre. Los arreglos adimensionales se examinan a profundidad, incluso arreglos de structs y arreglos de objetos de clase. El material sobre arreglos multidimensionales completa la discusión.

El capítulo 13 integra el material de los capítulos 11 y 12 definiendo la lista como un TDA. Debido a que ya se han introducido las clases y los arreglos, se puede distinguir claramente entre arreglos y listas desde el principio. El arreglo es una estructura de datos de tamaño fijo, integrada. La lista es una estructura de tamaño variable, definida por el usuario, representada en este capítulo como una variable de longitud y un arreglo de ítems aglutinados en un objeto de clase. Los elementos de la lista son aquellos elementos del arreglo de la posición 0 hasta la *longitud* de posición -1. En este capítulo, se diseñan clases de C++ para TDA de listas no clasificadas y clasificadas, y se codifican los algoritmos de lista como funciones de miembros de clase. Se usa la notación *Big-O* para comparar los distintos algoritmos de búsqueda y clasificación desarrollados para estos TDA. Por último, se examinan cadenas de C a fin de dar a los alumnos cierta visión de cómo se podría ejecutar la abstracción de nivel superior (una cadena como una lista de caracteres) en términos de abstracción de nivel bajo (un arreglo char con terminación nula).

En el capítulo 14 se amplían los conceptos de abstracción de datos y clases C++ a una exploración de desarrollo de software orientado a objetos. El diseño orientado a objetos, introducido de manera breve en el capítulo 4, se revisa con mayor profundidad. Los alumnos aprenden a distinguir entre relaciones de herencia y composición durante la fase de diseño y las clases derivadas de C++ se emplean para poner en práctica la herencia. En este capítulo se introducen también funciones virtuales de C++, que apoyan el polimorfismo en la forma de enlace de operaciones a objetos en tiempo de ejecución.

En el capítulo 15 se examinan tipos de punteros y referencia. Se presenta a los punteros como una forma de hacer más eficientes a los programas y de permitir la asignación en tiempo de ejecución de datos de programa. La cobertura de estructuras de datos dinámicos continúa en el capítulo 16, en el que se presentan listas enlazadas, algoritmos de listas enlazadas y representaciones alternas de listas enlazadas.

En el capítulo 17 se introducen plantillas de C++ y el manejo de excepción, y en el capítulo 18 se concluye el texto con la cobertura de la recursión. No hay consenso en cuanto al mejor lugar para introducir estos temas. Se cree que es mejor esperar hasta por lo menos el segundo semestre para cubrirlos. Sin embargo, se ha incluido este material para los profesores que lo han solicitado. Ambos capítulos han sido diseñados de modo que puedan ser asignados para leer junto con capítulos previos. Se sugiere la siguiente lectura de prerrequisitos para los temas de los capítulos 17 y 18:

Sección o secciones	Tema	Prerrequisito
17.1	Funciones de plantilla	Capítulo 10
17.2	Clases de plantilla	Capítulo 13
17.3	Excepciones	Capítulo 11
18.1-18.3	Recursión con variables simples	Capítulo 8
18.4	Recursión con arreglos	Capítulo 12
18.5	Recursión con variables de puntero	Capítulo 16

Secciones especiales Cinco tipos de características se hacen resaltar del texto principal. Las secciones de bases teóricas presentan el material relacionado con la teoría fundamental detrás de varias ramas de la computación. En los consejos prácticos de ingeniería de software se examinan métodos para hacer los programas más confiables, robustos o eficientes. Los asuntos de estilo atienden cuestiones de la codificación de programas. En las secciones de información básica se exploran cuestiones secundarias que mejoran el conocimiento general del alumno en computación. Asimismo se incluyen biografías de pioneros de la computación como Blaise Pascal, Ada Lovelace y Grace Murray Hopper.

Objetivos Como ya se describió, cada capítulo comienza con una lista de objetivos para el alumno, separados en dos categorías: objetivos de conocimiento y objetivos de habilidades. Éstos se refuerzan y prueban en los ejercicios de fin de capítulo.

Casos prácticos de resolución de problemas La resolución de problemas se demuestra mejor a través de casos prácticos. En cada caso práctico se presenta un problema y se emplean técnicas de resolución de problemas para desarrollar una solución manual. A continuación, se desarrolla la solución para un algoritmo por medio de descomposición funcional, diseño orientado a objetos, o ambos; luego se codifica el algoritmo en C++. Se muestran los datos de prueba muestrales y la salida, y se continúa con una explicación de lo que tiene que ver con la prueba completa del programa.

Prueba y depuración Las secciones de prueba y depurado siguen a los casos prácticos en cada capítulo y consideran a profundidad las implicaciones del material del capítulo en relación con la prueba completa de programas. Estas secciones concluyen con una lista de sugerencias de prueba y depuración.

Comprobaciones rápidas Al final de cada capítulo hay preguntas que prueban la capacidad del alumno de recordar los puntos principales relacionados con los objetivos del capítulo. Al leer cada pregunta, el alumno debe conocer la respuesta de inmediato, que se puede comprobar de un vistazo en las respuestas al final de cada sección. El número de página en el que se explicó el concepto aparece al final de cada pregunta de modo que el alumno puede revisar el material en el caso de una respuesta incorrecta.

Ejercicios de preparación para examen Estas preguntas ayudan al alumno a prepararse para pruebas. Las preguntas por lo general tienen respuestas objetivas y están diseñadas para ser contestadas con algunos minutos de trabajo.

Ejercicios de preparación para programación Esta sección proporciona al alumno experiencia en la escritura de fragmentos de código C++. El alumno puede practicar los constructos sintácticos en cada capítulo sin la carga de escribir un programa completo.

Problemas de programación Estos ejercicios, tomados de una amplia variedad de disciplinas, requieren que el alumno diseñe soluciones y escriba programas completos.

Seguimiento de caso práctico Mucha de la práctica de programación moderna requiere leer y modificar código existente. Estos ejercicios dan al alumno la oportunidad de fortalecer esta habilidad crítica al contestar preguntas acerca del código de caso práctico o al hacerle cambios.

Materiales de apoyo

Esta obra cuenta con interesantes complementos que fortalecen los procesos de enseñanza-aprendizaje, así como la evaluación de los mismos, los cuales se otorgan a profesores que adoptan este texto para sus cursos. Para obtener más información y conocer la política de entrega de estos materiales, contacte a su representante McGraw-Hill o envíe un correo electrónico a marketinghe@mcgraw-hill.com

Reconocimientos

Nos gustaría agradecer a las personas que ayudaron en la preparación de esta cuarta edición. Estamos en deuda con los miembros de las facultades de los departamentos de computación de la universidad de Texas en Austin y la Universidad de Massachusetts en Amherst.

Se agradece especialmente a Jeff Brumfiel por desarrollar el metalenguaje de plantilla de sintaxis y permitirnos usarlo en el texto.

Por sus muchas sugerencias útiles, se agradece a los profesores, asistentes de enseñanza, asesores y supervisores de alumnos quienes pusieron en práctica los cursos para los que fue escrito este libro, así como a los mismos alumnos.

Se agradece a las siguientes personas que se dieron tiempo para ofrecer sus comentarios o cambios posibles a ediciones previas: Trudee Bremer, Illinois Central College; Mira Carlson, Northeastern Illinois University; Kevin Daimi, University of Detroit, Mercy; Bruce Elenbogen, University of Michigan, Dearborn; Sandria Kerr, Winston-Salem State University; Alicia Kime, Fairmont State College; Shahadat Kowuser, University of Texas, Pan America; Bruce Maxim, University of Michigan, Dearborn; William McQuain, Virginia Tech; Xiannong Meng, University of Texas, Pan America; William Minervini, Broward University; Janet Remen, Washtenaw Community College; Viviana Sandor, Oakland University; Mehdi Setareh, Virginia Tech; Katy Snyder, University of Detroit, Mercy; Tom Steiner, University of Michigan, Dearborn; John Weaver, West Chester University; Charles Welty, University of Southern Maine; Cheer-Sun Yang, West Chester University.

Se agradece también a los editores. Agradecemos especialmente a Amy Rose, cuyas habilidades y naturaleza genial convirtieron el trabajo duro en placer.

Cualquiera que haya escrito un libro, o esté relacionado con alguien que lo haya hecho, puede apreciar la cantidad de tiempo requerida en tal proyecto. A nuestras familias, todo el clan Dale y la amplia familia Dale (demasiados para nombralos) y a Lisa, Charlie y Abby, gracias por su tremendo apoyo e indulgencia.

N. D.
C. W.

CONTENIDO

CAPÍTULO

1

Repaso de programación y resolución de problemas

Objetivos

Objetivos de conocimiento

- *Entender lo que es un programa de computadora.*
- *Comprender lo que es un algoritmo.*
- *Aprender lo que es un lenguaje de programación de alto nivel.*
- *Conocer los procesos de compilación y ejecución.*
- *Aprender la historia del lenguaje C++.*
- *Saber lo que son los principales componentes de una computadora y cómo funcionan juntos.*
- *Aprender acerca de algunos asuntos éticos básicos que enfrentan los profesionales de la computación.*

Objetivos de habilidades

Ser capaz de:

- *Listar las etapas básicas relacionadas con la escritura de un programa de computadora.*
- *Describir lo que es un compilador y lo que hace.*
- *Distinguir entre hardware y software.*
- *Elegir un método apropiado de resolución de problemas para desarrollar una solución algorítmica a un problema.*

1.1 Repaso de programación

En la nota al margen se da la definición de **computadora**. ¡Qué definición tan breve para algo que, en sólo unas cuantas décadas, ha cambiado la forma de vida de las sociedades industrializadas! Las computadoras tocan todas las áreas de nuestras vidas: pago de facturas, conducción de automóviles, uso del teléfono, ir de compras. De hecho, sería más fácil enumerar las áreas de nuestras vidas que no son afectadas por las computadoras.

Computadora Objeto que calcula; específicamente: dispositivo electrónico programable que puede almacenar, recuperar y procesar datos.

Es triste que un dispositivo que hace tanto bien sea, con mucha frecuencia, tratado de forma injusta y se le vea con temor. ¿Cuántas veces ha escuchado a alguien decir: "lo siento, la computadora echó a perder las cosas", o "no entiendo las computadoras, son muy complicadas para mí"? Sin embargo, el hecho de que usted esté leyendo este libro significa que está preparado para hacer a un lado los prejuicios y aprender acerca de las computadoras. Pero se advierte: este libro no sólo trata de computadoras. Es un texto para enseñarle a programar computadoras.

¿Qué es la programación?

Mucho del comportamiento y pensamiento humano se caracteriza por secuencias lógicas. Desde la infancia usted ha estado aprendiendo cómo actuar, cómo hacer las cosas. Y ha aprendido a esperar cierto comportamiento de otras personas.

Mucho de lo que hace todos los días lo hace de manera automática. Por fortuna no es necesario que piense conscientemente que todo paso requerido en un proceso tan simple como dar vuelta a la página:

1. Levantar la mano.
2. Mover la mano a la derecha del libro.
3. Asir la esquina derecha de la página.
4. Mover la mano de derecha a izquierda hasta que la página esté colocada de modo que pueda leer lo que está sobre la otra página.
5. Soltar la página.

Piense en cuántas neuronas debe encender y cuántos músculos deben responder, todo en cierto orden o secuencia, para mover su brazo y su mano. Sin embargo, lo hace de manera inconsciente.

Mucho de lo que hace de manera inconsciente lo tuvo que aprender una vez. Observe cómo un bebé se concentra en poner un pie antes que el otro mientras aprende a caminar. Luego, observe a un grupo de niños de tres años que juegan a la roña.

En una escala más amplia, las matemáticas nunca se podrían haber desarrollado sin secuencias lógicas de pasos para resolver problemas y demostrar teoremas. La producción en masa nunca habría funcionado sin operaciones que tienen lugar en cierto orden. La civilización se basa en el orden de las cosas y acciones.

Programación Planear o calendarizar el desempeño de una tarea o suceso.

Computadora Dispositivo programable que puede almacenar, recuperar y procesar datos.

Programa de computadora Secuencia de instrucciones que realizará una computadora.

Programación en computadora Proceso de planificar una secuencia de pasos para que los desarrolle una computadora.

Se crea orden, de manera consciente e inconsciente, en un proceso al que se denomina **programación**. Este libro tiene que ver con la programación de una de las herramientas, la **computadora**.

Del mismo modo que un programa de concierto lista el orden en que los músicos ejecutan las piezas, un **programa de computadora** lista la secuencia de pasos que realiza la computadora. De ahora en adelante, cuando se use la palabra *programación* y *programa*, se entenderá **programación en computadora** y *programa de computadora*.

La computadora permite hacer las tareas con más eficiencia, rapidez y exactitud de como se podrían hacer a mano, si acaso se

* Con autorización. De *Merriam-Webster's Collegiate Dictionary*. Décima edición. © 1994 de Merriam-Webster Inc.

pudieran hacer a mano. A fin de usar esta poderosa herramienta, se debe especificar lo que se desea hacer y el orden en que se desea hacerlo. Esto es posible por medio de la programación.

¿Cómo se escribe un programa?

Una computadora no es inteligente. No es capaz de analizar un problema y proponer una solución. Un humano (el *programador*) debe analizar el problema, desarrollar una secuencia de instrucciones para resolver el problema y luego comunicarlo a la computadora. ¿Cuál es la ventaja de usar una computadora si no puede resolver problemas? Una vez que se ha escrito la solución como una secuencia de instrucciones para la computadora, ésta puede repetir la solución de manera muy rápida y congruente, una y otra vez. La computadora libera a la gente de las tareas repetitivas y tediosas.

Para escribir una secuencia de instrucciones que efectuará una computadora, se debe ir por un proceso bifásico: *resolución de problema* e *implementación* (véase la figura 1-1).

Fase de resolución del problema

1. *Análisis y especificación.* Entender (definir) el problema y lo que debe hacer la solución.
2. *Solución general (algoritmo).* Desarrollar una secuencia lógica de pasos que resuelve el problema.
3. *Verificar.* Seguir los pasos exactamente para ver si la solución resuelve en realidad el problema.

Fase de implementación

1. *Solución concreta (programa).* Traducir el algoritmo en un lenguaje de programación.
2. *Prueba.* Ver que la computadora siga las instrucciones. Después, comprobar de manera manual los resultados. Si encuentra errores, analice el programa y el algoritmo para determinar la fuente de errores, y luego hacer correcciones.

   Una vez que se ha escrito el programa, entra a la tercera fase: *mantenimiento.*

Fase de mantenimiento

1. *Uso.* Utilice el programa.
2. *Mantenimiento.* Modifique el programa para satisfacer requisitos de cambio o corregir cualquier error que aparezca al usarlo.

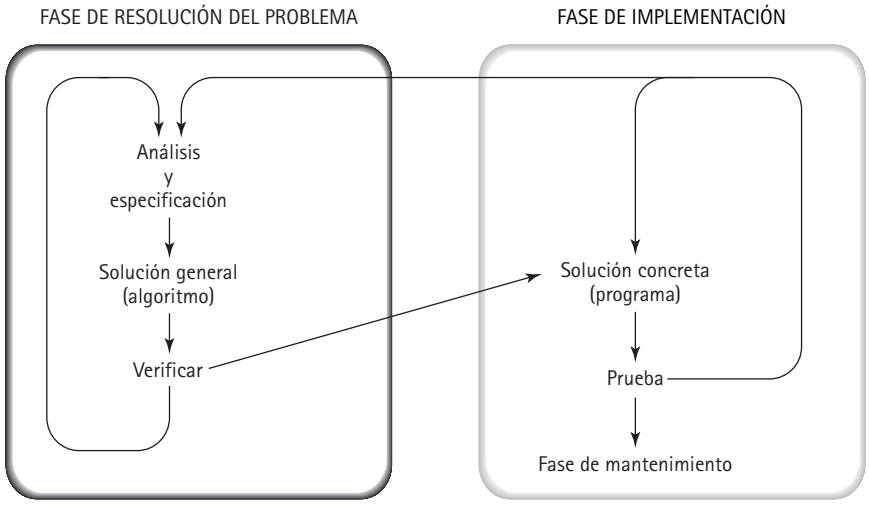


Figura 1-1 *Proceso de programación*

Algoritmo Procedimiento paso a paso para resolver un problema en una cantidad de tiempo finita.

El programador comienza el proceso de programación al analizar el problema y desarrollar una solución general llamada **algoritmo**. Entender y analizar un problema toma más tiempo del que implica la figura 1-1. Son el corazón del proceso de programación.

Si las definiciones de un programa de computadora y un algoritmo parecen similares, es porque todos los programas son algoritmos. Un programa es simplemente un algoritmo que ha sido escrito para una computadora.

Un algoritmo es una descripción verbal o escrita de una secuencia lógica de acciones. Se usan algoritmos todos los días. Recetas, instrucciones e indicaciones son ejemplos de algoritmos que no son programas.

Cuando enciende su automóvil, sigue un procedimiento paso a paso. El algoritmo podría parecer algo como esto:

1. Inserte la llave.
2. Asegúrese de que la transmisión esté en estacionar (o neutral).
3. Presione el pedal del acelerador.
4. Dé vuelta a la llave a la posición de encendido.
5. Si el motor enciende en seis segundos, libere la llave a la posición de encendido.
6. Si el motor no enciende en seis segundos, suelte la llave y el pedal del acelerador, espere diez segundos y repita los pasos 3 al 6, pero no más de cinco veces.
7. Si el automóvil no arranca, llame al taller mecánico.

Sin la frase "pero no más de cinco veces" en el paso 6, se podría estar intentando encender el automóvil por siempre. ¿Por qué? Debido a que si algo anda mal con el automóvil, repetir los pasos 3 al 6 una y otra vez no hará que encienda. Este tipo de situación de nunca acabar se llama *ciclo infinito*. Si se deja la frase "pero no más de cinco veces" fuera del paso 6, el procedimiento no se ajusta a la definición de un algoritmo. Un algoritmo debe terminar en una cantidad finita de tiempo para todas las condiciones posibles.

Suponga que un programador necesita un algoritmo para determinar el salario semanal de un empleado. El algoritmo refleja lo que se haría a mano:

1. Buscar la tasa de pago del empleado.
2. Determinar la cantidad de horas trabajadas durante la semana.
3. Si el número de horas trabajadas es menor o igual que 40, multiplique el número de horas por la tasa de pago para calcular salarios regulares.
4. Si el número de horas trabajadas es mayor que 40, multiplique 40 por la tasa de pago para calcular salarios regulares y luego multiplique la diferencia entre el número de horas trabajadas y 40 por 1½ veces la tasa de pago para calcular salarios de horas extras.
5. Sumar los salarios regulares a los de horas extras (si existen) para determinar salarios totales para la semana.

Los pasos que sigue la computadora con frecuencia son los mismos que usted usaría para hacer los cálculos a mano.

Después de desarrollar una solución general, el programador prueba el algoritmo, caminando por cada paso mental o manualmente. Si el algoritmo no funciona, el programador repite el proceso de resolver el problema, analiza de nuevo el problema y sugiere otro algoritmo. Por lo regular, el segundo algoritmo es sólo una variación del primero. Cuando el programador está satisfecho con el

algoritmo, lo traduce en un **lenguaje de programación**. En este libro se usa el lenguaje de programación C++.

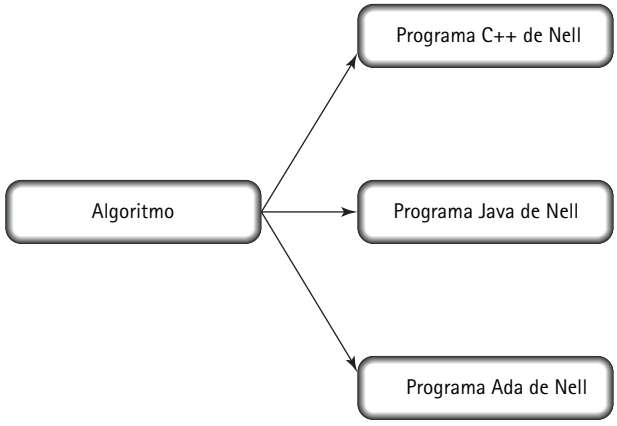
Lenguaje de programación Conjunto de reglas, símbolos y palabras especiales usado para implementar un programa de computadora.

Un lenguaje de programación es una forma simplificada del inglés (con símbolos matemáticos) que se adhiere a un conjunto estricto de reglas gramaticales. El inglés es un lenguaje demasiado complicado para que lo sigan las computadoras actuales. Los lenguajes de programación, debido a que limitan el vocabulario y la gramática, son mucho más simples.

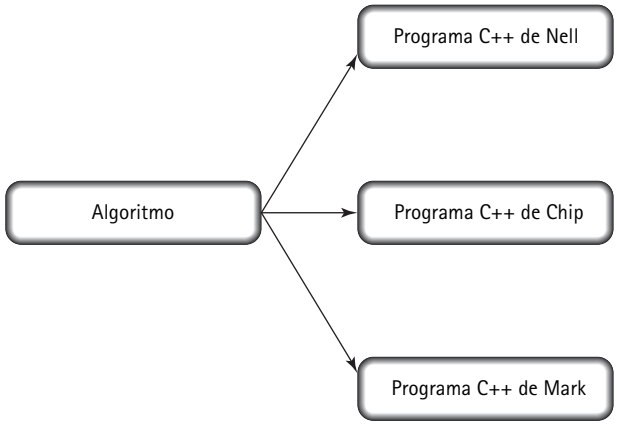
Aunque un lenguaje de programación es simple en forma, no siempre es fácil usarlo. Intente dar a alguien instrucciones para llegar al aeropuerto más cercano con un vocabulario de no más de 45 palabras y comenzará a ver el problema. La programación lo obliga a escribir instrucciones exactas muy simples.

Traducir un algoritmo en un lenguaje de programación se llama *codificar* el algoritmo. El producto de esa traducción, el programa, se prueba ejecutándolo en la computadora. Si el programa no produce los resultados deseados, el programador debe *depurarlo*; es decir, determinar qué está mal y luego modificar el programa, o incluso el algoritmo, para arreglarlo. La combinación de codificar y probar un algoritmo se llama *implementación*.

No hay una forma simple de implementar un algoritmo. Por ejemplo, un algoritmo se puede traducir en más de un lenguaje de programación. Cada traducción produce una implementación diferente. Incluso cuando dos personas traducen un algoritmo en el mismo lenguaje de programación, es probable que propongan implementaciones distintas (véase la figura 1-2). ¿Por qué? Porque todo



a) Algoritmo traducido en diferentes lenguajes



b) Algoritmo traducido por diferentes personas

Figura 1-2 *Diferencias en la implementación*

lenguaje de programación permite al programador cierta flexibilidad en cómo se traduce un algoritmo. Dada esta flexibilidad, las personas adoptan sus propios estilos al escribir programas, del mismo modo que lo hacen al escribir historias cortas o ensayos. Una vez que ya cuenta con algo de experiencia en la programación, desarrolla un estilo propio. En todo el libro se ofrecen consejos prácticos acerca del buen estilo de programación.

Algunas personas intentan acelerar el proceso de programación al ir directamente de la definición del problema a la codificación del programa (véase la figura 1-3). Un atajo aquí es muy tentador y en principio parece ahorrar mucho tiempo. Sin embargo, por muchas razones que se irán haciendo obvias a medida que lea este libro, esta clase de atajo toma en realidad *más* tiempo y esfuerzo. Desarrollar una solución general antes de escribir un programa ayuda a manejarlo, mantener claros sus pensamientos y evitar errores. Si al comienzo no se da tiempo para razonar y pulir su algoritmo, utilizará tiempo extra en la depuración y revisión de su programa. Así que ¡piense primero y codifique después! Mientras más pronto empiece a codificar, más tiempo le llevará elaborar un programa que funcione.

Una vez que un programa se ha puesto en uso, a menudo es necesario modificarlo. La modificación puede requerir arreglar un error descubierto durante el uso del programa o cambiar el programa en respuesta a cambios en los requisitos del usuario. Cada vez que se modifica el programa, es necesario repetir las fases de resolución del problema e implementación para los aspectos del programa que cambian. Esta fase del proceso de programación se conoce como mantenimiento y explica la mayor parte del esfuerzo empleado en la mayoría de los programas. Por ejemplo, un programa que se implementa en unos cuantos meses podría requerir que sea mantenido en un periodo de muchos años. Así, es una inversión económica de tiempo desarrollar la solución del problema inicial y la implementación del programa de manera cuidadosa. Juntas, las fases de resolución del problema, implementación y mantenimiento constituyen el *ciclo de vida* del programa.

Documentación Texto y comentarios que facilitan a otros la comprensión, uso y modificación de un programa.

Información Cualquier conocimiento que puede ser comunicado.

Datos Información en forma que una computadora puede usar.

Además de resolver el problema, ejecutar el algoritmo y mantener el programa, la **documentación** es una parte importante del proceso de programación. La documentación incluye explicaciones escritas del problema que se está resolviendo y la organización de la solución, comentarios insertados dentro del programa mismo y manuales del usuario que describen cómo usar el programa. Muchas personas distintas trabajan en la mayor parte de los programas durante un largo periodo. Cada una de esas personas debe poder leer y entender el código.

Después de escribir un programa, se debe dar a la computadora la información o datos necesarios para resolver el problema. La **información** es cualquier conocimiento que puede ser comunicado, incluso ideas y conceptos abstractos como "la Tierra es redonda". Los **datos** son la información en una forma que la computadora puede usar; por ejemplo, los números y letras constituyen las fórmulas que relacionan el radio de la Tierra con su volumen y área superficial. Pero los datos no están restringidos a

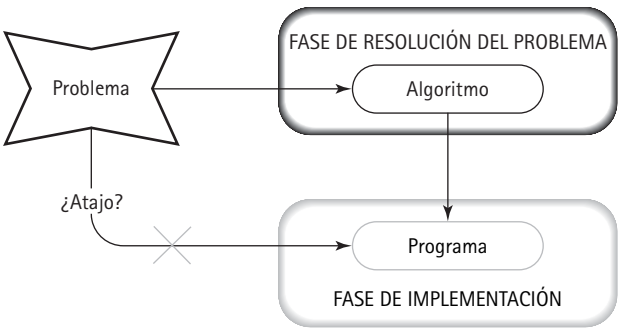


Figura 1-3 *¿Atajo de programación?*

números y letras. En la actualidad, las computadoras también procesan datos que representan sonido (que se reproducirá en las bocinas), imágenes gráficas (que se mostrarán en una pantalla de computadora o impresora), video (que se verá en un reproductor de DVD), etcétera.

Bases teóricas

Representación binaria de datos

En una computadora, los datos son representados electrónicamente por medio de pulsos de electricidad. Los circuitos eléctricos, en su forma más simple, están encendidos o apagados. Por lo común, un circuito encendido se representa por el número 1; un circuito apagado se representa por el número 0. Cualquier clase de datos se puede representar mediante combinaciones de "unos" y "ceros" suficientes. Sólo se tiene que elegir la combinación que representa cada conjunto de datos que se está usando. Por ejemplo, se podría elegir de manera arbitraria el patrón 1101000110 para representar el nombre C++.

Los datos representados por "unos" y "ceros" están en *forma binaria*. El sistema de números binarios (base 2) utiliza sólo "unos" y "ceros" para representar números. (El sistema de números decimales [base 10] emplea los dígitos 0 al 9.) La palabra *bit* (abreviatura para **b**inary dig**it**) se emplea con frecuencia para referirse a un solo 1 o 0. Así, el patrón 1101000110 tiene 10 bits. Un número binario con 10 bits puede representar 2^{10} (1 024) patrones diferentes. Un *byte* es un grupo de 8 bits; puede representar 2^8 (256) patrones. Dentro de la computadora, cada carácter (como la letra *A*, la letra *g* o un signo de interrogación) se representa normalmente por un byte. Cuatro bits, o la mitad de un byte, se llama *nibble* o *nybble* —nombre que originalmente fue propuesto en tono de burla, pero ahora es terminología estándar—. Los grupos de 16, 32 y 64 bits se llaman por lo general *palabras* (aunque a veces se emplean los términos *palabra corta* y *palabra larga* para hacer referencia a grupos de 16 y 64 bits, respectivamente).

El proceso de asignar patrones de bits a conjuntos de datos se llama *codificación*; se da el mismo nombre al proceso de traducir un algoritmo en un lenguaje de programación. Los nombres son los mismos porque el único lenguaje que reconocían las primeras computadoras era de forma binaria. Así, en los primeros días de las computadoras, programación significaba traducir datos y algoritmos en patrones de "unos" y "ceros".

Los esquemas de codificación binarios aún se emplean en la computadora para representar tanto las instrucciones que sigue como los datos que utiliza. Por ejemplo, 16 bits pueden representar los enteros decimales de 0 a $2^{16} - 1$(65 535). Los caracteres pueden ser representados también por combinaciones de bits. En un esquema de codificación, 01001101 representa a *M* y 01101101 representa a *m*. Para representar números negativos, números reales, números en notación científica, sonido, gráficas y video son necesarios esquemas de codificación más complicados. En el capítulo 10 se examina en detalle la representación de números y caracteres en la computadora.

Los patrones de bits que representan datos varían de una computadora a otra. Incluso en la misma computadora, lenguajes de programación diferentes pueden usar representaciones binarias distintas para los mismos datos. Un solo lenguaje de programación puede incluso usar el mismo patrón de bits para representar diversas cosas en contextos distintos. (Las personas hacen esto también. La palabra formada por las cuatro letras *tack* [en idioma inglés] tiene diferentes significados dependiendo de si habla acerca de tapicería, navegación, coser a máquina, pintura o montar a caballo.) La cuestión es que los patrones de bits por sí mismos carecen de significado. La manera en que se emplean los patrones es lo que les da su significado.

Por fortuna, ya no es necesario trabajar con esquemas de codificación binarios. Hoy día el proceso de codificar es normalmente sólo un asunto de escribir los datos con letras, números y símbolos. La computadora convierte de modo automático estas letras, números y símbolos a la forma binaria. Sin embargo, cuando trabaje con computadoras, se encontrará continuamente con números relacionados con potencias de 2 —números como 256, 32 768 y 65 536—, recordatorios de que el sistema de números binarios acecha en algún lugar cercano.

1.2 ¿Qué es un lenguaje de programación?

En la computadora, los datos —cualquiera que sea su forma— se almacenan y emplean en códigos binarios, cadenas de "unos" y "ceros". Las instrucciones y datos se almacenan en la memoria de la computadora por medio de estos códigos binarios. Si usted examinara los códigos binarios que representan instrucciones y datos en la memoria, no podría indicar la diferencia entre ellos; se distinguen sólo por la manera en que los usa la computadora. Esto hace posible que la computadora procese sus propias instrucciones como una forma de datos.

Lenguaje de máquina Lenguaje conformado por instrucciones en código binario, usado directamente por la computadora.

Lenguaje ensamblador Lenguaje de programación de bajo nivel en el que se emplea una ayuda nemotécnica para representar cada una de las instrucciones del lenguaje de máquina para una computadora particular.

En los inicios del desarrollo de las computadoras, el único lenguaje de programación disponible era la instrucción primitiva integrada en cada máquina, el **lenguaje de máquina** o *código de máquina*.

Aun cuando la mayoría de las computadoras realizan la misma clase de operaciones, sus diseñadores eligen diferentes conjuntos de códigos binarios para cada instrucción. Por tanto, el código de máquina para una computadora no es el mismo que para otra.

Cuando los programadores usaron el lenguaje de máquina para programar, tuvieron que introducir códigos binarios para las distintas instrucciones, un proceso tedioso propenso a error. Además, sus programas eran difíciles de leer y modificar. Con el tiempo, se desarrollaron los **lenguajes ensambladores** para facilitar el trabajo del programador.

Las instrucciones en un lenguaje ensamblador están en una forma fácil de recordar llamada *nemotécnica*. Las instrucciones características para la suma y la resta podrían parecerse a esto:

Lenguaje ensamblador	Lenguaje de máquina
ADD	100101
SUB	010011

Aunque el lenguaje ensamblador es más fácil para que los humanos trabajen con él, la computadora no puede ejecutar de modo directo las instrucciones. Uno de los descubrimientos fundamentales en la ciencia de la computación es que, debido a que una computadora puede procesar sus propias instrucciones como una forma de datos, es posible escribir un programa para traducir las instrucciones del lenguaje ensamblador en código de máquina. Esta clase de programa se llama **ensamblador**.

Ensamblador Programa que traduce lenguaje ensamblador en código de máquina.

Compilador Programa que traduce lenguaje de alto nivel en código de máquina.

Programa fuente Programa escrito en lenguaje de programación de alto nivel.

Programa objeto Versión del lenguaje de máquina de un programa fuente.

El lenguaje ensamblador es un paso en la dirección correcta, pero aún obliga a los programadores a pensar en términos de instrucciones de máquina individuales. Finalmente, los científicos de la computación desarrollaron lenguajes de programación de alto nivel. Estos lenguajes son más fáciles de usar que los lenguajes ensambladores o código de máquina porque se aproximan más al idioma inglés y a otros lenguajes naturales (véase la figura 1-4).

Un programa llamado **compilador** traduce los programas escritos en algunos lenguajes de alto nivel (C++, Pascal, FORTRAN, COBOL, Modula-2 y Ada, por ejemplo) en lenguaje de máquina. Si usted escribiera un programa en un lenguaje de alto nivel, puede ejecutarlo en cualquier computadora que tenga un compilador apropiado. Esto es posible porque la mayoría de los lenguajes de alto nivel están *estandarizados*, lo que significa que existe una descripción oficial del lenguaje.

Un programa en un lenguaje de alto nivel se llama **programa fuente**. Para el compilador, un programa fuente son sólo datos de entrada. Traduce el programa en un programa en lenguaje de máquina llamado **programa objeto** (véase la figura 1-5). Algunos compiladores producen también un *listado* (una copia del programa con mensajes de error y otra información insertada).

Un beneficio de los lenguajes de alto nivel estandarizados es que permiten escribir en *código portable* (o *independiente de la máquina*). Según se destaca en la figura 1-5, un programa escrito en lenguaje ensamblador o lenguaje de máquina no es transportable de una computadora a otra. Debido

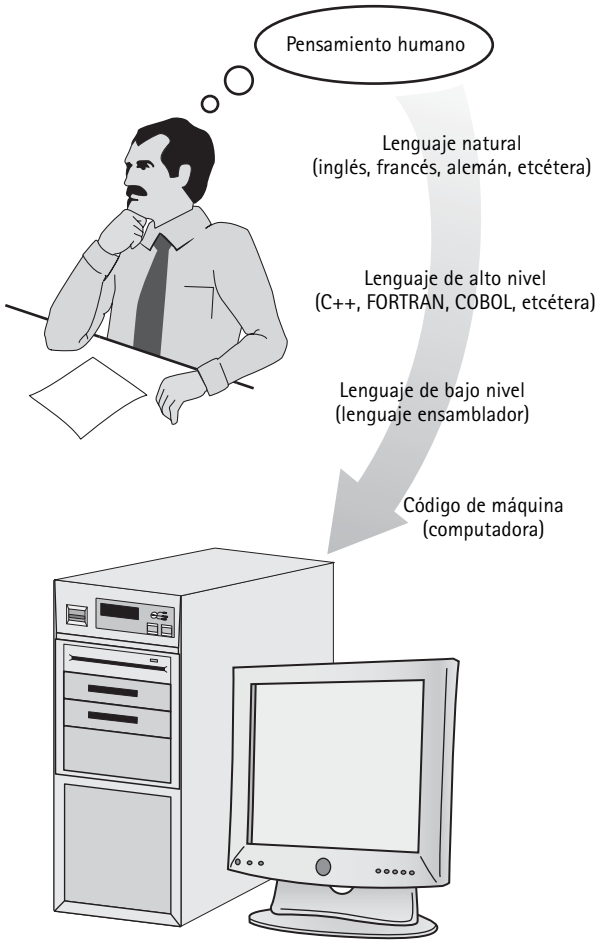


Figura 1-4 *Niveles de abstracción*

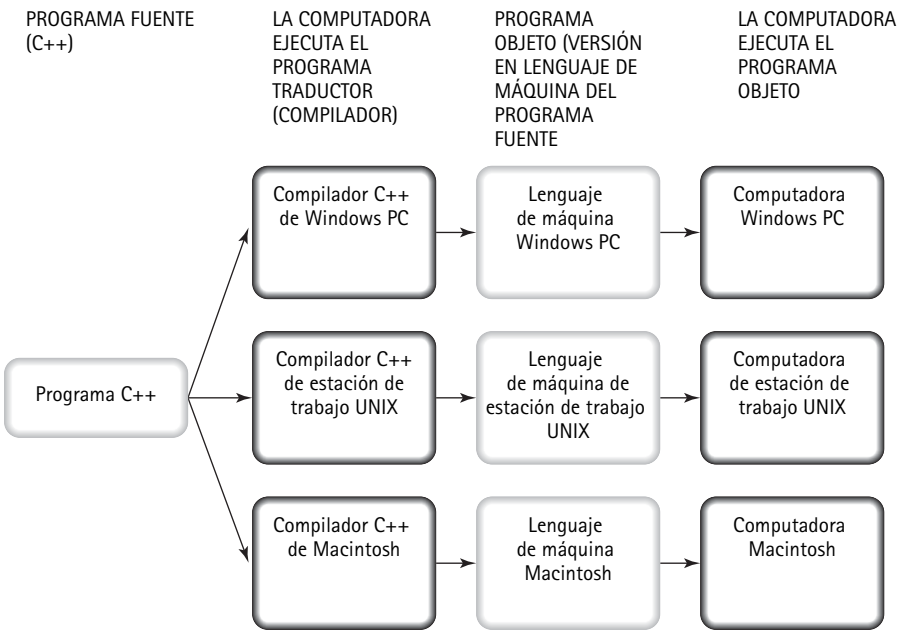


Figura 1-5 *Los lenguajes de programación de alto nivel permiten que los programas sean compilados en diferentes sistemas*

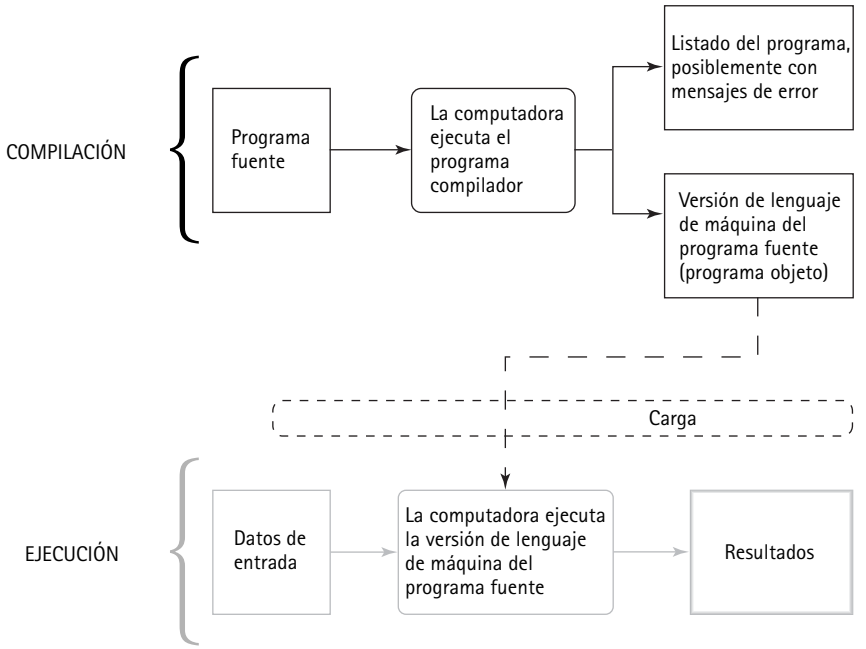


Figura 1-6 *Compilación y ejecución*

a que cada computadora tiene su propio lenguaje de máquina, un programa en lenguaje de máquina escrito para una computadora A no correrá en una computadora B.

Es importante entender que la compilación y la ejecución son dos procesos distintos. Durante la compilación, la computadora ejecuta el programa compilador. Durante la ejecución, el programa objeto se carga en la memoria de la computadora y remplaza al programa compilador. De esta manera, la computadora ejecuta el programa objeto y las instrucciones del programa (véase la figura 1-6).

Información básica

Compiladores e intérpretes

Algunos lenguajes de programación —LISP, Prolog y muchas versiones de BASIC, por ejemplo— son traducidos por un *intérprete* en vez de un compilador. Un intérprete traduce y *ejecuta* cada instrucción del programa fuente, una a la vez. En contraste, un compilador traduce todo el programa fuente en lenguaje de máquina, después de lo cual tiene lugar la ejecución del programa objeto.

El lenguaje Java emplea tanto un compilador como un intérprete. Primero, se compila un programa Java, no en un lenguaje de máquina de una determinada computadora, sino en un código intermedio llamado *bytecode*. A continuación, un programa llamado Máquina Virtual de Java (MVJ; JVM, por sus siglas en inglés) toma al programa bytecode y lo interpreta (traduce una instrucción de bytecode en lenguaje de máquina y la ejecuta, traduce la siguiente y la ejecuta, y así sucesivamente). De esta manera, un programa de Java compilado en bytecode es transportable a muchas computadoras diferentes, siempre y cuando cada computadora tenga su propia MVJ que pueda traducir el bytecode en el lenguaje de máquina de la computadora.

Las instrucciones en un lenguaje de programación reflejan las operaciones que puede realizar una computadora:

- Una computadora puede transferir datos de un dispositivo a otro.
- Una computadora puede introducir datos desde un dispositivo de entrada (un teclado o ratón, por ejemplo) y enviar datos a un dispositivo de salida (una pantalla).
- Una computadora almacena datos y los recupera de su memoria y área de almacenamiento secundario (las partes de una computadora se analizan en la siguiente sección).
- Una computadora compara dos valores de datos para igualdad y desigualdad.
- Una computadora puede efectuar operaciones aritméticas (suma y resta, por ejemplo) muy rápido.

Los lenguajes de programación requieren el uso de determinadas *estructuras de control* para expresar los algoritmos como programas. Hay cuatro formas básicas de estructurar sentencias (instrucciones) en la mayoría de los lenguajes de programación: de modo secuencial, condicional, repetitivo y con subprogramas (véase la figura 1-7). Una *secuencia* es una serie de sentencias que se ejecutan una después de otra. La *selección*, la estructura de control condicional, ejecuta sentencias diferentes dependiendo de determinadas condiciones. La estructura de control repetitiva, el *ciclo*, repite sentencias mientras se satisfacen ciertas condiciones. El *subprograma* permite estructurar un programa al descomponerlo en unidades más pequeñas. Cada una de estas formas de estructurar sentencias controla el orden en el cual la computadora ejecuta las sentencias, razón por la que se llaman estructuras de control.

Imagine que conduce un automóvil. Ir por un tramo recto de carretera es como seguir una secuencia de instrucciones. Cuando llega a una bifurcación, debe decidir por dónde ir y luego tomar una vía u otra. Esto es lo que hace la computadora cuando encuentra una estructura de control de selección (a veces llamada *bifurcación* o *decisión*) en un programa. Algunas veces se tiene que ir alrededor de una cuadra varias veces a fin de hallar un lugar para estacionarse. La computadora hace lo mismo cuando encuentra un ciclo en un programa.

Un subprograma es un proceso que consta de múltiples pasos. Todos los días, por ejemplo, usted sigue un procedimiento para ir de casa al trabajo. Tiene sentido entonces que alguien le dé instrucciones para llegar a una reunión diciendo: "dirígete a la oficina, luego recorre cuatro cuadras hacia el oeste", sin especificar todos los pasos que tuvo que efectuar para llegar a la oficina. Los subprogramas permiten escribir partes de los programas por separado y luego ensamblarlos en una forma final. Pueden simplificar en gran medida la tarea de escribir programas grandes.

1.3 ¿Qué es una computadora?

Usted puede aprender un lenguaje de programación, cómo escribir programas y cómo ejecutarlos sin saber mucho acerca de computadoras. Pero si sabe algo en relación con las partes de una computadora puede entender mejor el efecto de cada instrucción en un lenguaje de programación.

La mayoría de las computadoras tiene seis componentes básicos: unidad de memoria, unidad aritmética/lógica, unidad de control, dispositivos de entrada, dispositivos de salida y dispositivos de almacenamiento auxiliares. La figura 1-8 es un diagrama estilizado de los componentes básicos de una computadora.

Unidad de memoria Depósito interno para almacenamiento de datos en una computadora.

La **unidad de memoria** es una secuencia ordenada de celdas de almacenamiento, cada una capaz de contener un conjunto de datos. Cada celda de memoria tiene una dirección distinta a la que se hace referencia a fin de almacenar o recuperar datos de ella. Estas celdas de almacenamiento se llaman *celdas de memoria* o *localidades de memoria*.* La unidad de memoria contiene datos (datos de entrada o el producto de

* La unidad de memoria se conoce también como RAM, acrónimo para memoria de acceso aleatorio (llamada así porque se puede acceder a cualquier lugar de manera aleatoria).

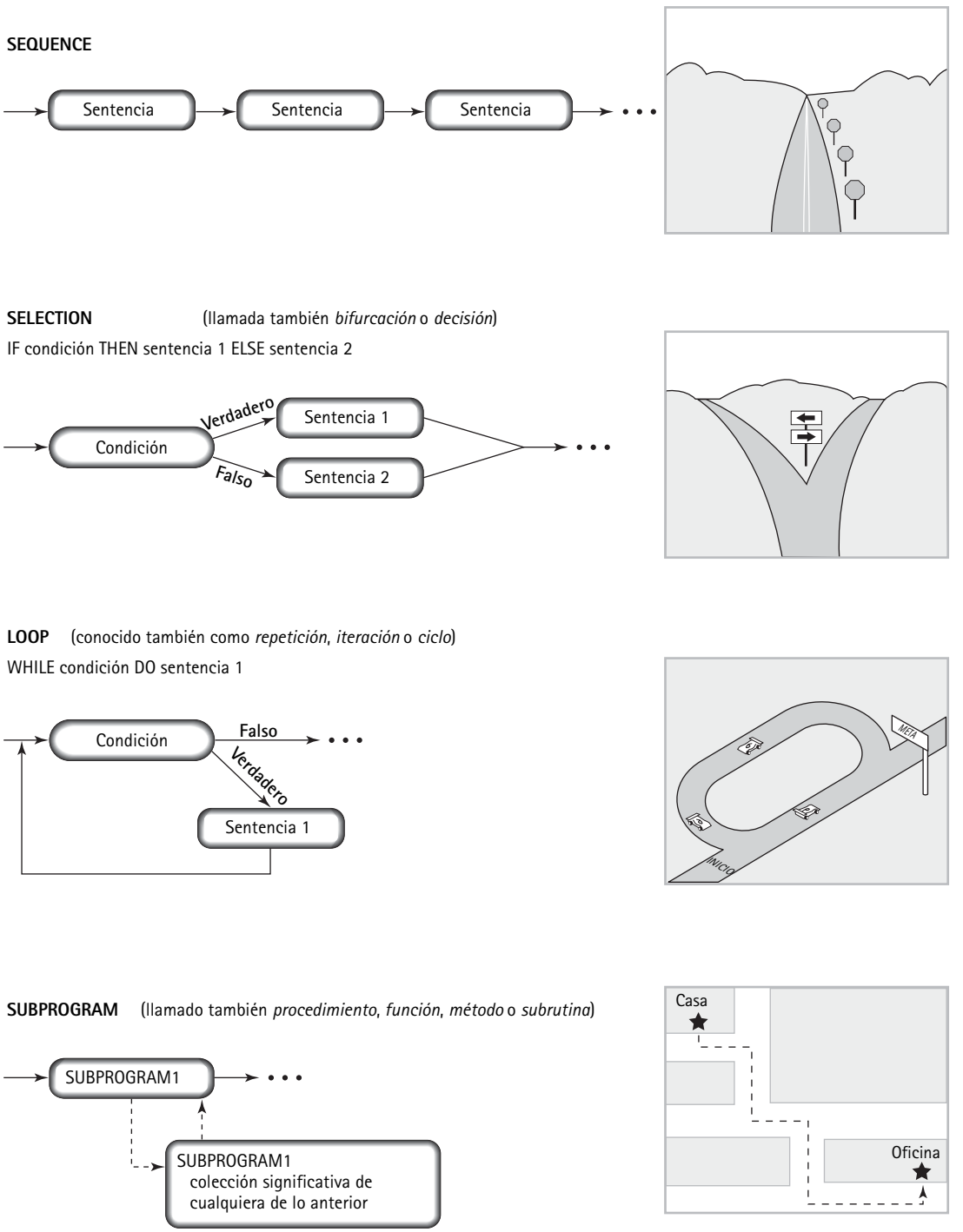


Figura 1-7 *Estructuras de control básicas de lenguajes de programación*

Unidad central de procesamiento (CPU) Parte de la computadora que ejecuta las instrucciones (programa) almacenadas en la memoria; está conformada por la unidad aritmética/lógica y la unidad de control.

Unidad aritmética/lógica (ALU) Componente de la unidad central de procesamiento que efectúa las operaciones aritméticas y lógicas.

un cálculo) e instrucciones (programas), como se muestra en la figura 1-9.

La parte de la computadora que sigue las instrucciones se llama **unidad central de procesamiento (CPU)**. Por lo común, el CPU tiene dos componentes. La **unidad aritmética/lógica (ALU)** efectúa las operaciones aritméticas (suma, resta, multiplicación y división) y las

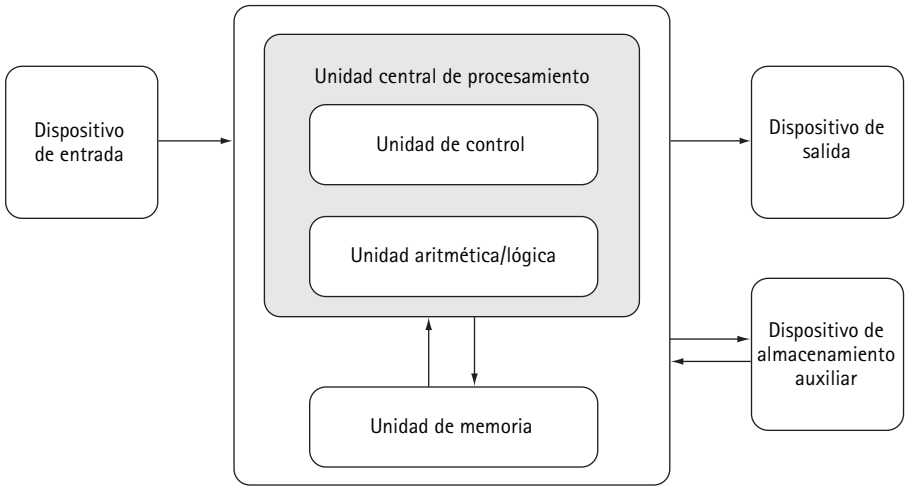


Figura 1–8 *Componentes básicos de una computadora*

operaciones lógicas (comparar dos valores). La **unidad de control** regula las acciones de los otros componentes de la computadora para que las instrucciones del programa se ejecuten en el orden correcto.

Unidad de control Componente de la unidad central de procesamiento que controla las acciones de los otros componentes para que se ejecuten las instrucciones (el programa) en el orden correcto.

Dispositivos de entrada/salida (I/O) Partes de la computadora que aceptan que los datos sean procesados (entrada) y presentan los resultados de ese procesamiento (salida).

Para poder usar las computadoras, se requiere alguna forma de hacer que los datos entren y salgan de ellas. Los **dispositivos de entada/salida (I/O)** aceptan los datos que serán procesados (entrada) y presentan valores de datos que han sido procesados (salida). Un teclado es un dispositivo de entrada común. Otro es un ratón, un dispositivo indicador. Una pantalla de video es un dispositivo de salida común, como lo son las impresoras y las pantallas de cristal líquido (LCD). Algunos dispositivos, como una conexión a una red de computadoras, se usan para entrada y salida.

En su mayoría, las computadoras simplemente mueven y combinan datos en la memoria. Los varios tipos de computadoras difieren sobre todo en el tamaño de su memoria, la velocidad con que pueden ser recuperados los datos, la eficiencia con que éstos se pueden mover o combinar, y las limitaciones en los dispositivos I/O.

Cuando se está ejecutando un programa, la computadora sigue una serie de pasos, el *ciclo buscar-ejecutar*:

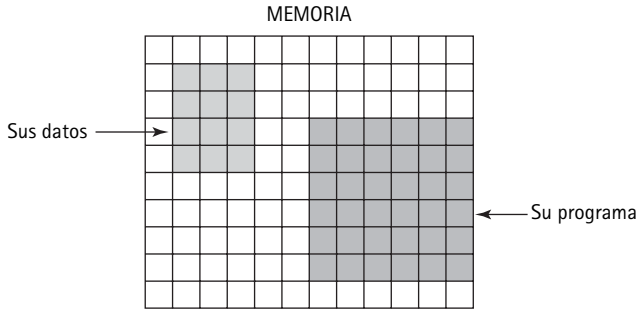


Figura 1–9 *Memoria*

1. La unidad de control recupera (*busca*) la siguiente instrucción codificada de la memoria.
2. La instrucción se traduce en señales de control.
3. Las señales de control indican la unidad apropiada (unidad aritmética/lógica, memoria, dispositivo I/O) para realizar (ejecutar) la instrucción.
4. La secuencia se repite desde el paso 1.

Dispositivos periféricos Dispositivo de entrada, salida o almacenamiento auxiliar que está conectado a la computatora.

Dispositivo de almacenamiento auxiliar Dispositivo que almacena datos en forma codificada de manera externa a la memoria principal de la computadora.

Las computadoras pueden tener una amplia variedad de **dispositivos periféricos** unidos a ellas. Un **dispositivo de almacenamiento auxiliar** o un *dispositivo de almacenamiento secundario*, retiene los datos codificados para la computadora hasta que se desee usarlos. En lugar de introducir datos cada vez, se pueden introducir sólo una vez y pedir a la computadora que los almacene en un dispositivo de almacenamiento auxiliar. Siempre que se requiera usar los datos, se indica a la computadora que transfiera los datos del dispositivo a su memoria. Por tanto, un dispositivo de almacenamiento auxiliar sirve como dispositivo de entrada y salida. Los dispositivos de almacenamiento auxiliar son unidades de disco y unidades de cinta magnética. Una *unidad de disco* es una cruza entre un reproductor de disco compacto y un grabador de cinta. Utiliza un disco delgado hecho de material magnético. Una cabeza de lectura/escritura (similar a la cabeza de grabar/reproducir en una grabadora de cinta) se desplaza sobre el disco que gira, recuperando o registrando datos. Una *unidad de cinta magnética* es como una grabadora y se usa con frecuencia para *respaldar* (hacer una copia de) los datos de un disco, en caso de que alguna vez se dañe.

Otros ejemplos de dispositivos periféricos son los siguientes:

- Escáneres, que "leen" imágenes visuales en papel y las convierten en datos binarios.
- Unidades CD-ROM (memoria de sólo lectura en disco compacto), que leen (pero no pueden escribir) datos almacenados en discos compactos removibles.
- Unidades CD-R (disco compacto-grabable), que pueden escribir en un CD particular una sola vez, pero pueden leerlo muchas veces.
- Unidades CD-RW (disco compacto-regrabable), que pueden escribir y leer de un CD particular muchas veces.
- Unidades DVD-ROM (memoria de sólo lectura en disco de video digital [o disco versátil digital]), que usa discos compactos con capacidad de almacenaje mucho mayor que la de los discos compactos comunes.
- Módems (moduladores/demoduladores), que convierten de una parte a otra entre datos binarios y señales que pueden ser enviadas en líneas telefónicas ordinarias.
- Tarjetas de audio y bocinas.
- Sintetizadores de voz.
- Cámaras digitales.

Hardware Componentes físicos de una computadora.

Software Programas de computadora; conjunto de programas disponibles en una computadora.

Juntos, todos estos componentes físicos se conocen como **hardware**. Los programas que permiten operar el hardware se denominan **software**. Normalmente, el hardware es de diseño fijo; el software se cambia con facilidad. De hecho, la facilidad con que se puede manejar el software es lo que hace de la computadora una herramienta tan versátil y poderosa.

Información básica

Computadoras personales, estaciones de trabajo y computadoras centrales

Existen computadoras de muchos tipos y tamaños. Las *computadoras centrales* son muy grandes (¡pueden llenar una habitación!) y muy rápidas. Una computadora central representativa consta de varios gabinetes llenos de componentes electrónicos. Dentro de esos gabinetes está la memoria, la unidad central de procesamiento y las unidades de entrada y salida. Es fácil localizar los distintos dispositivos periféricos: gabinetes separados contienen las unidades de disco y las unidades de cinta. Otras unidades son obviamente las impresoras y las terminales (monitores con teclados). Es común poder conectar cientos de terminales a una sola computadora central. Por ejemplo, todas las cajas registradoras en una cadena de tiendas departamentales podrían estar enlazadas a una sola computadora central.

En el otro extremo del espectro están las *computadoras personales*. Éstas son lo suficientemente pequeñas para colocarse de modo confortable sobre un escritorio. Debido a su tamaño, puede ser difícil localizar cada una de las partes dentro de las computadoras personales. Muchas son sólo una simple caja con una pantalla, un teclado y un ratón. Es necesario abrir la cubierta para ver la unidad central de procesamiento, que por lo común es sólo un componente electrónico llamado *circuito integrado* o *chip*.

Algunas computadoras personales tienen unidades de cinta, pero la mayoría opera sólo con unidades de disco, unidades de CD-ROM e impresoras. El CD-ROM y las unidades de disco para computadoras personales contienen, por lo regular, muchos menos datos que los discos empleados con las computadoras centrales. De manera similar, las impresoras conectadas a las computadoras personales son normalmente mucho más lentas que las utilizadas con las computadoras centrales.

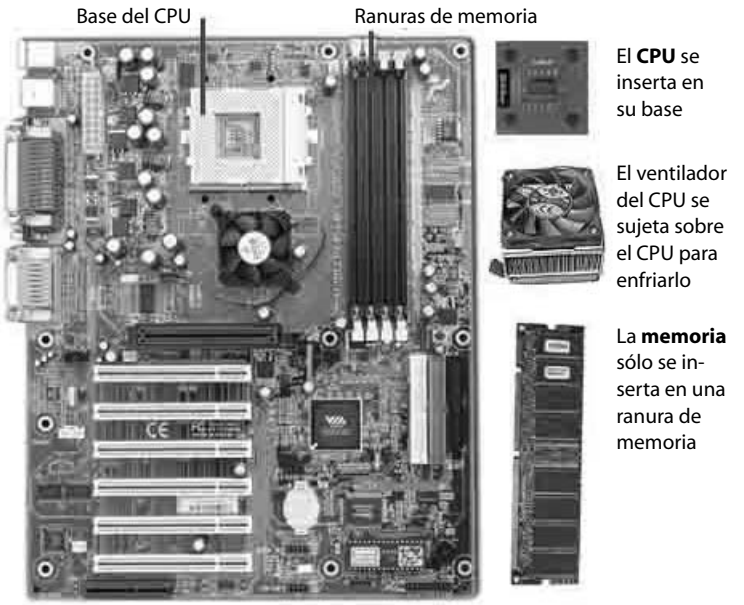
Las computadoras *laptop* o *notebook* son computadoras personales que han sido reducidas al tamaño de un cuaderno grande y operan con baterías para que sean portátiles. Constan de dos partes conectadas mediante una bisagra en la parte posterior de la cubierta. La parte superior contiene una pantalla plana de cristal líquido (LCD) y en la parte inferior están el teclado, el dispositivo de señalamiento, el procesador, la memoria y las unidades de disco.

Entre las computadoras centrales y las computadoras personales están las *estaciones de trabajo*. Estos sistemas computarizados de tamaño intermedio son, por lo común, más económicos que las computadoras centrales y más poderosos que las computadoras personales. Las estaciones de trabajo se preparan con frecuencia para que las use principalmente una persona a la vez. Una estación de trabajo se puede configurar también para que funcione como una computadora central pequeña, en cuyo caso se denomina *servidor*. Una estación de trabajo típica se parece mucho a una PC. De hecho, como las computadoras personales se han hecho más poderosas y las estaciones de trabajo son más compactas, la distinción entre ellas ha comenzado a desvanecerse.

Computadora central (Mainframe). *Foto cortesía de IBM*

(continúa)

Computadoras personales, estaciones de trabajo y computadoras centrales



A) Interior de una PC, unidad de sistema en partes *Foto cortesía de UltimaTec*

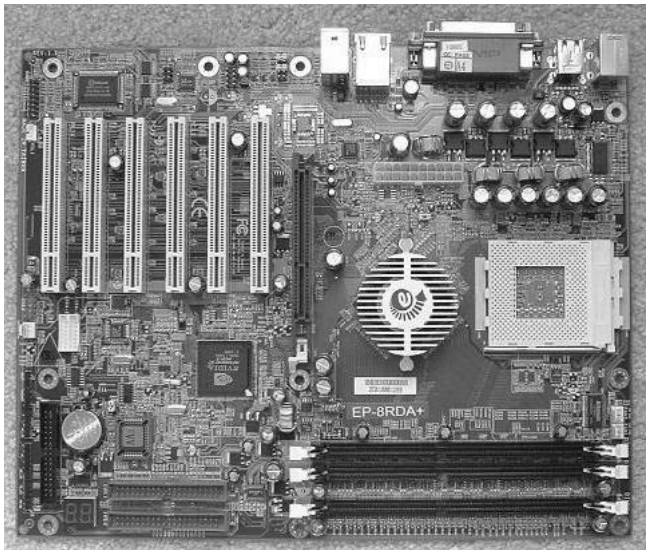
B) Computadora personal, Macintosh
Foto cortesía de Apple Computers

*C) Computadora personal**

* Las figuras (C) y (G) de las páginas 16 y 17 son cortesía de International Business Machines Corporation. Se prohíbe el uso no autorizado.

(continúa)

Computadoras personales, estaciones de trabajo y computadoras centrales



D) Interior de una PC, acercamiento de una tarjeta de sistema
Foto cortesía de Rob Williams

E) Computadora notebook
Foto cortesía de Apple Computers

*G) Estación de trabajo**

F) Supercomputadora
Foto cortesía de Cray, Inc.

Un último tipo de computadora que se debe mencionar es la *supercomputadora*, la clase de computadora más poderosa que existe. Las supercomputadoras están diseñadas normalmente para realizar cálculos científicos y de ingeniería en conjuntos de datos inmensos con gran velocidad. Son muy caras y, por tanto, su uso no es extenso.

Interfaz Enlace de conexión en un límite compartido que permite a los sistemas independientes satisfacer y actuar o comunicarse entre sí.

Sistema interactivo Sistema que permite la comunicación directa entre el usuario y la computadora.

Sistema operativo Conjunto de programas que controla todos los recursos de la computadora.

Editor Programa interactivo empleado para crear y modificar programas fuente o datos.

Además de los programas que compra o escribe el usuario, hay programas en la computadora que están diseñados para simplificar la **interfaz** usuario/computadora, lo que facilita el uso de la máquina. La interfaz entre el usuario y la computadora es un conjunto de dispositivos I/O —por ejemplo, un teclado, ratón y pantalla— que permiten al usuario comunicarse con la computadora. Los usuarios trabajan con teclado, ratón y pantalla en su lado del límite de interfaz; los cables conectados a estos dispositivos llevan los impulsos electrónicos con los que trabaja la computadora en su lado del límite de interfaz. En el límite mismo está un mecanismo que traduce la información para los dos lados.

Al comunicarse directamente con la computadora, se está usando un **sistema interactivo.** Los sistemas interactivos permiten la entrada directa de programas y datos y proporcionan retroalimentación inmediata al usuario. En contraste, los *sistemas por lotes* requieren que todos los datos sean introducidos antes de que se ejecute un programa y proporcionan retroalimentación sólo después de que se ha ejecutado un programa. En este texto se centra la atención en los sistemas interactivos, aunque en el capítulo 4 se examinan programas orientados a archivos, que comparten ciertas similitudes con los sistemas por lotes.

El conjunto de programas que simplifica la interfaz usuario/computadora y mejora la eficacia del procesamiento se denomina *software de sistema*. Incluye el compilador así como el sistema operativo y el editor (véase la figura 1-10). El **sistema operativo** controla todos los recursos de la computadora. Puede introducir programas, llamar al compilador, ejecutar programas objeto y realizar cualquier otra instrucción del sistema. El **editor** es un programa interactivo utilizado para crear y modificar programas fuente o datos.

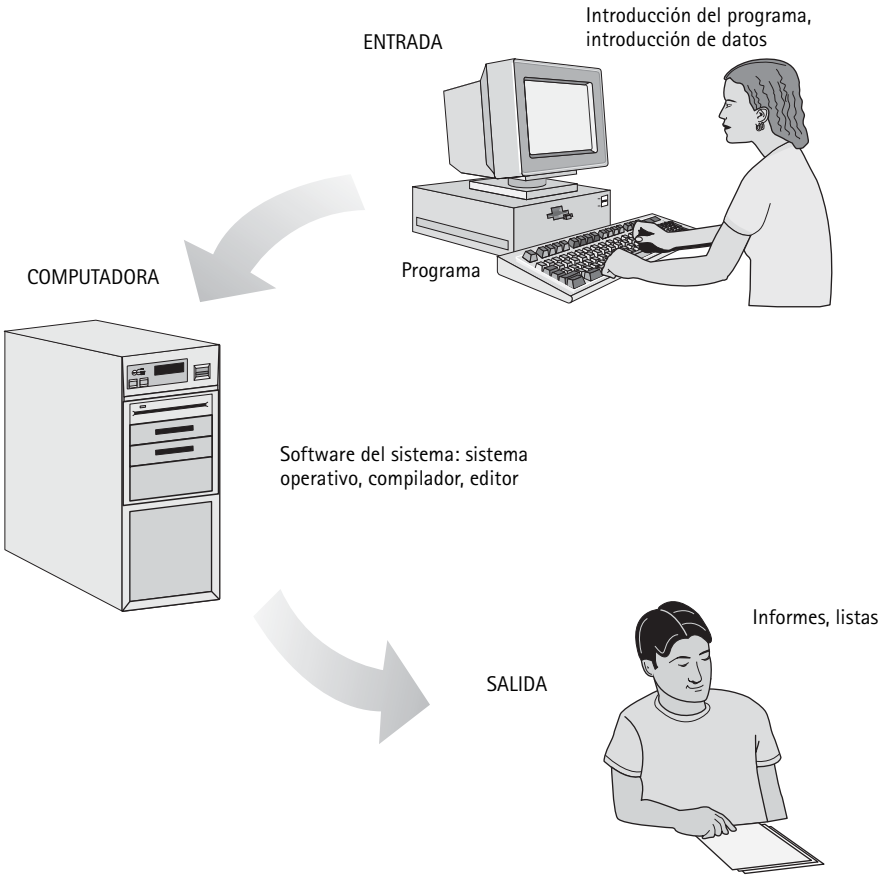


Figura 1-10 *Interfaz usuario/computadora*

Aunque las computadoras solitarias (*independientes*) son comunes en los hogares y negocios pequeños, es muy común que muchos usuarios se conecten juntos y formen una *red*. En una *red de área local* (*LAN, por sus siglas en inglés*) las computadoras están conectadas mediante cables y deben estar razonablemente cerca, como en un edificio de oficinas. En una *red de área amplia* (*WAN, por sus siglas en inglés*) o *red de largo alcance*, las computadoras están muy apartadas desde el punto de vista geográfico y se comunican por medio de líneas telefónicas, cable de fibra óptica u otros medios. La red de largo alcance mejor conocida es la Internet, que al principio se diseñó como un medio para que las universidades, negocios y agencias de gobierno intercambiaran información de investigación. La Internet se hizo popular con el establecimiento de la *World Wide Web*, un sistema de computadoras enlazadas por medio de Internet que soporta documentos formateados específicamente (*páginas web*) que contienen texto, gráficas, audio y video.

Información básica

Los orígenes de C++

A finales de la década de 1960 y principios de la de 1970, Dennis Ritchie creó el lenguaje de programación C en AT&T Bell Labs. En aquella época, un grupo de personas en los laboratorios Bell estaba diseñando el sistema operativo UNIX. Al inicio, UNIX se escribía en lenguaje ensamblador, como era costumbre para casi todo el software de sistema en esos días. Para evitar las dificultades de programar en lenguaje ensamblador, Ritchie inventó C como un lenguaje de programación de sistema. C combina las características de bajo nivel de un lenguaje ensamblador con la facilidad de uso y portabilidad de un lenguaje de alto nivel. UNIX se reprogramó de modo que casi 90% se escribió en C y el resto en lenguaje ensamblador.

Las personas suelen preguntarse de dónde viene el enigmático nombre C. En la década de 1960 un lenguaje de programación denominado BCPL (Basic Combined Programming Language) tuvo un seguimiento pequeño pero leal, sobre todo en Europa. De BCPL surgió otro lenguaje con B como abreviatura para su nombre. Para su lenguaje, Dennis Ritchie adoptó características del lenguaje B y decidió que el sucesor de B debería nombrarse C. Así que el avance fue de BCPL a B y después a C.

En 1985 Bjarne Stroustrup, también de laboratorios Bell, inventó el lenguaje de programación C++. Añadió al lenguaje C características para abstracción de datos y programación orientada a objetos (temas que se analizan más adelante en este libro). En lugar de nombrarlo lenguaje D, el grupo de laboratorios Bell de una manera humorística lo nombró C++. Como se verá más adelante, ++ significa la operación de incremento en los lenguajes C y C++. Dada una variable *x*, la expresión *x*++ indica incrementar (sumar uno a) el valor actual de *x*. Por tanto, el nombre C++ hace pensar en una versión mejorada ("incrementada") del lenguaje C.

Desde los años en que el doctor Stroustrup inventó C++, el lenguaje comenzó a evolucionar en formas un poco diferentes en distintos compiladores de C++. Aunque las características fundamentales de C++ fueron casi las mismas en los compiladores de las empresas, una compañía podía añadir una nueva característica de lenguaje, mientras que otra no. Como resultado, los programas de C++ no siempre fueron transportables de un compilador al siguiente. La comunidad dedicada a la programación acordó que el lenguaje necesitaba ser estandarizado, y un comité conjunto de la International Standards Organization (ISO) y el American National Standards Institute (ANSI) comenzó el largo proceso de crear un estándar de lenguaje C++. Después de varios años de análisis y debate, el estándar de lenguaje ISO/ANSI para C++ se aprobó de manera oficial a mediados de 1998. La mayoría de los compiladores actuales de C++ avalan el estándar ISO/ANSI (de aquí en adelante llamado *estándar C++*). Para ayudarlo si está usando un compilador preestándar, en todo el libro se indican las discrepancias entre las características de lenguaje antiguas y las nuevas que podrían afectar la manera como escribe sus programas.

Aunque en un principio C se propuso como un lenguaje de programación de sistema, tanto C como C++ se emplean mucho hoy día en negocios, industria y computación personal. C++ es poderoso y versátil e incorpora una amplia variedad de conceptos de programación. En este libro el alumno conocerá una parte sustancial del lenguaje, pero C++ incorpora características complejas que van más allá del alcance de un curso de programación introductorio.

1.4 Ética y responsabilidades en la profesión de computación

Toda profesión opera con un conjunto de principios éticos que ayudan a definir las responsabilidades de las personas que la practican. Por ejemplo, los médicos tienen la responsabilidad ética de mantener confidencial la información acerca de sus pacientes. Los ingenieros tienen una responsabilidad ética con sus patrones de proteger la información de propiedad exclusiva, pero también tienen la responsabilidad de proteger al público y al ambiente del daño que podría resultar de su trabajo. Los escritores están comprometidos éticamente a no plagiar el trabajo de otros, etcétera.

La computadora presenta un vasto y novedoso ámbito de capacidades que pueden afectar al ambiente de manera espectacular. Esto enfrenta a la sociedad con muchas situaciones éticas nuevas. Algunas de las prácticas éticas se aplican a la computadora, mientras que otras situaciones requieren nuevas reglas éticas. En algunos casos, podría no haber normas establecidas, pero corresponde a la persona decidir lo que es ético. En esta sección se examinan algunas situaciones comunes encontradas en la profesión de la computación que dan lugar a asuntos éticos.

Un profesional en la industria de la computación, como cualquier otro, tiene conocimiento que le permite hacer ciertas cosas que otros no pueden. Saber cómo tener acceso a las computadoras, cómo programarlas y cómo manejar datos da al profesional de las computadoras la capacidad para crear productos nuevos, resolver problemas importantes y ayudar a las personas a manejar sus interacciones con el mundo aún más complejo en el que vivimos. El conocimiento de las computadoras puede ser un medio poderoso para efectuar un cambio positivo.

El conocimiento puede ser empleado también de maneras no éticas. Una computadora puede ser programada para activar una bomba, sabotear la línea de producción de un competidor o robar dinero. Aunque estos ejemplos constituyen un punto extremo y son inmorales en cualquier contexto, hay ejemplos más sutiles que son únicos para las computadoras.

Piratería de software

Es fácil copiar el software de computadora. Pero al igual que los libros, el software está en general protegido por las leyes de los derechos de autor. Es ilegal copiar software sin permiso de su creador. Tal copia se llama **piratería de software**.

Piratería de software Copia no autorizada de software para uso personal o uso por parte de otros.

Las leyes de derechos de autor protegen a los creadores de software (y libros y arte), de modo que puedan obtener ganancias del esfuerzo y dinero gastado en el desarrollo de software. El desarrollo de un paquete de software puede costar millones de dólares, y este costo (junto con el de producir el paquete, enviarlo, apoyar a los clientes y permitir que el vendedor al menudeo incremente el precio) se refleja en el precio de compra. Si las personas hacen copias no autorizadas del software, entonces la compañía pierde esas ventas y tiene que aumentar sus precios para compensar o gastar menos dinero en el desarrollo de versiones mejoradas del software; en cualquier caso, se hace más difícil lograr un precio deseable del software.

En ocasiones los piratas de software racionalizan su robo de software con la excusa de que sólo hacen una copia para su propio uso. Después de todo, no es que estén vendiendo gran cantidad de copias ilegales. Pero si miles de personas hacen lo mismo, entonces la compañía pierde millones de dólares, lo que origina precios más altos para todos.

Los profesionales de la computación tienen la obligación ética de no participar en la piratería de software y deben tratar de evitar que ocurra. Nunca copie software sin autorización. Si alguien le pide una copia de una pieza de software, debe negarse a proporcionarla. Si alguien le pide "prestado" el software para probarlo, dígale que lo puede hacer en su máquina (o en la tienda de un vendedor al menudeo), pero no haga una copia.

Esta regla no está restringida a duplicar software protegido por las leyes del derecho de autor; incluye el plagio de todo o parte del código que pertenece a cualquier otro. Si alguien le autoriza copiar un poco de su código, entonces, como cualquier escritor responsable, debe agradecer a esa persona con una cita en el código.

Privacidad de datos

La computadora permite la compilación de bases de datos que contienen información útil acerca de las personas, compañías, regiones geográficas, etc. Estas bases de datos permiten que los patrones emitan cheques de nómina, que los bancos cambien el cheque de un cliente en cualquier sucursal, que el gobierno cobre impuestos y que los comerciantes en masa envíen publicidad por correo. Aunque no es posible cuidar todo uso de las bases de datos, tienen por lo general beneficios positivos. Sin embargo, pueden ser empleadas de maneras negativas.

Por ejemplo, un ladrón de automóviles que tiene acceso al registro estatal de vehículos motorizados podría imprimir una lista de modelos de automóviles lujosos junto con las direcciones de sus dueños. Un espía industrial podría robar datos de clientes de la base de datos de una compañía y venderlos a un competidor. Aunque es evidente que son actos ilegales, los profesionales de la computación enfrentan otras situaciones que no son tan obvias.

Suponga que su trabajo incluye administrar la base de datos de la nómina de la compañía. Podría estar tentado a indagar en la base de datos para ver cuál es su salario en comparación con el de sus compañeros; sin embargo, este acto es inmoral y una invasión al derecho de sus compañeros a la privacidad, porque dicha información es confidencial. Un ejemplo de información pública es un número telefónico listado en un directorio telefónico. La información privada incluye cualquier dato que se proporciona con el entendido de que se empleará sólo para un fin específico (como los datos acerca de una aplicación de tarjeta de crédito).

Un profesional de la computación tiene la responsabilidad de evitar sacar ventaja del acceso especial que pueda tener a datos confidenciales. El profesional tiene también la responsabilidad de proteger esos datos contra el acceso no autorizado. Proteger los datos puede requerir cosas simples como destruir impresiones viejas, mantener las copias de respaldo en un gabinete cerrado con llave y no usar claves que sean fáciles de adivinar (por ejemplo, un nombre o palabra), así como medidas más complejas como la *encriptación* (mantener los datos almacenados en una forma codificada secreta).

Uso de recursos de computadora

Si alguna vez ha comprado una computadora, sabe que cuesta dinero. Una computadora personal puede ser relativamente económica, pero todavía es una compra importante. Las computadoras más grandes pueden costar millones de dólares. Operar una PC puede costar unos cuantos dólares al mes por la electricidad y un desembolso ocasional por papel, discos y reparaciones. La operación de computadoras más grandes puede costar decenas de miles de dólares por mes. Sin importar el tipo de computadora, quien posea una tiene que pagar estos costos. Así lo hace porque la computadora es un recurso que justifica su costo.

La computadora es un recurso inusual porque es valioso sólo cuando se ejecuta un programa. Así, el tiempo de computadora es en realidad el recurso valioso. No hay diferencia física importante entre una computadora que está trabajando y una que está desocupada. En contraste, un automóvil está en movimiento cuando está funcionando. Por tanto, el uso no autorizado de una computadora es diferente del uso no autorizado de un automóvil. Si una persona usa el automóvil de otro sin permiso, ese individuo debe tomar posesión de él físicamente; es decir, robarlo. Si alguien usa una computadora sin autorización, la computadora no es robada de modo físico, pero al igual que en el caso del robo de automóvil, el dueño es privado de un recurso por el que está pagando.

Para algunas personas, el robo de los recursos de computadora es un juego, como pasear sin permiso en un automóvil. El ladrón en realidad no quiere los recursos, sólo el reto de burlar el sistema de seguridad de una computadora y ver cuán lejos puede llegar sin ser atrapado. El éxito proporciona un estímulo al ego de esta clase de personas. Muchos ladrones de recursos de computadoras piensan que sus acciones son aceptables si no dañan, pero en todos los casos en que el trabajo real es desplazado de la computadora por tales actividades, es claro que hay un daño. Aunque no sirva para otra cosa, el ladrón está transgrediendo la propiedad del dueño de la computadora. Por analogía, considere que aunque no causa daño alguien que entra a su recámara y toma una siesta mientras usted no está, tal acción es preocupante para usted porque implica una amenaza de posible daño físico.

Virus Programa de computadora que se reproduce por sí mismo, a menudo con el propósito de invadir otras computadoras sin autorización y tal vez con la intención de hacer daño.

Otros ladrones pueden ser maliciosos. Del mismo modo que un joven choca a propósito un automóvil que ha robado, estas personas destruyen o corrompen datos para causar daño. Es posible que tengan una sensación de poder por ser capaces de dañar a otros con impunidad. Algunas veces estas personas dejan programas que actúan como bombas de tiempo para causar daño cuando ellos ya se han ido. Otra clase de programa que puede ser dejado es un **virus**, un programa que se reproduce por sí mismo, a menudo con la finalidad de pasar a otras computadoras. Los virus pueden ser benignos, ya que el único daño que causan es usar algunos recursos. Otros pueden ser destructivos y causar que el daño se disemine a los datos. Han ocurrido incidentes en los que los virus han costado millones de dólares en pérdida de tiempo de computadora y datos.

Los profesionales de la computación tienen la responsabilidad ética de nunca usar los recursos de computadora sin permiso, lo cual incluye actividades como hacer trabajo personal en la computadora de un patrón. También se tiene la responsabilidad de ayudar a proteger los recursos a los que se tiene acceso; esto se logra al usar contraseñas indescifrables y mantenerlas en secreto, observar signos de uso inusual de la computadora, escribir programas que no generen conflictos en el sistema de seguridad de una computadora, etcétera.

Ingeniería de software

En gran medida, los humanos han llegado a depender de las computadoras en muchos aspectos de sus vidas. Esa confianza es propiciada por la percepción de que las computadoras funcionan de manera confiable; es decir, funcionan correctamente la mayor parte del tiempo. Sin embargo, la confiabilidad de una computadora depende del cuidado que se tome al escribir su software.

Los errores en un programa pueden tener consecuencias graves, como se ilustra en los siguientes ejemplos de incidentes relacionados con errores de software. Un error en el software de control del avión caza a reacción F-18 causó que se pusiera boca abajo la primera vez que voló en el ecuador. Un lanzamiento de cohete se salió de control y tuvo que ser destruido porque había una coma escrita en lugar de un punto en su software de control. Una máquina de terapia con radiación mató a varios pacientes debido a que un error de software causó que la máquina operara a plena potencia cuando el operador tecleó varios comandos demasiado rápido.

Aun cuando el software se usa en situaciones menos críticas, los errores pueden tener efectos importantes. Ejemplos de esta clase son los siguientes:

- Un error en su procesador de palabras que ocasiona que su trabajo se pierda horas antes de entregarlo.
- Un error en un programa estadístico que hace que un científico llegue a una conclusión equivocada y publique un artículo del que se debe retractar después.
- Un error en un programa de preparación de impuestos que produce una respuesta incorrecta, la cual da lugar a una multa.

Ingeniería de software Aplicación de metodologías y técnicas de ingeniería tradicionales para el desarrollo de software.

Los programadores tienen, por tanto, la responsabilidad de desarrollar software sin errores. El proceso que se emplea para desarrollar software correcto se conoce como **ingeniería de software**.

La ingeniería de software tiene muchos aspectos. El ciclo de vida del software descrito al comienzo de este capítulo presenta las etapas en el desarrollo de software. En cada una de estas etapas se emplean técnicas distintas. Muchas de las técnicas se atienden en este texto. En el capítulo 4 se introducen metodologías para desarrollar algoritmos correctos. Se analizan estrategias para probar y validar programas en cada capítulo. Se usa un lenguaje de programación moderno que permite escribir programas legibles bien organizados, etcétera. Algunos aspectos de la ingeniería de software, como el desarrollo de una especificación matemática formal para un programa, están más allá del alcance de este libro.

1.5 Técnicas de resolución de problemas

Usted debe resolver problemas todos los días, con frecuencia sin tener conciencia del proceso que se sigue para llegar a la solución. En un ambiente de aprendizaje, normalmente se tiene la mayoría de la información necesaria: un enunciado claro del problema, la entrada necesaria y la salida requerida. En la vida real, el proceso no siempre es tan simple. Por lo común hay que definir el problema y luego decidir con qué información se tiene que trabajar y cuáles deben ser los resultados.

Después de que comprende y analiza un problema, debe llegar a una solución, un algoritmo. Antes se definió un algoritmo como un procedimiento paso a paso para resolver un problema en una cantidad de tiempo finita. Aunque trabaje con algoritmos todo el tiempo, la mayor parte de su experiencia con ellos es en el contexto de *seguirlos*. Usted sigue una receta, participa en un juego, ensambla un juguete, toma medicina. En la fase de resolución de problemas de la programación de computadoras, se estará *diseñando* algoritmos, no siguiéndolos. Esto significa que debe estar consciente de las estrategias que emplea para resolver problemas a fin de aplicarlas a problemas de programación.

Haga preguntas

Si se le encomienda una tarea de manera verbal, usted hace preguntas —¿cuándo?, ¿por qué?, ¿dónde?—, hasta que entiende exactamente lo que tiene que hacer. Si las instrucciones son escritas, podría poner interrogaciones en el margen, subrayar una palabra u oración, o indicar de alguna otra manera que la tarea no está clara. Sus preguntas pueden ser contestadas en un párrafo posterior, o bien podría tener que discutirlas con la persona que le asignó la tarea.

Estas son algunas de las preguntas que podría hacer en el contexto de la programación:

- ¿Con qué hago el trabajo?; es decir, ¿cuáles son mis datos?
- ¿A qué se parecen los datos?
- ¿Cuántos datos hay?
- ¿Cómo sé que he procesado todos los datos?
- ¿A qué se debe parecer mi resultado?
- ¿Cuántas veces se tiene que repetir el proceso?
- ¿Qué condiciones especiales de error podrían surgir?

Busque cosas que sean familiares

Nunca reinvente la rueda. Si existe una solución, utilícela. Si ha resuelto antes el mismo problema o uno similar, sólo repita su solución. Las personas son buenas para reconocer situaciones similares. No tiene que aprender cómo ir a la tienda a comprar leche, luego comprar huevos y después dulces. Se sabe que ir a la tienda es siempre lo mismo; sólo lo que se compra es diferente.

En la programación, algunos problemas ocurren una y otra vez en modos distintos. Un buen programador reconoce de inmediato una subtarea que ha resuelto antes y la inserta en la solución. Por ejemplo, determinar las temperaturas diarias alta y baja es el mismo problema que hallar las calificaciones mayor y menor en una prueba. Se desean los valores máximo y mínimo en un conjunto de números (véase la figura 1-11).

Resuelva por analogía

Con frecuencia un problema le recuerda otro similar que ha visto antes. Puede tener a mano la solución de un problema con más facilidad si recuerda cómo resolvió el anterior. En otras palabras, haga una analogía entre los dos problemas. Por ejemplo, una solución a un problema de proyección en perspectiva de una clase de arte podría ayudar a entender cómo calcular la distancia hasta una señal cuando hace una caminata en el campo. A media que resuelva el nuevo problema, encontrará cosas distintas a las del problema anterior, pero por lo común son detalles que puede tratar uno a la vez.

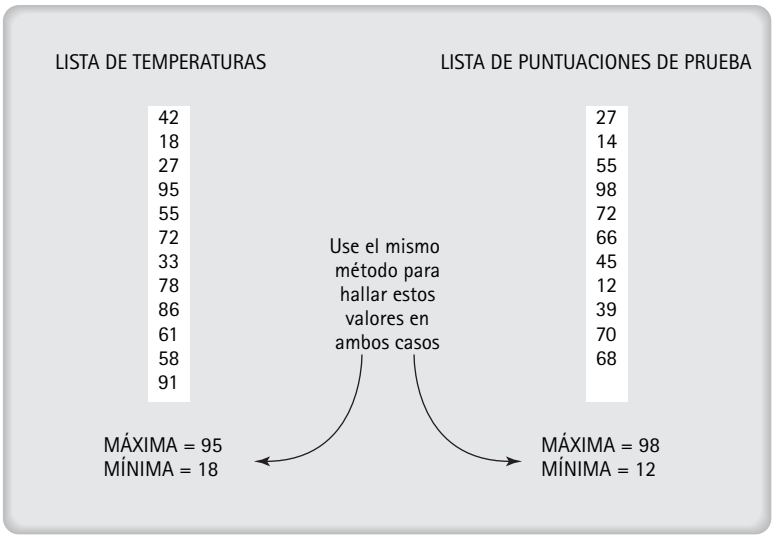


Figura 1-11 *Busque las cosas que le sean familiares*

La analogía es sólo una aplicación más amplia de la estrategia de buscar cosas que son familiares. Cuando intenta hallar un algoritmo para resolver un problema, no se limite a las soluciones orientadas a la computadora. Retroceda e intente obtener una visión más amplia del problema. No se preocupe si su analogía no tiene una correspondencia perfecta; la única razón para usar una analogía es que da un lugar para empezar (véase la figura 1-12). Los mejores programadores son personas con amplia experiencia en la resolución de toda clase de problemas.

Análisis de medios y fines

Con frecuencia se dan el lugar inicial y el lugar final; el problema es definir un conjunto de acciones que se puedan usar para ir de uno al otro. Suponga que quiere ir de Boston, Massachusetts, a Austin, Texas. Conoce el lugar inicial (usted está en Boston) y el lugar final (quiere estar en Austin). El problema es cómo ir de uno al otro. En este ejemplo, hay muchas opciones. Puede ir en avión, caminar, pedir un aventón, usar bicicleta o cualquier cosa. El método que elija depende de sus circunstancias. Si tiene prisa es probable que decida ir en avión.

Una vez que ha reducido el conjunto de acciones, tiene que resolver los detalles. Podría ser útil establecer objetivos intermedios que son más fáciles de satisfacer que el objetivo global. Suponga que hay un vuelo directo muy barato a Austin fuera de Newark, Nueva Jersey. Podría decidir dividir el viaje en tramos: Boston a Newark y luego Newark a Austin. Su objetivo intermedio es ir de Boston a Newark. Ahora sólo tiene que examinar los medios para cumplir ese objetivo intermedio (véase la figura 1-13).

Un sistema de catálogo de una biblioteca puede dar una idea de cómo organizar un inventario.

Figura 1-12 *Analogía*

Inicio: Boston **Objetivo:** Austin	**Medios:** *Volar*, caminar, pedir aventón, bicicleta, manejar, navegar, autobús
Inicio: Boston **Objetivo:** Austin	**Medios revisados:** Volar a Chicago y luego a Austin; *volar a Newark y luego a Austin*; volar a Atlanta y después a Austin
Inicio: Boston **Objetivo intermedio:** Newark **Objetivo:** Austin	**Medios para lograr el objetivo intermedio:** *Vuelo con transbordo*, caminar, pedir aventón, bicicleta, conducir, navegar, autobús
Solución: Tome el vuelo a Newark y luego el económico a Austin	

Figura 1-13 *Análisis de medios y fines*

La estrategia global de análisis de medios y fines es definir los fines y luego analizar los medios para lograrlos. El proceso se traduce con facilidad a la programación de computadoras. Se comienza por escribir lo que es la entrada y lo que debe ser la salida. Luego, se consideran las acciones que puede realizar una computadora y se elige una secuencia de acciones que pueden transformar los datos en resultados.

Dividir y vencer

Con frecuencia los problemas grandes se dividen en unidades más pequeñas que son más fáciles de manejar. Limpiar toda la casa podría parecer abrumador; limpiar las habitaciones una a la vez parece mucho más accesible. El mismo principio se aplica a la programación. Se descompone un problema grande en piezas más pequeñas que se pueden resolver de manera individual (véase la figura 1-14). De hecho, la descomposición funcional y las metodologías orientadas a objetos, que se describen en el capítulo 4, se basan en el principio de "divide y vencerás".

Método de bloques de construcción

Otra forma de enfrentar un problema grande es ver si hay soluciones para las piezas más pequeñas del problema. Sería posible juntar algunas de estas soluciones para resolver la mayor parte del problema grande. Esta estrategia es sólo una combinación de los métodos "buscar cosas familiares"

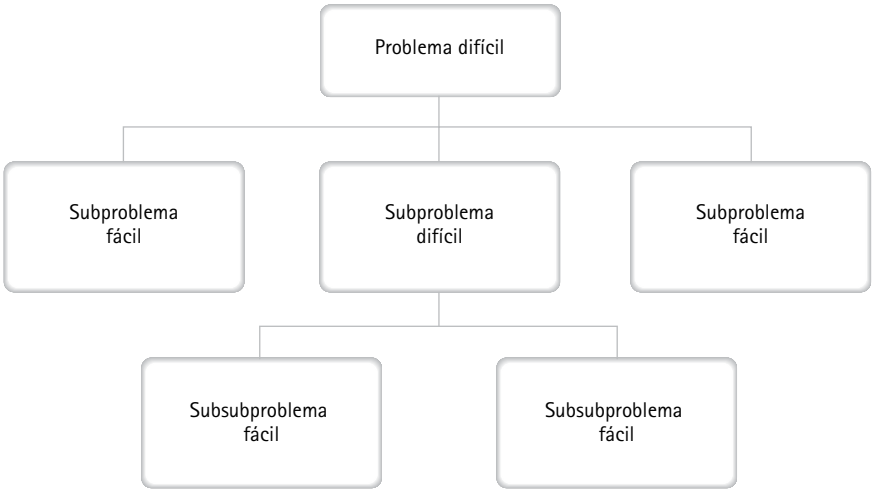


Figura 1-14 *Divide y vencerás*

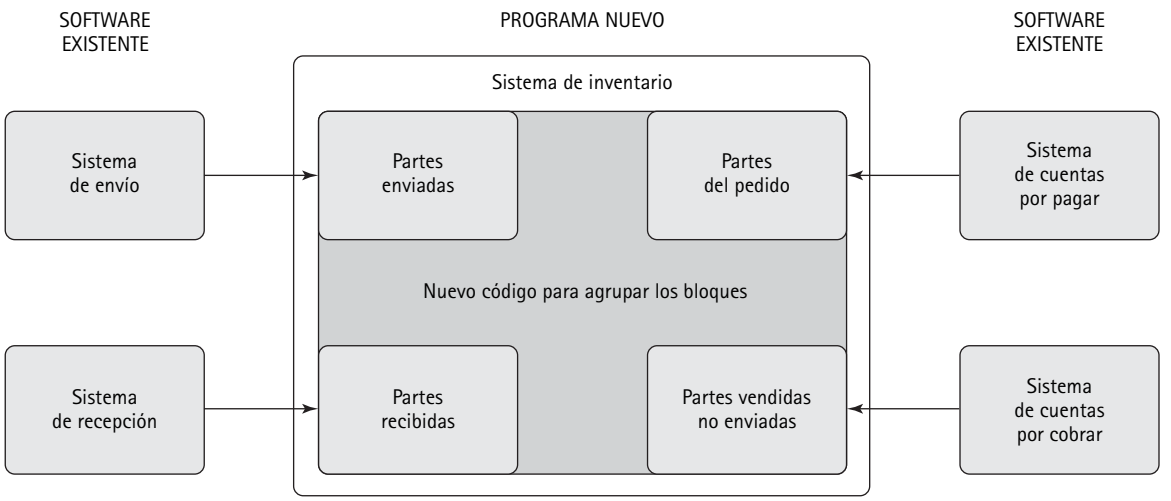


Figura 1-15 *Método de bloques de construcción*

y "divide y vencerás". Examine el problema grande y vea que se puede dividir en problemas más pequeños para los cuales ya existen soluciones. Resolver el problema grande es sólo una manera de agrupar las soluciones existentes, como pegar bloques o ladrillos para construir una pared (véase la figura 1-15).

Combinar soluciones

Otra forma para combinar las soluciones existentes es agruparlas en una base paso a paso. Por ejemplo, para calcular el promedio de una lista de valores, se debe sumar y contar los valores. Si ya se tienen soluciones separadas para sumar valores y para contarlos, es posible combinarlos. Pero si primero se hace la suma y luego se cuenta, se tiene que leer la lista dos veces. Se pueden ahorrar pasos si se combinan estas dos soluciones: leer un valor y luego sumarlo al total corriente y sumar 1 a la cuenta antes de ir al siguiente valor. Siempre que las soluciones a los subproblemas dupliquen los pasos, piense en agruparlas en lugar de unirlas extremo con extremo.

Bloqueos mentales: el temor de empezar

Los escritores están muy familiarizados con la experiencia de empezar una página en blanco, sin saber dónde empezar. Los programadores tienen la misma dificultad cuando enfrentan por vez primera un gran problema. Examinan el problema y éste parece abrumador (véase la figura 1-16).

Recuerde que siempre tiene una forma de empezar a resolver cualquier problema: escríbalo en papel con sus propias palabras, de modo que lo entienda. Una vez que interprete el problema, se puede enfocar en cada una de las subpartes por separado en lugar de intentar enfrentarlo en su totalidad. Esto le ayuda a ver piezas del problema que parezcan familiares o que sean análogas a otros problemas que ha resuelto, e identifica áreas donde algo no está claro, donde necesita más información.

Cuando usted escribe un problema, tiende a agruparlo en piezas pequeñas entendibles, que pueden ser lugares naturales para dividir el problema: "divide y vencerás". Su descripción del problema puede reunir toda la información acerca de los datos y resultados en un lugar para referencia fácil. Después puede ver los lugares inicial y final necesarios para el análisis de medios y fines.

La mayoría de los bloqueos mentales se deben a que no se entiende el problma. Reescribir el problema con sus propias palabras es una buena forma de centrarse en las subpartes que lo componen, una a la vez, y entender lo que se requiere para una solución.

Figura 1–16 *Bloqueo mental*

Resolución algorítmica de problemas

Sugerir un procedimiento paso a paso para resolver un problema particular no es siempre un proceso predeterminado. De hecho, normalmente es un proceso de prueba y error que requiere varios intentos y refinamientos. Se prueba cada intento para ver si en realidad resuelve el problema. Si es así, muy bien. De lo contrario, se intenta de nuevo. Resolver cualquier problema no trivial requiere, por lo común, una combinación de técnicas ya descritas.

Recuerde que la computadora sólo puede hacer ciertas cosas (véase la p. 11). Su interés principal, entonces, es cómo hacer que la computadora transforme, manipule, calcule o procese los datos de entrada para producir el resultado deseado. Si tiene en mente las instrucciones permisibles de su lenguaje de programación, no diseñará un algoritmo que sea difícil o imposible de codificar.

En el caso práctico que sigue, se desarrolla un programa para determinar si un año es bisiesto. Tipifica el proceso de pensamiento requerido para escribir un algoritmo y codificarlo como un programa, y muestra a lo que se parece un programa completo de C++.

Caso práctico de resolución de problemas

Algoritmo del año bisiesto

PROBLEMA Necesita escribir un conjunto de instrucciones que puedan emplearse para determinar si un año es bisiesto. Las instrucciones deben ser muy claras porque las utilizarán alumnos de cuarto grado que han aprendido la multiplicación y la división. Planean usar las instrucciones como parte de una tarea para determinar si alguno de sus parientes nació en año bisiesto. Para comprobar que el algoritmo funciona correctamente, lo codificará como un programa C++ y lo probará.

DISCUSIÓN La regla para determinar si un año es bisiesto es que un año debe ser divisible entre cuatro, pero no un múltiplo de 100. Cuando el año es un múltiplo de 400, es año bisiesto de cualquier modo. Se necesita escribir este conjunto de reglas como una serie de pasos (un algoritmo) que los alumnos de cuarto grado puedan seguir con facilidad.

Primero, se descompone en pasos principales por medio de "divide y vencerás". Hay tres pasos obvios en casi cualquier problema de este tipo:

1. Obtener los datos.
2. Calcular los resultados.
3. Producir los resultados.

¿Qué significa "obtener los datos"? Por *obtener*, se entiende *leer* o *introducir* los datos. Se necesita una pieza de los datos: un año de cuatro dígitos. Para que el usuario sepa cuándo introducir el valor, se debe pedir a la computadora que produzca un mensaje que indique cuándo está lista para aceptar el valor (esto se llama *mensaje indicador* o *indicador*). Por tanto, para que la computadora obtenga los datos, se tienen que efectuar estos dos pasos:

```
Indicar al usuario que introduzca un año de cuatro dígitos
    Leer el año
```

A continuación, se comprueba si el año que se leyó puede ser bisiesto (divisible entre cuatro), y luego probar si es uno de los casos excepcionales. Así que el algoritmo de alto nivel es:

Principal

```
Si el año no es divisible entre 4,
    entonces el año no es bisiesto
    en caso contrario, comprobar las excepciones
```

Es evidente que se requiere desarrollar estos pasos con instrucciones más detalladas, porque ninguno de los alumnos de cuarto grado sabe lo que significa "divisible". Se emplea el análisis de medios y fines para resolver el problema de cómo determinar si algo es divisible entre cuatro. Los alumnos de cuarto grado saben cómo hacer la división simple que dé como resultado un cociente y un residuo. Así, se les puede indicar que dividan el año entre cuatro y si el residuo es cero, entonces es divisible entre cuatro. De este modo, la primera línea se amplía en lo siguiente:

Principal revisado

```
Divida el año entre 4
Si el residuo no es 0,
    entonces el año no es bisiesto,
    de lo contrario, compruebe las excepciones
```

La comprobación de las excepciones cuando el año es divisible entre cuatro puede tener dos partes: comprobar si el año es divisible entre 100 y si es divisible entre 400. Dado como se hizo el primer paso, esto es fácil:

Comprobar las excepciones

```
Divida el año entre 100
Si el residuo es 0,
    entonces el año no es bisiesto
Divida el año entre 400
Si el residuo es 0,
    entonces el año es bisiesto
```

Estos pasos son confusos por sí mismos. Cuando el año es divisible entre 400, también es divisible entre 100, así que se tiene una prueba que dice que es un año bisiesto y una que dice que no. Lo que se necesita hacer es tratar los pasos como bloques de construcción y combinarlos. Una de las operaciones que se puede usar en tales situaciones

es comprobar también cuándo *no* existe una condición. Por ejemplo, si el año es divisible entre 4 pero no entre 100, entonces debe ser un año bisiesto. Si es divisible entre 100 pero no entre 400, entonces en definitiva no es un año bisiesto. Así, el tercer paso (comprobar las excepciones cuando el año es divisible entre 4) se amplía a las tres pruebas siguientes:

Comprobar las excepciones revisadas

```
Si el año no es divisible entre 100,
    entonces es un año bisiesto
Si el año es divisible entre 100, pero no entre 400,
    entonces no es un año bisiesto
Si el año es divisible entre 400,
    entonces es un año bisiesto
```

Se puede simplificar la segunda prueba porque la primera parte de ésta es sólo lo contrario de la primera; se puede decir simplemente "de otro modo" en lugar de repetir la prueba para divisibilidad entre 100. De manera similar, la última prueba es sólo el caso "de otro modo" de la segunda prueba. Una vez que se traduce cada una de estas pruebas simplificadas en pasos que los alumnos de cuarto grado saben cómo efectuar, se puede escribir el subalgoritmo que da verdadero si el año es bisiesto y falso en caso contrario. Llámese a este subalgoritmo Esañobisiesto, y ponga el año por probar entre paréntesis junto al nombre.

Esañobisiesto(año)

```
Divide el año entre 4
Si el residuo no es 0,
    Devuelve falso (el año no es bisiesto)
    De otro modo, divide el año entre 100 y
Si el residuo no es 0,
    Devuelve verdadero (el año es bisiesto)
    De otro modo, divide el año entre 400 y
Si el residuo no es 0
    Devuelve falso (el año no es bisiesto)
    De otro modo, devuelve verdadero (el año es bisiesto)
```

Lo único que falta es escribir los resultados. Si Esañobisiesto devuelve verdadero, entonces se escribe que el año es bisiesto; de otro modo, se escribe que no es bisiesto. Ahora se puede escribir el algoritmo completo para este problema.

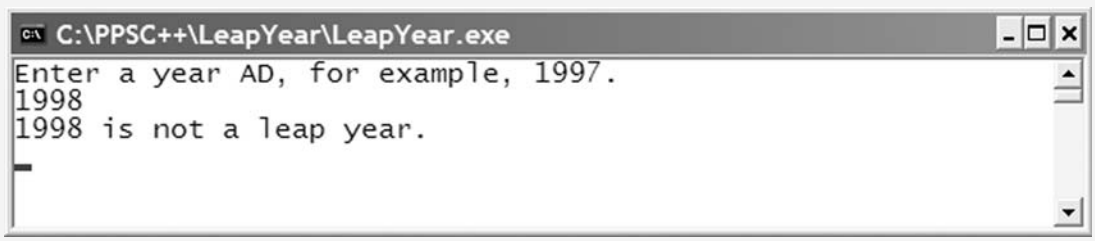
Algoritmo principal

```
Solicitar al usuario que introduzca un año de cuatro dígitos
Leer el año
Si Esañobisiesto(año)
    Escribir que el año es bisiesto
De lo contrario
    Escribir que el año no es bisiesto
```

Este algoritmo no sólo es claro, conciso y fácil de seguir, sino que una vez que haya terminado algunos de los siguientes capítulos, verá que es fácil pasar a un programa C++. Aquí se presenta el programa para que pueda compararlo con el algoritmo. No necesita saber cómo leer programas C++ para empezar a ver las similitudes. Note que el símbolo % es lo que C++ usa para calcular el residuo, y cualquier cosa que aparezca entre /* y */ o // y el fin de la línea es un comentario que es ignorado por el compilador.

```
//**************************************************************************
// Programa del año bisiesto
// Este programa ingresa un año e imprime el año
// es año bisiesto o no
//**************************************************************************

#include <iostream>           // Acceso de flujo de salida

using namespace std;

bool IsLeapYear( int );       // Prototipo para el subalgoritmo

int main()
{
    int year;                 // Año a ser probado
    cout << "Introduzca un año por ejemplo, 1997."
         << endl;             // Solicita la entrada
    cin >> year;              // Lee año

    if (IsLeapYear(year))     // Prueba para el año bisiesto
        cout << year << " es un año bisiesto."  << endl;
    else
        cout << year << " no es un año bisiesto."  << endl;

    return 0;                 // Indica que se completó satisfactoriamente
}

//**************************************************************************

bool IsLeapYear( int year )

// IsLeapYear regresa verdadero si year es un año bisiesto y
// falso en cualquier otro caso.

{
  if (year % 4 != 0)          // ¿Year no es divisible entre 4?
    return false;             // Si es así, no puede ser bisiesto
  else if (year % 100 !=  0)  // ¿Year no es múltiplo de 100?
    return true;              // Si es así, es año bisiesto
  else if (year % 400 != 0)   // ¿Year no es múltiplo de 400?
    return false;             // Si es así, entonces no es año bisiesto
  else
    return true;              // Es un año bisiesto
}
```

A continuación se presenta una imagen de pantalla de la entrada y la salida.



Resumen

Esto no tiene que ver de ninguna manera con prender la televisión y sentarse a verla. Es una herramienta de comunicación que se usa para mejorar nuestras vidas. Las computadoras se están haciendo tan comunes como las televisiones, una parte habitual de nuestras vidas. Y como las televisiones, las computadoras se basan en principios complejos pero están diseñadas para uso fácil.

Las computadoras son tontas, se les debe decir qué hacer. Un error de computadora verdadero es extremadamente raro (por lo común debido al mal funcionamiento de un componente o a una falla eléctrica). Debido a que se le dice a la computadora qué hacer, la mayoría de los errores en el resultado generado por computadora son en realidad errores humanos.

La programación de computadoras es el proceso de planificar una secuencia de pasos para que los siga una computadora. Requiere una fase de resolución de problemas y una fase de implementación. Después de analizar un problema, se desarrolla y prueba una solución general (algoritmo). Esta solución general se convierte en una solución concreta —el programa— cuado se escribe en un lenguaje de programación de alto nivel. La secuencia de instrucciones que conforma el programa se compila entonces en código de máquina, el lenguaje que usa la computadora. Después de corregir cualquier error que se presente durante la prueba, el programa es fácil de usar.

Una vez que se comienza a usar el programa, entra en la etapa de mantenimiento. El mantenimiento implica corregir errores descubiertos mientras se está usando el programa y modificarlo para reflejar los cambios en los requerimientos del usuario.

Los datos e instrucciones se representan como números binarios (números que constan de "unos" y "ceros") en las computadoras electrónicas. El proceso de convertir datos e instrucciones en una forma utilizable por la computadora se llama codificación.

Un lenguaje de programación refleja la variedad de operaciones que puede efectuar una computadora. Las estructuras de control básicas en un lenguaje de programación: secuencia, selección, ciclo y subprograma, se basan en estas operaciones fundamentales. En este texto se aprenderá a escribir programas en el lenguaje de programación de alto nivel C++.

Las computadoras se componen de seis partes básicas: unidad de memoria, unidad aritmética/lógica, unidad de control, dispositivos de entrada y salida, y dispositivos de almacenamiento auxiliares. La unidad aritmética/lógica y la unidad de control juntas se denominan unidad central de procesamiento. Las partes físicas de la computadora se llaman hardware. Los programas que ejecuta la computadora se llaman software.

El software del sistema es un conjunto de programas diseñados para simplificar la interfaz usuario/computadora. Incluye compilador, sistema operativo y editor.

Los profesionales de la computación se guían por un conjunto de principios éticos, como los miembros de otras profesiones. Entres las responsabilidades que se tienen están la de copiar software sólo con autorización, incluir la atribución a otros programadores cuando se hace uso de su código, proteger la privacidad de datos confidenciales, usar los recursos de computadora sólo con autorización y diseñar de manera cuidadosa los programas para que funcione correctamente.

Se ha dicho que la resolución de problemas es una parte integral del proceso de programación. Aunque es posible que se tenga poca experiencia en programar computadoras, se debe tener mucha experiencia en resolver problemas. La clave es detenerse y pensar en las estrategias que emplea para resolver problemas, y luego usar las estrategias para diseñar algoritmos prácticos. Entre esas estrategias está hacer preguntas, buscar cosas que sean familiares, resolver por analogía, aplicar el análisis de medios y fines, dividir el problema en subproblemas, usar soluciones existentes a pequeños problemas para resolver uno más grande, combinar soluciones y parafrasear el problema a fin de superar un bloqueo mental.

La computadora se usa ampliamente hoy día en ciencia, ingeniería, negocios, gobierno, medicina, artículos de consumo y las artes. Aprender a programar en C++ puede ayudar a usar esta poderosa herramienta de manera efectiva.

Comprobación rápida

La comprobación rápida está hecha para ayudarlo a corroborar si ha logrado los objetivos establecidos al comienzo de cada capítulo. Si entiende el material de cada capítulo, la respuesta a cada

pregunta debe ser bastante obvia. Después de leer una pregunta, compruebe su respuesta contra las respuestas listadas al final de la comprobación rápida. Si no conoce una respuesta o no entiende la respuesta dada, vaya a la página o páginas listadas al final de la pregunta para revisar el material.

1. ¿Qué nombre recibe la secuencia de instrucciones que es ejecutada por una computadora? (p. 3)
2. ¿Cómo difiere un algoritmo de un programa de computadora? (p. 4)
3. ¿En qué difiere un lenguaje de programación de un lenguaje humano? (p. 5)
4. ¿Qué es la entrada y salida de un compilador? (pp. 8-10)
5. ¿Quién inventó el lenguaje de programación C++? (p. 19)
6. ¿Qué nombre recibe la combinación de la unidad de control y la unidad aritmética/lógica? (pp. 12 y 13)
7. ¿De qué modo se puede ayudar a mantener la privacidad de los datos confidenciales? (p. 21)
8. ¿Cuáles son las tres fases del ciclo de vida del software? (p. 3)
9. ¿Cuál de las siguientes herramientas traduce un programa C++ en lenguaje de máquina: editor, sistema operativo, compilador o ensamblador? (pp. 8 y 9)
10. ¿Cuál es el término general que se emplea para hacer referencia a los componentes físicos de la computadora? (p. 14)
11. ¿Cuándo usaría el método de bloques de construcción para resolver un problema? (pp. 25 y 26)

Respuestas

1. Un programa de computadora. **2.** Un algoritmo se puede escribir en cualquier lenguaje, que llevará a cabo una persona o un procesador de cualquier clase. Un programa se escribe en un lenguaje de programación para que lo ejecute una computadora. **3.** Un lenguaje de programación tiene un vocabulario muy pequeño de palabras y símbolos, y un conjunto muy preciso de reglas que especifican la forma de términos de lenguaje válidos (sintaxis). **4.** El compilador introduce un programa fuente escrito en un lenguaje de alto nivel y produce un programa equivalente en lenguaje de máquina. Algunos compiladores producen también un listado, que es una copia del programa fuente con mensajes de error insertados. **5.** Bjarne Stroustrup en Bell Laboratories. **6.** Unidad central de procesamiento. **7.** Usar contraseñas que sean difíciles de adivinar y cambiarlas periódicamente. Encriptar datos almacenados. Asegurar que los medios de almacenamiento de datos se mantengan en un área segura. **8.** Resolución de problemas, implementación, mantenimiento. **9.** Compilador. **10.** Hardware. **11.** Cuando ve que un problema se puede dividir en piezas que pueden corresponder a subproblemas para los que ya se conocen soluciones.

Ejercicios de preparación para examen

1. Haga corresponder los siguientes términos con sus definiciones, dadas a continuación
   - ***a***) Programación.
   - ***b***) Computadora.
   - ***c***) Algoritmo.
   - ***d***) Programa de computadora.
   - ***e***) Lenguaje de programación.
   - ***f***) Documentación.
   - ***g***) Información.
   - ***h***) Datos.

   - ***i***) Dispositivo programable que puede almacenar, recuperar y procesar datos.
   - ***ii***) Información en forma que una computadora puede usar.
   - ***iii***) Secuencia de instrucciones que realizará una computadora.
   - ***iv***) Un conjunto de reglas, símbolos y palabras especiales empleadas para construir un programa de computadora.
   - ***v***) Planificar o programar el desempeño de una tarea o suceso.
   - ***vi***) Cualquier conocimiento que pueda ser transmitido.

vii) Texto escrito y comentarios que hacen un programa más fácil para que otros lo entiendan, usen o modifiquen.
viii) Procedimiento paso a paso para resolver un problema en una cantidad de tiempo finita.

2. Liste los tres pasos de la fase de resolución de problemas del ciclo de vida del software.
3. Liste los pasos de la fase de implementación del ciclo de vida del software.
4. Si en la prueba se descubre un error, ¿a qué paso del ciclo de vida del software regresa el programador?
5. Explique por qué la siguiente serie de pasos no es un algoritmo, y luego reescriba los pasos para hacer un algoritmo válido:

   Despertar.
   Ir a la escuela.
   Llegar a casa.
   Ir a dormir.
   Repetir desde el primer paso.

6. Haga corresponder los siguientes términos con sus definiciones, dadas a continuación.
   a) Lenguaje de máquina.
   b) Lenguaje ensamblador.
   c) Ensamblador.
   d) Compilador.
   e) Programa fuente.
   f) Programa objeto.

   i) Programa que traduce un lenguaje de alto nivel en código de máquina.
   ii) Lenguaje de programación de bajo nivel en que se emplean ayudas nemónicas para representar cada una de las instrucciones para una computadora particular.
   iii) Versión en lenguaje de máquina de un programa fuente.
   iv) Programa que traduce un programa en lenguaje ensamblador en código de máquina.
   v) Lenguaje, hecho de instrucciones en código binario, que la computadora emplea de modo directo.
   vi) Programa escrito en lenguaje de programación de alto nivel.
7. ¿Cuál es la ventaja de escribir un programa en un lenguaje de programación estandarizado?
8. ¿Qué hace la unidad de control?
9. El editor es un dispositivo periférico. ¿Cierto o falso?
10. La memoria RAM es un dispositivo periférico. ¿Cierto o falso?
11. ¿Es un caso de piratería de software si usted y un amigo compran una pieza de software y la instalan en sus computadoras? La licencia para el software dice que puede ser registrada para un solo usuario.
12. Establezca una correspondencia entre las estrategias de resolución de problemas y las descripciones siguientes.
    a) Hacer preguntas.
    b) Buscar cosas que sean familiares.
    c) Resolver por analogía.
    d) Análisis de medios y fines.
    e) Divide y vencerás.
    f) Método de bloques de construcción.
    g) Combinar o agrupar soluciones.

    i) Descomponer el problema en piezas más manejables.
    ii) Reunir más información para ayudar a descubrir una solución.
    iii) Identificar aspectos del problema que son similares a un problema en un dominio diferente.
    iv) Combinar los pasos de dos o más algoritmos diferentes.
    v) Identificar aspectos del problema que usted ha resuelto antes.

vi) Reunir las soluciones que existen para el problema.
vii) Examinar la entrada, salida y las operaciones disponibles y hallar una secuencia de operaciones que transforman la entrada hacia la salida.

Ejercicios de preparación para la programación

1. En el siguiente algoritmo para hacer fotografías en blanco y negro, identifique los pasos que son bifurcaciones (selección), ciclos o referencias a subalgoritmos definidos en alguna otra parte.
   a) Mezclar el revelador según las instrucciones del paquete.
   b) Vaciar el revelador en una charola.
   c) Mezclar el baño de detención de acuerdo con las instrucciones del paquete.
   d) Vaciar el baño de detención en la charola.
   e) Mezclar el fijador según las instrucciones del paquete.
   f) Vaciar el fijador en la charola.
   g) Apagar la luz blanca y encender la luz de seguridad.
   h) Colocar el negativo en el amplificador y encender.
   i) Ajustar el tamaño de la imagen y enfocar, y luego apagar el amplificador.
   j) Retirar una pieza de papel de impresión del seguro y colocar en el atril de amplificación.
   k) Encender el amplificador durante treinta segundos y después apagar.
   l) Colocar el papel en el revelador durante 1 minuto, en el baño de detención por 30 segundos y en el fijador por 1 minuto.
   m) Encender las luces blancas e inspeccionar la primera impresión, y luego apagar las luces blancas.
   *n*1) Si la primera impresión es demasiado clara:
   Retire una pieza de papel del seguro, colóquela en el atril y exponga durante 60 segundos para crear una impresión muy oscura. Luego coloque en el revelador durante 1 minuto, en el baño de detención durante 30 segundos y en el fijador durante 1 minuto.
   *n*2) Si la primera impresión es demasiado oscura:
   Retire una pieza de papel del seguro, colóquela en el atril y exponga durante 60 segundos para crear una impresión muy clara. Luego coloque en el revelador durante 1 minuto, en el baño de detención durante 30 segundos y en el fijador durante 1 minuto.
   *n*3) Si la primera impresión es más o menos correcta:
   Retire una pieza de papel del seguro, colóquela en el atril y exponga durante 60 segundos para crear una impresión muy oscura. Después coloque en el revelador durante 1 minuto, en el baño de detención durante 30 segundos y en el fijador durante 1 minuto. Saque una pieza de papel del seguro, colóquela en el atril y exponga durante 15 segundos para crear una impresión muy clara. Luego coloque en el revelador durante 1 minuto, en el baño de detención durante 30 segundos y en el fijador durante 1 minuto.
   o) Analice las impresiones demasiado clara y demasiado oscura para estimar el tiempo de exposición base, y luego identifique los toques de luz y las sombras que requieren menos o más exposición para estimar el tiempo necesario para cada área.
   p) Retire una pieza de papel del seguro, coloque en el atril y exponga durante el tiempo de exposición base, cubriendo las áreas sombreadas por el tiempo estimado, y luego cubra toda la impresión excepto las áreas claras, que se dejan en exposición más tiempo como se estimó. Luego coloque la impresión en el revelador durante 1 minuto, en el baño de detención durante 30 segundos y en el fijador durante 1 minuto.
   q) Analice la impresión desde el paso `p` y ajuste las estimaciones de tiempos según sea apropiado.
   r) Repita los pasos `p` y `q` hasta que se obtenga una impresión con la exposición deseada.
   s) Documente los tiempos de exposición que dan como resultado la impresión deseada.
   t) Retire una pieza de papel del seguro, colóquela en el atril y exponga de acuerdo con la información del paso `s`. Luego coloque en el revelador durante 1 minuto, en el baño de detención durante 30 segundos y en el fijador durante 4 minutos. Coloque la impresión en el lavador.
   u) Repita el paso t para crear las impresiones necesarias.

v) Lave las impresiones durante 1 hora.
w) Coloque las impresiones en el secador.

2. Escriba un algoritmo para el cepillado de los dientes. Las instrucciones deben ser muy simples y exactas porque la persona que las seguirá nunca lo ha hecho antes y todo lo entiende de manera literal.
3. Identifique los pasos en su solución del ejercicio 2 que son bifurcaciones, ciclos y referencias a subalgoritmos definidos en cualquier otro lado.
4. Cambie el algoritmo del ejercicio 1 de modo que se creen 10 impresiones expuestas apropiadamente cada vez que se ejecute.

Seguimiento de caso práctico

1. Use el algoritmo del caso práctico del año bisiesto para decidir si los siguientes años son bisiestos.
   a) 1900
   b) 2000
   c) 1996
   d) 1998
2. Dado el algoritmo del caso práctico del año bisiesto con las líneas numeradas como sigue:

```
1. Dividir el año entre 4 y si el residuo no es 0
2.     Devolver falso (el año no es bisiesto)
3.     De otro modo, dividir el año entre 100 y si el residuo no es 0,
4.         Devolver verdadero (el año es bisiesto)
5.         De otro modo, dividir el año entre 400 y si el residuo no es 0,
6.             Devolver falso (el año no es bisiesto)
7.             De otro modo, devolver verdadero (el año es bisiesto)
```

   Mencione cuál línea del algoritmo indica si la fecha es un año bisiesto en cada uno de los siguientes casos.

   a) 1900 Línea =
   b) 1945 Línea =
   c) 1600 Línea =
   d) 1492 Línea =
   e) 1776 Línea =

3. ¿Cómo ampliaría el algoritmo del año bisiesto para que le indique cuándo un año es un año milenio (un múltiplo de 1 000)?
4. Use el algoritmo del año bisiesto para determinar si usted nació en un año bisiesto.
5. Amplíe el algoritmo del año bisiesto de modo que le indique cuándo será el siguiente año bisiesto, si el año de entrada no es bisiesto.
6. Compare el algoritmo para determinar el año bisiesto con el programa C++ que se muestra en el caso práctico. Utilice el esquema de numeración para el algoritmo de la pregunta 2 para decidir qué línea (o líneas) del algoritmo corresponden a qué línea (o líneas) del programa mostrado aquí.

Número de línea

```
          {
_________   if (year % 4 != 0)
_________     return false;
_________   else if (year % 100 !=  0)
_________     return true;
_________   else if (year % 400 != 0)
_________     return false;
_________   else return true;
          }
```

CAPÍTULO

2

Sintaxis y semántica de C++, y el proceso de desarrollo de programa

Objetivos

Objetivos de conocimiento

- *Entender cómo se compone un programa en C++ de uno o más subprogramas (funciones).*
- *Conocer qué es un metalenguaje y cómo se emplea.*
- *Entender el concepto de un tipo de dato.*
- *Aprender los pasos requeridos para introducir y ejecutar un programa.*

Objetivos de habilidades

Ser capaz de:

- *Leer plantillas de sintaxis a fin de entender las reglas formales que gobiernan a los programa en C++.*
- *Crear y reconocer identificadores legales en C++.*
- *Declarar constantes nombradas y variables de tipo* `char` *y* `string`.
- *Distinguir palabras reservadas en C++ de identificadores definidos por el usuario.*
- *Asignar valores a variables.*
- *Elaborar expresiones de cadena simple constituidas por constantes, variables y el operador de concatenación.*
- *Construir una sentencia que se escribe para un flujo de salida.*
- *Determinar lo que se imprime mediante una sentencia de salida.*
- *Usar comentarios para aclarar los programas.*
- *Crear programas simples en lenguaje C++.*

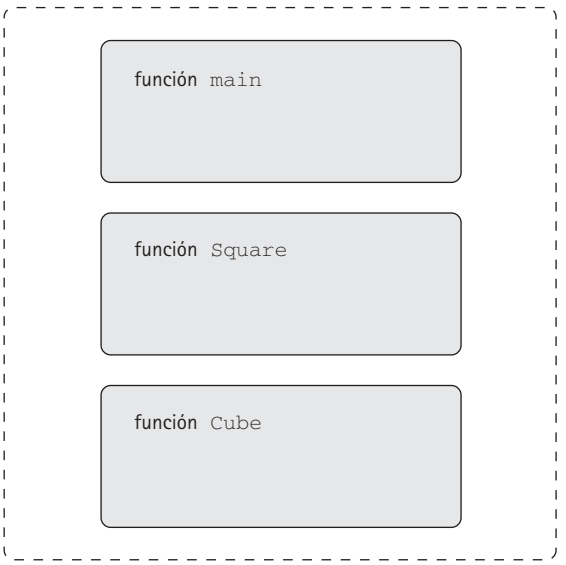
2.1 Elementos de programas C++

Los programadores desarrollan soluciones para problemas por medio de un lenguaje de programación. En este capítulo se empieza examinando las reglas y símbolos que conforman el lenguaje de programación C++. Asimismo, se revisan los pasos requeridos para crear un programa y hacer que funcione en una computadora.

Estructura de un programa C++

En el capítulo 1 se habló acerca de las cuatro estructuras básicas para expresar acciones en un lenguaje de programación: secuencia, selección, ciclo y subprograma. Se dice que los subprogramas permiten escribir por separado partes del programa y luego ensamblarlas en una forma final. En C++, los subprogramas se denominan **funciones**, y un programa C++ es una colección de una o más funciones. Cada función realiza alguna función particular y, de manera colectiva, todas cooperan para resolver un problema completo.

Función Subprograma en C++.



Cada programa C++ debe tener una función llamada `main`. La ejecución del programa comienza siempre con la función `main`. Se puede pensar en `main` como el maestro, y las otras funciones como los sirvientes. Cuando `main` quiere que la función `Square` realice una tarea, `main` llama (o *invoca*) a `Square`. Cuando la función `Square` completa la ejecución de sus sentencias, de manera obediente devuelve el control al maestro, `main`, para que éste continúe la ejecución.

Considérese un ejemplo de un programa C++ con tres funciones principales: `main`, `Square` y `Cube`. No se preocupe por los detalles del programa, sólo observe su apariencia y estructura globales.

```
#include <iostream>


using namespace std;

int Square( int );
int Cube( int );
```

```
int main()
{
    cout << "El cuadrado de 27 es" << Square(27) << endl;
    cout << "y el cubo de 27 es " << Cube(27) << endl;
    return 0;
}

int Square( int n )

{
    return n * n;
}

int Cube( int n )

{
    return n * n * n;
}
```

En cada una de las tres funciones, la llave izquierda ({) y la llave derecha (}) marcan el comienzo y el final de las sentencias que serán ejecutadas. Las sentencias que aparecen entre llaves se conocen como *cuerpo* de la función.

La ejecución de un programa siempre comienza con la primera sentencia de la función `main`. En el programa, la primera declaración es

```
cout << "El cuadrado de 27 es " << Square (27) << end1;
```

Ésta es una sentencia de salida que causa que se imprima la información en la pantalla de la computadora. Se aprenderá cómo crear sentencias como ésta más adelante en este capítulo. De manera breve, esta sentencia imprime dos elementos o ítems. El primero es el mensaje

```
El cuadrado de 27 es
```

El segundo elemento que se imprimirá es el valor obtenido al llamar (invocar) a la función `Square`, con el valor 27 como el número que será elevado al cuadrado. Como sirviente, la función `Square` realiza la tarea de elevar al cuadrado el número y enviar de nuevo el resultado calculado (729) a su *invocador*, la función `main`. Ahora `main` puede continuar la ejecución al imprimir el valor 729 y proceder a su siguiente sentencia.

De manera similar, la segunda sentencia en `main` imprime el mensaje

```
y el cubo de 27 es
```

y luego invoca la función `cube` e imprime el resultado 19683. Por tanto, el resultado completo al ejecutar este programa es

```
El cuadrado de 27 es 729
y el cubo de 27 es 19683
```

Tanto `Square` como `Cube` son ejemplos de *funciones que devuelven un valor*. Una función de devolución de valor devuelve un solo valor a su invocador. La palabra `int` al inicio de la primera línea de la función `Square`

```
int Square( int n )
```

expresa que la función devuelve un valor entero.

Ahora examine de nuevo la función `main`. Se verá que la primera línea de la función es

```
int main()
```

La palabra `int` indica que `main` es una función de devolución de valor que debe devolver un valor entero. Y eso es lo que hace. Después de imprimir el cuadrado y el cubo de 27, `main` ejecuta la sentencia

```
return 0;
```

para devolver el valor 0 a su invocador. Pero, ¿quién llama a la función `main`? La respuesta es: el sistema operativo de la computadora.

Cuando se trabaja con programas C++, el sistema operativo es considerado el invocador de la función `main`. El sistema operativo espera que `main` devuelva un valor cuando `main` termina la ejecución. Por convención, un valor de retorno 0 significa que todo estuvo correcto. Un valor de retorno que no sea cero (por lo común, 1, 2, ...) significa que algo estuvo mal. Más adelante en este libro se consideran situaciones en las que se podría desear que `main` devuelva un valor distinto de cero. Por el momento, se concluye que la ejecución de `main` devuelve el valor 0.

Se ha considerado sólo de manera breve la representación general de la apariencia de un programa C++, una colección de una o más funciones, incluida `main`. Se ha mencionado también lo que es especial acerca de la función `main` —es una función requerida, la ejecución comienza allí, y devuelve un valor al sistema operativo—. Ahora es tiempo de comenzar a ver los detalles del lenguaje C++.

Sintaxis y semántica

Un lenguaje de programación es un conjunto de reglas, símbolos y palabras especiales usadas para crear un programa. Hay reglas para la **sintaxis** (gramática) y la **semántica** (significado).

Sintaxis Reglas formales que gobiernan cómo se escriben instrucciones válidas en un lenguaje de programación.

Semántica Conjunto de reglas que determina el significado de las instrucciones escritas en un lenguaje de programación.

Metalenguaje Lenguaje que se emplea para escribir las reglas de sintaxis para otro lenguaje.

La sintaxis es un conjunto formal de reglas que define de manera exacta qué combinaciones de letras, números y símbolos se puede usar en un lenguaje de programación. No hay lugar para la ambigüedad en la sintaxis de un lenguaje de programación porque la computadora no puede pensar; no "sabe lo que se quiere dar a entender". Para evitar ambigüedad, las reglas de sintaxis deben ser escritas en un lenguaje muy simple, preciso y formal, llamado **metalenguaje**.

Aprender a leer un metalenguaje es como aprender a leer las notaciones usadas en las reglas de un deporte. Una vez que se entienden las notaciones, se puede aprender a leer el libro de reglas. Es cierto que muchas personas aprenden un deporte simplemente viendo jugar a otros, pero lo que aprenden es por lo general sólo lo suficiente para que tomen parte en juegos casuales. Se puede aprender C++ siguiendo los ejemplos de este libro, pero un programador serio, al igual que un atleta serio, debe darse tiempo para leer y entender las reglas.

Las reglas de sintaxis son los anteproyectos que se usan para establecer las instrucciones de un programa. Permiten tomar los elementos de un lenguaje de programación —los bloques de construcción básicos del lenguaje— y ensamblarlos en *construcciones*, estructuras correctas desde el punto de vista de la sintaxis. Si el programa viola cualquiera de las reglas del lenguaje (como al escribir mal una palabra crucial u omitir una coma importante), se dice que el programa tiene *errores de sintaxis* y no puede compilar de manera correcta hasta arreglarlos.

Bases teóricas

Metalenguajes

Metalenguaje es la palabra *lenguaje* con el prefijo *meta-*, que significa "más allá" o "más comprensivo". Un metalenguaje es un lenguaje que va más allá de un lenguaje normal al permitir hablar con precisión respecto a ese lenguaje. Es un lenguaje para hablar acerca de lenguajes. Uno de los metalenguajes más viejos orientados a las computadoras es *Backus-Naur Form* (*BNF*), en honor a John Backus y Peter Naur, quienes lo desarrollaron en 1960. Las definiciones de sintaxis BNF se escriben con letras, números y símbolos especiales. Por ejemplo, un *identificador* (un nombre para algo en un programa) en C++ debe ser por lo menos una letra o raya de subrayado (_), que puede ir seguida o no de letras adicionales, rayas de subrayado o dígitos. La definición BNF de un identificador en C++ es como sigue.

```
<Identifier> ::= <Nondigit> | <Nondigit> <NondigitOrDigitSequence>
<NondigitOrDigitSequence> ::= <NondigitOrDigit> | <NondigitOrDigit> <NondigitOrDigitSequence>
<NondigitOrDigit> ::= <Nondigit> | <Digit>
<Nondigit> ::= _ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
               a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
<Digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Donde el símbolo ::= significa "se define como", el símbolo | significa "o", los símbolos < y > se emplean para encerrar palabras denominadas *símbolos no terminales* (símbolos que aún necesitan ser definidos), y cualquier otra cosa se denomina *símbolo terminal*.

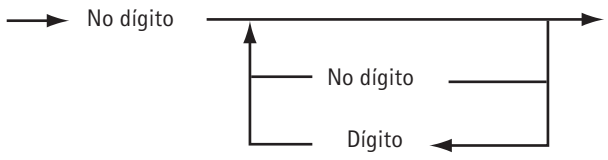
La primera línea de la definición se lee como sigue: "un identificador se define como un no dígito o un no dígito seguido por una secuencia de no dígito-o-dígito". Esta línea contiene símbolos no terminales que deben ser definidos. En la segunda línea, el símbolo no terminal NondigitOrDigitSequence se define como un NondigitOrDigit o un NondigitOrDigit seguido por otra NondigitOrDigitSequence. La autorreferencia en la definición es una manera indirecta de decir que una NondigitOrDigitSequence puede ser una serie de uno o más no dígitos o dígitos. La tercer línea define a NondigitOrDigit como un no dígito o dígito. En las líneas cuarta y la última, se encuentran los símbolos terminales, que definen a Nondigit como una raya de subrayado (guión bajo) o cualquier letra mayúscula o minúscula y a Digit como cualquiera de los símbolos numéricos 0 al 9.

BNF es un lenguaje muy simple, pero esa simplicidad da lugar a definiciones de sintaxis que pueden ser largas y difíciles de leer. Un metalenguaje alternativo, el diagrama de sintaxis, es más fácil de seguir. Éste utiliza flechas para indicar cómo se puede combinar símbolos. Los diagramas de sintaxis que definen a un identificador en C++ se muestran a continuación y en la página siguiente.

Para leer los diagramas, empiece a la izquierda y siga las flechas. Cuando llegue a una bifurcación, tome cualquiera de las trayectorias de bifurcación. Los símbolos en negrita son símbolos terminales, y las palabras que no están en negrita son símbolos no terminales.

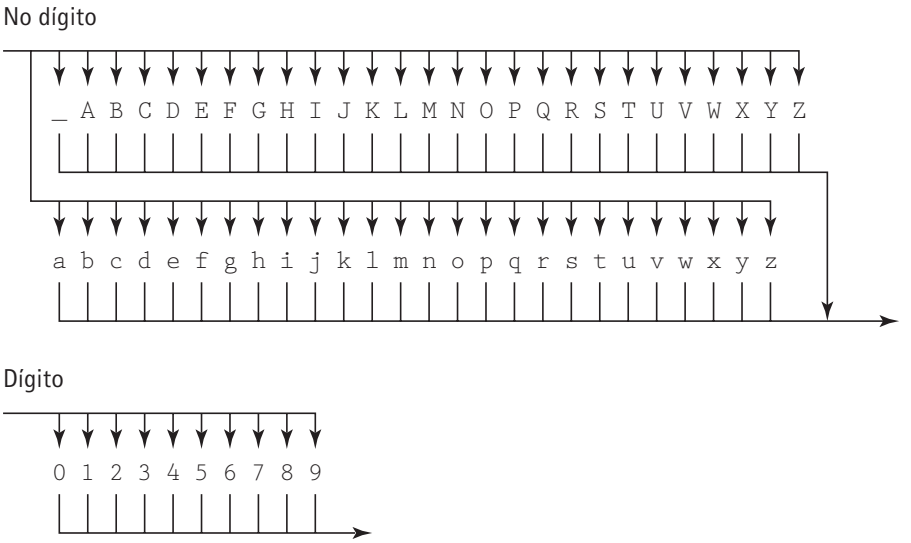
En el primer diagrama se muestra que un identificador consta de un no dígito seguido, opcionalmente, por cualquier cantidad de no dígitos o dígitos. En el segundo diagrama se define el símbolo no terminal Nondigit como un guión bajo o cualquiera de los caracteres del alfabeto. En el tercer diagrama se define a Digit como uno de los caracteres numéricos. Aquí, se han eliminado los símbolos no terminales BNF NondigitOrDigitSequence o NondigitOrDigit por medio de flechas en el primer diagrama de sintaxis para permitir una secuencia de no dígitos o dígitos consecutivos.

Identificador



(continúa)

Metalenguajes



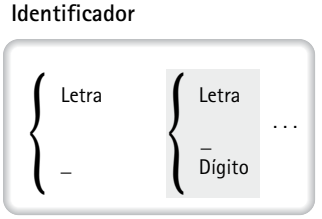
Los diagramas de sintaxis son más fáciles de interpretar que las definiciones BNF, pero aún pueden ser difíciles de leer. En este texto se introduce otro metalenguaje, llamado *plantilla de sintaxis*. Las plantillas de sintaxis muestran de un vistazo la forma que toma una construcción en C++.

Nota final: los metalenguajes sólo muestran cómo escribir construcciones que el compilador puede traducir. No definen lo que hacen esas instrucciones (su semántica). Existen lenguajes formales para definir la semántica de un lenguaje de programación, pero están fuera del alcance de este libro. En todo este libro se describe en español la semántica de C++.

Plantillas de sintaxis

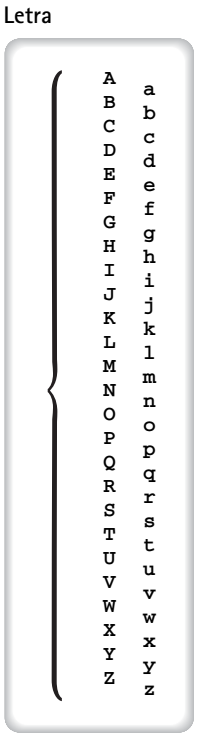
En este libro, se escriben las reglas de sintaxis para C++ con un metalenguaje llamado *plantilla de sintaxis*. Una plantilla de sintaxis es un ejemplo genérico de construcción en C++ que se define. Las convenciones gráficas muestran qué porciones son opcionales y cuáles pueden ser repetidas. Una palabra o símbolo en negrita es una palabra literal o símbolo en el lenguaje C++. Una palabra en negrita se puede remplazar por otro ejemplo. Se usa una llave para indicar una lista de ítems, de los que se puede elegir uno.

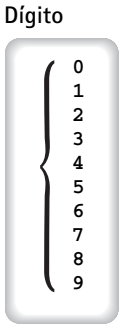
Veamos un ejemplo. Esta plantilla define a un identificador en C++:



La zona sombreada indica una parte de la definición que es opcional. Los tres puntos (...) significan que el símbolo precedente o bloque sombreado puede repetirse. Así, un identificador en C++ debe comenzar con una letra o guión bajo y va seguido opcionalmente por una o más letras, rayas de subrayado o dígitos.

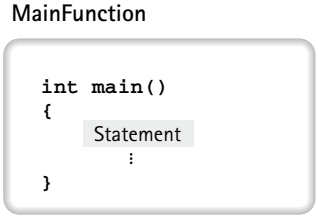
Recuerde que una palabra no escrita en negrita se puede remplazar con otra plantilla. Éstas son las plantillas para Letra y Dígito:





En estas plantillas, las llaves indican de nuevo listas de ítems, de los cuales cualquiera puede ser elegido. Así, una letra puede ser cualquiera de las letras mayúsculas o minúsculas, y un dígito puede ser cualquiera de los caracteres numéricos del 0 al 9.

Ahora se examinará la plantilla de sintaxis para la función `main` de C++:



La función `main` comienza con la palabra `int`, seguida de la palabra `main` y luego los paréntesis izquierdo y derecho. Esta primera línea de la función es el *encabezado*. Después del encabezado, la llave izquierda señala el comienzo de las sentencias en la función (su cuerpo). El área sombreada y los tres puntos indican que el cuerpo de la función consta de cero o más sentencias. (En este diagrama se han colocado verticalmente los tres puntos para sugerir que las sentencias están arregladas, por lo general, de modo vertical, una arriba de la siguiente.) Por último, la llave derecha indica el fin de la función.

En principio, la plantilla de sintaxis permite que el cuerpo de la función no tenga sentencias en absoluto. Sin embargo, en la práctica, el cuerpo debe incluir una sentencia Return porque la palabra `int` en el encabezado de la función expresa que `main` devuelve un valor entero. Así, el programa C++ más corto es

```
int main()
{
    return 0;
}
```

Como puede suponer, este programa ¡no hace nada absolutamente útil cuando se ejecuta!

A medida que se introducen las construcciones del lenguaje C++ en todo el libro, se usan plantillas de sintaxis para mostrar la sintaxis apropiada. En el sitio web de la editorial encontrará reunidas estas plantillas de sintaxis en una ubicación central.

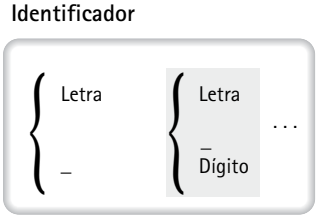
Cuando termine este capítulo, debe saber lo suficiente acerca de la sintaxis y semántica de las sentencias en C++ para escribir programas simples. Pero antes de poder hablar acerca de la escritura de sentencias, se debe examinar cómo se escriben los nombres en C++ y algunos de los elementos de un programa.

Cómo nombrar los elementos de programa: identificadores

Como se hizo notar en la explicación acerca de los metalenguajes, los **identificadores** se emplean en C++ para nombrar cosas, como subprogramas y localidades en la memoria de la computadora. Los identificadores están constituidos por letras (A-Z, a-z), dígitos (0-9) y el carácter de subrayado (_), pero deben comenzar con una letra o raya de subrayado.

Identificador Nombre asociado con una función u objeto de datos y empleado para referir a esa función u objeto de datos.

Recuerde que un identificador *debe* comenzar con una letra o raya de subrayado (guión bajo):



(Los identificadores que comienzan con un guión bajo tienen significados especiales en algunos sistemas C++, así que es mejor empezar un identificador con una letra.)

A continuación se dan algunos ejemplos de identificadores válidos:

```
sum_of_squares  J9  box_22A  GetData  Bin3D4  count
```

Y aquí se presentan algunos ejemplos de identificadores no válidos y las razones por las que no son válidos:

Identificador no válido	Explicación
`40Hours`	Los identificadores no pueden comenzar con un dígito.
`Get Data`	En los identificadores no se permiten espacios en blanco.
`box-22`	El guión (–) es un símbolo matemático (menos) en C++.
`cost_in_$`	No se permiten símbolos especiales como $.
`int`	La palabra `int` está predefinida (reservada) en el lenguaje C++.

El último identificador de la tabla, `int`, es un ejemplo de **palabra reservada**. Las palabras reservadas son palabras que tienen usos específicos en C++; no es posible usarlas como identificadores definidos por el programador. En el apéndice A se listan todas las palabras reservadas en C++.

Palabra reservada Palabra que tiene significado especial en C++; no puede ser usada como un identificador definido por el programador.

El programa LeapYear (año bisiesto) del capítulo 1 emplea los identificadores definidos por el programador de la siguiente lista. (La mayoría de los otros identificadores en el programa son palabras reservadas en C++.) Observe que se eligen los nombres para transmitir cómo se usan los identificadores.

Identificador	Cómo se usa
`year`	Los identificadores no pueden comenzar con un dígito.
`IsLeapYear`	Siempre que el año sea bisiesto

Cuestiones de estilo

Uso significativo, identificadores legibles

Los nombres que se emplean para hacer referencia a cosas en los programas carecen de sentido para la computadora. Ésta se comporta de la misma forma si al valor 3.14159265 se le llama `pi` o `cake`, siempre y cuando se refiera a lo mismo. Sin embargo, es mucho más fácil que alguien se imagine cómo funciona un programa si los nombres elegidos para los elementos dicen en realidad algo acerca de ellos. Siempre que tenga que inventar un nombre para algo en un programa, trate de elegir uno que sea significativo para una persona que lee el programa.

C++ es un lenguaje *sensible a mayúsculas y minúsculas*. Las letras mayúsculas son distintas de las minúsculas. Los identificadores

```
PRINTTOPPORTION  printtopportion  pRiNtToPpOrTiOn  PrintTopPortion
```

son cuatro nombres distintos y de ninguna manera son intercambiables. Como puede ver, la última de estas formas es la más fácil de leer. En este libro se usan combinaciones de letras mayúsculas, minúsculas y rayas de subrayado en los identificadores. Se explican las convenciones para elegir entre mayúsculas y minúsculas conforme se avanza en este capítulo.

Ahora que se ha visto cómo escribir identificadores, se examinan algunas de las cosas que C++ permite nombrar.

Datos y tipos de datos

Un programa de computadora opera sobre datos (almacenados internamente en la memoria, almacenados en medios externos como discos o cintas, o bien, introducidos desde un dispositivo como un teclado, escáner o sensor eléctrico) y produce un resultado. En C++, cada pieza de los datos debe ser de un **tipo de datos** específico. El tipo de datos determina cómo se repre-

Tipo de datos Conjunto específico de valores de datos, junto con un conjunto de operaciones en esos valores.

sentan los datos en la computadora y la clase de proceso que puede realizar la computadora con ellos.

Algunos tipos de datos se emplean con tanta frecuencia que C++ los define para nosotros. Ejemplos de estos *tipos estándar* (o *integrados*) son `int` (para trabajar con números enteros), `float` (para trabajar con números reales que tienen puntos decimales) y `char` (para trabajar con datos de caracteres).

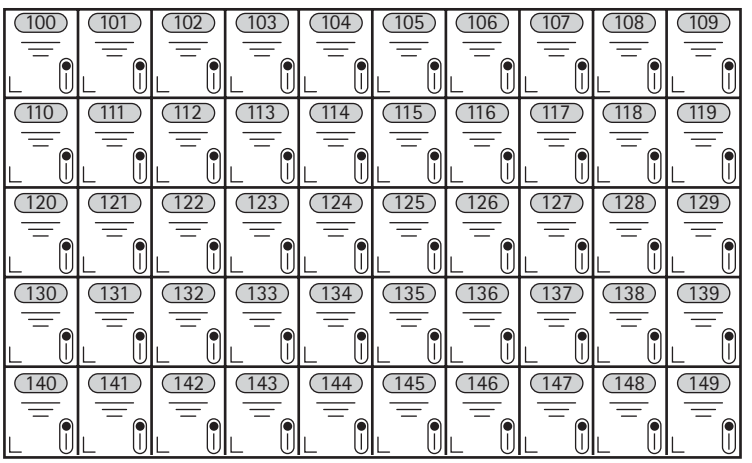
Adicionalmente, C++ permite a los programadores definir sus propios datos: *tipos definidos por el programador* (o *definidos por el usuario*). Al comienzo del capítulo 10 se le muestra cómo definir sus propios tipos de datos.

En este capítulo se centra la atención en dos tipos de datos —uno para representar datos que constan de un solo carácter, el otro para representar cadenas de caracteres—. En el siguiente capítulo se examinan en detalle los tipos numéricos (como `int` y `float`).

Información básica

Almacenamiento de datos

¿De dónde obtiene un programa los datos que necesita para operar? Los datos se almacenan en la memoria de la computadora. Recuerde que la memoria se divide en gran cantidad de localidades o celdas separadas, cada una de las cuales puede contener una pieza de datos. Cada ubicación de memoria tiene una dirección única a la que se hace referencia cuando se almacenan o recuperan datos. Se puede representar la memoria como un conjunto de buzones, con los números de caja como las direcciones usadas para designar lugares particulares.



Por supuesto, la dirección real de cada localidad en la memoria es un número binario en un código de lenguaje de máquina. En C++ se usan identificadores para nombrar localidades de memoria; el compilador los traduce entonces en binario para nosotros. Ésta es una de las ventajas de un lenguaje de programación de alto nivel: nos libera de tener que seguir la pista de direcciones numéricas de las localidades de memoria en los que están almacenados datos e instrucciones.

Tipo de datos `char` El tipo integrado `char` describe datos que constan de un carácter alfanumérico, una letra, dígito o símbolo especial:

```
'A'   'a'   '8'   '2'   '+'   '-'   '$'   '?'   '*'   ' '
```

Cada máquina usa un conjunto de *caracteres* particular, el conjunto de caracteres alfanuméricos que puede representar. (Véanse en el apéndice E algunos conjuntos de caracteres muestra.) Observe que cada carácter está encerrado en comillas simples (apóstrofos). El compilador C++ requiere los apóstrofos para diferenciar, por ejemplo, entre el dato de caracteres '8' y el valor entero 8 porque los dos se almacenan de manera diferente dentro de la máquina. Observe también que el espacio en blanco ' ', es un carácter válido.*

Usted no desearía añadir el carácter 'A' al carácter 'B' o restar el carácter '3' del carácter '8', sino quizá quiera comparar valores de caracteres. Cada conjunto de caracteres tiene una secuencia de clasificación, un orden predefinido de todos los caracteres. Aunque esta secuencia varía de un conjunto de caracteres a otro, 'A' siempre se compara menor que 'B', 'B' menor que 'C', y así sucesivamente. Y '1' se compara menor que '2', '2' menor que '3', etcétera. Ninguno de los identificadores del programa del año bisiesto es de tipo `char`.

Tipo de datos `string` Mientras que un valor de tipo `char` está limitado a un solo carácter, una *cadena* es una secuencia de caracteres, como una palabra, nombre o enunciado, encerrados entre comillas. Por ejemplo, las siguientes son cadenas en C++:

```
"Problem Solving"    "C++"    "Programming and "    "  .  "
```

Una cadena debe ser escrita por completo en una línea. Por ejemplo, la cadena

```
"Esta cadena no es válida porque está
escrita en más de una línea."
```

no es válida porque se divide en dos líneas. En este caso, el compilador C++ produce un mensaje de error en la primera línea. El mensaje podría decir algo como "UNTERMINATED STRING", lo cual depende del compilador particular.

Las comillas no se consideran parte de la cadena sino simplemente están para distinguir la cadena de otras partes de un programa C++. Por ejemplo, `"amount"` (entre comillas) es la cadena de caracteres constituida por las letras *a*, *m*, *o*, *u*, *n* y *t* en ese orden. Por otro lado, `amount` (sin comillas) es un identificador, quizás el nombre de una localidad en la memoria. Los símbolos `"12345"` representan una cadena conformada por los caracteres *1*, *2*, *3*, *4* y *5* en ese orden. Si se escribe `12345` sin las comillas, es una cantidad entera que se puede usar en cálculos.

Una cadena que no contiene caracteres se llama *cadena nula* (o *cadena vacía*). Se escribe la cadena nula con dos comillas sin nada (ni siquiera espacios) entre ellas:

```
""
```

La cadena nula no es equivalente a una cadena de espacios; es una cadena especial que no contiene caracteres.

Para trabajar con datos de cadena, en este libro se usa un tipo de datos llamado `string`. Este tipo de datos no es parte del lenguaje C++ (es decir, no es un tipo integrado). Además, `string` es un tipo definido por el programador que es suministrado por la *biblioteca estándar* C++, una colección grande de funciones y tipos de datos prescritos que cualquier programador de C++ puede usar. Las

---

* En la mayoría de los lenguajes de programación se emplea ASCII (American Standard Code for Information Interchange) para representar el alfabeto inglés y otros símbolos. Cada carácter ASCII se almacena en un solo byte de memoria.

Un conjunto de caracteres más nuevo denominado Unicode incluye los alfabetos de muchos idiomas. Un solo carácter Unicode ocupa dos bytes de memoria. C++ proporciona los datos tipo `wchar_t` (para "carácter amplio") a fin de acomodar conjuntos de caracteres más grandes como Unicode. En C++, la notación L `'algo'` denota un valor de tipo `wchar_t`, donde el algo depende del conjunto de caracteres amplios particular que se emplea. En este libro ya no se examinan caracteres amplios.

operaciones en datos `string` incluyen comparar los valores de `string`, buscar una cadena para un carácter particular y unir una cadena a otra. Se examinan algunas de estas operaciones después en este capítulo y se cubren más operaciones en capítulos posteriores. Ninguno de los identificadores del programa del año bisiesto es del tipo `string`, aunque los valores de cadena se emplean directamente en varios lugares del programa.

Cómo nombrar elementos: declaraciones

Los identificadores se pueden usar para nombrar constantes y variables. En otras palabras, un identificador puede ser el nombre de una localidad de la memoria cuyo contenido no está permitido cambiar o puede ser un nombre de una localidad de la memoria cuyo contenido cambia.

Declaración Enunciado que relaciona a un identificador con un objeto de datos, una función o tipo de datos para que el programador pueda hacer referencia a ese ítem por nombre.

¿Cómo se indica a la computadora qué representa un identificador? Mediante una **declaración**, una sentencia que relaciona un nombre (un identificador) con una descripción de un elemento en un programa C++ (del mismo modo que una definición de diccionario relaciona un nombre con una descripción del mismo). En una declaración, se nombra un identificador y lo que representa. Por ejemplo, el programa del año bisiesto emplea la declaración

```
int year:
```

para anunciar que `year` es el nombre de una variable cuyo contenido es del tipo `int`. Cuando se declara una variable, el compilador selecciona una localidad de la memoria que será relacionada con el identificador. No se tendría que saber la dirección real de la localidad en la memoria porque la computadora sigue el rastro de modo automático.

Suponga que cuando se envía por correo una carta, sólo se tiene que escribir el nombre de una persona en ella y la oficina de correo buscaría la dirección. Por supuesto, todos los habitantes del mundo necesitarían un nombre distinto; de lo contrario, la oficina no podría dilucidar de qué dirección se trata. Lo mismo se cumple en C++. Cada identificador representa sólo una cosa (excepto en circunstancias especiales, de las que se hablará en los capítulos 7 y 8). Todo identificador que se emplea en un programa debe ser diferente de los demás.

Las constantes y variables se llaman en conjunto *objetos de datos*. Ambos objetos de datos y las instrucciones actuales de un programa se almacenan en varias localidades de la memoria. Se ha visto que un grupo de instrucciones, una función, puede recibir un nombre. Un nombre también se puede relacionar con un tipo de datos definido por el programador.

En C++, se debe declarar todo identificador antes de usarlo. Esto permite que el compilador compruebe que el uso del identificador no sea otro que el establecido. Si se declara que un identificador sea una constante y después se intenta cambiar su valor, el compilador detecta esta incongruencia y presenta un mensaje de error.

Hay una forma diferente de sentencias de declaración para cada clase de objeto de datos, función o tipo de datos en C++. Las formas de declaraciones para variables y constantes se introducen aquí; otras se estudian en capítulos posteriores.

Variables Un programa opera en datos. Los datos se almacenan en la memoria. Mientras se ejecuta un programa, valores diferentes pueden ser almacenados en la misma localidad de memoria en tiempos distintos. Esta clase de localidad de memoria se llama **variable**, y su contenido es el *valor de variable*. El nombre simbólico que se relaciona con la localidad de memoria es el *nombre de variable* o *identificador de variable* (véase la figura 2-1). En la práctica, es común referirse al nombre de variable en forma más breve como la *variable*.

Variable Una localidad en memoria, referenciada por un identificador, que contiene un valor de dato que puede ser cambiado.

Declarar una variable significa especificar su nombre y tipo de datos. Esto dice al compilador que relacione un nombre con una localidad de memoria cuyo contenido es de un tipo específico (por ejemplo, `char` o `string`). La siguiente sentencia declara que `myChar` es una variable de tipo `char`:

```
char myChar:
```

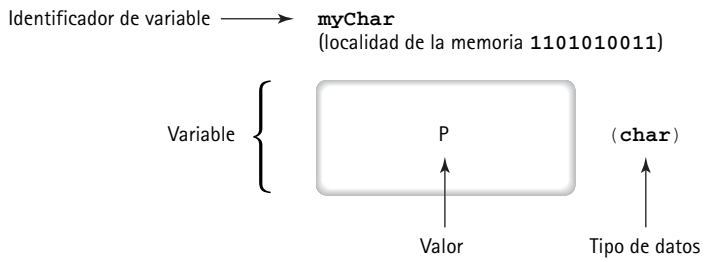


Figura 2-1 *Variable*

En C++, una variable puede contener un valor de datos sólo del tipo especificado en su declaración. Debido a la declaración anterior, la variable `myChar` puede contener *sólo* un valor `char`. Si el compilador C++ encuentra una instrucción que trata de almacenar un valor `float` en `myChar`, genera instrucciones extra para convertir el valor `float` al tipo adecuado. En el capítulo 3 se examina cómo tiene lugar dicho tipo de conversión.

A continuación se muestra la plantilla de sintaxis para una declaración de variable:

VariableDeclaration

```
DataType Identifier , Identifier. . . ;
```

donde DataType es el mismo nombre de un tipo de datos como `char` o `string`. Observe que una declaración de variable termina siempre con un punto y coma.

De la plantilla de sintaxis, se puede ver que es posible declarar varias variables en una sentencia:

```
char letter, middleInitial, ch;
```

Aquí las tres variables se declaran como variables `char`. Sin embargo, se prefiere declarar cada variable con una sentencia separada:

```
char letter;
char middleInitial;
char ch;
```

Con esta forma es más fácil, al modificar un programa, agregar nuevas variables a la lista o borrar las que ya no interesan.

Declarar cada variable con una sentencia separada permite también anexar comentarios a la derecha de cada declaración, como se muestra aquí:

```
float payRate;       // Tasa de pago del empleado
float hours;         // Horas trabajadas
float wages;         // Salario ganado
int   empNum;        // Número de identificación del empleado
```

Estas declaraciones indican al compilador que reserve espacio de memoria para tres variables `float`, `payRate`, `hours` y `wages`, y una variable `int`, `empNum`. Los comentarios explican a alguien que lea el programa que representa cada variable.

Ahora que se ha visto cómo declarar variables en C++, considérese cómo declarar constantes.

Constantes Todos los caracteres simples (encerrados entre apóstrofos) y cadenas (encerradas entre comillas) son constantes.

```
'A'   '@'   "Howdy boys"   "Please enter an employee number:"
```

Valor literal Cualquier valor constante escrito en un programa.

Constante nombrada (constante simbólica) Lugar de la memoria, al que se hace referencia mediante un identificador, que contiene un valor de datos que no puede ser cambiado.

En C++ como en matemáticas, una constante es algo cuyo valor nunca cambia. Cuando se usa el valor actual de una constate en un programa, se está usando un **valor literal** (o *literal*).

Una alternativa para la constante literal es la **constante nombrada** (o **constante simbólica**), que se introduce en un enunciado de sentencia. Una constante nombrada es sólo otra forma de representar un valor literal. En lugar de usar el valor literal en una instrucción, se le da un nombre en un enunciado de sentencia, y después se usa ese nombre en la instrucción. Por ejemplo, se puede escribir una instrucción que imprima el título de este libro usando la cadena literal `"Programación y resolución de problemas con C++"`. O bien, se puede declarar una constante nombrada que se denomina `BOOK_TITLE`, que es igual a la misma cadena, y luego usar el nombre constante en la instrucción. Es decir, se puede usar

```
"Programación y resolución de problemas con C++"
```

o bien

```
BOOK_TITLE
```

en la instrucción.

Usar el valor literal de una constante puede parecer más fácil que darle un nombre y luego referirse a ella mediante ese nombre. Pero, de hecho, las constantes nombradas facilitan la lectura de un programa porque aclaran el significado de las constantes literales. Las constantes nombradas también hacen más fácil hacer cambios posteriores en un programa.

Ésta es una plantilla de sintaxis para una declaración constante:

ConstantDeclaration

```
const DataType Identifier = LiteralValue;
```

Observe que la palabra reservada `const` comienza con la declaración, y un signo igual (=) aparece entre el identificador y el valor literal.

Lo siguiente son ejemplos de declaraciones constantes:

```
const string STARS = "********";
const char   BLANK = ' ';
const string BOOK_TITLE = "Programming and Problem Solving with C++";
const string MESSAGE = "Error condition";
```

Como se hizo antes, muchos programadores de C++ escriben en mayúsculas el identificador completo de una constante nombrada y separan las palabras con una rayita de subrayado. La idea es permitir que el lector distinga rápidamente entre nombres de variables y nombres de constantes cuando aparecen a la mitad de un programa.

Es una buena idea añadir comentarios tanto a declaraciones de constantes como a declaraciones de variables. Se describe en comentarios lo que representa cada constante:

```
const float MAX_HOURS = 40.0;    // Horas de trabajo normal máximas
const float OVERTIME = 1.5;      // Factor de pago por horas extras
```

Cuestiones de estilo

Escritura de identificadores en letras mayúsculas

Los programadores emplean con frecuencia letras mayúsculas como una pista visual rápida para lo que representa un identificador. Diferentes programadores adoptan distintas convenciones para usar letras mayúsculas y minúsculas. Algunas personas sólo emplean minúsculas, y separan las palabras en un identificador con el carácter de subrayado:

```
pay_rate   emp_num  pay_file
```

Las convenciones que se emplean en este libro son como sigue:

- Para identificadores que representan variables, se empieza con una letra minúscula y la inicial de cada palabra sucesiva se escribe con mayúscula.

  ```
  lengthInYards   middleInitial   hours
  ```

- Los nombres de funciones escritas por el programador y los tipos de datos definidos por el programador (que se examinan más adelante en el libro) se escriben de la misma manera que los nombres de variables, excepto que comienzan con letras mayúsculas.

  ```
  CalcPay(payRate, hours, wages)   Cube(27)   MyDataType
  ```

  Escribir con mayúscula la primera letra permite que una persona lea el programa para saber, de un vistazo, que un identificador representa un nombre de función o tipo de datos y no una variable. Sin embargo, no se puede usar esta convención de uso de mayúsculas en todas partes. C++ espera que todo programa tenga una función llamada `main`, escrita en minúsculas, así que no es posible llamarla `Main`. Tampoco se puede usar `Char` para el tipo de datos integrados `char`. Las palabras reservadas de C++ se escriben en minúsculas, así como la mayoría de los identificadores declarados en la biblioteca estándar (como `string`).

- Para identificadores que representan constantes nombradas, se usan mayúsculas para cada letra y se usan rayas de subrayado para separar las palabras.

  ```
  BOOK_TITLE   OVERTIME   MAX_LENGTH
  ```

  Esta convención, de uso extenso entre los programadores de C++, es una señal inmediata de que `BOOK_TITLE` es una constante nombrada y no una variable, función o tipo de datos.

Estas convenciones son sólo eso, convenciones. C++ no requiere este estilo particular de uso de mayúsculas en los identificadores. Quizá desee usar las mayúsculas de un modo diferente. Pero cualquiera que sea el sistema que use, es esencial que utilice un estilo congruente en su programa. Una persona que lea su programa se confundirá si usted usa un estilo aleatorio de uso de mayúsculas.

Manos a la obra: sentencias ejecutables

Hasta este punto, se ha considerado la forma de declarar objetos de datos en un programa. Ahora se dirige la atención a las formas de actuar, o efectuar operaciones, en datos.

Asignación El valor de una variable puede ser establecido o cambiado a través de una **sentencia de asignación**. Por ejemplo,

Sentencia de asignación Sentencia que almacena el valor de una expresión en una variable.

```
lastName = "Lincoln";
```

asigna el valor de cadena `"Lincoln"` a la variable `lastName` (es decir, almacena la secuencia de caracteres "Lincoln" en la memoria relacionada con la variable llamada `lastName`).

A continuación se da la plantilla de sintaxis para una sentencia de asignación:

AssignmentStatement

```
Variable = Expression;
```

Expresión Disposición de identificadores, literales y operadores que puede ser evaluada para calcular un valor de un tipo dado.

Evaluar Calcular un nuevo valor al efectuar un conjunto especificado de operaciones sobre valores dados.

La semántica (significado) del operador de asignación (=) es "almacenar"; el valor de la **expresión** se *almacena* en la variable. Cualquier valor previo de la variable se destruye y reemplaza por el valor de la expresión.

Sólo una variable puede estar del lado izquierdo de una sentencia de asignación. Una sentencia de asignación *no* es como una expresión matemática ($x + y = z + 4$); la expresión (lo que está a la derecha del operador de asignación) se **evalúa**, y el valor resultante se almacena en la variable única a la izquierda del operador de asignación. Una variable mantiene su valor asignado hasta que otra sentencia almacena un nuevo valor en ella.

Dadas las declaraciones

```
string firstName;
string middleName;
string lastName;
string title;
char   middleInitial;
char   letter;
```

las siguientes sentencias de asignación son válidas:

```
firstName = "Abraham";
middleName = firstName;
middleName = "";
lastName = "Lincoln";
title = "President";
middleInitial = ' ';
letter = middleInitial;
```

Sin embargo, estas asignaciones no son válidas:

Sentencia de asignación no válida	Razón
`middleInitial = "A.";`	`middleInitial` es de tipo `char`; `"A."` es una cadena.
`letter = firstName;`	`letter` es de tipo `char`; `firstName` es de tipo `string`.
`firstName = Thomas;`	`Thomas` es un identificador no declarado.
`"Edison" = lastName;`	Sólo puede aparecer una variable a la izquierda de =.

Expresiones de cadena Aunque no es posible hacer expresiones aritméticas con cadenas, el tipo de datos `string` proporciona una operación de cadena especial, llamada *concatenación*, que usa el ope-

rador +. El resultado de concatenar (unir) dos cadenas es una nueva cadena que contiene los caracteres de ambas cadenas. Por ejemplo, dadas las sentencias

```
string bookTitle;
string phrase1;
string phrase2;

wphrase1 = "Programming and ";
phrase2 = "Problem Solving";
```

se podría escribir

```
bookTitle = phrase1 + phrase2;
```

Esta sentencia recupera el valor de `phrase1` de la memoria y concatena el valor de `phrase2` para formar una cadena temporal nueva que contiene los caracteres

```
"Programming and Problem Solving"
```

Esta cadena temporal (que es de tipo `string`) se asigna entonces (almacena en) `book-Title`.

El orden de las cadenas en la expresión determina cómo aparecen en la cadena resultante. Si en cambio se escribe

```
bookTitle = phrase2 + phrase1;
```

entonces `bookTitle` contiene

```
"Problem SolvingProgramming and "
```

La concatenación funciona con constantes `string` nombradas, cadenas literales y datos `char`, así como con variables `string`. La única restricción es que por lo menos uno de los operandos del operador + *debe* ser una variable `string` o constante nombrada (así que no se pueden usar expresiones como `"Hi" + "ther"` o bien `'A' + 'B'`). Por ejemplo, si se han declarado las siguientes constantes:

```
const string WORD1 = "rogramming";
const string WORD3 = "Solving";
const string WORD5 = "C++";
```

entonces se podría escribir la siguiente sentencia de asignación para almacenar el título de este libro en la variable `bookTitle`:

```
bookTitle = 'P' + WORD1 + " and Problem " + WORD3 + " with " + WORD5;
```

Como resultado, `bookTitle` contiene la cadena

```
"Programming and Problem Solving with C++"
```

En el ejemplo precedente se demuestra cómo es posible combinar identificadores, datos `char` y cadenas literales en una expresión de concatenación. Por supuesto, si sólo se quiere asignar la cadena completa a `bookTitle`, se puede hacer de modo directo:

```
bookTitle = "Programming and Problem Solving with C++";
```

Pero ocasionalmente se encuentra una situación en la que se quiere añadir algunos caracteres a un valor de cadena existente. Suponga que `bookTitle` ya contiene `"Programming and Problem Solving"` y que se desea completar el título. Se podría usar una sentencia de la forma

```
bookTitle = bookTitle + " with C++";
```

Tal declaración recupera el valor de `bookTitle` de la memoria, concatena la cadena `" with C++"` para formar una nueva cadena, y luego almacena la nueva cadena en `bookTitle`. La nueva cadena remplaza al valor anterior de `bookTitle` (que es destruido).

Recuerde que la concatenación funciona sólo con valores de tipo `string`. Aunque se usa un signo más para la operación, no se puede concatenar valores de tipos de datos numéricos, como `int` y `float`, con cadenas.

Si usted está usando C++ pre-estándar (cualquier versión de C++ previa al estándar ISO/ANSI) y su biblioteca estándar no proporciona el tipo `string`, véase en la sección D.1 del apéndice D una explicación de cómo proceder.

Salida ¿Alguna vez ha preguntado a alguien, "¿sabes qué hora es?" sólo para ver que la persona sonríe con satisfacción y dice, "sí, si sé", y se aleja? Esta situación es como la que existe en realidad entre usted y la computadora. Ahora sabe suficiente sintaxis de C++ para indicarle a la computadora que asigne valores a las variables y concatene cadenas, pero la computadora no le dará resultados hasta que le diga que los escriba.

En C++ los valores de variables y expresiones se escriben por medio de una variable especial llamada `cout` junto con el *operador de inserción* (<<):

```
cout << "Hello";
```

Esta sentencia muestra los caracteres `Hello` en el *dispositivo de salida estándar*, por lo general la pantalla.

La variable `cout` está predefinida en sistemas C++ para denotar un *flujo de salida*. Se puede pensar en un flujo de salida como una secuencia interminable de caracteres que van a un dispositivo de salida. En el caso de `cout`, el flujo de salida va al dispositivo de salida estándar.

El operador de inserción << toma dos operandos. Su operando izquierdo es una expresión de flujo (en el caso más sencillo, sólo una variable de flujo como `cout`). Su operando derecho es una expresión, que podría ser tan simple como una cadena literal:

```
cout << "The title is ";
cout << bookTitle + ", 2nd Edition";
```

El operador de inserción convierte a su operando derecho en una secuencia de caracteres y los inserta en (o, de manera más precisa, agrega a) el flujo de salida. Observe cómo << apunta en la dirección que van los datos, *de* la expresión escrita a la derecha al flujo de salida a la izquierda.

Se puede usar el operador << varias veces en una sola sentencia de salida. Cada vez que aparece anexa el siguiente ítem de datos al flujo de salida. Por ejemplo, se pueden escribir las dos sentencias de salida como

```
cout << "The title is " << bookTitle + ", 2nd Edition";
```

Si `bookTitle` contiene `"American History"`, ambas versiones producen la misma salida:

```
The title is American History, 2nd Edition
```

La sentencia de salida tiene la siguiente forma:

OutputStatement

```
cout << Expression << Expression . . . ;
```

Las siguientes sentencias de salida producen el resultado mostrado. En estos ejemplos se supone que la variable `char ch` contiene el valor `'2'`, la variable de cadena `firtName` contiene `"Marie"` y la variable `string`, `lastName`, contiene `"Curie"`.

Sentencia	Lo que se imprime (␣means blank)
`cout << ch;`	`2`
`cout << "ch = " <<ch;`	`ch␣=␣2`
`cout << firstName + " " + lastName;`	`Marie␣Curie`
`cout << firstName << lastName;`	`MarieCurie`
`cout << firstName << ' ' << lastName;`	`Marie␣Curie`
`cout << "ERROR MESSAGE";`	`ERRORMESSAGE`
`cout << "Error=" << ch;`	`Error=2`

Una sentencia de salida imprime cadenas literales exactamente como aparecen. Para permitir que la computadora sepa que usted quiere imprimir una cadena literal —no una constante nombrada o variable— debe recordar usar comillas para encerrar la cadena. Si no escribe la cadena entre comillas, es probable que obtenga un mensaje de error (como "UNDECLARED IDENTIFIER") del compilador de C++. Si desea imprimir una cadena que incluye un par de comillas, se debe escribir el carácter (\), barra inclinada a la izquierda, y un par de comillas, sin espacio entre ellas, en la cadena. Por ejemplo, para imprimir los caracteres

```
Al "Butch" Jones
```

la declaración de salida se parece a esto:

```
cout << "Al \"Butch\" Jones";
```

Para concluir esta consideración introductoria de la salida de C++, se debe mencionar cómo terminar una línea de salida. Normalmente, las sentencias de salida sucesivas causan que la salida continúe a lo largo de la misma línea de la pantalla. La secuencia

```
cout << "Hi";
cout << "there";
```

escribe lo siguiente en la pantalla, todo en la misma línea:

```
Hithere
```

Para imprimir las dos palabras en líneas separadas, se puede hacer esto:

```
cout << "Hi" << endl;
cout << "there" << endl;
```

La salida de estas declaraciones es

```
Hi
there
```

El identificador `end1` (que significa "termina la línea") es una característica especial de C++ llamada *manipulador*. En el capítulo siguiente se analizan los manipuladores. Por ahora, lo importante es notar que `end1` permite terminar una línea de salida e ir a la siguiente siempre que lo desee.

Más allá del minimalismo: añadir comentarios a un programa

Todo lo que necesita para crear un programa funcional es la correcta combinación de declaraciones y sentencias ejecutables. El compilador ignora los comentarios, pero son de gran ayuda para cualquiera que deba leer el programa. Los comentarios pueden aparecer en cualquier parte en un programa, excepto a la mitad de un identificador, una palabra reservada o una constante literal.

Los comentarios de C++ vienen en dos formas. La primera es cualquier secuencia de caracteres encerrada por el par `/* */`. El compilador ignora cualquier cosa dentro del par. Lo siguiente es un ejemplo:

```
string idNumber;   /* Identification number of the aircraft */
```

La segunda forma, y más común, comienza con dos diagonales (//) y se extiende hasta el final de esa línea del programa:

```
string idNumber;   // Identification number of the aircraft
```

El compilador ignora cualquier cosa después de las dos diagonales.

Escribir programas completamente comentados es un buen estilo de programación. Un comentario debe aparecer al comienzo de un programa para explicar lo que hace el programa:

```
// Este programa calcula el peso y equilibrio de un avión Beechcraft
// Starship-1, dada la cantidad de combustible, número de
// pasajeros y el peso del equipaje en los compartimientos de proa y popa.
// Se supone que hay dos pilotos y un complemento estándar
// de equipo, y que los pasajeros pesan 170 libras cada uno
```

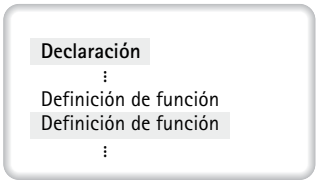
Otro buen lugar para comentarios es en las declaraciones constante y variable, donde los comentarios explican cómo se emplea cada identificador. Además, los comentarios deben introducir cada paso principal en un programa largo y explicar cualquier cosa que sea inusual o difícil de leer (por ejemplo, una fórmula larga).

Es importante hacer comentarios concisos y ordenarlos en el programa, de modo que sean fáciles de ver y claros respecto a lo que se refieren. Si los comentarios son demasiado largos o atestan las sentencias del programa, hacen que éste sea más difícil de leer, ¡justo lo contrario de lo que se pretende!

2.2 Construcción del programa

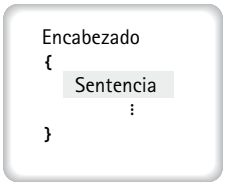
Se han considerado los elementos básicos de programas C++: identificadores, declaraciones, variables, constantes, expresiones, sentencias y comentarios. Ahora se verá cómo reunir estos elementos en un programa. Como se vio antes, los programas C++ están hechos de funciones, una de las cuales debe ser nombrada `main`. Un programa también puede tener declaraciones que quedan fuera de cualquier función. La plantilla de sintaxis para un programa se parece a esto:

Programa



Una definición de función consta del encabezado de la función y su cuerpo, que está delimitada por las llaves izquierda y derecha:

Definición de función



A continuación se presenta un ejemplo de un programa con sólo una función, la función `main`:

```
//*************************************************************
//Programa PrintName
//Este programa imprime un nombre en dos formatos distintos
//*************************************************************
#include <iostream>
#include <string>

using namespace std;

const string FIRST = "Herman";   // Nombre de la persona
const string LAST = "Smith";     // Apellido de la persona
const char   MIDDLE = 'G';       // Inicial intermedia de la persona

int main()
{
    string firstLast;     // Nombre en formato nombre-apellido
    string lastFirst;     // Nombre en formato apellido-nombre

    firstLast = FIRST + " " + LAST;
    cout << "El nombre en formato nombre-apellido es " << firstLast << endl;

    lastFirst = LAST + ", " + FIRST + ", ";
    cout << "El nombre en formato apellido-nombre-inicial es ";
    cout << lastFirst << MIDDLE << '.' << endl;

    return 0;
}
```

El programa comienza con un comentario que explica el código del programa. Inmediatamente después del comentario aparecen las siguientes líneas:

```
#include <iostream>
#include <string>

using namespace std;
```

Las líneas `#include` instruyen al compilador C++ que inserte en el programa el contenido de los archivos llamados `iostream` y `string`. El primer archivo contiene información que C++ necesita para enviar valores a un flujo como `cout`. El segundo archivo contiene información acerca del tipo de datos `string` definidos por el programador. Más adelante, en este capítulo, se analiza el propósito de las líneas `#include` y la sentencia `using`.

A continuación viene una sección de declaración en la que se definen las constantes `FIRST`, `LAST` y `MIDDLE`. Los comentarios explican cómo se usa cada identificador. El resto del programa es la definición de función para la función `main`. La primera línea es el encabezado de la función: la palabra reservada `int`, el nombre de la función y luego los paréntesis de apertura y cierre. (Los paréntesis informan al compilador que `main` es el nombre de una función, no una variable o constante nombrada.) El cuerpo de la función incluye las declaraciones de dos variables, `firstLast` y `lastFirst`, seguidas de una lista de sentencias ejecutables. El compilador traduce estas sentencias ejecutables en instrucciones de lenguaje de máquina. Durante la fase de ejecución del programa, éstas son las instrucciones que se ejecutan.

La función `main` termina devolviendo a 0 como el valor de la función:

```
return 0;
```

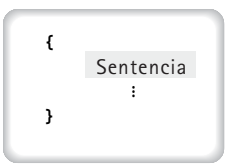
Recuerde que `main` devuelve un valor entero al sistema operativo cuando se completa la ejecución. Este valor entero se llama *estado de salida*. En la mayoría de las computadoras se devuelve un estado de salida 0 para indicar que fue exitosa la terminación del programa; de lo contrario, se obtiene un valor distinto de cero.

Observe cómo se usa el espaciamiento (uso de sangrías) en el programa PrintName para facilitar su lectura. Se emplean líneas en blanco para separar sentencias en grupos relacionados, y luego se emplean sangrías en todo el cuerpo de la función `main`. El compilador no requiere que se formatee de esta manera el programa; esto se hace para hacerlo más legible. En el siguiente capítulo se proporciona información acerca del formato de un programa.

Bloques (sentencias compuestas)

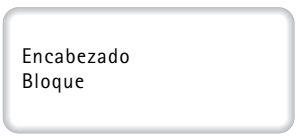
El cuerpo de una función es un ejemplo de un *bloque* (o *sentencia compuesta*). Ésta es la plantilla de sintaxis para un bloque:

Bloque



Un bloque es sólo una secuencia de cero o más sentencias encerradas (delimitadas) por un par de llaves `{ }`. Ahora se puede determinar de nuevo una definición de función como un encabezado seguido de un bloque:

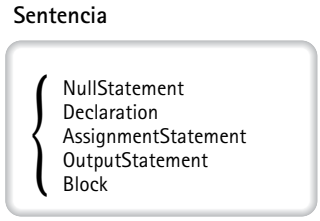
Definición de función



En capítulos posteriores, cuando se aprenda cómo escribir funciones distintas a `main`, se define más precisamente la sintaxis del Encabezado. En el caso de la función `main`, el Encabezado es simplemente

```
int main()
```

A continuación se proporciona la plantilla de sintaxis para una sentencia, limitada a las sentencias de C++ analizadas en este capítulo:



Una sentencia puede ser vacía (*sentencia nula*). La sentencia nula es sólo un punto y coma (`;`) y se ve como esto:

```
;
```

No hace nada en absoluto al momento de la ejecución; ésta procede a la siguiente sentencia. Esto no se utiliza con frecuencia.

Según se observa en la plantilla de sintaxis, una sentencia o proposición puede ser también una declaración, una sentencia ejecutable o incluso un bloque. Este último significa que se puede usar un bloque completo siempre que se permita una sentencia sencilla. En capítulos posteriores en que se introduce la sintaxis para estructuras de bifurcación y ciclos, este hecho es muy importante.

El uso de bloques es frecuente, en particular como partes de otras sentencias. Dejar un par de llaves `{ }` puede cambiar de manera drástica tanto el significado como la ejecución de un programa. Esto es porque dentro de un bloque siempre se usan sentencias con sangría; ésta hace que sea fácil detectar un bloque en un programa largo y complicado.

Observe que en las plantillas de sintaxis para el bloque y la sentencia no se hace mención del punto y coma. Sin embargo, el programa PrintName contiene muchos signos de punto y coma. Si examina de nuevo las plantillas para declaración constante, declaración variable, sentencia de asignación y sentencia de salida, se puede ver que se requiere un punto y coma al final de cada tipo de sentencia. Sin embargo, en la plantilla de sintaxis para el bloque no se observa ningún punto y coma después de la llave derecha. La regla para usar punto y coma en C++ es, por tanto, muy sencilla: termine cada sentencia, *excepto* una sentencia compuesta (bloque), con un punto y coma.

Una cosa más acerca de los bloques y sentencias: de acuerdo con la plantilla de sintaxis para una sentencia, una declaración es considerada oficialmente como una sentencia. Por tanto, una declaración puede aparecer siempre que aparezca una sentencia ejecutable. En un bloque se pueden combinar declaraciones y sentencias ejecutables, si se desea:

```
{
    char ch;
    ch = 'A';
    cout << ch;
    string str;
    str = "Hello";
    cout << str;
}
```

Sin embargo, es mucho más común que los programadores agrupen las declaraciones antes del inicio de sentencias ejecutables:

```
{
    char   ch;
    string str;

    ch = 'A';
    cout << ch;
```

```
    str = "Hello";
    cout << str;
}
```

El preprocesador de C++

Imagine que usted es el compilador de C++ y se le presenta el siguiente programa en el que comprobará los errores de sintaxis y, si no existen, lo traducirá en código de lenguaje de máquina.

```
//************************************
// Este programa imprime Feliz cumpleaños
//************************************

int main()
{
    cout << "Feliz cumpleaños" << endl;
    return 0;
}
```

Usted, el compilador, reconoce al identificador `int` como una palabra reservada de C++ y al identificador `main` como el nombre de una función requerida. ¿Pero qué hay acerca de los identificadores `cout` y `endl`? El programador no los ha declarado como variables o constantes nombradas, y no son palabras reservadas. Usted no tiene más opción que emitir un mensaje de error y desistir.

Para arreglar este programa, lo primero que se debe hacer es insertar una línea cerca de la parte superior que diga

```
#include <iostream>
```

del mismo modo que se hizo en el programa PrintName (así como en el programa de muestra al comienzo de este capítulo y el del año bisiesto [LeapYear] del capítulo 1).

La línea dice insertar el contenido de un archivo llamado `iostream` en el programa. Este archivo contiene declaraciones de `cout`, `endl`, y otros elementos necesarios para efectuar el flujo de entrada y salida. La línea `#include` no es manejada por el compilador de C++, sino por un programa conocido como *preprocesador*.

El concepto de preprocesador es fundamental en C++. El preprocesador es un programa que actúa como filtro durante la fase de compilación. Su programa fuente pasa por el preprocesador en su camino hacia el compilador (véase la figura 2-2).

Una línea que comienza con un signo de libra (`#`) no se considera una sentencia del leguaje C++ (y, por tanto, no termina con punto y coma). Se llama *directiva de preprocesador*. El preprocesador expande una directiva `#include` al insertar físicamente el contenido del archivo nombrado en su programa fuente. Un archivo cuyo nombre aparece en una directiva `#include` se llama *archivo de encabezado*. Los archivos de encabezado contienen declaraciones de constante, variable, tipo de datos y función requeridas por un programa.

En las directivas

```
#include <iostream>
#include <string>
```

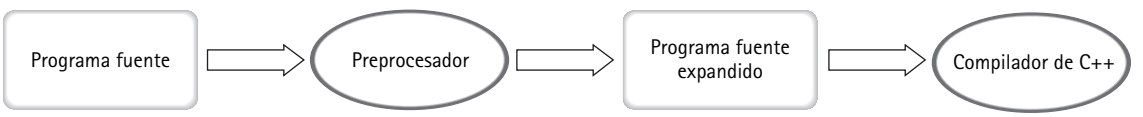


Figura 2-2 *El preprocesador de C++*

son necesarios los paréntesis angulares < >. Ellos indican al procesador que busque archivos en el *directorio include* estándar, un lugar en el sistema de la computadora que contiene los archivos de encabezado que se relacionan con la biblioteca estándar de C++. El archivo `iostream` contiene declaraciones de medios de entrada o salida, y el archivo `string` contiene declaraciones acerca del tipo de datos `string`. En el capítulo 3 se hace uso de archivos de encabezado estándares distintos a `iostream` y `string`.

En el lenguaje C y en C++ pre-estándar, los archivos de encabezado estándares terminan en el sufijo `.h` (como, `iostream.h`), donde *h* hace pensar en "archivo de encabezado". En ISO/ANSI C++, los archivos de encabezado estándares ya no emplean el sufijo `.h`.

Introducción a los espacios de nombres (Namespaces)

En el programa Feliz cumpleaños, incluso si se añade la directiva al preprocesador `#include <iostream>`, el programa no compilará. El compilador *aún* no reconoce a los identificadores `cout` y `end1`. El problema es que el archivo de encabezado iostream (y, de hecho, todo archivo de encabezado estándar) declara que todos sus identificadores están en un *namespace* llamado `std`:

```
namespace std
{
    :  Declaraciones de variables, tipos de datos, etcétera
}
```

Se puede tener acceso directo a un identificador declarado dentro de un bloque de espacio de nombre (namespace) sólo mediante las sentencias dentro de ese bloque. Para tener acceso a un identificador que está "oculto" dentro de un espacio de nombres, el programador tiene varias opciones. Aquí se describen dos de ellas. En el capítulo 8 se describen espacios de nombres con más detalle.

La primera opción es usar un *nombre calificado* para el identificador. Un nombre calificado consta del nombre del namespace, luego el operador `::` (el *operador de resolución de alcance*) y después el identificador deseado:

```
std::cout
```

Con este método el programa se parece a lo siguiente:

```
#include <iostream>

int main()
{
    std::cout << "Feliz cumpleaños" << std::endl;
    return 0;
}
```

Observe que tanto `cout` como `end1` deben ser calificadas.

La segunda opción es usar una sentencia denominada *directiva using*:

```
using namespace std;
```

Cuando se coloca esta sentencia cerca de la parte superior de un programa antes de la función `main`, se logra que *todos* los identificadores del espacio de nombres `std` tengan acceso al programa sin tener que calificarlos:

```
#include <iostream>

using namespace std;

int main()
{
```

```
    cout << "Happy Birthday" << endl;
    return 0;
}
```

Esta segunda opción es la que se emplea en el programa PrintName y el programa muestra del principio de este capítulo. En muchos de los capítulos siguientes se continúa usando este método. Sin embargo, en el capítulo 8 se explica por qué no es recomendable el uso del método en programas grandes.

Si está usando un compilador pre-estándar que no reconoce espacios de nombre y los encabezados de archivo más recientes (`iostream`, `string`, etc.), debe volver a la sección D.2 del apéndice D para ver una explicación acerca de incompatibilidades.

2.3 Más acerca de la salida

Se puede controlar tanto el espaciamiento horizontal como el vertical de la salida para hacerla más atractiva (y comprensible). Considérese primero un espaciamiento vertical.

Crear líneas en blanco

El espaciamiento vertical se controla por medio del manipulador `endl` en una sentencia de salida. Se ha visto que una secuencia de sentencias de salida continúa escribiendo caracteres en la línea de flujo hasta que `endl` termina la línea. A continuación se dan algunos ejemplos:

Sentencias	Salida producida*
`cout << "Hi there, ";`	
`cout << "Lois Lane. " << endl;`	`Hi there, Lois Lane.`
`cout << "Have you seen ";`	
`cout << "Clark Kent?" << endl;`	`Have you seen Clark Kent?`
`cout << "Hi there, " << endl;`	`Hi there,`
`cout << "Lois Lane. " << endl;`	`Lois Lane.`
`cout << "Have you seen " << endl;`	`Have you seen`
`cout << "Clark Kent?" << endl;`	`Clark Kent?`
`cout << "Hi there, " << endl;`	`Hi there,`
`cout << "Lois Lane. ";`	
`cout << "Have you seen " << endl;`	`Lois Lane. Have you seen`
`cout << "Clark Kent?" << endl;`	`Clark Kent?`

* Las líneas de salida se muestran a continuación de la sentencia de salida que termina a cada una de ellas. No hay líneas en blanco en la salida real de estas sentencias.

¿Qué cree que impriman las siguientes sentencias?

```
cout << "Hi there, " << endl;
cout << endl;
cout << "Lois Lane." << endl;
```

La primera sentencia de salida ocasiona que se impriman las palabras *Hi there*; `endl` causa que el cursor de pantalla vaya a la siguiente línea. La siguiente sentencia no imprime nada pero va a la

siguiente línea. La tercera sentencia imprime las palabras *Lois Lane.* y termina la línea. La salida resultante son las tres líneas.

```
Hi there,

Lois Lane.
```

Siempre que use una `endl` inmediatamente después de otra `endl`, se produce una línea en blanco. Como podría suponer, tres usos consecutivos de `endl` producen dos líneas en blanco, cuatro usos consecutivos producen tres líneas en blanco, etcétera.

Note que se tiene mucha flexibilidad en cómo escribir una sentencia de salida en un programa C++. Se podría combinar las tres sentencias precedentes en dos:

```
cout << "Hi there, " << endl << endl;
cout << "Lois Lane." << endl;
```

De hecho, esto se podría hacer en una sentencia. Una posibilidad es

```
cout << "Hi there, " << endl << endl << "Lois Lane." << endl;
```

Aquí está otra:

```
cout << "Hi there, " << endl << endl
     << "Lois Lane." << endl;
```

En el último ejemplo se muestra que es posible extender una sola sentencia de C++ en más de una línea del programa. El compilador trata al punto y coma no como el final físico de una línea sino como el fin de una sentencia.

Inserción de espacios en blanco dentro de una línea

Para controlar el espaciamiento horizontal de la salida, una técnica es enviar caracteres extra en blanco al flujo de salida. (Recuerde que el carácter en blanco, generado al oprimir la barra espaciadora en un teclado, es un carácter perfectamente válido en C++.)

Por ejemplo, para producir esta salida:

```
  *   *   *   *   *   *   *   *   *   *

*   *   *   *   *   *   *   *   *   *   *

  *   *   *   *   *   *   *   *   *   *
```

se usarían estos enunciados:

```
cout << "  *   *   *   *   *   *   *   *   *   *" << endl << endl;
cout << "*   *   *   *   *   *   *   *   *   *   *" << endl << endl;
cout << "  *   *   *   *   *   *   *   *   *   *" << endl;
```

Todos los espacios en blanco y asteriscos están entre comillas, así que se imprimen literalmente como están escritos en el programa. Los manipuladores extra `endl` proporcionan las líneas en blanco entre los renglones de asteriscos.

Si quiere que se impriman espacios en blanco, *debe* encerrarlos entre apóstrofos. La sentencia

```
cout << '*' <<                                    '*';
```

produce la salida

```
**
```

A pesar de los espacios en blanco incluidos en la sentencia de salida, los asteriscos se imprimen lado a lado porque los espacios en blanco no están encerrados entre apóstrofos.

2.4 Introducción de programa, corrección y ejecución

Una vez que tiene un programa en papel, debe introducirlo por medio del teclado. En esta sección se examina el proceso de introducción de programa en general. Debe consultar el manual de su computadora para conocer los detalles.

Introducción de un programa

El primer paso para introducir un programa es lograr la atención de la computadora. Con una computadora personal, esto normalmente significa prenderla si no está encendida. Las estaciones de trabajo conectadas a una red por lo regular se dejan en funcionamiento todo el tiempo. Es necesario *iniciar el procedimiento de entrada al sistema* para poder tener acceso a la máquina. Esto significa introducir un nombre de usuario y una contraseña. El sistema de contraseña protege la información al evitar que la información que usted almacenó en la computadora sea trastocada o destruida por alguna otra persona.

Archivo Área nombrada en un almacenamiento secundario que se usa para contener un conjunto de datos; el conjunto mismo de datos.

Una vez que la computadora está lista para aceptar sus instrucciones, indíquele que desea introducir un programa solicitándole que ejecute el editor. El editor es un programa que permite crear y modificar programas al introducir información en un área del almacenamiento secundario de la computadora denominado **archivo**.

Un archivo en un sistema de computadora es como una carpeta de archivos en un archivero. Es un conjunto de datos relacionado con un nombre. Por lo común, se elige el nombre para el archivo cuando lo crea con el editor. A partir de ese punto, se hace referencia al archivo por el nombre que se le asignó.

Hay muchos tipos de editores, cada uno con características distintas, que no es posible comenzar a describirlos todos aquí. Sin embargo, se puede describir algunas de sus características generales.

La unidad básica de información en un editor es una pantalla llena de caracteres. El editor permite cambiar cualquier cosa que usted vea en la pantalla. Cuando crea un nuevo archivo, el editor limpia la pantalla para mostrarle que el archivo está vacío. Luego usted introduce su programa, con el ratón y el teclado para regresar y hacer correcciones según sea necesario. En la figura 2-3 se ilustra un ejemplo de la pantalla de un editor.

Compilación y ejecución de un programa

Una vez que su programa se almacena en un archivo, lo compila por medio de un comando para ejecutar el compilador de C++. El compilador traduce el programa, luego almacena la versión en lenguaje de máquina en un archivo. El compilador puede mostrar una ventana con mensajes que indican errores en el programa. Algunos sistemas permiten dar clic en un mensaje de error para ubicar automáticamente el cursor en la ventana del editor en el punto donde se detectó el error.

Si el compilador encuentra errores en su programa (errores de sintaxis), usted tiene que determinar la causa, ir de nuevo al editor y arreglarlos, y luego ejecutar de nuevo el compilador. Una vez que su programa compila sin errores, puede correrlo (ejecutarlo).

Algunos sistemas ejecutan de modo automático un programa cuando se compila con éxito. En otros sistemas se tiene que emitir un comando separado para ejecutar el programa. No obstante, otros

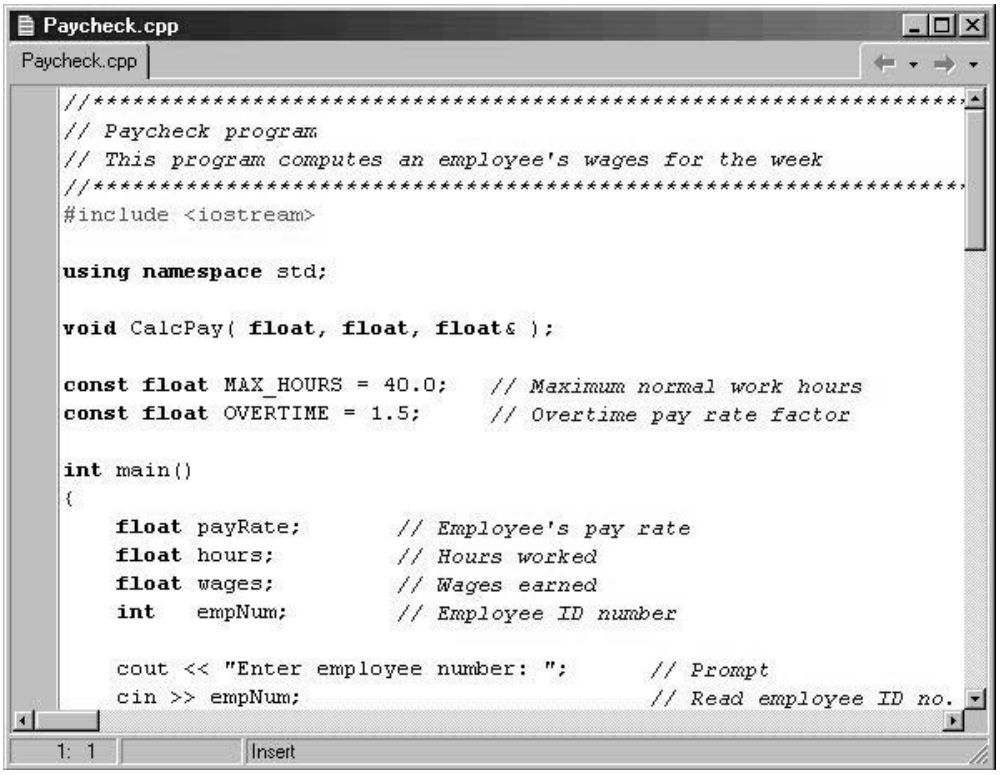


Figura 2-3 *Pantalla para un editor*

sistemas requieren que se especifique un paso extra llamado *ligado (enlace)* entre compilar y ejecutar un programa. Cualquiera que sea la serie de comando que use su sistema, el resultado es el mismo: su programa se carga en la memoria y la computadora lo ejecuta.

Aun cuando un programa pueda correr, suele tener errores en su diseño. La computadora hace exactamente lo que se le indica, incluso si eso no era lo que quería que hiciera. Si su programa no hace lo que debe (un *error lógico*), tiene que volver al algoritmo y arreglarlo, y luego ir al editor y componer el programa. Por último, usted compila y corre de nuevo el programa. Este proceso de *depuración* se repite hasta que el programa hace lo que se supone tiene que hacer (véase la figura 2-4).

Terminado

En una estación de trabajo, una vez que desea trabajar en su programa tiene que *salir del sistema* al proporcionar un comando con el ratón y el teclado. Esto libera a la estación de trabajo para que alguna otra persona pueda usarla. También evita que alguien intente manipular sus archivos.

En una computadora personal, una vez que realiza su trabajo guarda sus archivos y sale del editor. Al apagar la computadora se anula lo que hay en la memoria de la computadora, pero los archivos se guardan de modo seguro en el disco. Es una precaución sabia respaldar (hacer una copia de) periódicamente sus archivos de programa en un disco removible. Cuando un disco en una computadora experimenta una falla de hardware, con frecuencia es imposible recuperar los archivos. Con una copia de respaldo en un disquete, puede restaurar sus archivos al disco una vez que está reparado.

Asegúrese de leer el manual de su sistema y editor antes de introducir su primer programa. No se alarme si al principio tiene problemas, esto le pasa casi a todos. Con la práctica esto se hace mucho más fácil. Por esa razón es buena idea ir primero a través del proceso con un programa como PrintName, donde no importan los errores, a diferencia de una asignación de programación de clase.

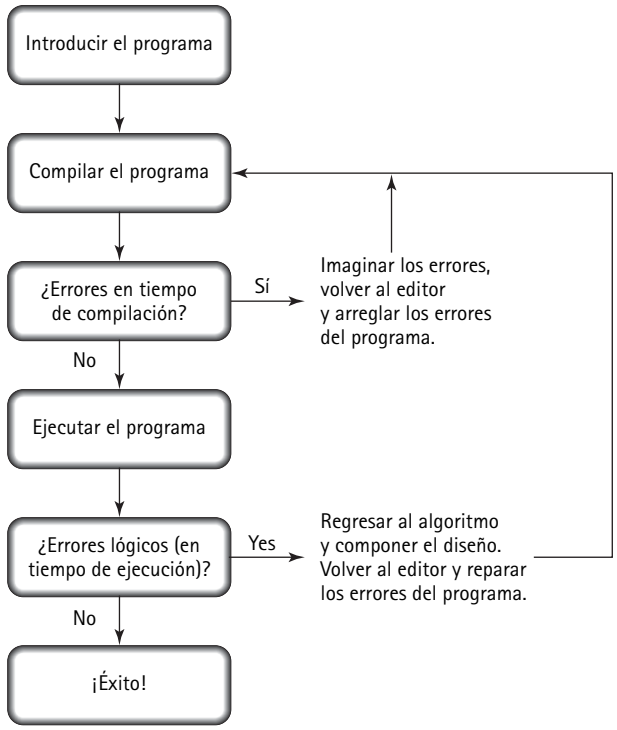


Figura 2–4 *Proceso de depuración*

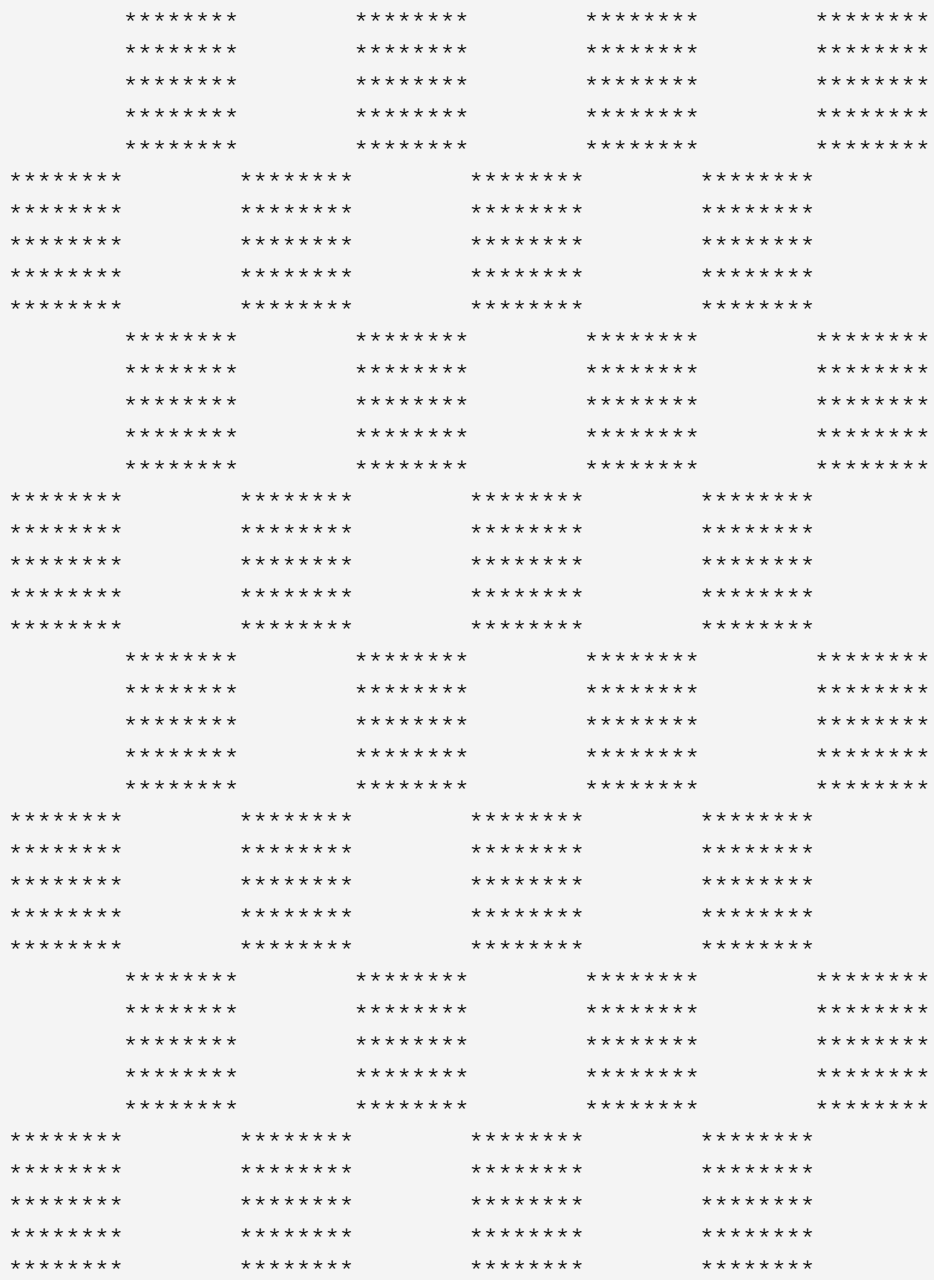
Caso práctico de resolución de problemas

Impresión de un tablero de ajedrez

PROBLEMA En su escuela se realiza un torneo de ajedrez y las personas que dirigen el torneo quieren registrar las posiciones finales de las piezas en cada juego sobre una hoja de papel con un tablero preimpreso en ella. Su trabajo es escribir un programa para preimprimir estas piezas de papel. El tablero es un patrón ocho por ocho de cuadros que alternan entre negro y blanco, donde el cuadro superior izquierdo es blanco. Se requiere imprimir cuadros de caracteres claros (espacios) y caracteres oscuros (por ejemplo, *) en este patrón para formar el tablero.

DISCUSIÓN Se podría escribir simplemente un programa que consta de una serie de sentencias de salida con patrones alternantes de espacios en blanco y asteriscos. Pero los organizadores no están exactamente seguros de la apariencia que desean para el tablero, y usted decide que sería más seguro escribir un programa de una manera que permite que el diseño del tablero se pueda modificar con facilidad.

Es fácil hacer un tablero modificable si el diseño se elabora por medio de variables de cadena. Puede comenzar por definir constantes de cadena para renglones de caracteres que forman las áreas negras y blancas. Luego, puede concatenar éstas para formar variables que contienen patrones alternantes de negro y blanco que se repiten para imprimir un renglón de cuadros de tablero de ajedrez. Por ejemplo, suponga que su diseño inicial para el tablero se parece a esto:



Puede comenzar por definir constantes para BLANCO y NEGRO que contengan ocho espacios en blanco y ocho asteriscos respectivamente, y puede concatenarlas en variables que siguen los patrones

```
BLANCO + NEGRO + BLANCO + NEGRO + BLANCO + NEGRO + BLANCO + NEGRO
```

y

```
NEGRO + BLANCO + NEGRO + BLANCO + NEGRO + BLANCO + NEGRO + BLANCO
```

que pueden imprimirse con sentencias cout la cantidad de veces apropiada para crear el tablero de ajedrez.

De esta discusión se sabe que hay dos constantes y dos variables, como se resume en las tablas siguientes:

Constantes

Nombre	Valor	Función
NEGRO	"********"	Caracteres que forman una línea de un cuadro negro
BLANCO	" "	Caracteres que forman una línea de un cuadro blanco

Variables

Nombre	Tipo de dato	Descripción
Renglón blanco	string	Un renglón que comienza con un cuadro blanco
Renglón negro	string	Un renglón que comienza con un cuadro negro

Si se observa con detenimiento el tablero de ajedrez, el algoritmo salta a la vista. Es necesario producir cinco renglones blancos, cinco renglones negros, cinco renglones blancos, cinco renglones negros, cinco renglones blancos, cinco renglones negros, cinco renglones blancos y cinco renglones negros. Se puede resumir esto en el algoritmo para producir los cinco renglones blancos y cinco renglones negros cuatro veces.

```
Repetir cuatro veces
   Producir cinco renglones blancos
   Producir cinco renglones negros
```

```
//*************************************************************
// Programa Tablero de ajedrez
// Este programa imprime un patrón de tablero de ajedrez que se
// construye a partir de cadenas básicas de caracteres blancos y negros.
//*************************************************************

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

const string BLACK = "********";  // Define una línea de un cuadro negro
const string WHITE = "        ";  // Define una línea de un cuadro blanco

int main ()
{
    string whiteRow;            // Un renglón que comienza con un cuadro blanco
    string blackRow;            // Un renglón que comienza con un cuadro negro

    // Crear un renglón blanco-negro al concatenar las cadenas básicas
    whiteRow = WHITE + BLACK + WHITE + BLACK +
               WHITE + BLACK + WHITE + BLACK;
    // Crear un renglón negro-blanco al concatenar las cadenas básicas
    blackRow = BLACK + WHITE + BLACK + WHITE +
               BLACK + WHITE + BLACK + WHITE;
```

```
// Imprimir cinco renglones blanco-negro
cout << whiteRow << endl;
cout << whiteRow << endl;
cout << whiteRow << endl;
cout << whiteRow << endl;
cout << whiteRow << endl;

// Imprimir cinco renglones negro-blanco
cout << blackRow << endl;
cout << blackRow << endl;
cout << blackRow << endl;
cout << blackRow << endl;
cout << blackRow << endl;

// Imprimir cinco renglones blanco-negro
cout << whiteRow << endl;
cout << whiteRow << endl;
cout << whiteRow << endl;
cout << whiteRow << endl;
cout << whiteRow << endl;

// Imprimir cinco renglones negro-blanco
cout << blackRow << endl;
cout << blackRow << endl;
cout << blackRow << endl;
cout << blackRow << endl;
cout << blackRow << endl;

// Imprimir cinco renglones blanco-negro
cout << whiteRow << endl;
cout << whiteRow << endl;
cout << whiteRow << endl;
cout << whiteRow << endl;
cout << whiteRow << endl;

// Imprimir cinco renglones negro-blanco
cout << blackRow << endl;
cout << blackRow << endl;
cout << blackRow << endl;
cout << blackRow << endl;
cout << blackRow << endl;

// Imprimir cinco renglones blanco-negro
cout << whiteRow << endl;
cout << whiteRow << endl;
cout << whiteRow << endl;
cout << whiteRow << endl;
cout << whiteRow << endl;

// Imprimir cinco renglones negro-blanco
cout << blackRow << endl;
cout << blackRow << endl;
```

```
    cout << blackRow << endl;
    cout << blackRow << endl;
    cout << blackRow << endl;

    return 0;
}
```

Existen ocho bloques de cinco líneas que lucen parecidos. En el capítulo 6, se introduce la sentencia looping que nos permite acortar considerablemente este programa.

Después que muestre la impresión de este programa a los organizadores del torneo, sugerirán que el tablero se vería mejor con el carácter "#" en lugar de "*" para llenar los cuadros negros. Usted acepta con agrado hacer este cambio porque sabe que la única diferencia en el programa es que el valor de la constante NEGRO se convierte en "########".

Prueba y depuración

1. Todo identificador que no es una palabra reservada en C++ debe ser declarada. Si usa un nombre que no ha sido declarado, ya sea mediante sus propias sentencias de declaración o al incluir un archivo de encabezado, obtiene un mensaje de error.
2. Si intenta declarar un identificador que es lo mismo que una palabra reservada en C++, se obtiene un mensaje de error del compilador. Véase en el apéndice A una lista de palabras reservadas.
3. C++ es un lenguaje sensible al uso de minúsculas y mayúsculas. Dos identificadores en los que las mayúsculas se emplean de manera distinta son tratados como dos identificadores distintos. La palabra `main` y todas las palabras reservadas en C++ usan sólo letras minúsculas.
4. Para usar identificadores de la biblioteca estándar, como `cout` y `string`, debe *a*) dar un nombre calificado como `std :: cout` o *b*) poner una directiva `using` cerca de la parte superior de su programa:

   ```
   using namespace std;
   ```

5. Compruebe comillas incompatibles en literales `char` y de cadena. Cada literal `char` comienza y termina con un apóstrofo (comilla simple). Cada literal de cadena comienza y termina con una comilla doble.
6. Asegúrese de usar sólo el apóstrofo (`'`) para encerrar literales `char`. La mayoría de los teclados tiene un apóstrofo inverso (`‘`), que se confunde fácilmente con el apóstrofo. Si usa el apóstrofo inverso, el compilador emite un mensaje de error.
7. Para usar una comilla doble en una cadena literal, use los dos símbolos `\"` en un renglón. Si usa sólo una comilla doble, termina la cadena, y entonces el compilador ve el resto de la cadena como un error.
8. En una declaración de asignación, asegúrese de que el identificador a la izquierda de = es una variable y no una constante nombrada.
9. Al asignar un valor a una variable `string`, la expresión a la derecha de = debe ser una expresión `string`, una cadena literal, o un `char`.
10. En una expresión de concatenación, por lo menos uno de los operandos de + debe ser del tipo `string`. Por ejemplo, los operandos no pueden ser cadenas literales o valores `char`.*
11. Asegúrese de que sus sentencias terminan en punto y coma (excepto sentencias compuestas, que no tienen punto y coma después de la llave derecha).

* La expresión de concatenación no válida `"Hi" + "there"` da como resultado un mensaje de error de sintaxis, como “INVALID POINTER ADDITION”. Esto puede ser confuso, en particular porque el tema de los apuntadores no se trata hasta mucho más adelante en este libro.

Resumen

La sintaxis (gramática) del lenguaje C++ se define por un metalenguaje. En este texto se usa una forma de metalenguaje llamada plantillas de sintaxis. Se describe en español la semántica (significado) de las sentencias de C++.

Los identificadores se usan en C++ para nombrar cosas. Algunos identificadores, denominados palabras reservadas, tienen significados predefinidos en el lenguaje; otros los crea el programador. Los identificadores que usted inventa están restringidos a los *no* reservados por el lenguaje C++. Las palabras reservadas se listan en el apéndice A.

Los identificadores se relacionan con localidades de la memoria mediante declaraciones. Una declaración puede dar un nombre a un lugar cuyo valor no cambia (una constante) o a uno cuyo valor puede cambiar (una variable). Toda constante y variable tiene relacionado un tipo de dato. C++ proporciona muchos tipos de datos integrados, de los cuales los más comunes son `int`, `float` y `char`. Adicionalmente, C++ permite tipos definidos por el programador, tales como `string` de la biblioteca estándar.

El operador de asignación se usa para cambiar el valor de una variable al asignarle el valor de una expresión. En tiempo de ejecución, la expresión se evalúa y el resultado se almacena en la variable. Con el tipo `string`, el signo (+) es un operador que concatena dos cadenas. Una expresión de cadena puede concatenar cualquier número de cadenas para formar un nuevo valor de `string` (cadena).

La salida de programa se efectúa por medio de la variable de flujo de salida `cout`, junto con el operador de inserción (<<). Cada operación de inserción envía datos de salida al dispositivo de salida estándar. Cuando un manipulador `end1` aparece en lugar de un ítem de salida, la computadora termina la línea de salida actual y va a la siguiente línea.

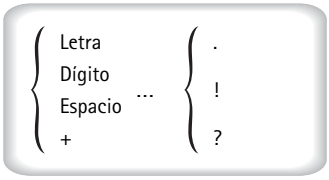
La salida debe ser clara, comprensible y estar dispuesta con nitidez. Los mensajes en la salida deben describir la importancia de los valores. Las líneas en blanco (producidas por usos sucesivos del manipulador `end1`) y los espacios en blanco dentro de las líneas ayudan a organizar la salida y mejorar la apariencia.

Un programa en C++ es una colección de una o más definiciones de función (y de modo optativo algunas declaraciones fuera de cualquier función). Una de las funciones *debe* llamarse `main`. La ejecución de un programa comienza siempre con la función `main`. Colectivamente, las funciones cooperan para producir los resultados deseados.

Comprobación rápida

1. ¿Cuál es el nombre de una función que debe tener todo programa de C++? (p. 38)
2. ¿Cuál es el propósito de un metalenguaje? (p. 41)
3. ¿Qué es un tipo de datos? (pp. 45 y 46)
4. Si descubre un error lógico en su programa, ¿va directamente al editor y comienza a cambiar el código? (pp. 64-65)
5. Use la siguiente plantilla de sintaxis para decidir si la cadena “C++ 4a edición” es una “oración” válida. (pp. 42-43)

Oración



6. ¿Cuáles de las siguientes expresiones son identificadores válidos de C++? (pp. 44 y 45)

```
Hello    Bob    4th-edition    C++    maximum    all_4_one
```

7. La única diferencia entre una declaración de variable y una declaración de constante es que la palabra reservada `const` precede a la declaración. ¿Verdadero o falso? (pp. 48-49)
8. Las palabras reservadas listadas en el apéndice A no pueden ser empleadas como identificadores. ¿Verdadero o falso? (p. 45)
9. Escriba una sentencia para asignar la letra "A" a la variable `char initial`. (pp. 51 y 52)
10. ¿Qué valor se almacena en la variable `name` mediante la siguiente sentencia? (pp. 53 y 54)

```
name = "Alexander" + " Q. " + "Smith";
```

11. Escriba una sentencia que envíe el valor de la variable `name` al flujo `cout`. (pp. 54-56)
12. ¿Qué se imprime mediante la siguiente sentencia, dado el valor asignado a `name` en el ejercicio 10? (pp. 55 y 56)

```
cout << name << " Jr." << endl;
```

13. ¿Cuáles son las dos formas de escribir comentarios en C++? (pp. 55 y 56)
14. Llene los espacios en blanco en las siguientes sentencias, que aparecen al comienzo de un programa. (pp. 57-59)

```
#include <__________>
#include <__________>

using namespace _______;
```

Respuestas

1. `main` **2.** Proporcionar un lenguaje para expresar las reglas de sintaxis de un lenguaje de programación. **3.** Es un conjunto específico de valores, junto con un conjunto de operaciones que pueden aplicarse a esos valores. **4.** No. Regresa al algoritmo, corrige la solución para el problema y luego traduce la corrección en código antes que use el editor para cambiar su programa. **5.** No. De acuerdo con la plantilla, una Oración debe terminar con un punto, signo de exclamación o de interrogación. **6.** `Hello, Bob, maximum y all_4_one` son identificadores válidos en C++. **7.** Falso. La declaración de constante asigna también un valor literal al identificador. **8.** Verdadero. Los identificadores difieren de palabras reservadas. **9.** `initial = 'A';` **10.** `Alexander Q. Smith` **11.** `cout << name;` **12.** Se imprime la cadena `Alexander Q. Smith Jr.` y el resultado sucesivo aparecerá en la línea siguiente. **13.** Entre los delimitadores /* y */, o después de //al final de la línea.

14.
```
#include <iostream>
#include <string>

using namespace std
```

Ejercicios de preparación para examen

1. Marque los siguientes identificadores como válidos o no válidos.

		Válido	No válido
a)	`theDog`	_____	_____
b)	`all-In-One`	_____	_____
c)	`const`	_____	_____
d)	`recycling`	_____	_____
e)	`DVD_ROM`	_____	_____
f)	`elizabeth_the_2nd`	_____	_____
g)	`2morrow`	_____	_____
h)	`page#`	_____	_____

2. Haga corresponder los siguientes términos con las definiciones dadas a continuación.
   a) Función
   b) Sintaxis
   c) Semántica
   d) Metalenguaje
   e) Identificador
   f) Tipo de datos
   g) Varible
   h) Constante nombrada
   i) Literal
   j) Expresión
      i) Identificador que se refiere a un valor que puede ser cambiado.
      ii) Conjunto específico de valores, junto con operaciones que pueden ser aplicadas a esos valores.
      iii) Valor que aparece en un programa.
      iv) Conjunto de reglas que determina el significado de las instrucciones escritas en un leguaje de programación.
      v) Lenguaje que se emplea para escribir las reglas para otro lenguaje.
      vi) Subprograma en C++.
      vii) Nombre que se usa para hacer referencia a una función u objeto de datos.
      viii) Disposición de identificadores, literales y operadores que puede ser evaluada.
      ix) Reglas formales mediante las que se rige la escritura de instrucciones válidas en un lenguaje de programación.
      x) Identificador que hace referencia a un valor que no puede ser cambiado.
3. Una palabra reservada es una constante nombrada que está predefinida en C++. ¿Verdadero o falso?
4. ¿Qué está mal con la siguiente plantilla de sintaxis para un identificador de C++?

5. Una literal `char` puede estar encerrada en comillas simples o dobles. ¿Verdadero o falso?
6. La cadena nula representa una cadena que no contiene caracteres. ¿Verdadero o falso?
7. La concatenación sólo funciona con un valor char cuando su otro operando es un valor de cadena. ¿Verdadero o falso?
8. ¿Cuál es el resultado de la siguiente sentencia?

```
cout << "Four score and" << endl << "seven years ago"
     << "our fathers" << endl
     << "brought forth on this" << endl
     << "continent a new nation..." << end
```

9. Dadas estas asignaciones a variables de cadena:

```
string1 = "Bjarne Stroustrup";
string2 = "C";
string3 = "programming language";
string4 = "++ because it is a successor to the ";
string5 = " named his new ";
```

¿Cuál es el valor de la siguiente expresión?

```
string1 + string5 + string3 + " " + string2 + string4 + string2 + " " +
string3 + "."
```

10. ¿Cómo se inserta una comilla doble en una cadena?
11. ¿Qué peligro se relaciona con el uso de la forma de comentarios /* y */ y cómo se evita al usar la forma de comentario //?
12. ¿Cuáles son las limitaciones asociadas con la forma de comentario //?
13. ¿Cuál es el nombre del operador <<, y cómo se pronunciaría al leer una línea de código que lo contenga?
14. ¿Se puede concatenar `end1` con una cadena en una expresión? Explique.
15. ¿Qué hace la directiva de preprocesador `#include`?
16. Si no incluye la línea

```
using namespace std;
```

al comienzo de un programa, ¿cómo debe cambiar la siguiente sentencia de salida para que funcione de manera correcta?

```
cout << "Hello everybody!" << endl;
```

17. ¿Cuál es el nombre de la construcción de C++ que empieza con { y termina con }?
18. ¿Cómo escribe una cadena que es demasiado larga y no cabe en una sola línea?
19. Reordene las siguientes líneas para construir un programa operante.

```
{
}
#include <iostream>

const string TITLE = "Dr.";
cout << "Hello " + TITLE + " Stroustrup!";
int main()
#include <string>
return 0;
using namespace std;
```

Ejercicios de preparación para la programación

1. Escriba una sentencia de salida que imprima la fecha de hoy y luego pase a la siguiente línea.
2. Escriba una sola sentencia de salida que produce las siguientes tres líneas:

```
He said, "How is that possible?"
She replied, "Using manipulators."
"Of course!" he exclaimed.
```

3. Escriba las declaraciones de lo siguiente:
   a) Una constante nombrada, llamada `RESPUESTA`, de tipo `string` con el valor `"Verdadero"`
   b) Una variable `char` con el nombre `middleInitial`
   c) Una variable `string` con el nombre `courseTitle`
   d) Una constante `char` nombrada, llamada `PORCIENTO`, con el valor `'%'`
4. Cambie las tres declaraciones del programa printName en la página 57 de modo que imprima su nombre.

5. Dadas las siguientes declaraciones:

```
const string PART1 = "Pro";
const string PART2 = "gramming and ";
const string PART3 = "blem Solving with C++";
```

escriba una sentencia de salida que imprima el título de este libro, usando sólo el flujo `cout`, el operador de inserción de flujo y las constantes nombradas arriba.

6. Escriba una expresión que dé como resultado una cadena que contenga el título de este libro, usando sólo el operador de concatenación y las declaraciones del ejercicio 5.
7. Escriba sentencias de salida C++ para imprimir lo siguiente exactamente como se muestra aquí (una porción del texto se encuentra en la página vi de este libro, que relaciona la acción del telar de una tejedora con la complejidad del pensamiento humano). Debe escribir la sentencia de salida para cada línea de texto.

```
Sin embargo la red del pensamiento no tiene tales pliegues
Y tiene más parecido con las obras maestras de un tejedor;
Un paso, miles de hilos surgen,
Aquí y allá dispara cada lanzadera,
Los hilos fluyen, invisibles y sutiles,
Cada golpe afecta miles de enlaces.
```

8. Llene las líneas que faltan en el siguiente programa, que produce:

```
Rev. H.G. Jones

_____________________
_____________________

_____________________

const string TITLE = "Rev. ";
const char FIRST = 'H';
const char MID 'G';
_____________________

_____________________
{
    cout << TITLE << FIRST << DOT << MID << DOT << " Jones";
_____________________
```

9. Introduzca y ejecute el siguiente programa. Asegúrese de escribirlo exactamente como aparece aquí, pero sustituya su nombre y la fecha como se indica.

```
//*************************************************************
// Ejercicio de introducción de programa y compilación
// Su nombre va aquí
// La fecha de hoy va aquí
//*************************************************************

#include <iostream>
#include <string>

using namespace std;

const string STARS35 = "***********************************";
const char   STAR = '*';
const string BLANKS33 = "                                 ";
```

```
const string MSG = "¡Bienvenido a la programación en C++!";
const string BLANKS3 = "    ";

int main()

{
     cout << STARS35 << endl << STAR << BLANKS33 << STAR << endl;
     cout << STAR << BLANKS3 << MSG << BLANKS3 << STAR << endl;
     cout << STAR << BLANKS33 << STAR << endl << STARS35 << endl;
}
```

Problemas de programación

1. Escriba un programa que imprima su horario de cursos para una sola semana. A continuación se da un ejemplo de la salida para un día:

   ```
   Lunes 9:00 Computación 101
   Lunes 11:00 Física 201
   Lunes 2:00 Danza 153
   ```

   Use constantes de cadena nombradas siempre que sea posible para evitar volver a escribir palabras o números. Asegúrese de incluir comentarios apropiados en su código, elija identificadores significativos y use sangrías como se hizo en los programas de este capítulo.
2. Escriba un programa que imprima las seis permutaciones del orden de las siguientes tres líneas. Declare una constante nombrada para cada línea y escriba una sentencia de salida para cada permutación. Asegúrese de incluir comentarios apropiados en su código, elija identificadores significativos y use sangrías como se hizo en los programas de este capítulo.

   ```
   Vi al gran oso café.
   El gran oso café me vio.
   ¡Oh! ¡Qué aterradora experiencia!
   ```

3. Escriba un programa que muestre un patrón de tablero de ajedrez hecho de estrellas y espacios en blanco, como se muestra a continuación. Un tablero de ajedrez consta de ocho cuadros por ocho cuadros. Esto será más fácil si primero declara dos constantes de cadena nombradas que representan a los dos patrones de renglones distintos. Asegúrese de incluir comentarios apropiados en su código, elija identificadores significativos y use sangrías como se hizo en los programas de este capítulo.

   ```
     * * * *
   * * * *
     * * * *
   * * * *
     * * * *
   * * * *
     * * * *
   * * * *
   ```

4. Escriba un programa que imprima tarjetas de presentación para usted mismo. Una tarjeta debe incluir su nombre, dirección, número o números telefónicos y dirección de correo electrónico. Para ahorrar papel, el programa debe imprimir ocho tarjetas por página, dispuestas en dos columnas de cuatro tarjetas. Así, para teclear menos, debe declarar una constante de cadena nombrada para cada línea de la tarjeta y luego escribir sentencias de salida para imprimir las ocho tarjetas con esas constantes. Asegúrese de incluir comentarios apropiados en su código, elija identificadores y use sangrías como se hace en los programas de este capítulo.

Seguimiento de caso práctico

1. ¿Qué cambio tendría que hacer al programa Tablero de ajedrez para hacer que los cuadros negros se impriman con el carácter "%" en lugar de "*"?
2. ¿Cómo cambiaría el programa Tablero de ajedrez para imprimir puntos en los cuadros blancos en lugar de espacios en blanco?
3. ¿Cómo cambiaría el programa Tablero de ajedrez si quisiera invertir los colores (es decir, hacer los cuadros negros blancos y viceversa) sin cambiar las declaraciones constantes o los valores de `whiteRow` y `blackRow`?
4. Cambie el programa Tablero de ajedrez para que los cuadrados sean de 10 caracteres de ancho por 8 renglones de alto.
5. Los organizadores del torneo de ajedrez tienen dificultades para escribir en los cuadros negros del tablero impreso. Quieren que usted cambie el programa de modo que los cuadros negros tengan un espacio en blanco en su centro, que es de cuatro caracteres de ancho y dos líneas de alto. (*Sugerencia:* tiene que definir otra constante de cadena.)
6. ¿Cuántos caracteres se almacenan en cada una de las variables de cadena en el programa Tablero de ajedrez?

CAPÍTULO

3

Tipos numéricos, expresiones y salida

Objetivos

Objetivos de conocimiento

- *Entender la coerción tipo implícita y la conversión tipo explícita.*
- *Reconocer y entender el propósito de los argumentos de función.*
- *Aprender a usar operaciones adicionales relacionadas con el tipo* `string`.
- *Aprender cómo formatear sentencias de programa de una manera clara y legible.*

Objetivos de habilidades

Ser capaz de:

- *Declarar constantes y variables nombradas de tipo* `int` *y* `float`.
- *Construir expresiones aritméticas simples.*
- *Evaluar expresiones aritméticas simples.*
- *Construir y evaluar expresiones que contienen operaciones aritméticas múltiples.*
- *Llamar (invocar) una función que devuelve un valor.*
- *Usar funciones de biblioteca C++ en expresiones.*
- *Llamar (invocar) una función void (una que no devuelve un valor de función).*
- *Usar manipuladores de C++ para formatear la salida.*

En el capítulo 2 se examinó la suficiente sintaxis de C++ para poder elaborar programas sencillos por medio de asignación y salida. Se centró la atención en los tipos `char` y `string` y se vio cómo construir expresiones con el operador de concatenación. En este capítulo se continúa con la escritura de programas que usan asignación y salida, pero se pone atención a tipos de datos integrados adicionales: `int` y `float`. Estos tipos numéricos son apoyados por numerosos operadores que permiten construir expresiones aritméticas complejas. Se muestra cómo hacer expresiones incluso más poderosas mediante funciones de biblioteca, funciones prescritas que son parte de todo sistema C++ y están disponibles para ser usadas por cualquier programa.

También se regresa al tema de formatear la salida. En particular, se consideran las características especiales que C++ proporciona para formatear números en la salida. Se termina considerando algunas operaciones adicionales en datos `string`.

3.1 Repaso de tipo de datos de C++

Los tipos de datos integrados C++ están organizados en tipos simples, tipos estructurados y tipos de dirección (véase la figura 3-1). No se sienta abrumado por la cantidad de tipos de datos mostrados en la figura. El propósito es ofrecer una representación global de lo que está disponible en C++. En este capítulo se pone atención en los tipos integral y flotante. Los detalles de los otros tipos vienen más adelante en el libro. Primero se consideran los tipos integrales (los que se emplean sobre todo para representar enteros), y luego se examinan los tipos flotantes (usados para representar números reales que contienen puntos decimales).

3.2 Tipos de datos numéricos

Ya se está familiarizado con los conceptos básicos de números enteros y reales en matemáticas. Sin embargo, cuando se usan en una computadora, los tipos de datos correspondientes tienen ciertas limitaciones, que se consideran ahora.

Tipos integrales

Los tipos de datos `char`, `short`, `int` y `long` se conocen como tipos integrales (o enteros) porque se refieren a valores enteros, números enteros sin parte fraccionaria. (Se pospone hablar acerca del tipo integral restante, `bool`, hasta el capítulo 5.)

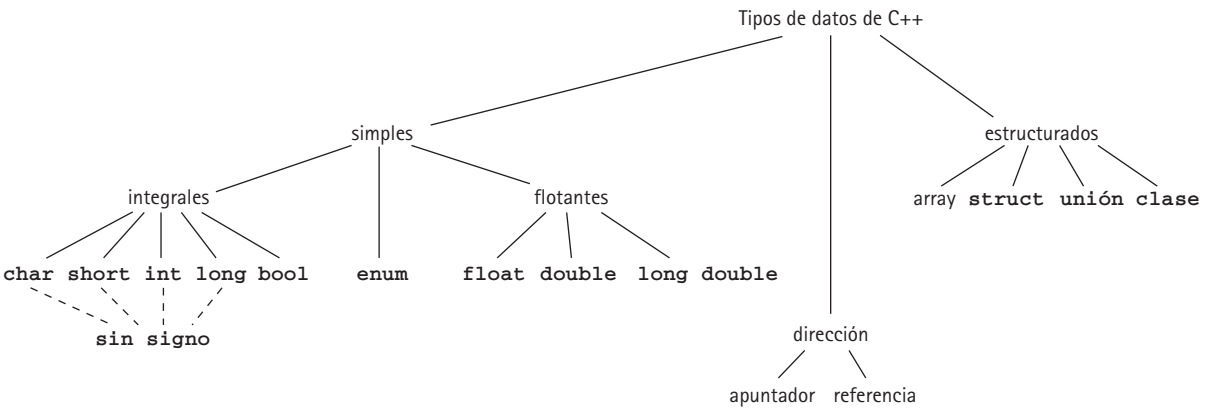


Figura 3-1 *Tipos de datos en C++*

En C++, la forma más simple de valor entero es una secuencia de uno o más dígitos:

```
22   16   1   498   0   4600
```

No se permiten comas.

En la mayor parte de los casos, un signo menos que precede a un valor entero lo hace negativo:

```
-378  -912
```

La excepción es cuando se añade de modo explícito la palabra reservada `unsigned` al nombre de tipo de datos:

```
unsigned int
```

Se supone que un valor entero `unsigned` es sólo positivo o cero. Los tipos `unsigned` se emplean sobre todo en situaciones especializadas. En este libro rara vez se usa `unsigned`.

Los tipos de datos `char`, `short`, `int` y `long` tienen como fin representar tamaños distintos de enteros, de menor (pocos bits) a más grande (más bits). Los tamaños dependen de la máquina (es decir, pueden variar de una máquina a otra). Para una determinada máquina, se podrían ilustrar los tamaños de la siguiente forma:

En otra máquina, el tamaño de un tipo `int` debe ser el mismo que el tamaño de uno `long`. En general, mientras más bits haya en la celda de memoria, más grande es el entero que se puede almacenar.

Aunque en el capítulo 2 se usó el tipo `char` para guardar datos de caracteres como `'A'`, hay razones de por qué C++ clasifica `char` como un tipo integral. En el capítulo 10 se examinan las razones.

`int` es por mucho el tipo de dato más común para manejar datos enteros. En el programa del año bisiesto del capítulo 1, el identificador, `year`, es del tipo de datos `int`. Casi siempre se usa `int` para manejar valores enteros, pero algunas veces es necesario usar `long` si el programa requiere valores más grandes que el valor `int` máximo. (En algunas computadoras personales, el intervalo de valores `int` va de `-32768` hasta `+32767`. Más comúnmente, `int` varía de `-2147483648` a `+2147483647`.) Si su programa intenta calcular un valor más grande que el valor máximo de su máquina, el resultado es *integer overflow* (desbordamiento). Algunas máquinas dan un mensaje de error cuando hay desbordamiento, pero otras no. En capítulos posteriores se habla más de desbordamiento.

Una precaución acerca de valores enteros en C++: una constante literal que comienza con un cero se considera como un número octal (base 8) en lugar de un número decimal (base 10). Si escribe

```
015
```

el compilador C++ toma esto para promediar el número decimal 13. Si no está familiarizado con el sistema de números octales, no se preocupe de por qué un 15 octal es lo mismo que un 13 decimal. Lo importante a recordar es no iniciar una constante entera decimal con un cero (a menos que sólo quiera el número 0, que es el mismo en la base octal y decimal). En el capítulo 10 se analizan con más detalle los distintos tipos integrales.

Tipos de punto flotante

Los tipos de punto flotante (o tipos flotantes), la segunda categoría principal de tipos simples en C++, se emplean para representar números reales. Los números de punto flotante tienen una parte entera y una parte fraccionaria, con un punto decimal entre ellas. Puede faltar la parte entera o la fraccionaria, pero no ambas. Aquí se dan algunos ejemplos:

```
18.0   127.54   0.57   4.   193145.8523   .8
```

Si `0.57` empieza con cero no es un número octal. Es sólo con valores enteros que un cero principal indica un número octal.

Así como los tipos integrales en C++ vienen en tamaños distintos (`char`, `short`, `int` y `long`), también los tipos de punto flotante. En orden creciente de tamaño, los tipos de punto flotante son `float`, `double` (que significa doble precisión) y `long double`. De nuevo, los tamaños exactos dependen de la máquina. Cada tamaño más grande da potencialmente un intervalo más amplio de valores y más precisión (la cantidad de dígitos significativos en el número), pero a expensas de más espacio para contener el número.

Los valores de punto flotante también pueden tener un exponente, como en la notación científica. (En notación científica, un número se escribe como un valor multiplicado por 10 elevado a alguna potencia.) En lugar de escribir 3.504×10^{12}, en C++ se escribe `3.504E12`. La `E` significa exponente de base 10. El número que precede a la letra `E` puede no incluir un punto decimal. A continuación se dan algunos ejemplos de números de punto flotante en notación científica.

```
1.74536E-12   3.652442E4   7E20
```

La mayoría de los programas no requiere tipos `double` o `long double`. El tipo `float` en general proporciona precisión suficiente e intervalo de valores para números de punto flotante. La mayoría de las computadoras personales proporcionan valores `float` con una precisión de seis o siete dígitos significativos y un valor máximo de aproximadamente `3.4E+38`. Se usan valores de punto flotante para representar dinero y tasas de interés en el caso práctico al final del capítulo.

En el capítulo 10 se habla más acerca de los números de punto flotante. Pero hay algo más que debe saber acerca de ellos ahora. Las computadoras no pueden representar siempre números de punto flotante de modo exacto. En el capítulo 1 se aprendió que la computadora almacena los datos en forma binaria (base 2). Muchos valores de punto flotante sólo pueden ser aproximados en el sistema de números binarios. No se sorprenda si su programa imprime el número 4.8 como 4.7999998. En la mayoría de los casos, se esperan ligeras inexactitudes en los dígitos fraccionarios más a la derecha y no son resultado de error del programador.

3.3 Declaraciones para tipos numéricos

Así como con los tipos `char` y `string`, se pueden declarar constantes y variables nombradas de tipo `int` y `float`. Tales declaraciones usan la misma sintaxis que antes, excepto que las literales y los nombres de los tipos de datos son diferentes.

Declaraciones constantes nombradas

En el caso de declaraciones constantes nombradas, los valores literales en las declaraciones son numéricos en lugar de ser caracteres en comillas simples o dobles. Por ejemplo, aquí están algunas declaraciones constantes que definen valores de tipo `int` y `float`. Por comparación, se incluyen valores de las declaraciones `char` y `string`.

```
const float   PI = 3.14159;
const float   E = 2.71828;
const int     MAX_SCORE = 100;
const int     MIN_SCORE = -100;
const char    LETTER = 'W';
const string NAME = "Elizabeth";
```

Aunque las literales de carácter y cadena se escriben entre comillas, no sucede lo mismo con los enteros literales y los números de punto flotante, ya que no hay manera de confundirlos con identificadores. ¿Por qué? Porque los identificadores deben empezar con una letra o guión bajo y los números deben empezar con un dígito o signo.

Consejo práctico de ingeniería de software

Uso de constantes nombradas en lugar de literales

Es una buena idea usar constantes nombradas en lugar de literales. Además de hacer su programa más legible, las constantes nombradas pueden hacer que la modificación de su programa sea más fácil. Suponga que escribió un programa el año pasado que calcula impuestos. En varios lugares usó la literal 0.05, que fue la tasa de impuestos para las ventas en aquel entonces. Ahora la tasa ha subido a 0.06. Para cambiar su programa, debe localizar toda literal 0.05 y cambiarla a 0.06. Y si 0.05 se emplea por alguna otra razón para calcular deducciones, por ejemplo, necesita examinar cada lugar donde se usó, recordar para qué se empleó y luego decidir si la cambia.

El proceso es mucho más simple si usa una constante nombrada. En lugar de usar una constante literal, suponga que ha declarado una constante nombrada, `TAX_RATE`, con un valor de 0.05. Para cambiar su programa, simplemente cambiaría la declaración, y fijaría `TAX_RATE` igual a 0.06. Esta modificación cambia todos los cálculos de tasa de impuestos sin afectar los otros lugares donde se usa 0.05.

C++ permite declarar constantes con diferentes nombres pero el mismo valor. Si un valor tiene significados distintos en diferentes partes de un programa, tiene sentido declarar y usar una constante con un nombre apropiado para cada significado.

Las constantes nombradas también son confiables; protegen de errores. Si escribe mal el nombre `PI` como `PO`, el compilador C++ le dice que el nombre `PO` no ha sido declarado. Por otro lado, aunque se reconozca que el número 3.14149 es una versión mal introducida de pi (3.1416), el número es perfectamente aceptable para el compilador. No advierte que algo está mal.

Declaraciones de variables

Las variables numéricas se declaran de la misma manera que las variables `char` y `string`, excepto que usan los nombres de tipos numéricos. Las siguientes son declaraciones válidas para variables:

```
int    studentCount;   // Número de alumnos
int    sumOfScores;    // Suma de sus puntuaciones
float  average;        // Promedio de las puntuaciones
char   grade;          // Calificación del alumno
string stuName;        // Nombre del alumno
```

Dadas las declaraciones

```
int   num;
int   alpha;
float rate;
char  ch;
```

las siguientes son sentencias de asignación apropiadas:

Variable	Expresión
`alpha =`	`2856;`
`rate =`	`0.36;`
`ch =`	`'B';`
`num =`	`alpha`

En cada una de estas sentencias de asignación, el tipo de datos de la expresión concuerda con el de la variable a la que se asignó. Más adelante, en este capítulo, se explica lo que sucede si no concuerdan los tipos de datos.

3.4 Expresiones aritméticas simples

Ahora que se ha examinado la declaración y la asignación, se considera cómo calcular con valores de tipos numéricos. Los cálculos se efectúan con expresiones. Primero se consideran las expresiones simples que tienen que ver con a lo sumo un operador de modo que se examine cada operador en detalle. Luego, se pasa a las expresiones compuestas que combinan operaciones múltiples.

Operadores aritméticos

Las expresiones están constituidas por constantes, variables y operadores. Las siguientes son expresiones válidas:

```
alpha + 2     rate - 6.0     4 - alpha     rate     alpha * num
```

Los operadores permitidos en una expresión dependen de los tipos de datos de las constantes y variables en la expresión. Los *operadores aritméticos* son

+	Signo positivo unario
-	Signo negativo unario
+	Suma
-	Resta
*	Multiplicación
/	División de punto flotante (resultado de punto flotante)
/	División de enteros (sin parte fraccionaria)
%	Módulo (residuo de la división de enteros)

Operador unario Operador que sólo tiene un operando.
Operador binario Operador que tiene dos operandos.

Los dos primeros operadores son **operadores unarios**, toman sólo un operando. Los cinco restantes son **operadores binarios**, toman dos operandos. Los signos unarios positivo y negativo se usan como sigue:

```
-54    +259.65    -rate
```

Los programadores rara vez usan el más unario. Sin signo, se supone de cualquier manera que una constante numérica es positiva.

Quizá no esté familiarizado con la división de enteros y el módulo (%). Examinémoslos más de cerca. Observe que el signo % se emplea sólo con enteros. Cuando divide un entero entre otro, se obtiene un cociente entero y un residuo. La división de enteros da sólo el cociente entero, y % da sólo el residuo. (Si el operando es negativo, el signo del residuo puede variar de un compilador a otro de C++.)

```
  3  ← 6 / 2        3  ← 7 / 2
2 )6             2 )7
  6                 6
  0  ← 6 % 2        1  ← 7 % 2
```

En contraste, la división de punto flotante produce un resultado de punto flotante. La expresión

```
7.0 / 2.0
```

produce el valor 3.5.

A continuación se dan algunas expresiones con operadores aritméticos y sus valores:

Expresión	Valor
`3 + 6`	9
`3.4 - 6.1`	–2.7
`2 * 3`	6
`8 / 2`	4
`8.0 / 2.0`	4.0
`8 / 8`	1
`8 / 9`	0
`8 / 7`	1
`8 % 8`	0
`8 % 9`	8
`8 % 7`	1
`0 % 7`	0
`5 % 2.3`	error (ambos operandos deben ser enteros)

Tenga cuidado con la división y el módulo. Las expresiones `7.0 / 0.0`, `7 / 0` y `7 % 0` producen errores. La computadora no puede dividir entre cero.

Debido a que en las expresiones se permiten variables, las siguientes son asignaciones válidas:

```
alpha = num + 6;
alpha = num / 2;
num = alpha * 2;
num = 6 % alpha;
alpha = alpha + 1;
num = num + alpha;
```

Como se vio con las sentencias de asignación relacionadas con expresiones `string`, la misma variable puede aparecer en ambos lados del operador de asignación. En el caso de

```
num = num + alpha;
```

el valor en `num` y el valor en `alpha` se suman y, después, la suma de los dos valores se almacena de nuevo en `num`, remplazando el valor previo almacenado ahí. Este ejemplo muestra la diferencia entre igualdad matemática y asignación. La igualdad matemática

num = num + alpha

es verdadera sólo cuando *alpha* es igual a 0. La sentencia de asignación

```
num = num + alpha;
```

es válida para *cualquier* valor de `alpha`.

A continuación se muestra un programa simple que usa expresiones aritméticas:

```
//*****************************************************************
// Programa FreezeBoil
// Este programa calcula el punto medio entre
// los puntos de congelación y ebullición del agua
//*****************************************************************
#include <iostream>

using namespace std;

const float FREEZE_PT = 32.0;  // Punto de congelación del agua
const float BOIL_PT = 212.0;   // Punto de ebullición del agua

int main()
{
    float avgTemp;             // Contiene el resultado de promediar
                               //   FREEZE_PT y BOIL_PT

    cout << "El agua se congela a " << FREEZE_PT << endl;
    cout << "y hierve a " << BOIL_PT << "grados. " << endl;

    avgTemp = FREEZE_PT + BOIL_PT;
    avgTemp = avgTemp / 2.0;

    cout << "La mitad es ";
    cout << avgTemp << " grados. " << endl;

    return 0;
}
```

El programa comienza con un comentario que explica lo que hace el programa. A continuación viene una sección de declaraciones donde se definen las constantes `FREEZE_PT` y `BOIL_PT`. El cuerpo de la función `main` incluye una declaración de la variable `avgTemp` y luego una serie de sentencias ejecutables. Estas sentencias imprimen un mensaje, suman `FREEZE_PT` y `BOIL_PT`, dividen la suma entre 2 y, por último, imprimen el resultado.

Operadores de incremento y decremento

Además de los operadores aritméticos, C++ provee los operadores de *incremento* y *decremento*

`++`	Incremento
`--`	Decremento

Éstos son operadores unarios que toman un solo nombre de variable como un operando. Para operandos de punto flotante, el efecto es sumar 1 a (o restar 1 de) el operando. Si `num` contiene actualmente el valor 8, la sentencia

```
num++;
```

ocasiona que `num` contenga a 9. Se puede lograr el mismo efecto al escribir la sentencia de asignación

```
num = num + 1;
```

pero los programadores de C++ prefieren, en general, el operador de incremento. (Recuerde del capítulo 1 cómo obtuvo su nombre el lenguaje C++: C++ es una versión mejorada ["incrementada"] del lenguaje C.)

Los operadores `++` y `--` pueden ser operadores *prefijos*

```
++num;
```

o posfijos

```
num++;
```

Ambas sentencias se comportan exactamente de la misma forma; suman 1 a lo que esté en `num`. La elección entre los dos es cuestión de preferencia personal.

C++ permite el uso de `++` y `--` a la mitad de una expresión más grande:

```
alpha = num++ * 3;
```

En este caso, la forma posfija de `++` *no* da el mismo resultado que la forma prefija. En el capítulo 10 se explican en detalle los operadores `++` y `--`. Mientras tanto, debe usarlos sólo para incrementar o disminuir una variable como una sentencia independiente, separada:

IncrementStatement

Variable ++ ;

++ Variable ;

DecrementStatement

Variable -- ;

-- Variable ;

3.5 Expresiones aritméticas compuestas

Las expresiones que se han usado hasta aquí contienen a lo sumo un operador aritmético. También se ha tenido cuidado de no combinar valores enteros y de punto flotante en la misma expresión. Ahora se consideran expresiones más complicadas, compuestas de varios operadores y que contienen tipos de datos mixtos.

Reglas de precedencia

Las expresiones aritméticas pueden estar constituidas por muchas constantes, variables, operadores y paréntesis. ¿En qué orden se ejecutan las operaciones? Por ejemplo, en la sentencia de asignación

```
avgTemp = FREEZE_PT + BOIL_PT / 2.0;
```

¿se calcula primero `FREEZE_PT + BOIL_PT` o `BOIL_PT / 2.0`?

Los operadores aritméticos básicos (unario +, unario -, + para suma, - para resta, * para multiplicación, / para división y % para módulo) son ordenados de la misma forma que los operadores matemáticos, según las *reglas de precedencia*:

Nivel de precedencia más alto:	Unario + Unario -
Nivel medio:	* / %
Nivel mínimo:	+ -

Debido a que la división tiene precedencia más alta que la suma, la expresión del ejemplo anterior se encierra entre paréntesis de manera implícita como

```
FREEZE_PT + (BOIL_PT / 2.0)
```

Es decir, se divide primero `BOIL_PT` entre 2.0 y luego se suma `FREEZE_PT` al resultado.

Puede cambiar el orden de evaluación por medio de paréntesis. En la sentencia

```
avgTemp = (FREEZE_PT + BOIL_PT) / 2.0;
```

`FREEZE_PT` y `BOIL_PT` se suman primero, y luego su suma se divide entre 2.0. Se evalúan primero las subexpresiones en el paréntesis y luego sigue la precedencia de los operadores.

Cuando una expresión aritmética tiene varios operadores binarios con la misma precedencia, su *orden de agrupamiento* (o *asociatividad*) es de izquierda a derecha. La expresión

```
int1 - int2 + int3
```

significa (`int1 - int2`) + `int3`, no `int1` - (`int2` + `int3`). Como otro ejemplo, se usaría la expresión

```
(float1 + float2) / float1 * 3.0
```

para evaluar primero la expresión entre paréntesis, dividir después la suma entre `float1`, y multiplicar el resultado por 3.0. A continuación se ilustran algunos ejemplos más.

Expresión	Valor
`10 / 2 * 3`	15
`10 % 3 - 4 / 2`	–1
`5.0 * 2.0 / 4.0 * 2.0`	5.0
`5.0 * 2.0 / (4.0 * 2.0)`	1.25
`5.0 + 2.0 / (4.0 * 2.0)`	5.25

En C++, los operadores unarios (como unario + y unario -) tienen asociatividad de derecha a izquierda. Aunque al principio esto podría parecer extraño, resulta ser el orden de agrupamiento natural. Por ejemplo, `- + x` significa `- (+ x)` y no el orden `(- +) x` carente de significado.

Coerción y conversión de tipo (Moldeo de tipo)

Los valores enteros y los valores de punto flotante se almacenan de modo diferente en la memoria de una computadora. El patrón de bits que representa la constante 2 no se parece en absoluto al patrón de bits que representa a la constante 2.0. (En el capítulo 10 se examina por qué los números de punto flotante necesitan una representación especial dentro de la computadora.) ¿Qué sucede si se combinan valores enteros y de punto flotante en una sentencia de asignación o una expresión aritmética? Considérense primero las sentencias de asignación.

Sentencias de asignación Si hace las declaraciones

```
int   someInt;
float someFloat;
```

entonces `someInt` puede contener *sólo* valores enteros y `someFloat` puede contener *sólo* valores de punto flotante. La sentencia de asignación

```
someFloat = 12;
```

podría parecer que almacena el valor entero 12 en `someFloat`, pero esto no es cierto. La computadora rechaza almacenar cualquier otra cosa que no sea un valor `float` en `someFloat`. El compilador inserta instrucciones de lenguaje de máquina extra que primero convierten a 12 en 12.0 y luego almacenan a 12.0 en `someFloat`. Esta conversión implícita (automática) de un valor de un tipo de datos a otro se conoce como **coerción de tipo**.

Coerción de tipo Conversión implícita (automática) de un valor de un tipo de dato a otro.

La sentencia

```
someInt = 4.8;
```

también causa coerción de tipo. Cuando se asigna un valor de punto flotante a una variable `int`, se trunca la parte fraccionaria. Como reusltado, a `someInt` se le asigna el valor 4.

Con las dos sentencias de asignación anteriores, el programa resulta menos confuso para que alguien lo lea si se evita combinar tipos de datos:

```
someFloat = 12.0;
someInt = 4;
```

La mayoría de las veces, no son sólo constantes sino expresiones enteras las que intervienen en la coerción de tipo. Las dos asignaciones siguientes

```
someFloat = 3 * someInt + 2;
someInt = 5.2 / someFloat - anotherFloat;
```

conducen a tipo de coerción. Almacenar el resultado de una expresión `int` en una variable `float` en general no causa pérdida de información; un número entero como 24 se puede representar en forma de punto flotante como 24.0. Sin embargo, almacenar el resultado de una expresión de punto flotante en una variable `int` puede causar pérdida de información porque se trunca la parte fraccionaria. Es fácil pasar por alto la asignación de una expresión de punto flotante para una variable `int` cuando se intenta descubrir por qué un programa produce respuestas erróneas.

Para hacer los programas lo más claro posible (y libres de errores), se puede usar el moldeo de **tipos explícito** (o **conversión de tipos**). Una *operación de moldeo* en C++ consiste en nombrar tipos de datos y luego, entre paréntesis, la expresión por convertir

Moldeo de tipos Conversión explícita de un valor de un tipo de datos a otro; conocida también como conversión de tipos.

```
someFloat = float(3 * someInt + 2);
someInt = int(5.2 / someFloat - anotherFloat);
```

Ambas sentencias

```
someInt = someFloat + 8.2;
someInt = int(someFloat + 8.2);
```

producen resultados idénticos. La única diferencia es en claridad. Con la operación de conversión, es perfectamente claro para el programador y otros que leen el programa que la combinación de tipos es intencional, no un descuido. Han resultado errores incontables de la combinación no intencional de tipos.

Observe que hay una forma sutil de redondear en vez de truncar un valor de punto flotante antes de almacenarlo en una variable `int`. A continuación se muestra la forma de hacerlo:

```
someInt = int(someFloat + 0.5);
```

Con lápiz y papel, vea por sí mismo lo que se almacena en `someInt` cuando `someFloat` contiene 4.7. Ahora pruebe de nuevo, suponiendo que `someFloat` contiene 4.2. (Con esta técnica de redondeo sumando 0.5 se supone que `someFloat` es un número positivo.)

Expresiones aritméticas Hasta aquí se ha hablado acerca de combinar tipos de datos a través del operador de asignación (=). También es posible combinar tipos de datos dentro de una expresión:

```
someInt * someFloat
4.8 + someInt - 3
```

Expresión de tipo mixto Expresión que contiene operandos de tipos de datos diferentes; llamada también expresión de modo mixto.

Esta clase de expresiones se conoce como **expresiones de tipo mixto** (o **modo mixto**).

Siempre que un valor entero y un valor de punto flotante están unidos por un operador, la coerción de tipos implícita ocurre como sigue.

1. El valor entero es forzado de forma temporal a un valor de punto flotante.
2. Se ejecuta la operación.
3. El resultado es un valor de punto flotante.

Observe cómo la máquina evalúa la expresión `4.8 + someInt - 3`, donde `someInt` contiene el valor 2. Primero, los operandos del operador + tienen tipos mixtos, así que el valor de `someInt` es forzado a 2.0. (Esta conversión es sólo temporal; no afecta al valor que se almacena en `someInt`.) La suma se realiza y produce un valor de 6.8. A continuación, el operador de resta (-) une un valor de punto flotante (6.8) y un valor entero (3). El valor 3 es forzado a 3.0, la resta se efectúa y el resultado es el valor de punto flotante 3.8.

Del mismo modo que con las sentencias de asignación, se pueden usar moldeos explícitos de tipos dentro de las expresiones para reducir el riesgo de errores. Escribir expresiones como

```
float(someInt) * someFloat
4.8 + float(someInt - 3)
```

aclara cuáles son sus intenciones.

Los moldeos explícitos de tipos no sólo son valiosos para claridad del programa, sino que en algunos casos son obligatorios para la programación correcta. Dadas las sentencias

```
int   sum;
int   count;
float average;
```

suponga que `sum` y `count` contienen 60 y 80, respectivamente. Si `sum` representa la suma de un grupo de valores enteros y `count` representa el número de valores, determínese el valor promedio:

```
average = sum / count;      // Wrong
```

Infortunadamente, esta sentencia almacena el valor 0.0 en `average`. La razón es que la expresión a la derecha del operador de asignación no es una expresión de tipo mixto. Ambos operandos del ope-

rador / son de tipo `int`, así que se efectúa la división de enteros. 60 dividido entre 80 produce el valor entero 0. A continuación, la máquina obliga de modo implícito a 0 al valor 0.0 antes de almacenarlo en `average`. La manera de hallar el promedio de forma correcta y clara es ésta:

```
average = float(sum) / float(count);
```

Esta sentencia da una división de punto flotante en lugar de una división entera. Como resultado, el valor 0.75 se almacena en `average`.

Como observación final acerca de la coerción y conversión de tipos, quizá haya observado que sólo se enfocó la atención en los tipos `int` y `float`. También es posible mezclar en la olla los valores `char`, `short` y `double`. Los resultados pueden ser confusos e inesperados. En el capítulo 10 se vuelve al tema con una descripción más detallada. Mientras, se debe evitar combinar valores de estos tipos dentro de una expresión.

Conozca a...

Blaise Pascal

Una de las grandes figuras históricas en el mundo de la computación fue el matemático y filósofo religioso francés Blaise Pascal (1623-1662), inventor de una de las primeras calculadoras mecánicas conocidas.

El padre de Pascal, Etienne, fue un noble de la corte francesa, recaudador de impuestos y matemático. La madre de Pascal murió cuando éste tenía 3 años de edad. Cinco años después, la familia se trasladó a París y Etienne se hizo cargo de la educación de los hijos. Rápidamente Pascal mostró talento para las matemáticas. A la edad de 17 años publicó un ensayo matemático que ganó la envidia de René Descartes, uno de los fundadores de la geometría moderna. (En realidad el trabajo de Pascal se había completado antes de que cumpliera los 16 años de edad.) Se basó en un teorema, al que llamó *hexagrammum mysticum* o hexagrama místico, que describió la inscripción de hexágonos en secciones cónicas (parábolas, hipérbolas y elipses). Además del teorema (ahora conocido como teorema de Pascal), su ensayo incluyó más de 400 corolarios.

Cuando Pascal tenía cerca de 20 años de edad, construyó una calculadora mecánica que realizaba la suma y resta de números de ocho dígitos. Esa calculadora requería que el usuario marcara los números a sumar o restar; entonces la suma o diferencia aparecía en un conjunto de ventanas. Se cree que su motivación para construir esta máquina fue ayudar a su padre en la recaudación de impuestos. La primera versión de la máquina dividía los números en seis dígitos decimales y dos dígitos fraccionarios, ya que se usaría para calcular sumas de dinero. La máquina fue aclamada por sus contemporáneos como un gran avance en matemáticas, y Pascal construyó varias más en diferentes formas. Logró tal popularidad que otros construyeron muchas copias falsas e inútiles mostradas como novedades. En algunos museos aún existen varias calculadoras de Pascal.

La caja de Pascal, como se llamó, fue considerada por mucho tiempo la primera calculadora mecánica. Sin embargo, en 1950, se descubrió una carta de Wilhelm Shickard a Johannes Kepler escrita en 1624. En esta carta se describía una calculadora incluso más compleja construida por Shickard 20 años antes que la caja de Pascal. Por desgracia, la máquina fue destruida en un incendio y nunca se volvió a construir.

Durante sus años veinte, Pascal resolvió varios problemas difíciles relacionados con la curva cicloide, que de manera indirecta contribuyeron al desarrollo del cálculo diferencial. Trabajando con Pierre de Fermat, estableció las bases del cálculo de probabilidades y el análisis combinatorial. Uno de los resultados de este trabajo llegó a ser conocido como triángulo de Pascal, que simplifica el cálculo de los coeficientes del desarrollo de $(x + y)^n$, donde n es un entero positivo.

(continúa) ▼

Blaise Pascal

Pascal publicó también un tratado acerca de la presión de aire y realizó experimentos que mostraban que la presión barométrica disminuye con la altitud, lo que ayudó a confirmar teorías que habían sido propuestas por Galileo y Torricelli. Su trabajo sobre mecánica de fluidos forma parte significativa de las bases de ese campo. Entre las más famosas de sus contribuciones está la ley de Pascal, que establece que la presión aplicada a un fluido en un recipiente cerrado se transmite de modo uniforme por el fluido.

Cuando Pascal tenía 23 años, su padre enfermó, y la familia fue visitada por dos discípulos del jansenismo, movimiento de reforma en la iglesia católica que había comenzado seis años antes. La familia adoptó esta doctrina y 5 años después una de sus hermanas ingresó a un convento. Al inicio, Pascal se mantuvo al margen de este nuevo movimiento, pero cuando tenía 31 años, su hermana lo persuadió de abandonar el mundo y dedicarse a la religión.

Sus trabajos religiosos son considerados no menos brillantes que sus escritos matemáticos y científicos. Algunos consideran Provincial Letters, su serie de 18 ensayos en varios aspectos de la religión, como el comienzo de la prosa francesa moderna.

Pascal volvió brevemente a las matemáticas cuando tenía 35 años, pero un año después su salud, que siempre había sido mala, empeoró. Incapaz de realizar su trabajo habitual, se dedicó a ayudar a los menos afortunados. Tres años después, murió durante una estancia con su hermana, tras haber cedido su casa a una familia pobre.

3.6 Llamadas de función y funciones de biblioteca

Funciones de devolución de valor

Al comienzo del capítulo 2 se mostró un programa que consta de tres funciones: `main`, `Square` y `Cube`. A continuación se presenta una parte del programa:

```
int main()
{
    cout << "El cuadrado de 27 es " << Square(27) << endl;
    cout << "y el cubo de 27 es " << Cube(27) << endl;
    return 0;
}

int Square( int n )
{
    return n * n;
}

int Cube( int n )
{
    return n * n * n;
}
```

Se dice que las tres funciones son funciones de devolución de valor. `Square` devuelve a su invocador un valor, el cuadrado del número enviado a él. `Cube` devuelve un valor, el cubo del número enviado a él. Y `main` devuelve al sistema operativo un valor, el estado de salida del programa.

Ponga atención por un momento en la función `Cube`. La función `main` contiene una sentencia

```
cout << " y el cubo de 27 es " << Cube(27) << endl;
```

En esta sentencia, el maestro (`main`) causa que el sirviente (`Cube`) calcule el cubo de 27 y dé el resultado de nuevo a `main`. La secuencia de símbolos

```
Cube(27)
```

es una **llamada de función** o **invocación de función**. La computadora desactiva temporalmente la función `main` y comienza la ejecución de la función `Cube`. Cuando `Cube` ha terminado de hacer su trabajo, la computadora regresa a `main` y reinicia donde la dejó.

Llamada de función (invocación de función) Mecanismo que transfiere el control a una función.

En la llamada de función anterior, el número 27 se conoce como *argumento* (o *parámetro real*). Los argumentos hacen posible que la misma función trabaje en muchos valores distintos. Por ejemplo, es posible escribir sentencias como éstas:

```
cout << Cube(4);
cout << Cube(16);
```

Enseguida se muestra la plantilla de sintaxis para una llamada de función:

FunctionCall

FunctionName (ArgumentList)

La **lista de argumentos** es una vía para que las funciones se comuniquen entre sí. Algunas funciones, como `Square` y `Cube`, tienen un solo argumento en la lista. Otras, como `main`, no tienen argumentos en la lista. Algunas funciones tienen dos, tres o más argumentos en la lista, separados por comas.

Lista de argumentos Mecanismo por medio del cual las funciones se comunican entre sí.

Las funciones de devolución de valor se emplean en expresiones de forma muy parecida a como se usan las variables y constantes. El valor calculado mediante una función simplemente toma su lugar en la expresión. Por ejemplo, la sentencia

```
someInt = Cube(2) * 10;
```

guarda el valor 80 en `someInt`. Primero se ejecuta la función `Cube` para calcular el cubo de 2, que es 8. Ahora el valor 8 está disponible para usarlo en el resto de la expresión; es multiplicado por 10. Observe que una llamada de función tiene mayor precedencia que la multiplicación, lo cual tiene sentido si considera que el resultado de la función debe estar disponible antes de que tenga lugar la multiplicación.

Aquí se presentan varios hechos acerca de la función de devolución de valor:

- La llamada de función se emplea dentro de una expresión; no aparece como una sentencia separada.
- La función calcula un valor (*resultado*) que después está disponible para uso en la expresión.
- La función devuelve exactamente un resultado.

La función `Cube` espera que se le dé (o *pase*) un argumento de tipo `int`. ¿Qué sucede si el invocador pasa un argumento `float`? La respuesta es que el compilador aplica coerción implícita de tipos. La llamada de función `Cube (6.9)` calcula el cubo de 6, no de 6.9.

Aunque se continúan usando constantes literales como argumentos para `Cube`, el argumento podría ser fácilmente una variable o constante nombrada. De hecho, el argumento para una función de devolución de valor puede ser cualquier expresión del tipo apropiado. En la sentencia

```
alpha = Cube(int1 * int1 + int2 * int2);
```

se evalúa primero la expresión en la lista de argumentos, y sólo su resultado se pasa a la función. Por ejemplo, si `int1` contiene a 3 e `int2` contiene a 5, la llamada de función pasa a 34 como el argumento a `Cube`.

Una expresión en una lista de argumentos de función puede incluir hasta llamadas para funciones. Por ejemplo, se podría usar la función `Square` para reescribir la sentencia de asignación anterior como sigue:

```
alpha = Cube(Square(int1) + Square(int2))
```

Funciones de biblioteca

Algunos cálculos, como sacar raíz cuadrada o hallar el valor absoluto de un número, son muy comunes en los programas. Sería una enorme pérdida de tiempo si todo programador tuviera que empezar de cero y crear funciones que efectúen estas tareas. Para hacer más fácil la vida del programador, todo sistema de C++ incluye una biblioteca estándar, una gran colección de funciones preescritas, tipos de datos y otros elementos que cualquier programador de C++ pudiera usar. A continuación se ofrece una muestra muy pequeña de algunas funciones de biblioteca estándar:

Archivo de encabezado†	Función	Tipo(s) de argumento(s)	Tipos de resultado	Resultado (valor devuelto)
`<cstdlib>`	`abs(i)`	`int`	`int`	Valor absoluto de `i`
`<cmath>`	`cos(x)`	`float`	`float`	Coseno de `x` (`x` está en radianes)
`<cmath>`	`fabs(x)`	`float`	`float`	Valor absoluto de `x`
`<cstdlib>`	`labs(j)`	`long`	`long`	Valor absoluto de `j`
`<cmath>`	`pow(x, y)`	`float`	`float`	`x` elevada a la potencia `y` (si `x` = 0.0, `y` debe ser positiva; si `x` $\leq$ 0.0, `y` debe ser un número entero)
`<cmath>`	`sin(x)`	`float`	`float`	Seno de `x` (`x` está en radianes)
`<cmath>`	`sqrt(x)`	`float`	`float`	Raíz cuadrada de `x` (`x` $\geq$ 0.0)

† Los nombres de estos archivos de encabezado no son los mismos que en C++ pre-estándar. Si está trabajando con C++ pre-estándar, véase la sección D.2 del apéndice D.

Técnicamente, los elementos de la tabla marcados con `float` deben decir `double`. Estas funciones de biblioteca realizan su trabajo con valores de punto flotante de precisión doble. Pero como resultado de la coerción de tipos, las funciones trabajan como a usted le gustaría cuando les pasa valores `float`.

Usar una función de biblioteca es fácil. Primero, se coloca una directiva `#include` cerca de la parte superior de su programa, la cual especifica el archivo de encabezado apropiado. Esta directiva asegura que el preprocesador C++ inserte sentencias en el programa que dan al compilador cierta información acerca de la función. Entonces, si desea usar la función, sólo haga una llamada de función.* Aquí se presenta un ejemplo:

```
#include <iostream>
#include <cmath>       // For sqrt() and fabs()

using namespace std;
  :
float alpha;
float beta;
  :
alpha = sqrt(7.3 + fabs(beta));
```

* Algunos sistemas requieren que se especifique una opción de compilador particular si usa funciones matemáticas. Por ejemplo, con algunas versiones de UNIX, se debe añadir la opción `-lm` al compilar su programa.

Recuerde del capítulo 2 que los identificadores de la biblioteca estándar están en el espacio de nombre `std`. Si se omite la directiva `using` del código anterior, se deben usar nombres calificados para las funciones de biblioteca (`std::sqrt`, `std::fabs`, etcétera).

La biblioteca estándar C++ proporciona docenas de funciones para que usted las use. En el apéndice C se enlista una selección mucho más grande que la presentada aquí. En este momento debe echar un vistazo breve sin olvidar que mucha de la terminología y notación del lenguaje C++ tendrá sentido sólo después de avanzar más en la lectura del libro.

Funciones void (vacías)

En este capítulo, la única clase de función que se ha examinado es la función de devolución de valor. C++ proporciona también otra clase de función. Por ejemplo, la siguiente definición para la función `CalcPay` comienza con la palabra `void` en lugar del tipo de datos como `int` o `float`:

```
void CalcPay( . . .   )
{
    :
}
```

`CalcPay` es un ejemplo de una función que no devuelve un valor de función a su invocador. En cambio, efectúa alguna acción y luego termina. A una función como ésta se le denomina *función sin devolución de valor*, una *función de devolución void*, o, de manera más breve, una **función void.** En algunos lenguajes de programación, una función void se conoce como **procedimiento.**

Función void (procedimiento) Función que no devuelve un valor de función a su invocador y es invocada como una sentencia separada.

Función de devolución de valor Función que devuelve un solo valor a su invocador y es invocada desde dentro de una expresión.

Las funciones void se invocan de modo distinto que las **funciones de devolución de valor.** Con una función de devolución de valor, la llamada de función aparece en una expresión. Con una función void, la llamada de función es una sentencia separada, independiente. En el programa del año bisiesto, `main` llama a la función `IsLeapYear` en una expresión como ésta:

```
if (IsLeapYear(year))
```

Por otro lado, una llamada para una función void tiene el sabor de un comando o instrucción integrada:

```
DoThis(x, y, z);
DoThat();
```

En los siguientes capítulos no escribiremos nuestras propias funciones (excepto `main`). En cambio, se pondrá atención en cómo usar las funciones existentes, incluso funciones para llevar a cabo entrada y salida de flujo. Algunas de estas funciones son de las que devuelven un valor; otras son funciones void. De nuevo, se destaca la diferencia en cómo invocar estas dos clases de funciones: en una expresión ocurre una llamada para una función de devolución de valor, mientras que una llamada para una función void ocurre como una sentencia separada.

3.7 Formateo del resultado

Formatear la salida o resultado de un programa significa controlar cómo aparece visualmente en la pantalla o en una impresora. En el capítulo 2 se consideraron dos modos de formatear la salida: crear líneas en blanco extra por medio del manipulador `endl` e insertar espacios en blanco dentro de una línea escribiendo espacios en blanco extra en cadenas literales. En esta sección se examina cómo formatear los valores de salida.

Enteros y cadenas

Por omisión, los valores enteros y de cadena consecutivos son producidos sin espacios entre ellos. Si las variables `i`, `j` y `k` contienen los valores 15, 2 y 6, respectivamente, la sentencia

```
cout << "Results: " << i << j << k;
```

produce el flujo de caracteres

```
Results: 1526
```

Sin espacio entre los números, la interpretación de esta salida es difícil.

Para separar los valores de salida, se podría imprimir un solo espacio en blanco (como una constante `char`) entre los números:

```
cout << "Results: " << i << ' ' << j << ' ' << k;
```

Esta sentencia produce la salida

```
Results: 15 2 6
```

Si quiere incluso más espacio entre elementos, puede usar cadenas literales que contengan espacios en blanco, como se explicó en el capítulo 2:

```
cout << "Results: " << i << "    " << j << "    " << k;
```

Aquí, la salida obtenida es

```
Results: 15    2    6
```

Otra forma de controlar el espaciamiento horizontal de la salida es usar *manipuladores*. Desde hace ya algún tiempo se ha estado usando el manipulador `endl` para terminar una línea de resultado. En C++, un manipulador es algo muy curioso que se comporta como una función pero viaja disfrazado de un objeto de datos. Como una función, un manipulador genera alguna acción. Pero como un objeto de datos, un manipulador puede aparecer a la mitad de una serie de operaciones de inserción:

```
cout << someInt << endl << someFloat;
```

Los manipuladores se emplean sólo en sentencias de entrada y salida.

A continuación se muestra una plantilla de sintaxis revisada para la sentencia de salida, que muestra que no sólo se permiten expresiones aritméticas y de cadena sino también manipuladores:

OutputStatement

```
cout << ExpressionOrManipulator << ExpressionOrManipulator ...;
```

La biblioteca estándar de C++ proporciona muchos manipuladores, pero por ahora se consideran sólo cinco de ellos: `endl`, `setw`, `fixed`, `showpoint` y `setprecision`. Los manipuladores `endl`, `fixed` y `showpoint` vienen "gratis" cuando se incluye (`#include`) el archivo de encabezado `iostream` para efectuar I/O. Los otros dos manipuladores, `setw` y `setprecision`, requieren que también se incluya el archivo de encabezado `iomanip`:

```
#include <iostream>
#include <iomanip>

using namespace std;
  :
cout << setw(5) << someInt;
```

El manipulador `setw`, que significa "ancho de conjunto", permite controlar las posiciones de carácter que debe ocupar el siguiente grupo de datos cuando se produce. (`setw` sólo es para formatear números y cadenas, no datos `char`.) El argumento para `setw` es una expresión de enteros llamada *especificación de ancho de campo*; el grupo de posiciones de caracteres se conoce como *campo*. El siguiente grupo de datos por salir se imprime *justificado a la derecha* (con espacios en blanco a la izquierda para llenar el campo).

Considérese un ejemplo. Suponga que a dos variables `int` se les han asignado valores de la siguiente manera:

```
ans = 33;
num = 7132;
```

entonces las siguientes sentencias de salida producen el resultado mostrado a la derecha.

Sentencia	Salida (␣ significa espacio en blanco)

1. `cout << setw(4) << ans`
`<< setw(5) << num`
`<< setw(4) << "Hi";`
	␣␣33␣7132␣␣Hi
	4 5 4

2. `cout << setw(2) << ans`
`<< setw(4) << num`
`<< setw(2) << "Hi";`
	337132Hi
	2 4 2

3. `cout << setw(6) << ans`
`<< setw(3) << "Hi"`
`<< setw(5) << num;`
	␣␣␣␣33␣Hi␣7132
	6 3 5

4. `cout << setw(7) << "Hi"` `<< setw(4) << num;`	␣␣␣␣␣Hi7132 7 4
5. `cout << setw(1) << ans` `<< setw(5) << num;`	33␣7132 ↑ 5 El campo se expande automáticamente para ajustar el valor de dos dígitos

En (1), cada valor se especifica para ocupar posiciones suficientes de modo que haya por lo menos un espacio de separación entre ellos. En (2), los valores van juntos porque el ancho de campo especificado para cada valor es lo suficientemente grande para contenerlo. Es evidente que este resultado no es muy legible. Es mejor hacer el ancho de campo más grande que el tamaño mínimo requerido para que quede algún espacio entre valores. En (3) hay espacios en blanco extra para legibilidad; en (4), no los hay. En (5) el ancho de campo no es suficientemente grande para el valor en `ans`, de modo que se expande de manera automática a fin de crear espacio para todos los dígitos.

Fijar el ancho de campo es una acción única. Se mantiene sólo para que salga el siguiente elemento. Después de esta salida, el ancho de campo se restablece en 0, lo cual significa "extender el campo a exactamente las posiciones necesarias". En la sentencia

```
cout << "Hi" << setw(5) << ans << num;
```

el ancho de campo se restablece a 0 después de que se produce ans. Como resultado, se obtiene la salida

```
Hi   337132
```

Números de punto flotante

Se puede especificar un ancho de campo para valores de punto flotante así como para valores enteros. Pero hay que recordar que es necesario permitir el punto decimal cuando se especifica el número de caracteres. El valor 4.85 requiere cuatro posiciones de salida, no tres. Si `x` contiene el valor 4.85, la sentencia

```
cout << setw(4) << x << endl
     << setw(6) << x << endl
     << setw(3) << x << endl;
```

produce la salida

```
4.85
  4.85
4.85
```

En la tercera línea, un ancho de campo de 3 es insuficiente, así que el campo se expande de manera automática para acomodar el número.

Hay otras cuestiones relacionadas con la salida de números de punto flotante. Primero, los valores grandes de punto flotante se imprimen en notación científica (E). El valor 123456789.5 podría imprimirse en algunos sistemas como

```
1.23457E+08
```

Se puede usar el manipulador denominado `fixed` para obligar a que la salida de punto flotante subsiguiente aparezca en forma decimal y no en notación científica:

```
cout << fixed << 3.8 * x;
```

Segundo, si el número es un entero, C++ no imprime un punto decimal. El valor 95.0 se imprime como

```
95
```

Para obligar a los puntos decimales a mostrarse en la subsiguiente salida de punto flotante, incluso para números enteros, se puede usar el manipulador `showpoint`:

```
cout << showpoint << floatVar;
```

(Si usa una versión pre-estándar de C++, es posible que no estén disponibles los manipuladores `fixed` y `showpoint`. Véase en la sección D.3 del apéndice D una forma alternativa de lograr los mismos resultados.)

Tercero, con frecuencia se desearía controlar el número de *lugares decimales* (dígitos a la derecha del punto decimal) que se muestran. Si se supone que su programa imprime el impuesto de ventas de 50% en cierta cantidad, la sentencia

```
cout << "El impuesto es $" << price * 0.05;
```

podría producir

```
El impuesto es $17.7435
```

Aquí, claramente se preferiría mostrar el resultado a dos decimales. Para esto se usa el manipulador `setprecision` de la siguiente manera:

```
cout << fixed << setprecision(2) << "El impuesto es $" << price * 0.05;
```

Puesto que se especificó `fixed`, el argumento para `setprecision` especifica el número deseado de lugares decimales. A diferencia de `setw`, que se aplica sólo al siguiente elemento impreso, el valor enviado a `setprecision` permanece para toda salida subsiguiente (hasta que sea cambiado con otra llamada para `setprecision`). A continuación se muestran algunos ejemplos de cómo usar `setprecision` junto con `setw`:

Valor de x	Sentencia	Salida (␣ significa espacio en blanco)
	`cout << fixed;`	
`310.0`	`cout << setw(10)`	
	`<< setprecision(2) << x;`	␣␣␣␣`310.00`
`310.0`	`cout << setw(10)`	
	`<< setprecision(5) << x;`	␣`310.00000`
`310.0`	`cout << setw(7)`	
	`<< setprecision(5) << x;`	`310.00000` (se expande a nueve posiciones)
`4.827`	`cout << setw(6)`	
	`<< setprecision(2) << x;`	␣␣`4.83` (se redondea el último dígito mostrado)
`4.827`	`cout << setw(6)`	
	`<< setprecision(1) << x;`	␣␣␣`4.8` (se redondea el último dígito mostrado)

De nuevo, el número total de posiciones impresas se expande si el ancho de campo especificado por `setw` es demasiado estrecho. Sin embargo, el número de posiciones para dígitos fraccionarios se controla por completo mediante el argumento para `setprecision`.

En la tabla siguiente se resumen los manipuladores analizados en esta sección. Los manipuladores sin argumentos están disponibles por medio del archivo de encabezado `iostream`. Los que tienen argumentos requieren el archivo de encabezado `iomanip`.

Archivo de encabezado	Manipulador	Tipo de argumento	Efecto
`<iostream>`	`endl`	Ninguno	Termina la línea de salida actual
`<iostream>`	`showpoint`	Ninguno	Obliga la presentación de punto decimal en la salida de punto flotante
`<iostream>`	`fixed`	Ninguno	Suprime la notación científica en salida de punto flotante
`<iomanip>`	`setw(n)`	`int`	Fija el ancho de campo a `n`*
`<iomanip>`	`setprecision(n)`	`int`	Fija la precisión de punto flotante a `n` dígitos

* `setw` es sólo para números y cadenas, no para datos `char`. Asimismo, `setw` se aplica sólo a ítem de salida siguiente, después de lo cual el ancho de campo se restablece a 0 (que significa "use sólo las posiciones necesarias").

Cuestiones de estilo

Formato de programa

Respecto al compilador, las sentencias de C++ son de *formato libre*: pueden aparecer en cualquier parte sobre una línea, más de una puede aparecer en una sola línea, y una sentencia puede abarcar varias líneas. El compilador sólo necesita espacios en blanco (o comentarios o líneas nuevas) para separar símbolos importantes, y requiere símbolos de punto y coma para terminar las sentencias. Sin embargo, es muy importante que los programas sean legibles, tanto para su beneficio como para el de cualquier otra persona que los examine.

Cuando escribe una descripción para un documento, usted sigue ciertas reglas de sangrado para hacerlo legible. El mismo tipo de reglas puede hacer que sus programas sean más fáciles de leer. Es mucho más fácil localizar un error en un programa bien formateado que en uno desorganizado. Así que debe mantener bien formateado su programa mientras trabaja en él. Si tiene pereza y deja que su programa se desorganice mientras hace una serie de cambios, tómese el tiempo necesario para arreglarlo. Con frecuencia la fuente de un error se hace evidente durante el proceso de formatear el código.

Examine el siguiente programa para calcular el costo por pie cuadrado de una casa. Aunque se compila y ejecuta de modo correcto, no cumple ningún estándar de formato.

```
// Programa HouseCost
// Este programa calcula el costo por pie cuadrado de
   // espacio vital para una casa, dadas las dimensiones de
// la casa, el número de pisos, el tamaño del espacio no vital
// y el costo total menos el terreno
#include <iostream>
#include <iomanip>// Para setw() y setprecision()
using namespace
std;
const float WIDTH = 30.0; // Ancho de la casa
    const float LENGTH = 40.0; // Longitud de la casa
const float STORIES = 2.5; // Número de pisos completos
const float NON_LIVING_SPACE = 825.0;// Cochera, armarios, etc.

const float PRICE = 150000.0; // Precio de venta menos el terreno
int main() { float grossFootage; // Total de pies cuadrados
    float livingFootage;         // Área vital
float costPerFoot;     // Costo/pie de área vital
   cout << fixed << showpoint;  // Establecimiento de punto flotante
//   Formato de salida

grossFootage = LENGTH * WIDTH * STORIES; livingFootage =
grossFootage - NON_LIVING_SPACE; costPerFoot = PRICE /
livingFootage; cout << " El costo por pie cuadrado es "
<< setw(6) << setprecision(2) << costPerFoot << endl;
return 0; }
```

Ahora vea el mismo programa con el formato adecuado:

```
//*************************************************************
// Programa HouseCost
// Este programa calcula el costo por pie cuadrado de
// espacio vital para una casa, dadas las dimensiones de
```

(continúa) ▼

Formato de programa

```
// la casa, el número de pisos, el tamaño del
// espacio no vital y el costo total menos el terreno
//************************************************************
#include <iostream>
#include <iomanip>    // Para setw() y setprecision()
using namespace std;
const float WIDTH = 30.0;             // Ancho de la casa
const float LENGTH = 40.0;            // Longitud de la casa
const float STORIES = 2.5;            // Número de pisos completos
const float NON_LIVING_SPACE = 825.0; // Cochera, armarios, etc.
const float PRICE = 150000.0;         // Precio de venta menos el terreno

int main()
{
    float grossFootage;       // Total de pies cuadrados
    float livingFootage;      // Área vital
    float costPerFoot;        // Costo/pie de área vital
cout << fixed << showpoint;   // Establecer el punto flotante
                              //   formato de salida

    grossFootage = LENGTH * WIDTH * STORIES;
    livingFootage = grossFootage - NON_LIVING_SPACE;
    costPerFoot = PRICE / livingFootage;

    cout << "El costo por pie cuadrado es "
         << setw(6) << setprecision(2) << costPerFoot << endl;
    return 0;
}
```

¿Es necesario decir más?

En el apéndice F se habla acerca del estilo de programación. Utilícelo como una guía al momento de escribir programas.

3.8 Más operaciones de cadena

Ahora que se han introducido los tipos numéricos y las llamadas de función, es posible aprovechar las características adicionales del tipo de datos `string`. En esta sección se introducen cuatro funciones que operan en cadenas: `length`, `size`, `find` y `substr`.

Las funciones `length` y `size`

La función `length`, cuando se aplica a una variable `string`, devuelve un valor entero sin signo que es igual al número de caracteres actualmente en la cadena. Si `myName` es una variable `string`, una llamada a la función `length` se parece a esto:

```
myName.length()
```

Usted especifica el nombre de una variable `string` (aquí, `myName`), luego un punto y después el nombre de la función y la lista de argumentos. La función `length` no requiere que le pasen argumentos, pero se deben usar paréntesis para indicar una lista de argumentos vacía. Asimismo, `length` es una función de devolución de valor, así que la llamada de función debe aparecer dentro de una expresión:

```
string firstName;
string fullName;

firstName = "Alexandra";
cout << firstName.length() << endl;       // Prints 9
fullName = firstName + " Jones";
cout << fullName.length() << endl;        // Prints 15
```

Quizá se pregunte acerca de la sintaxis en una llamada de función como

```
firstName.length()
```

Esta expresión usa una notación C++ llamada *notación de punto*. Hay un punto entre el nombre de variable `firstName` y el nombre de función `length`. Algunos tipos de datos definidos por el programador, como `string`, tienen funciones que están estrechamente relacionadas con ellos, y la notación de punto se requiere en las llamadas de función. Si olvida usar la notación de punto y escribe la llamada de función como

```
length()
```

obtiene un mensaje de error al momento de compilar, algo como "UNDECLARED IDENTIFIER". El compilador piensa que usted está tratando de llamar a una función ordinaria denominada `length`, no la función `length` relacionada con el tipo `string`. En el capítulo 4 se analiza el significado de la notación de punto.

Algunas personas se refieren a la longitud de una cadena como su *tamaño*. Para acomodar ambos términos, el tipo `string` proporciona una función llamada `size`. Tanto `firstName.size()` como `firstName.length()` devuelven el mismo valor.

Se dice que la función `length` devuelve un valor entero sin signo. Si se desea guardar el resultado en una variable `len`, como en

```
len = firstName.length();
```

entonces, ¿qué se debe declarar en cuanto a la identidad del tipo de datos de `len`? Para evitar tener que adivinar si `unsigned int` o `unsigned long` es correcto para el compilador particular con que se está trabajando, el tipo `string` define un tipo de datos `size_type` para que se pueda usar

```
string firstName;
string::size_type len;

firstName = "Alexandra";
len = firstName.length();
```

Observe que es necesario usar el nombre calificado `string::size_type` (de la manera como se hizo con los identificadores en los espacios de nombre) porque de otro modo la definición de `size_type` se oculta dentro de la definición del tipo `string`.

Antes de dejar las funciones `length` y `size`, se debe hacer una observación acerca del uso de mayúsculas en los identificadores. En las normas dadas en el capítulo 2 se dijo que en este libro se comienzan con mayúscula los nombres de funciones definidas por el usuario y tipos de datos. Se

sigue esta convención al escribir nuestras propias funciones y tipos de datos en capítulos posteriores. Sin embargo, no se tiene control sobre el uso de mayúsculas en elementos provistos por la biblioteca estándar de C++. Los identificadores de la biblioteca estándar se escriben por lo general con minúsculas.

Función `find`

La función `find` busca una cadena para hallar la primera concurrencia de una subcadena particular y devuelve un valor entero sin signo (de tipo `string:: size_type`) que da el resultado de la búsqueda. La subcadena, pasada como un argumento para la función, puede ser una cadena literal o una expresión `string`. Si `str1` y `str2` son de tipo `string`, las siguientes son llamadas de función válidas:

```
str1.find("the")       str1.find(str2)       str1.find(str2 + "abc")
```

En cada caso anterior, se busca en `str1` para ver si dentro de ella se puede hallar la subcadena especificada. Si es así, la función devuelve la posición en `str1` donde comienza la correspondencia. (Las posiciones se numeran comenzando en 0, de modo que el primer carácter de una cadena está en la posición 0, el segundo está en la posición 1, y así sucesivamente.) Para una búsqueda exitosa, la correspondencia debe ser exacta, incluso el uso idéntico de mayúsculas. Si no se puede hallar la subcadena, la función devuelve el valor especial `string::npos`, una constante nombrada que significa "no es una posición dentro de la cadena". (`string::npos` es el valor más grande de tipo `string::size_type`, un número como `4294967295` en muchas máquinas. Este valor es adecuado para "no es una posición válida" porque las operaciones `string` no permiten que ninguna cadena adquiera esta longitud.)

Dado el segmento de código

```
string phrase;
string::size_type position;

phrase = "El perro y el gato";
```

la sentencia

```
position = phrase.find("el");
```

asigna a `position` el valor 12, mientras que la sentencia

```
position = phrase.find("rata");
```

asigna a `position` el valor `string::npos`, porque no hubo corresponencia.

El argumento para la función `find` puede ser también un valor `char`. En este caso, `find` busca la primera ocurrencia de ese carácter dentro de la cadena y devuelve su posición (o `string::npos`, si no se encontró el carácter). Por ejemplo, el segmento de código

```
string theString;

theString = "Abracadabra";
cout << theString.find('a');
```

produce el valor 3, que está en la posición de la primera ocurrencia de una *a* minúscula en `theString`.

A continuación se dan algunos ejemplos más de llamadas para la función `find`, suponiendo que se ha ejecutado el siguiente segmento de código:

```
string str1;
string str2;

str1 = "Programming and Problem Solving";
str2 = "gram";
```

Llamada de función	Valor devuelto por la función
`str1.find("and")`	12
`str1.find("Programming")`	0
`str2.find("and")`	`string::npos`
`str1.find("Pro")`	0
`str1.find("ro" + str2)`	1
`str1.find("Pr" + str2)`	`string::npos`
`str1.find(' ')`	11

Observe en el cuarto ejemplo que hay dos copias de la subcadena `"Pro"` en `str1`, pero `find` devuelve sólo la posición de la primera copia. Observe también que las copias pueden ser palabras separadas o partes de palabras; `find` solamente intenta hacer corresponder la secuencia de caracteres dada en la lista de argumentos. En el ejemplo final se demuestra que el argumento puede ser tan simple como un solo carácter, incluso un solo espacio en blanco.

Función `substr`

La función `substr` devuelve una subcadena particular de una cadena. Suponiendo que `myString` es de tipo `string`, aquí se da una llamada de función:

```
myString.substr(5, 20)
```

El primer argumento es un entero sin signo que especifica una posición dentro de la cadena, y el segundo es un entero sin signo que especifica la longitud de la subcadena deseada. La función devuelve la parte de la cadena que empieza con la posición especificada y continúa para el número de caracteres dados por el segundo argumento. Observe que `substr` no cambia a `myString`; devuelve un nuevo valor `string` temporal, que es copia de una parte de la cadena. A continuación se dan algunos ejemplos, suponiendo que la sentencia

```
myString = "Programming and Problem Solving";
```

se ha ejecutado.

Llamada de función	Cadena contenida en el valor devuelto por la función
`myString.substr(0, 7)`	`"Program"`
`myString.substr(7, 8)`	`"ming and"`
`myString.substr(10, 0)`	`""`
`myString.substr(24, 40)`	`"Solving"`
`myString.substr(40, 24)`	Ninguna. El programa termina con un mensaje de error de ejecución.

En el tercer ejemplo, especificar una longitud de 0 produce como resultado la cadena nula. En el cuarto ejemplo se muestra lo que sucede si el segundo argumento especifica más caracteres de los que están presentes después de la posición de inicio: `substr` devuelve los caracteres desde la posición inicial hasta el fin de la cadena. En el último ejemplo se ilustra que el primer argumento, la posición, no debe estar más allá del fin de la cadena.

Debido a que `substr` devuelve un valor de tipo `string`, puede usarlo con el operador de concatenación (+) para copiar partes de cadenas y unirlas para formar nuevas cadenas. Las funciones

find y length pueden ser útiles para determinar la ubicación y el fin de una parte de una cadena que será pasada a substr como argumentos.

A continuación se ilustra un programa en el que se emplean varias de las operaciones string:

```
//*****************************************************************
// Programa operaciones sobre cadenas
// Este programa demuestra varias operaciones sobre cadenas
//*****************************************************************
#include <iostream>
#include <string>     // Para tipo string

using namespace std;

int main()
{
    string fullName;
    string name;
    string::size_type startPos;

    fullName = "Jonathan Alexander Peterson";
    startPos = fullName.find("Peterson");
    name = "Mr. " + fullName.substr(startPos, 8);
    cout << name << endl;
    return 0;
}
```

Este programa produce Mr. Peterson cuando se ejecuta. Primero almacena una cadena en la variable fullName, y luego emplea find para localizar el inicio del nombre Peterson dentro de la cadena. A continuación, construye una nueva cadena al concatenar la literal "Mr. " con los caracteres Peterson, que son copiados de la cadena original. Por último, imprime la nueva cadena. Como se ve en capítulos posteriores, las operaciones de cadena son un aspecto importante de muchos programas de computadora.

En la tabla siguiente se resumen las operaciones string que se han considerado en este capítulo.

Llamada de función (s es de tipo string)	Tipo(s) de argumento(s)	Tipo de resultado	Resultado (valor obtenido)
s.length() s.size()	Ninguno	string::size_type	Número de caracteres en la cadena
s.find(arg)	string, cadena literal, o char	string::size_type	Posición de inicio en s donde se encontró arg; si no se encontró, el resultado es string::npos
s.substr(pos,len)	string::size_type	string	Subcadena de cuando mucho len caracteres, comenzando en la posición pos de s. Si len es demasiado grande, significa "hasta el fin" de la cadena s. Si pos es demasiado grande, se termina la ejecución del programa.*

* Técnicamente, si pos es demasiado grande, el programa genera lo que se llama *excepción fuera de alcance*, tema que se trata en el capítulo 17. A menos que se escriba un código de programa adicional para tratar de modo explícito con esta excepción fuera de alcance, el programa termina simplemente con un mensaje como "ABNORMAL PROGRAM TERMINATION".

Consejo práctico de ingeniería de software

Comprender antes de cambiar

Cuando se está a la mitad de lograr la ejecución de un programa y se encuentra un error, es tentador comenzar a cambiar partes del programa para intentar hacer que funcione. ¡No lo haga! Casi siempre empeorará las cosas. Es esencial que entienda qué es lo que causa el error y piense con detenimiento la solución. Lo único que debe intentar es ejecutar el programa con datos diferentes para determinar el patrón del comportamiento inesperado.

No hay truco mágico que pueda arreglar de manera automática un programa. Si el compilador indica que falta un punto y coma o una llave derecha, necesita examinar su programa y determinar con precisión cuál es el problema. Quizás escribió accidentalmente una coma en lugar de punto y coma. O quizá haya una llave izquierda extra.

Si el origen de un problema no es evidente de inmediato, una buena regla empírica es dejar la computadora e ir a alguna parte donde pueda ver con tranquilidad una copia impresa del programa. Los estudios muestran que las personas que hacen la depuración lejos de la computadora logran en menos tiempo que funcionen sus programas y al final producen mejores programas que quienes continúan trabajando en la máquina, una demostración más de que aún no hay sustituto mecánico para el pensamiento humano.*

* Basili, V. R. y Selby, R. W., "Comparing the Effectiveness of Software Testing Strategies", *IEEE Trans. on Software Engineering* SE-13, núm. 12 (1987): 1278-1296.

Caso práctico de resolución de problemas

Calculadora de pago de hipoteca

PROBLEMA Sus padres están pensando en refinanciar su hipoteca y le piden ayuda con los cálculos. Ahora que está aprendiendo C++, comprende que puede ahorrarse el trabajo de teclear en la calculadora si escribe un programa que realice los cálculos de manera automática.

ANÁLISIS En el caso práctico del capítulo 1, se dijo que con frecuencia hay pasos obvios en casi cualquier problema de este tipo:

1. Obtener los datos.
2. Calcular los resultados.
3. Producir los resultados.

Los datos necesarios en este programa son la cantidad de dinero por ahorrar, el número de años para el préstamo y la tasa de interés. De estos tres valores, se puede calcular el pago mensual. Aunque es posible resolver este problema con lápiz y papel, también podría escribir un programa para resolverlo. Por ahora puede hacer constantes los valores de datos y después, cuando aprenda cómo introducir valores, puede reescribir el programa.

Después de intercambiar ideas con sus padres, encuentra que deben $50 000 y tienen exactamente 7 años para pagar la deuda. La última cotización de su institución de crédito es por una tasa de interés de 5.24 sin costos de finiquito.

Definir constantes

```
Establecer LOAN_AMOUNT = 50000.00
Establecer NUMBER_OF_YEARS = 7
Establecer INTEREST_RATE = 0.0524
```

Usted recuerda vagamente haber visto la fórmula para determinar pagos por medio del interés compuesto, pero no la recuerda, así que decide buscar en Internet.

$$\frac{\text{Cantidad} * (1 + \text{Interés mensual})^{\text{número de pagos}} * \text{Interés mensual}}{(1 + \text{Interés mensual})^{\text{número de pagos}} - 1}$$

En realidad esto parece más fácil de hacer en una computadora que en una calculadora. Dos valores tomados para la potencia número de pagos parece desalentador. Por fortuna, el archivo de encabezado `<cmath>` de C++, que se examinó antes, contiene varias funciones matemáticas que incluyen la función de potencia. Antes de introducir los valores en la fórmula, es necesario calcular dos valores intermedios: la tasa de interés mensual y el número de pagos.

Calcular valores

```
Establecer monthlyInterest  to YEARLY_INTEREST dividido entre 12
Establecer numberOfPayments to NUMBER_OF_YEARS  multiplicado por 12
Establecer payment to (LOAN_AMOUNT * pow(monthlyInterest+1,
  numberOfPayments) * monthlyInterest) /
  (pow(monthlyInterest+1, numberOfPayments) – 1)
```

Ahora todo lo que queda es imprimir la respuesta en un formato claro y conciso.

Salida de resultados

```
Imprimir "Para una cantidad prestada de" LOAN_AMOUNT "con una tasa de
    interés de" YEARLY_INTEREST "y un" NUMBER_OF_YEARS "hipoteca anual",
Imprimir "sus pagos mensuales son $" payment "."
```

Del algoritmo se pueden crear tablas de constantes y variables como ayuda para escribir las declaraciones del programa.

Constantes

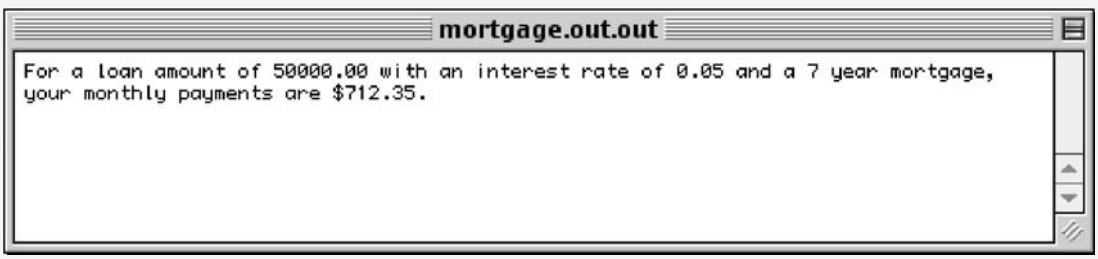
Nombre	Valor	Función
`LOAN_AMOUNT`	`50000.00`	Cantidad del préstamo
`YEARLY_INTEREST`	`0.0524`	Tasa de interés anual
`NUMBER_OF_YEARS`	`7`	Número de años

Variables

Nombre	Tipo de datos	Descripción
`monthlyInterest`	`float`	Tasa de interés mensual
`numberOfPayments`	`int`	Número total de pagos
`payment`	`int`	Pago mensual

```
//*************************************************************
// Programa Calculadora para pago de hipoteca
// Este programa determina los pagos mensuales en una hipoteca dadas
// la cantidad prestada, el interés anual y el número de años.
//*************************************************************

#include <iostream>                     // Acceso a cout
#include <cmath>                        // Acceso a función de potencia
#include <iomanip>                      // Acceso a manipuladores
using namespace std;

const float LOAN_AMOUNT = 50000.00;     // Cantidad del préstamo
const float YEARLY_INTEREST = 0.0524;   // Tasa de interés anual
const int NUMBER_OF_YEARS = 7;          // Número de años

int main()
{
    // variables locales
    float monthlyInterest;              // Tasa de interés mensual
    int numberOfPayments;               // Número total de pagos
    float payment;                      // Pago mensual

    // Calcular valores
    monthlyInterest = YEARLY_INTEREST / 12;
    numberOfPayments = NUMBER_OF_YEARS * 12;
    payment = (LOAN_AMOUNT *
               pow(monthlyInterest + 1, numberOfPayments)
               * monthlyInterest)/(pow(monthlyInterest + 1,
               numberOfPayments) - 1 );

    // Salida de resultados
    cout << fixed << setprecision(2) << "Para una cantidad prestada de "
         << LOAN_AMOUNT  << " con una tasa de interés de "
         << YEARLY_INTEREST << " y una " << NUMBER_OF_YEARS
         << " hipoteca anual, " << endl;
    cout << " sus pagos mensuales son de $" << payment
         << "." << endl;
    return 0;
}
```

A continuación se muestra una pantalla de la salida.



Se ve algo extraño en la salida: el interés debe ser 0.0524, no 0.05. La decisión de usar una precisión de 2 fue correcta para dólares y centavos, pero no para tasas de interés, que rara vez son porcentajes enteros. Se pidió hacer esta corrección en el ejercicio 1 del Seguimiento de caso práctico.

Prueba y depuración

1. Una constante `int` distinta de 0 no debe empezar con un cero. Si empieza con un cero, es un número octal (base 8).
2. Tenga cuidado con la división de enteros. La expresión `47/100` produce 0, el cociente entero. Ésta es una de las fuentes principales de resultados erróneos en programas de C++.
3. Al usar los operadores / y %, recuerde que no se permite la división entre cero.
4. Realice una doble comprobación de cada expresión de acuerdo con las reglas de precedencia para asegurarse de que las operaciones se efectúan en el orden deseado.
5. Evite combinar enteros y valores de punto flotante en las expresiones. Si los combina, considere usar moldeos explícitos de tipos para reducir la posibilidad de errores.
6. Para cada sentencia de asignación, compruebe que el resultado de la expresión tiene el mismo tipo de datos que la variable a la izquierda del operador de asignación (=). Si no, considere usar una conversión explícita de tipos para claridad y seguridad. Recuerde que guardar un valor de punto flotante en una variable `int` trunca la parte fraccionaria.
7. Para toda función de biblioteca que use en su programa, asegúrese de incluir (`#include`) el archivo de encabezado apropiado.
8. Examine cada llamada a una función de biblioteca para comprobar que usted tiene el número correcto de argumentos y que los tipos de datos de los argumentos son correctos.
9. Con el tipo `string`, las posiciones de los caracteres dentro de una cadena se numeran comenzando en 0, no en 1.
10. Si la causa de error en un programa no es obvia, deje la computadora y estudie una lista impresa. Cambie su programa sólo cuando haya comprendido la fuente del error.

Resumen

C++ proporciona varios tipos de datos numéricos integrados, de los cuales los más usados son `int` y `float`. Los tipos integrales se basan en los enteros matemáticos, pero la computadora limita el intervalo de valores enteros que pueden ser representados. Los tipos de punto flotante se basan en la noción matemática de números reales. Como con los enteros, la computadora limita el intervalo de números de punto flotante que se pueden representar. Asimismo, limita el número de dígitos de precisión en valores de punto flotante. Se pueden escribir literales de tipo `float` en varias formas, incluso en notación científica (E).

Gran parte del cálculo de un programa se efectúa en expresiones aritméticas. Las expresiones pueden contener más de un operador. El orden en que se llevan a cabo las operaciones se determina mediante las reglas de precedencia. En expresiones aritméticas, la multiplicación, la división y el módulo se efectúan primero, luego la suma y la resta. Las operaciones binarias múltiples (dos operandos) de la misma precedencia se agrupan de izquierda a derecha. Se pueden usar paréntesis para invalidar las reglas de precedencia.

Las expresiones pueden incluir llamadas de función. C++ soporta dos clases de funciones: funciones de devolución de valor y funciones void. Una función de devolución de valor se llama al escribir su nombre y lista de argumentos como parte de una expresión. Una función void se llama al escribir su nombre y lista de argumentos como una sentencia completa de C++.

La biblioteca estándar de C++ es una parte integral de todo sistema C++. La biblioteca contiene muchos tipos de datos preescritos, funciones y otros elementos que cualquier programador puede usar. El acceso a estos elementos es por medio de directivas `#include` para el preprocesador de C++, que inserta archivos de encabezado apropiados en el programa.

En sentencias de salida, los manipuladores `setw`, `showpoint`, `fixed` y `setprecision` controlan la apariencia de los valores en la salida. Estos manipuladores no afectan los valores almacenados en la memoria, sólo su apariencia cuando se muestran en el dispositivo de salida.

No sólo la salida producida por un programa debe ser fácil de leer, sino también el formato del programa debe ser claro y legible. C++ es un lenguaje de formato libre. Un estilo congruente en el que se usan sangrías, líneas en blanco y espacios dentro de las líneas le ayuda a usted (y a otros programadores) a entender y trabajar con sus programas.

Comprobación rápida

1. ¿Qué sucede con la parte fraccionaria de un número de punto flotante cuando se convierte a un tipo entero? (p. 89)
2. ¿Dónde aparecen los argumentos y cuál es su propósito? (pp. 89-91)
3. ¿Cuál es el valor de la siguiente expresión, dado que la variable de cadena `quickCheck` contiene la cadena `"My friend I shall be pedagogic"`? (pp. 101-105)

```
quickCheck.substr(10, 20) + " " + quickCheck.substr(0, 9) + "."
```

4. El formato nítido de un programa facilita la detección de errores. ¿Verdadero o falso? (pp. 100-102)
5. ¿Cómo difiere la declaración de constantes nombradas y variables de tipo `int` y `float` de las declaraciones de constantes nombradas y variables de tipo `string`? (pp. 82-84)
6. ¿Cómo escribiría una expresión que da el residuo de dividir integer1 entre integer2? (pp. 84-87)
7. Si `integer1` contiene 37 e `integer2` contiene 7, ¿cuál es el resultado de la expresión del ejercicio 6? (pp. 84-87)
8. ¿Cuál es el resultado de la siguiente expresión? (pp. 87 y 88)

```
27 + 8 * 6 - 44 % 5
```

9. Escriba una expresión que calcule la raíz cuadrada de 17.5. (pp. 92-95)
10. ¿Cómo difiere una llamada a una función void de una llamada a una función de devolución de valor? (pp. 96 y 97)
11. ¿Qué manipulador de flujo usaría para fijar la precisión de salida para valores de punto flotante? (pp. 95-99)

Respuestas

1. Se trunca la parte fraccionaria. **2.** Aparecen en la llamada para una función, entre paréntesis, y se usan para pasar datos hacia o desde una función. **3.** "I Shall be pedagogic My Friend." **4.** Verdadero. **5.** Las declaraciones son exactamente las mismas, excepto porque se usa la palabra reservada `int` o `float` en lugar de `string`, y se asigna un valor numérico para la constante en vez de un valor de cadena. **6.** `integer1 % integer2`. **7.** 37 % 7 = 2 **8.** 27 + (8 * 6) – (44 % 5) = 27 + 48 - 4 = 71 **9.** `sqrt(17.5)` **10.** Una función void aparece como una sentencia separada en vez de ser parte de una expresión. **11.** `setprecision`

Ejercicios de preparación para examen

1. Los tipos enteros y flotantes en C++ se consideran tipos de datos (simples, de dirección, estructurados). (Circule uno.)
2. ¿Cuáles son los cuatro tipos integrales en C++? Lístelos en orden de tamaño, del más pequeño al más grande.
3. ¿Cuál es el resultado si la computadora intenta calcular un valor que es más grande que el entero máximo permitido para un tipo integral determinado?
4. En un valor de punto flotante, ¿qué significa cuando la letra E aparece como parte de un número?
5. Etiquete cada una de las siguientes declaraciones como entero o punto flotante y si es constante o variable.

		entero/ flotante	constante/ variable
a)	`const int tracksOnDisk = 17;`	_____	_____
b)	`float timeOfTrack;`	_____	_____
c)	`const float maxTimeOnDisk = 74.0;`	_____	_____
d)	`short tracksLeft;`	_____	_____
e)	`float timeLeft;`	_____	_____
f)	`long samplesInTrack;`	_____	_____
g)	`const double sampleRate = 262144.5;`	_____	_____

6. ¿Cuáles son los dos significados del operador /?
7. ¿Cuál es el resultado de cada una de las siguientes expresiones?
   a) `27 + 8 / 5 - 7`
   b) `27.0 + 8.0 / 5.0 - 7.0`
   c) `25 % 7 + 9.0`
   d) `17++`
   e) `int(15.0 + 12.0 * 2.2 - 3 * 7)`
   f) `23--`
   g) `18 / 1.0`
8. Enliste los siguientes operadores en el orden de precedencia máxima a precedencia mínima. Si el conjunto de operadores tiene la misma precedencia, escríbalos encerrados en corchetes dentro de la lista ordenada.

   ```
   *    +    %    /    -    unary -    ()
   ```

9. Los operadores de incremento y decremento pueden preceder o seguir a su operando. ¿Verdadero o falso?
10. Establezca una correspondencia entre los siguientes términos y las definiciones dadas a continuación.
    a) Operador unario
    b) Operador binario
    c) Coerción de tipos
    d) Conversión de tipos (moldeo)
    e) Expresión de tipo mixto
    f) Lista de argumentos
    g) Función void

    i) Cálculo en que intervienen valores de punto flotante y enteros.
    ii) Operador con dos operandos.
    iii) Función que es llamada como una sentencia separada.
    iv) Cambiar de modo explícito el valor de un tipo en otro.
    v) Valores que aparecen entre los paréntesis en una llamada de función.
    vi) Operador con un solo operando.
    vii) Cambiar implícitamente el valor de un tipo en otro.
11. ¿A qué operador de C++ es equivalente esta sentencia?

    ```
    count = count + 1;
    ```

12. ¿Cómo escribe una operación de moldeo en C++?
13. ¿`main` es una función de devolución de valor o una función void?
14. Muestre de modo preciso lo que produce la siguiente sentencia.

    ```
    cout << setw(6) << showpoint << setprecision(2) << 215.0
    ```

15. Las formas prefijo y sufijo del operador de incremento (++) siempre se comportan de la misma manera. Se pueden usar indistintamente en cualquier parte del código de C++. ¿Verdadero o falso? Explique su respuesta.
16. ¿Qué tipo de datos se usan para declarar una variable que contenga el resultado de aplicar la función `length` a una cadena?
17. Dado que las variables de cadena, `str1` y `str2` contienen

    ```
    "you ought to start with logic"
    ```

    y

    ```
    "ou"
    ```

    respectivamente, ¿cuál es el resultado de cada una de las siguientes expresiones?

a) `str1.length()`
b) `str1.find(str2)`
c) `str1.substr(4, 25)`
d) `str1.substr(4, 25).find(str2)`
e) `str1.substr.(str1.find("logic"), 3)`
f) `str1.substr(24, 5).find(str2.substr(0,1))`
g) `str1.find("end")`

18. ¿Qué hace el manipulador fixed?

Ejercicios de preparación para la programación

1. Escriba una expresión para convertir un tiempo almacenado en las variables `int`, `hours`, `minutes` y `seconds`, en el número de segundos representado por el tiempo. Por ejemplo, si `hours` contiene 2, `minutes` contiene 20 y `seconds` contiene 12, entonces el resultado de su expresión debe ser 8412.
2. Dada una variable `int`, `days`, que contiene un número de días,
   a) Escriba una expresión que dé el número de semanas completas que corresponda a `days`. Por ejemplo, si `days` contiene 23, entonces el número de semanas completas es 3.
   b) Escriba una expresión que dé el número de días que quedan después de eliminar las semanas completas del valor en `days`. Por ejemplo, si `days` contiene 23, entonces el número de días que restan después de 3 semanas completas es 2.
3. Dadas las variables `int` llamadas `dollars`, `quarters`, `dimes`, `nickels` y `pennies`, escriba una expresión que calcule la cantidad total del dinero representado en las variables. El resultado debe ser un valor entero que represente el número total de monedas de un centavo (*pennies*).
4. Dadas las mismas variables que en el ejercicio 3, calcule el total pero guárdelo en una variable de punto flotante de modo que la parte entera esté en dólares y la parte fraccionaria en centavos.
5. Escriba una sentencia de asignación que sume 3 al valor de la variable `int` llamada `count`.
6. Escriba expresiones que ejecuten las siguientes fórmulas.
   a) $3X + Y$
   b) $A^2 + 2B + C$
   c) $\left(\frac{A+B}{C-D}\right) \times \left(\frac{X}{Y}\right)$
   d) $\frac{\left(\frac{A^2+2B+C}{D}\right)}{XY}$
   e) $\sqrt{|A-B|}$
   f) $X^{-\cos(Y)}$
7. Escriba una serie de sentencias de asignación que determinen las tres primeras posiciones de la cadena `"and"` en una variable `string` denominada `sentence`. Las posiciones se deben guardar en las variables `int` llamadas `first`, `second` y `third`. Puede declarar variables adicionales si es necesario. El contenido de `sentence` debe permanecer invariable.
8. Escriba una sentencia de asignación para hallar el primer espacio en blanco en una variable `string` denominada `name`. Guarde el resultado más uno en la variable `int` nombrada `startOfMiddle`.
9. Escriba una sentencia de salida que imprima el valor en la variable `money` tipo `float` en ocho espacios sobre la línea, con un signo delantero $ y dos dígitos de precisión decimal.
10. Escriba una sentencia de salida que imprima el valor en la variable `double` denominada `distance` en quince espacios sobre una línea con cinco dígitos de precisión decimal.
11. Si incluye el archivo de encabezado `climits` en un programa, se tienen las constantes `INT_MAX` e `INT_MIN`, que proporcionan los valores enteros máximo y mínimo que pueden ser representados. Escriba la sentencia include para este archivo y una sentencia de salida que muestre los dos valores, identificados con etiquetas apropiadas.

12. Complete el siguiente programa de C++. El programa debe calcular y producir el valor Celsius correspondiente al valor Fahrenheit dado.

```
//*********************************************
// Programa Celsius
// Este programa produce la temperatura Celsius
// correspondiente a una determinada temperatura Fahrenheit
//*********************************************

#include <iostream>

using namespace std;

int main()
{
    const float fahrenheit = 72.0;
```

Problemas de programación

1. Escriba un programa en C++ que calcule y produzca el volumen de un cono, dados el diámetro de su base y su altura. La fórmula para calcular el volumen del cono es:

   $$\frac{1}{3}\,\pi \times Radio^2 \times Altura$$

   Asegúrese de usar el formato y comentarios apropiados en su código. La salida debe ser etiquetada de manera clara.
2. Escriba un programa en C++ que calcule la media y la desviación estándar de un conjunto de cuatro valores enteros. La media es la suma de los cuatro valores divididos entre 4 y la fórmula de la desviación estándar es

   $$s = \sqrt{\frac{\sum_{i=1}^{n}\left(x_i - \bar{x}\right)^2}{n-1}}$$

   Donde $n = 4$, x_i se refiere a cada uno de los cuatro valores y $\bar{x}$ es la media. Observe que aunque cada uno de los valores son enteros, los resultados son valores de punto flotante. Asegúrese de usar el formato adecuado y los comentarios apropiados en su código. El resultado debe ser marcado de manera clara y tener un formato nítido.
3. El factorial de un número n (escrito $n!$) es el número multiplicado por el factorial de sí mismo menos uno. Esta definición es más fácil de entender con un ejemplo. El factorial de 2 es 2 * 1. El factorial de 3 es 3 * 2 * 1. El factorial de 4 es 4 * 3 * 2 * 1, etcétera. Los factoriales crecen mucho y muy rápido. La fórmula de Stirling da una aproximación del factorial para valores grandes, la cual es

   $$n! = e^{-n} n^n \sqrt{2\pi n}$$

   La función `exp` en el archivo de encabezado `<cmath>` da el valor de e elevado a una potencia dada (véase el apéndice C.5). Ya se han explicado las otras funciones necesarias para escribir esta fórmula. Escriba un programa en C++ que calcule el factorial de 15 tanto de manera directa como con la fórmula de Stirling y que produzca ambos resultados junto con su diferencia. Requerirá usar el tipo `double` para este cálculo. Asegúrese de usar el formato adecuado y los comentarios apropiados en su código. El resultado debe estar marcado con claridad y tener un formato nítido.

4. El número de permutaciones de un conjunto de n elementos tomados r a la vez que está dado por la fórmula:

$$\frac{n!}{r!(n-r)!}$$

Donde $n!$ es el factorial de n. (Véase, en el problema de programación 3, una explicación de las formas de calcular el factorial.) Si hay 18 personas en su grupo y quiere dividirlo en equipos de programación de 3 miembros, puede calcular el número de equipos distintos que se pueden ordenar con la fórmula anterior. Escriba un programa en C++ que determine el número de posibles disposiciones de equipos. Necesitará usar un tipo doble para este cálculo. Asegúrese de usar el formato adecuado y los comentarios apropiados en su código. El resultado se debe marcar con claridad y tener un formato nítido.

5. Escriba un programa en C++ que tome una cadena que contiene un nombre completo y produce cada parte del nombre por separado. El nombre debe estar en la forma de nombre, segundo nombre y apellido separados entre sí por un solo espacio. Por ejemplo, si el nombre contiene la cadena

```
"John Jacob Schmidt"
```

entonces el resultado del programa sería

```
Nombre: John
Segundo nombre: Jacob
Apellido: Schmidt
```

6. Amplíe el problema 5 para producir la longitud de cada una de las partes del nombre. Este problema se puede resolver por medio de una combinación de operaciones de cadena presentadas en este capítulo. Asegúrese de usar el formato adecuado y los comentarios apropiados en su código. El resultado debe ser marcado con claridad y tener un formato nítido.

Seguimiento de caso práctico

1. Cambie las sentencias de salida de modo que la tasa de interés se imprima hasta con cuatro decimales, pero que las cantidades en dólares permanezcan en dos decimales.
2. El programa asume que el número de años restantes en la antigua hipoteca es un múltiplo para 12. Cambie el programa de modo que la constante sea el número de meses que restan, no el número de años.
3. Por lo común se habla de tasas de interés como porcentajes. Reescriba el programa de modo que la tasa de interés se establezca como un porcentaje; es decir, 5.24 en vez de 0.0524.
4. El cálculo de `monthlyInterest + 1` se hace dos veces. Reescriba el programa de modo que realice este cálculo sólo una vez. A su juicio, ¿cuál versión del programa es mejor? Justifique su respuesta.

CAPÍTULO

4

Entrada de datos al programa y el proceso de diseño de software

Objetivos

Objetivos de conocimiento

- *Entender el valor de los mensajes en pantalla apropiados para programas interactivos.*
- *Conocer cuándo la entrada (o salida) no interactiva es apropiada y cómo difiere de la entrada (o salida) interactiva.*
- *Entender los principios básicos del diseño orientado a objetos.*

Objetivos de habilidades

Ser capaz de:

- *Construir sentencias de entrada para leer valores en un programa.*
- *Determinar el contenido de valores asignados a variables mediante sentencias de entrada.*
- *Escribir programas que usan archivos de datos para entrada y salida.*
- *Aplicar la metodología de descomposición funcional para resolver un problema simple.*
- *Tomar una descomposición funcional y codificarla en C++ por medio del código de autodocumentación.*

Un programa necesita datos en los cuales operar. Se han estado escribiendo todos los valores de datos en el programa mismo, en constantes nombradas y literales. Si esta fuera la única forma de introducir datos, se tendría que rescribir el programa cada vez que se quisiera aplicarlo a un conjunto distinto de valores. En este capítulo, se examinan formas de introducir datos en un programa mientras se ejecuta.

Una vez que se sabe cómo introducir datos, procesarlos y obtener resultados, se puede comenzar a pensar en diseñar programas más complicados. Se ha hablado acerca de estrategias generales para resolver problemas y de la escritura de programas sencillos. Para un problema sencillo, es fácil elegir una estrategia, escribir el algoritmo y codificar el programa. Pero a medida que los problemas se tornan más complejos, se tiene que usar un método mejor organizado. En la segunda parte de este capítulo se examinan dos metodologías generales para desarrollar software: diseño orientado a objetos y descomposición funcional.

4.1 Ingreso de datos en programas

Una de las más grandes ventajas de las computadoras es que un programa se puede usar con muchos conjuntos distintos de datos. Para esto, se deben mantener los datos separados del programa hasta que éste se ejecute. Entonces las instrucciones del programa copian valores del conjunto de datos a las variables del programa. Después de almacenar estos valores en las variables, el programa puede efectuar cálculos con ellos (véase la figura 4-1).

El proceso de colocar valores de un conjunto de datos externos en variables de un programa se llama *entrada*. En terminología usada ampliamente, se dice que la computadora *lee* datos externos hacia las variables. Los datos para el programa pueden venir de un dispositivo de entrada o de un archivo en un dispositivo de almacenamiento auxiliar. Más adelante, en este capítulo, se analiza la introducción de archivos; aquí se considera el teclado como el *dispositivo estándar de inserción de datos*.

Flujos de entrada y operador de extracción (>>)

El concepto de flujo es fundamental para entrada y salida en C++. Como se expresó en el capítulo 3, se puede pensar en un flujo de salida como una serie interminable de caracteres que van de su programa a un dispositivo de salida. Del mismo modo, piense en un *flujo de entrada* como una serie interminable de caracteres que entran a su programa desde un dispositivo de entrada.

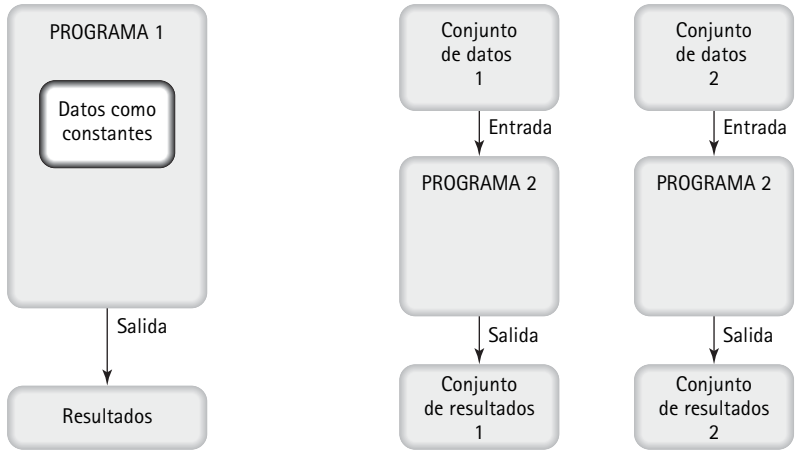


Figura 4-1 *Separar los datos del programa*

Para usar el flujo I/O, se debe usar la directiva de preprocesador

```
#include <iostream>
```

El archivo de encabezado `iostream` contiene, entre otras cosas, las definiciones de dos tipos de datos: `istream` y `ostream`. Éstos son tipos de datos que representan flujos de entrada y flujos de salida, respectivamente. El archivo de encabezado contiene también declaraciones que se parecen a esto:

```
istream cin;
ostream cout;
```

La primera declaración dice que `cin` es una variable de tipo `istream`. La segunda dice que `cout` es una variable de tipo `ostream`. Además, `cin` se relaciona con el dispositivo de entrada estándar (el teclado), y `cout` se relaciona con el dispositivo de salida estándar (por lo general, la pantalla).

Como ha visto, es posible producir valores para `cout` mediante el operador de inserción `(<<)`:

```
cout << 3 * price;
```

De modo similar, puede introducir datos desde `cin` por medio del *operador de extracción* `(>>)`:

```
cin >> cost;
```

Cuando la computadora ejecuta esta sentencia, introduce el siguiente número que usted introdujo en el teclado (425, por ejemplo) y lo guarda en la variable `cost`.

El operador de extracción `>>` lleva dos operandos. Su operando izquierdo es una expresión de flujo (en el caso más simple, sólo la variable `cin`). Su operando derecho es una variable donde se guardan los datos de entrada. Por el momento, se supone que la variable es de tipo simple (`char`, `int`, `float`, etc.). Más adelante, en este capítulo, se analiza la inserción de datos de cadena.

Se puede usar el operador `>>` varias veces en una sola sentencia de entrada. Cada ocurrencia extrae (introduce) el siguiente registro de datos desde el flujo de entrada. Por ejemplo, no hay diferencia entre la sentencia

```
cin >> length >> width;
```

y el par de sentencias

```
cin >> length;
cin >> width;
```

Usar una secuencia de extracciones en una sentencia es un beneficio para el programador.

Cuando no se tiene experiencia con C++ existe la posibilidad de invertir el operador de extracción `(>>)` y el operador de inserción `(<<)`. A continuación se expresa una forma fácil de recordar cuál es cuál: comience siempre la sentencia con `cin` o `cout` y use el operador que apunta en la dirección en la que van los datos. La sentencia

```
cout << someInt;
```

envía datos de la variable `someInt` *al* flujo de salida. La sentencia

```
cin >> someInt;
```

envía datos del flujo de entrada *a* la variable `someInt`.

La siguiente es una plantilla de sintaxis para la sentencia de entrada:

InputStatement

```
cin >> Variable >> Variable ...;
```

A diferencia de los elementos especificados en una sentencia de salida, que pueden ser constantes, variables o expresiones complicadas, los elementos especificados en una sentencia de entrada pueden ser *sólo* nombres de variables. ¿Por qué? Porque una sentencia de entrada indica dónde se deben almacenar los valores de datos de entrada. Sólo los nombres de variables se refieren a lugares de la memoria donde se pueden guardar valores mientras un programa está en ejecución.

Cuando introduce datos en el teclado, debe asegurarse de que cada valor es apropiado para el tipo de datos de la variable en la sentencia de entrada.

Tipo de datos de variable en una operación>>	Datos de entrada válidos
`char`	Un solo carácter imprimible distinto a un espacio en blanco
`int`	Una constante literal `int`, precedida opcionalmente por un signo
`float`	Una constante literal `int` o `float` (posiblemente en notación científica, E), precedida opcionalmente por un signo

Observe que cuando introduce un número en una variable `float`, el valor de entrada no tiene un punto decimal. El valor entero es forzado automáticamente a un valor `float`. Cualquier otra incongruencia, como tratar de introducir un valor `float` en una variable `int` o un valor `char` en una variable `float`, puede originar resultados inesperados o graves. Más adelante, en este capítulo, se analiza lo que podría suceder.

Al buscar el siguiente valor de entrada en el flujo, el operador `>>` omite *caracteres de espacio en blanco*. Estos caracteres no se imprimen, como el carácter que marca el fin de una línea. (En la siguiente sección se habla acerca de este carácter de fin de línea.) Después de omitir algún carácter de espacio en blanco, el operador `>>` procede a extraer el valor de datos deseado del flujo de entrada. Si este valor de datos es un valor `char`, la introducción se detiene tan pronto se introduce un solo carácter. Si el valor de datos es `int` o `float`, la inserción del número se detiene en el primer carácter que es inapropiado para el tipo de datos, como un carácter de espacio en blanco. A continuación se dan algunos ejemplos, donde `i`, `j` y `k` son variables `int`, `ch` es una variable `char`, y `x` es una variable `float`:

Sentencia	Datos	Contenido después del ingreso
1. `cin >> i;`	`32`	`i` = 32
2. `cin >> i >> j;`	`4 60`	`i` = 4, `j` = 60
3. `cin >> i >> ch >> x;`	`25 A 16.9`	`i` = 25, `ch` = 'A', `x = 16.9`

4. `cin >> i >> ch >> x;`	`25`
	`A`
	`16.9`
		`i` = 25, `ch` = 'A', `x` = 16.9

5. `cin >> i >> ch >> x;`	`25A16.9`	`i` = 25, `ch` = 'A', `x` = 16.9
6. `cin >> i >> j >> x;`	`12 8`	`i` = 12, `j` = 8 (La computadora espera un tercer número)
7. `cin >> i >> x;`	`46 32.4 15`	`i` = 46, `x` = 32.4 (15 se mantiene para ingreso posterior)

Los ejemplos (1) y (2) son directos de introducción de enteros. El ejemplo (3) muestra que los valores de datos de caracteres no van entre comillas cuando se introducen (las constantes de caracteres entre comillas son necesarias en un programa para distinguirlas de los identificadores). En el ejemplo 4 se muestra cómo el proceso de omitir caracteres de espacio en blanco incluye pasar a la siguiente línea de entrada si es necesario. En el ejemplo (5) se ilustra que el primer carácter encontrado, que es inapropiado para un tipo de datos numérico, termina el número. El ingreso para la variable `i` se detiene en el carácter de entrada `A`, después `A` se guarda en `ch`, y después el ingreso para `x` se detiene al final de la línea de entrada. El ejemplo 6 muestra que si usted está en el teclado y no ha introducido valores suficientes para satisfacer la sentencia de entrada, la computadora espera (y se mantiene en espera de) más datos. El ejemplo (7) muestra que si se introducen más valores que las variables existentes en la sentencia de entrada, los valores extra permanecen en espera en el flujo de entrada hasta que pueden ser leídos por la siguiente sentencia de entrada. Si hay valores extra restantes cuando termina el programa, la computadora los ignora.

Marcador de lectura y carácter de nueva línea

Para ayudar a explicar el ingreso de flujo con más detalle, se introduce el concepto del *marcador de lectura*. El marcador de lectura funciona como un separador de libro, pero en lugar de marcar un lugar en un libro, sigue la pista del punto en el flujo de entrada donde la computadora debe continuar leyendo. El marcador de lectura indica el siguiente carácter que habrá de leerse. El operador de extracción `>>` deja el marcador de lectura en el carácter que sigue a la última pieza de datos que se introdujo.

Cada línea de entrada tiene un carácter invisible de fin de línea (el *carácter de nueva línea*) que dice a la computadora dónde termina una línea y comienza la siguiente. Para hallar el siguiente valor de entrada, el operador `>>` cruza los límites de línea (caracteres de nueva línea) si tiene que ser así.

¿De dónde viene el carácter de nueva línea? ¿Qué es? La respuesta a la primera pregunta es fácil. Cuando usted está trabajando en un teclado, genera un carácter de nueva línea cada vez que oprime la tecla Return o Enter. Su programa genera también un carácter de nueva línea cuando usa el manipulador `endl` en una sentencia de salida. El manipulador `endl` produce una línea nueva, indicando al cursor en pantalla que vaya a la siguiente línea. La respuesta a la segunda pregunta varía de una computadora a otra. El carácter de nueva línea es un carácter de control no imprimible que el sistema reconoce como indicador de fin de una línea, si no es una línea de entrada o una línea de salida.

En un programa C++, se puede hacer referencia al carácter de nueva línea por medio de los símbolos `\n`, una barra inversa y una `n` sin ningún espacio entre ellas. Aunque `\n` consta de dos símbolos, se refiere a un solo carácter, el carácter de nueva línea. De la misma manera es posible almacenar la letra *A* en una variable `char ch`, como ésta:

```
ch = 'A';
```

se puede guardar el carácter de nueva línea en una variable:

```
ch = '\n';
```

También se puede colocar el carácter de nueva línea en una cadena, del mismo modo que colocaría cualquier carácter imprimible:

```
cout << "Hello\n";
```

Esta sentencia tiene exactamente el mismo efecto que la sentencia

```
cout << "Hello" << endl;
```

Regresemos a la explicación de ingreso de datos. Considérense algunos ejemplos con el marcador de lectura y el carácter de nueva línea. En la tabla siguiente, `i` es una variable `int`, `ch` es una variable `char` y `x` es una variable `float`. Las sentencias de entrada producen los resultados mostrados. La parte del flujo de entrada sombreada es lo que se ha extraído mediante las sentencias de entrada. El marcador de lectura, denotado por el bloque sombreado, indica el siguiente carácter que habrá de leerse. El símbolo `\n` denota el carácter de nueva línea producido al oprimir la tecla de retorno.

Sentencias	Contenido después del ingreso	Posición del marcador en el flujo de entrada
1.		25 A 16.9\n
cin >> i;	i = 25	**25** A 16.9\n
cin >> ch;	ch = 'A'	**25 A** 16.9\n
cin >> x;	x = 16.9	**25 A 16.9**\n
2.		25\n A\n 16.9\n
cin >> i;	i = 25	**25**\n A\n 16.9\n
cin >> ch;	ch = 'A'	**25\n** **A**\n 16.9\n
cin >> x;	x = 16.9	**25\n** **A\n** **16.9**\n
3.		25A16.9\n
cin >> i;	i = 25	**25**A16.9\n
cin >> ch;	ch = 'A'	**25A**16.9\n
cin >> x;	x = 16.9	**25A16.9**\n

Lectura de datos de caracteres con la función `get`

Como se explicó, el operador `>>` omite caracteres de espacio en blanco iniciales (como espacios en blanco y caracteres de nueva línea) mientras busca el siguiente valor de datos en el flujo de entrada. Suponga que `ch1` y `ch2` son variables `char` y el programa ejecuta la sentencia

```
cin >> ch1 >> ch2;
```

Si el flujo de entrada consta de

```
R 1
```

entonces el operador de extracción almacena 'R' en `ch1`, omite el espacio en blanco y guarda '1' en `ch2`. (Observe que el valor `char` '1' no es lo mismo que el valor `int` 1. Los dos se almacenan de manera completamente distinta en la memoria de una computadora. El operador de extracción interpreta los mismos datos en formas distintas, lo cual depende del tipo de datos de la variable que se está llenando.)

¿Qué pasa si se desea introducir *tres* caracteres desde la línea de entrada: la *R*, el espacio en blanco y el 1? Con el operador de extracción, no es posible. Se omiten los caracteres de espacio en blanco.

El tipo de datos `istream` proporciona una segunda forma de leer datos de caracteres, además del operador `>>`. Se puede usar la función `get`, que introduce el siguiente carácter en el flujo de entrada sin omitir ningún carácter de espacio en blanco. Una llamada de función se parece a esto:

```
cin.get(someChar);
```

La función `get` se relaciona con el tipo de datos `istream`, y se requiere notación de punto para hacer una llamada de función. (Recuerde que la notación de punto se empleó en el capítulo 3 para invocar ciertas funciones relacionadas con el tipo `string`. Más adelante, en este capítulo, se explica la razón para la notación de punto.) Para usar la función `get`, se da el nombre de una variable `istream` (aquí,

cin), luego un punto y después el nombre de la función y la lista de argumentos. Observe que la llamada para get emplea la sintaxis para llamar una función void, no una función de devolución de valor. La llamada de función es una sentencia completa; no es parte de una expresión más grande.

El efecto de la llamada de función anterior es introducir el siguiente carácter que espera en el flujo, incluso si es un carácter de espacio en blanco, y lo guarda en la variable someChar. El argumento para la función get *debe* ser una variable, no una constante o expresión arbitraria; se debe decir a la función dónde se quiere que guarde el carácter de entrada.

Por medio de la función get ahora se pueden introducir los tres caracteres de la línea de entrada

```
R 1
```

Se pueden usar tres llamadas consecutivas para la función get:

```
cin.get(ch1);
cin.get(ch2);
cin.get(ch3);
```

o se puede hacer de esta manera:

```
cin >> ch1;
cin.get(ch2);
cin >> ch3;
```

La primera versión es tal vez un poco más clara para que alguien la lea y la entienda.

A continuación se proporcionan algunos ejemplos adicionales de ingreso de caracteres por medio del operador >> y la función get. ch1, ch2 y ch3 son variables char. Como antes, \n denota el carácter de nueva línea.

Sentencias	Contenido después del ingreso	Posición del marcador en el flujo de entrada
1.		A B\n CD\n
cin >> ch1;	ch1 = 'A'	**A** B\n CD\n
cin >> ch2;	ch2 = 'B'	**A B**\n CD\n
cin >> ch3;	ch3 = 'C'	**A B\n** **C**D\n
2.		A B\n CD\n
cin.get(ch1);	ch1 = 'A'	**A** B\n CD\n
cin.get(ch2);	ch2 = ' '	**A** B\n CD\n
cin.get(ch3);	ch3 = 'B'	**A B**\n CD\n
3.		A B\n CD\n
cin >> ch1;	ch1 = 'A'	**A** B\n CD\n
cin >> ch2;	ch2 = 'B'	**A B**\n CD\n
cin.get(ch3);	ch3 = '\n'	**A B\n** CD\n

Bases teóricas

Más acerca de funciones y argumentos

Cuando la función `main` indica a la computadora salir y seguir las instrucciones de otra función, `SomeFunc`, la función `main` *llama* a `SomeFunc`. En la llamada a `SomeFunc`, los argumentos de la lista se *pasan* a la función. Cuando `SomeFunc` termina, la computadora *vuelve* a la función `main`.

Con algunas funciones que usted ya ha visto, como `sqrt` y `abs`, puede pasar constantes, variables y expresiones arbitrarias a la función. Sin embargo, la función `get` para leer datos de caracteres sólo acepta una variable como argumento. La función `get` almacena un valor en su argumento cuando vuelve, y sólo las variables pueden tener valores almacenados en ellas mientras un programa está en ejecución. Aun cuando `get` se llama como una función void, no una función de devolución de valor, *devuelve* o *pasa* un valor por su lista de argumentos. El punto a recordar es que se pueden usar argumentos tanto para enviar datos a una función como para obtener resultados.

Omitir caracteres con la función `ignore`

La mayoría de nosotros tenemos una herramienta especializada en un cajón de la cocina o en una caja de herramientas. Se llena de polvo y telarañas porque casi nunca se utiliza. Pero cuando de repente se necesita, se siente uno feliz de contar con ella. La función `ignore` relacionada con el tipo `istream` es como esta herramienta especializada. Rara vez se tiene ocasión de usar `ignore`; pero cuando la necesita, se siente feliz de que esté disponible.

La función `ignore` se emplea para omitir (leer y descartar) caracteres en el flujo de entrada. Es una función con dos argumentos, llamada de esta manera:

```
cin.ignore(200, '\n');
```

El primer argumento es una expresión `int`; la segunda un valor `char`. Esta llamada de función particular dice a la computadora que omita los siguientes 200 caracteres de entrada *u* omita los caracteres hasta que se lea un carácter de nueva línea, lo que venga primero.

A continuación se muestran algunos ejemplos que usan una variable `char`, `ch`, y tres variable `int` (`i`, `j` y `k`):

Sentencias	Contenido después del ingreso	Posición del marcador en el flujo de entrada
1.		957 34 1235\n 128 96\n
`cin >> i >> j;`	`i = 957, j = 34`	**957 34** 1235\n 128 96\n
`cin.ignore(100, '\n');`		**957 34 1235\n** 128 96\n
`cin >> k;`	`k = 128`	**957 34 1235\n** **128** 96\n
2.		A 22 B 16 C 19\n
`cin >> ch;`	`ch = 'A'`	**A** 22 B 16 C 19\n
`cin.ignore(100, 'B');`		**A 22 B** 16 C 19\n
`cin >> i;`	`i = 16`	**A 22 B 16** C 19\n
3.		ABCDEF\n
`cin.ignore(2, '\n');`		**AB**CDEF\n
`cin >> ch;`	`ch = 'C'`	**ABC**DEF\n

En el ejemplo (1) se muestra el uso más común de la función `ignore`, que es omitir el resto de los datos en la línea de entrada actual. En el ejemplo (2) se muestra el uso de un carácter distinto a `'\n'` como el segundo argumento. Se omiten todos los caracteres de entrada hasta que se encuentra una *B*, luego se lee el siguiente número de entrada en `i`. Tanto en (1) como en (2), se atiende el segundo argumento para la función `ignore`, y se elige en forma arbitraria cualquier número grande, como 100, para el primer argumento. En (3), se cambia de enfoque y se centra la atención en el primer argumento. La intención es omitir los siguientes dos caracteres en la línea actual.

Lectura de datos de cadena

Para introducir una cadena de caracteres en una variable `string`, se tienen dos opciones. La primera es usar el operador de extracción `(>>)`. Al leer los caracteres de entrada en una variable `string`, el operador `>>` omite cualquier carácter de espacio en blanco inicial. Después lee caracteres sucesivos hacia la variable, *deteniéndose* en el primer carácter de espacio en blanco posterior (que no es consumido, sino que permanece como el primer carácter en espera en el flujo de entrada). Por ejemplo, suponga que se tiene el siguiente código.

```
string firstName;
string lastName;

cin >> firstName >> lastName;
```

Si el flujo de entrada inicialmente se ve como esto (donde □ denota un espacio en blanco):

```
□□Mary□Smith□□□18
```

entonces la sentencia de entrada almacena los cuatro caracteres `Mary` en `firstName`, guarda los cinco caracteres `Smith` en `lastName` y deja al flujo de entrada como

```
□□□18
```

Aunque el operador `>>` se emplea ampliamente para introducción de cadena, tiene una posible desventaja: no se puede usar para introducir una cadena con espacios en blanco en su interior. (Recuerde que la lectura se detiene tan pronto como encuentra un carácter de espacio en blanco.) Este hecho conduce a la segunda opción para llevar a cabo la introducción de cadena: la función `getline`. Una llamada a esta función se parece a esto:

```
getline(cin, myString);
```

La llamada de función, que no usa la notación de punto, requiere dos argumentos. El primero es una variable de flujo de entrada (aquí, `cin`) y el segundo, una variable `string`. La función `getline` no omite los caracteres de espacio en blanco iniciales y continúa hasta que llega al carácter de nueva línea `'\n'`. Es decir, `getline` lee y almacena una línea de entrada completa, espacios en blanco incrustados y todo. Observe que con `getline`, el carácter de nueva línea se consume (pero no se almacena en la variable de cadena). Dado el segmento de código

```
string inputStr;

getline(cin, inputStr);
```

y la línea de entrada

```
□□Mary□Smith□□□18
```

el resultado de la llamada para `getline` es que los 17 caracteres en la línea de entrada (incluso los espacios en blanco) se almacenan en `inputStr`, y el marcador de lectura se traslada al comienzo de la siguiente línea de entrada.

En la tabla siguiente se resumen las diferencias entre el operador `>>` y la función `getline` al leer datos de cadena hacia variables `string`.

Sentencia	¿Omite el espacio en blanco principal?	¿Cuándo se detiene la lectura?
`cin >> inputStr;`	Sí	Cuando se encuentra un carácter de espacio en blanco posterior (que *no* es consumido)
`getline(cin, inputStr);`	No	Cuando se encuentra '\n' (que *es* consumido)

4.2 Entrada/salida interactiva

En el capítulo 1 se definió como programa interactivo aquel en el que el usuario se comunica directamente con la computadora. Muchos de los programas que se escriben son interactivos. Hay cierta "formalidad" en la escritura de programas interactivos que tienen que ver con las instrucciones que debe seguir el usuario.

Para meter datos en un programa interactivo, se comienza con *indicadores de entrada*, mensajes impresos que explican lo que el usuario debe introducir. Sin estos mensajes, el usuario no tiene idea de qué valores teclear. En muchos casos, un programa debe imprimir todos los valores escritos a fin de que el usuario compruebe que fueron introducidos de manera correcta. Imprimir los valores de entrada se llama *impresión por eco*. A continuación se presenta un programa que muestra el uso apropiado de los indicadores o mensajes al operador:

```
//*****************************************************************
// Programa Prompts
// Este programa demuestra el uso de los mensajes de entrada
//*****************************************************************
#include <iostream>
#include <iomanip>    // Para setprecision ()

using namespace std;

int main()
{
    int  partNumber;
    int  quantity;
    float unitPrice;
    float totalPrice;

    cout << fixed << showpoint          // Establecer punto flotante
         << setprecision(2);            //   Formato de salida

    cout << "Introduzca el número de parte:" << endl;        // Prompt
    cin >> partNumber;

    cout << "Introduzca la cantidad pedida de esta parte:"   // Prompt
         << endl;
    cin >> quantity;
```

```
    cout << "Introduzca el precio unitario para esta parte:" // Prompt
         << endl;
    cin >> unitPrice;

    totalPrice = quantity * unitPrice;
    cout << "Parte " << partNumber                          // Echo print
         << ", cantidad " << quantity
         << ", a $ " << unitPrice << " cada una" << endl;
    cout << "total $ " << totalPrice << endl;
    return 0;
}
```

Aquí está el resultado del programa, donde lo que introdujo el usuario se muestra en negritas:

```
Introduzca el número de parte:
4671
Introduzca la cantidad pedida de esta parte:
10
Introduzca el precio unitario para esta parte:
27.25
Parte 4671, cantidad 10, a $ 27.25 cada una da un
total de $ 272.50
```

La cantidad de información que debe poner en sus mensajes depende de quién vaya a usar un programa. Si está escribiendo un programa para personas que no están familiarizadas con las computadoras, sus mensajes deben ser más detallados. Por ejemplo, "Teclee un número de pieza de cuatro dígitos, luego oprima la tecla Enter". Si el programa lo va a usar de modo frecuente la misma persona, podría acortar los mensajes: "Introduzca NP" e Introduzca cant.". Si el programa es para usuarios muy experimentados, puede solicitar varios valores a la vez y pedir que los escriban todos en una línea de entrada:

```
Introduzca NP, cant., precio unitario:
4176 10 27.25
```

En programas que usan grandes cantidades de datos, este método ahorra golpes de tecla y tiempo. Sin embargo, provoca que el usuario introduzca valores en el orden equivocado. En esos casos, la impresión de datos por eco es especialmente importante.

Si un programa debe imprimir o no su entrada por eco depende también de cuán experimentados son los usuarios y de la tarea que el programa vaya a efectuar. Si los usuarios son experimentados y los mensajes son claros, como en el primer ejemplo, entonces probablemente no se requiere la impresión por eco. Si los usuarios son principiantes o se pueden introducir múltiples valores a la vez, se debe usar la impresión por eco. Si el programa introduce una gran cantidad de datos y los usuarios son experimentados, en lugar de imprimir los datos por eco, se pueden almacenar en un archivo separado que se puede comprobar después de que se introduzcan todos los datos. Más adelante, en este capítulo, se analiza cómo guardar datos en un archivo.

Los mensajes no son la única forma en que los programas interactúan con los usuarios. Puede ser útil pedir a un programa que imprima algunas instrucciones generales al comienzo ("Oprima Enter después de teclear cada valor de datos. Introduzca un número negativo al terminar."). Cuando el dato no se introduce en forma correcta, se debe imprimir un mensaje que indique el problema. Para usuarios que no han trabajado mucho con computadoras, es importante que estos mensajes sean informativos y cordiales. Es probable que el mensaje

```
¡¡¡VALORES DE DATOS INCORRECTOS!!!
```

incomode a un usuario poco experimentado. Además, no ofrece ninguna información constructiva. Un mensaje mucho mejor sería

```
Ése no es un número de parte válido.
Los números de parte no deben ser de más de cuatro dígitos de largo.
Por favor vuelva a introducir el número en forma apropiada.
```

En el capítulo 5 se mencionan sentencias que permiten probar datos erróneos.

4.3 Entrada/salida no interactiva

Aunque en este texto se tiende a usar ejemplos de entradas y salidas (I/O) interactivas, muchos programas se escriben con I/O no interactivas. Un ejemplo común de I/O no interactiva en sistemas computarizados grandes es el procesamiento por lotes (véase el capítulo 1). Recuerde que en el procesamiento por lotes el usuario y la computadora no interactúan mientras el programa está en ejecución. Este método es más efectivo cuando un programa va a introducir o a producir grandes cantidades de datos. Un ejemplo de procesamiento por lotes es un programa que introduce un archivo que contiene calificaciones semestrales para miles de alumnos e imprime informes de calificaciones que serán enviados por correo.

Cuando un programa debe leer muchos valores de datos, la práctica habitual es prepararlos con anticipación y archivarlos en un disco. Esto permite al usuario volver y hacer cambios o correcciones a los datos según sea necesario antes de ejecutar el programa. Cuando se diseña un programa para imprimir muchos datos, el resultado se puede enviar a una impresora de alta velocidad u otro archivo de disco. Después de que el programa ha ejecutado, el usuario puede examinar los datos cuando sea más conveniente. En la siguiente sección se analizan la entrada y salida con archivos de disco.

Los programas diseñados para I/O no interactiva no imprimen mensajes de indicación para ingreso. Sin embargo, es una buena idea imprimir por eco cada valor de datos que se lea. La impresión por eco permite que la persona lea el resultado para comprobar que los valores de entrada se prepararon de manera correcta. Debido a que los programas no interactivos tienden a imprimir grandes cantidades de datos, su resultado es con frecuencia en la forma de una tabla; esto es, columnas con encabezados descriptivos.

La mayoría de los programas C++ se escriben para uso interactivo. Pero la flexibilidad del lenguaje permite escribir también programas no interactivos. La gran diferencia reside en los requerimientos de entrada o salida. Los programas no interactivos son en general más rígidos respecto a la organización y formato de los datos de entrada y salida.

4.4 Ingreso y salida de archivos

En todo lo que se ha visto hasta aquí, se ha supuesto que el ingreso de datos para los programas viene del teclado y que el resultado va a la pantalla. Ahora se examina la introducción y salida de datos hacia y desde archivos.

Archivos

Antes se definió un archivo como un área nombrada en el almacenamiento secundario que contiene una colección de información (por ejemplo, el código de programa que se ha tecleado en el editor). La información en un archivo se guarda en un dispositivo de almacenamiento auxiliar, como un disco. Los programas pueden leer datos de un archivo de la misma forma que leen datos desde el teclado, y pueden escribir el resultado en un archivo de disco del mismo modo que lo escriben en pantalla.

¿Por qué se querría que un programa leyera datos de un archivo en lugar de hacerlo desde el teclado? Si un programa va a leer una gran cantidad de datos, es más fácil introducirlos en un archivo con un editor que hacerlo mientras el programa está en ejecución. Con el editor, se puede volver y corregir errores. Asimismo, no es necesario meter todos los datos a la vez; se puede interrumpir la tarea y volver después. Y si se quiere volver a ejecutar el programa, tener almacenados los datos en un archivo permite hacerlo sin volver a teclear los datos.

¿Por qué se desearía escribir el resultado de un programa en un archivo de disco? El contenido de un archivo se puede mostrar en una pantalla o imprimir. Esto da la opción de examinar el resultado una y otra vez sin volver a ejecutar el programa. También, el resultado almacenado en un archivo se puede leer en otro programa como entrada.

Uso de archivos

Si se quiere que un programa use el archivo I/O, se tienen que hacer cuatro cosas:

1. Solicitar al preprocesador que incluya el archivo de encabezado `fstream`.
2. Usar las sentencias de declaración para declarar los flujos de archivo que se van a usar.
3. Preparar cada archivo para lectura o escritura al usar una función llamada `open`.
4. Especificar el nombre del flujo de archivo en cada sentencia de entrada o salida.

Incluir el archivo de encabezado `fstream` Suponga que se desea que el programa Hipoteca del capítulo 3 (p. 108) lea los datos de un archivo y escriba su resultado en un archivo. Lo primero que se debe hacer es usar la directiva de preprocesador

```
#include <fstream>
```

Por medio del archivo de encabezado `fstream`, la biblioteca estándar de C++ define dos tipos de datos, `ifstream` y `ofstream` (que significan *flujo de archivos de entrada* y *flujo de archivos de salida*). Congruente con la idea general de flujos en C++, el tipo de datos `ifstream` representa un flujo de caracteres proveniente de un archivo de entrada, y `ofstream` representa un flujo de caracteres que van a un archivo de salida.

Todas las operaciones `istream` que ha aprendido —el operador de extracción `(>>)`, la función `get` y la función `ignore`— son válidas también para el tipo `ifstream`. Y todas las operaciones `ostream`, como el operador de inserción `(<<)` y los manipuladores `endl`, `setw` y `setprecision`, se aplican también al tipo `ofstream`. A estas operaciones básicas, los tipos `ifstream` y `ofstream` agregan algunas operaciones más, diseñadas específicamente para ingreso o salida de archivos.

Declarar flujos de archivos En un programa, las variables de flujo se declaran de la misma forma que cualquier variable, se especifica el tipo de datos y luego el nombre de la variable:

```
int      someInt;
float    someFloat;
ifstream inFile;
ofstream outFile;
```

(Las variables de flujo `cin` y `cout` no se tienen que declarar. El encabezado de archivo `iostream` lo hace por usted.)

Para el programa Hipoteca, se nombrarán los flujos de archivo de entrada y salida `inData` y `outData`. Se declaran de esta manera:

```
ifstream inData;        // Contiene la cantidad prestada, el interés y la
                           longitud
ofstream outData;       // Contiene valores de entrada y pagos mensuales
```

Observe que el tipo `ifstream` es sólo para archivos de entrada y el tipo `ofstream` es sólo para archivos de salida. Con estos tipos de datos, no se puede escribir y leer del mismo archivo.

Apertura de archivos La tercera cosa que se tiene que hacer es preparar cada archivo para lectura o escritura, un acto denominado *apertura de archivo*. Abrir un archivo ocasiona que el sistema operativo de la computadora lleve a cabo ciertas acciones que permiten proceder con la entrada o salida de archivos.

En el ejemplo, se quiere leer del flujo de archivos `inData` y escribir al flujo de archivos `outData`. Los archivos importantes se abren por medio de estas sentencias:

```
inData.open("loan.in");
outData.open("loan.out");
```

Ambas sentencias son llamadas de función (observe los argumentos delatores, la marca de una función). En cada llamada de función, el argumento es una cadena literal entre comillas. La primera sentencia es una llamada a una función denominada `open`, que se relaciona con el tipo de datos `ifstream`. La segunda es una llamada a otra función (denominada también `open`) relacionada con el tipo de datos `ofstream`. Como se ha visto antes, se emplea la notación de punto (como en `inData.open`) para llamar ciertas funciones de biblioteca estrechamente relacionadas con tipos de datos.

¿Exactamente qué hace una función `open`? Primero, relaciona una variable de flujo empleada en su programa con un archivo físico en disco. La primera llamada de función crea una conexión entre la variable de flujo `inData` y el archivo de disco real, nombrado `loan.in`. (Los nombres de flujos de archivo deben ser identificadores; son variables en su programa. Pero algunos sistemas computarizados no usan esta sintaxis para nombres de archivo en disco. Por ejemplo, muchos sistemas permiten o incluso requieren un punto dentro de un nombre de archivo.) De manera similar, la segunda llamada de función relaciona también la variable de flujo `outData` con el archivo de disco `loan.out`. Asociar un nombre de programa para un archivo (`outData`) con el nombre real para el archivo (`loan.out`) es casi lo mismo que asociar el nombre de un programa para el dispositivo de salida estándar (`cout`) con el dispositivo real (la pantalla).

La tarea siguiente que realiza la función `open` depende de si el archivo es de entrada o de salida. Con un archivo de entrada, la función `open` fija el marcador de lectura del archivo a la primera pieza de datos en el archivo. (Cada archivo de entrada tiene su propio marcador de lectura.)

Con un archivo de salida, la función `open` comprueba si ya existe el archivo. Si no existe, `open` crea para usted un nuevo archivo vacío. Si el archivo ya existe, `open` borra el contenido anterior del archivo. Luego, el marcador de escritura se fija al comienzo del archivo vacío (véase la figura 4-2). Conforme procede la salida, cada operación de salida sucesiva hace avanzar el marcador de escritura para añadir datos al final del archivo.

Debido a que la razón para abrir archivos es *prepararlos* para lectura o escritura, se deben abrir los archivos antes de usar cualquier sentencia de entrada o salida que se refiera a los archivos. En un programa, es una buena idea abrir de inmediato los archivos para asegurarse de que están preparados antes de que el programa intente cualquier ingreso o salida de archivos.

```
    ⋮

int main()
{
     ⋮     } Declaraciones

   // Abrir los archivos
```

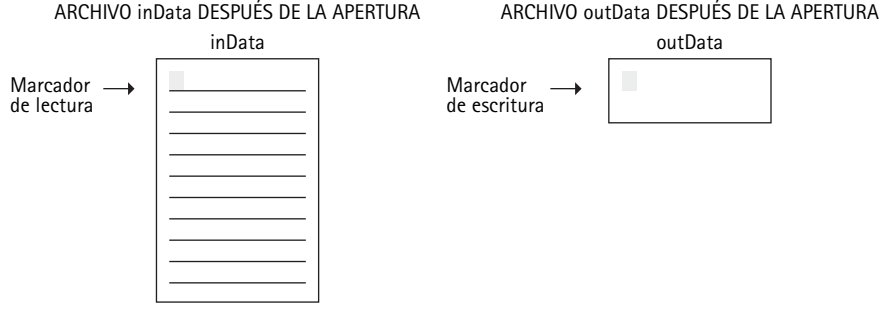


Figura 4-2 *Efecto de abrir un archivo*

```
    inData.open("loan.in");
    outData.open("loan.out");
      ⋮
}
```

Además de la función `open`, los tipos `ifstream` y `ofstream` tienen una función `close` relacionada con cada uno. Esta función no tiene argumentos y se puede usar como sigue.

```
ifstream inFile;

inFile.open("mydata.dat");     // Abrir el archivo
 ⋮                             // Leer y procesar los datos de archivo
inFile.close();                // Cerrar el archivo
 ⋮
```

Cerrar un archivo ocasiona que el sistema operativo lleve a cabo ciertas actividades finales en el disco e interrumpa la conexión entre la variable de flujo y el archivo de disco.

¿Debe llamar siempre a la función `close` cuando usted termine de leer o escribir un archivo? En algunos lenguajes de programación, es extremadamente importante que recuerde hacerlo. Sin embargo, en C++ un archivo se cierra de modo automático cuando el control de programa deja el bloque (sentencia compuesta) en la que se declara la variable de flujo. (Hasta llegar al capítulo 7, este bloque es el cuerpo de la función `main`.) Cuando el control deja este bloque, se ejecuta de manera implícita una función especial relacionada con `ifstream` y `ofstream` denominada *destructora*, y esta función destructora cierra el archivo por usted. En consecuencia, no se observa con frecuencia que los programas en C++ llamen de modo explícito a la función `close`. Por otro lado, muchos programadores prefieren que sea un hábito llamar a la función `close` de modo explícito, y quizás usted mismo desee hacerlo así.

Especificar flujos de archivo en sentencias de entrada y salida Hay sólo una cosa más que se tiene que hacer cuando se usan archivos. Como se mencionó, todas las operaciones `istream` son válidas también para el tipo `ifstream`, y todas las operaciones `ostream` son válidas para el tipo `ofstream`. Así, para leer o escribir en un archivo, todo lo que se necesita hacer en las sentencias de entrada y salida es sustituir la variable de flujo de archivos apropiada para `cin` o `cout`. En el programa Hipoteca, se usaría una sentencia como

```
inData >> loanAmount >> yearlyInterest >> numberOfYears;
```

para instruir a la computadora para que lea datos del archivo `inData`. De manera similar, todas las sentencias de salida que escriben en el archivo `outData` especificarían `outData`, y no `cout`, como el destino:

```
    outData << fixed << setprecision(2) << "Para una cantidad prestada de "
            << loanAmount
            << " con una tasa de interés de " << setprecision(4)
            << yearlyInterest  << " y una "
            << numberOfYears << " hipoteca anual, " << endl;
    outData << fixed << setprecision(2)
            << "sus pagos mensuales son de $" << payment
            << "." << endl;
```

Lo bueno en la entrada o salida de flujo de C++ es que se tiene una sintaxis uniforme para efectuar operaciones de entrada o salida (I/O), sin importar si se trabajó con el teclado o con la pantalla, con archivos o con otros dispositivos I/O.

Programa de ejemplo con archivos

A continuación se muestra el programa Hipoteca revisado. Ahora lee su entrada del archivo `inData` y escribe su salida al archivo `outData`. Compare este programa con la versión original de la página 108 y observe que las constantes nombradas han desaparecido porque los datos son ahora la entrada en tiempo de ejecución. Observe que para establecer el formato de resultado de punto flotante, los manipuladores `fixed` y `setprecision` se aplican a la variable de flujo `outData`, no a `cout`.

```
//*****************************************************************
// Programa Calculadora de pago de hipoteca
// Este programa determina los pagos mensuales de una hipoteca dadas
// la cantidad prestada, la tasa de interés anual y el número de
// años.
//*****************************************************************

#include <fstream>
#include <cmath>
#include <iomanip>

using namespace std;
#include <fstream>
#include <cmath>
#include <iomanip>

using namespace std;

int main()
{

    // Input variables
    float loanAmount;
    float yearlyInterest;
    int numberOfYears;
    ofstream outData;
    ifstream inData;

    // Local variables
    float monthlyInterest;
    int numberOfPayments;
    float payment;

    inData.open("loan.in");
    outData.open("loan.out");

    // Read values from the file
    inData >> loanAmount >> yearlyInterest >> numberOfYears;

    // Calculate values
    monthlyInterest = yearlyInterest * 0.01 / 12;
    numberOfPayments = numberOfYears * 12;
    payment = (loanAmount
               * pow(1 + monthlyInterest, numberOfPayments)
               *  monthlyInterest)
               /(pow(1 + monthlyInterest, numberOfPayments) - 1 );
```

```
    // Output results
    outData << fixed << setprecision(2) << "Para una cantidad prestada de"
            << loanAmount
            << "con una tasa de interés de" << setprecision(4)
            << yearlyInterest  << "y una"
            << numberOfYears << "hipoteca anual" << endl;
    outData << fixed << setprecision(2)
            << "sus pagos mensuales son $" << payment
            << "." << endl;
    inData.close();
    outData.close();
    return 0;
}
```

Antes de ejecutar el programa, se usaría el editor para crear y guardar un archivo `loan.in` para servir de entrada. El contenido del archivo podría parecerse a esto:

```
50000.00 0.05 5
```

Al escribir el nuevo programa Hipoteca, ¿qué sucede si erróneamente especifica `cout` en lugar de `outData` en una de las sentencias de salida? Nada grave; la salida de esa sentencia sólo va a la pantalla en lugar de al archivo de salida. ¿Y qué pasa si, por error, especifica `cin` en lugar de `inData` en la sentencia de salida? Las consecuencias no son tan gratas. Cuando se ejecuta el programa, parecerá que la computadora está bloqueada (deja de funcionar). Aquí está la razón: la ejecución llega a la sentencia de entrada y la computadora espera que usted introduzca los datos desde el teclado. Pero usted no sabe que la computadora está esperando. No hay mensaje en la pantalla que lo indique, y usted supone (erróneamente) que el programa obtiene su entrada desde el archivo de datos, así que la computadora espera, y usted también, y los dos siguen esperando. Todo programador alguna vez ha tenido la experiencia de enfrentarse con que la computadora ya no funciona, cuando, de hecho, está trabajando bien, esperando en silencio el ingreso de datos desde el teclado.

Ingreso de nombres de archivo en tiempo de ejecución

Hasta ahora, los ejemplos para abrir un archivo de entrada han incluido un código similar a lo siguiente:

```
ifstream inFile;

inFile.open("datafile.dat");
 ⋮
```

La función `open` relacionada con el tipo de datos `ifstream` requiere un argumento que especifique el nombre del archivo de datos real en disco. Al usar una cadena literal, como en el ejemplo anterior, el nombre de archivo se fija al momento de compilar. Por tanto, el programa funciona sólo para este archivo de disco particular.

Con frecuencia se desea hacer más flexible un programa al permitir que el nombre de archivo sea determinado en *tiempo de ejecución*. Una técnica común es solicitar al usuario el nombre del archivo, leer la respuesta del usuario en una variable y pasar la variable como un argumento a la función `open`. En principio, el siguiente código debe llevar a cabo lo que se desea. Infortunadamente, el compilador no lo permite.

```
ifstream inFile;
string   fileName;

cout << "Introduzca el nombre del archivo de entrada: ";
cin >> fileName;
inFile.open(fileName);                // Compile-time error
```

El problema es que la función open no espera un argumento del tipo `string`. En cambio, espera una *cadena C*. Una cadena C (llamada así porque se originó en el lenguaje C, el precursor de C++) es una forma limitada de cadena cuyas propiedades se analizan más adelante en este libro. Una cadena literal, como `"datafile.dat"`, es una cadena C y, por tanto, es aceptable como argumento para la función `open`.

Para hacer que el código anterior funcione de manera correcta, es necesario convertir una variable `string` en una cadena C. El tipo de datos `string` proporciona una función de devolución de valor llamada `c_str` que se aplica a una variable `string` como sigue:

```
fileName.c_str()
```

Esta función devuelve la cadena C, que es equivalente a la contenida en la variable `fileName`. (La cadena original contenida en `fileName` no se cambia por la función invocadora.) El propósito principal de la función `c_str` es permitir que los programadores llamen funciones de biblioteca que esperan cadenas C, no cadenas `string`, como argumentos.

Con la función `c_str`, se puede codificar la entrada en tiempo de ejecución de un nombre de archivo como sigue:

```
ifstream inFile;
string   fileName;

cout << "Introduzca el nombre del archivo de entrada: ";
cin >> fileName;
inFile.open(fileName.c_str());
```

4.5 Falla de la entrada

Cuando un programa introduce datos desde el teclado o un archivo de entrada, las cosas pueden ir mal. Suponga que se está ejecutando un programa. Éste solicita que se introduzca un valor entero, pero por distracción se teclean algunas letras del alfabeto. La operación de entrada falla debido a los datos no válidos. En terminología C++, el flujo `cin` ha introducido el *estado de falla*. Una vez que un flujo ha introducido el estado de falla, cualquier operación adicional I/O con ese flujo es considerada una operación nula; es decir, no tiene efecto en absoluto. Por desgracia para nosotros, *la computadora no detiene el programa o emite un mensaje de error*. La computadora sólo continúa la ejecución del programa, en silencio, ignorando cada intento adicional de usar ese flujo.

La introducción de datos no válidos es la causa más común para la falla de entrada. Cuando su programa introduce un valor `int`, está esperando hallar sólo dígitos en el flujo de entrada, precedidos quizá por un signo más o menos. Si hay un punto decimal en alguna parte dentro de los dígitos, ¿falla la operación de entrada? No necesariamente; depende de dónde esté el marcador de lectura. Considérese un ejemplo.

Suponga que un programa tiene variables `int i`, `j` y `k`, cuyo contenido es 10, 20 y 30, respectivamente. El programa ejecuta ahora los dos enunciados siguientes:

```
cin >> i >> j >> k;
cout << "i: " << i << "  j: " << j << "  k: " << k;
```

Si se teclean estos caracteres para los datos de entrada:

```
1234.56 7 89
```

entonces el programa produce este resultado:

```
i: 1234  j: 20  k: 30
```

Veamos por qué.

Recuerde que al leer datos `int` o `float`, el operador de extracción `>>` detiene la lectura en el primer carácter que es inapropiado para el tipo de datos (espacio en blanco o algún otro). En nuestro ejemplo, la operación de entrada para `i` tiene éxito. La computadora extrae los primeros cuatro caracteres del flujo de entrada y almacena el valor entero 1234 en `i`. El marcador de lectura ahora está en el punto decimal:

```
1234.56 7 89
```

La siguiente operación de entrada (para `j`) falla; un valor `int` no puede comenzar con un punto decimal. El flujo `cin` está ahora en el estado de falla, y el valor actual de `j` (20) permanece sin cambio. Se ignora la tercera operación de entrada (para `k`), al igual que el resto de las sentencias del programa que leen de `cin`.

Otra forma de hacer que un flujo entre al estado de falla es tratar de abrir un archivo de entrada que no existe. Suponga que en su disco tiene un archivo de datos llamado `myfile.dat`. En su programa tiene las siguientes sentencias:

```
ifstream inFile;


inFile.open("myfil.dat");
inFile >> i >> j >> k;
```

En la llamada para la función `open`, usted escribió mal el nombre de su archivo en disco. Al momento de la ejecución, falla el intento de abrir el archivo, así que el flujo `inFile` entra al estado de falla. Las tres operaciones de entrada siguientes (para `i`, `j` y `k`) son operaciones nulas. Sin emitir ningún mensaje de error, el programa procede a usar el contenido (desconocido) de `i`, `j` y `k` en los cálculos. Los resultados de estos cálculos son confusos.

El punto de esta cuestión no es asustarlo en cuanto a I/O, sino hacer que esté consciente. En la sección de Prueba y depuración, al final de este capítulo, se ofrecen sugerencias para evitar la falla de entrada, y en los capítulos 5 y 6 se presentan sentencias de programa que permiten probar el estado de un flujo.

4.6 Metodologías de diseño de software

En los dos capítulos anteriores, y en la primera parte de éste, se introdujeron elementos del lenguaje C++ que permiten insertar datos, realizar cálculos y producir resultados. Los programas que se escribieron fueron cortos y directos debido a que los problemas por resolver fueron simples. Ya se está listo para escribir programas para problemas más complicados, pero primero es necesario volver atrás y considerar el proceso global de programar.

Según se aprendió en el capítulo 1, el proceso de programación consta de una fase de resolución de problemas y una fase de ejecución. La fase de resolución de problemas incluye *análisis* (analizar y comprender el problema por resolver) y *diseño* (diseñar una solución para el problema). Dado un problema complejo —uno que da como resultado un programa de 10 000 líneas, por ejemplo—, simplemente no es razonable omitir el proceso de diseño e ir directo a la escritura de código C++. Es necesaria una forma sistemática de diseñar una solución para un problema, sin importar cuán complicado sea.

En el resto de este capítulo se describen dos metodologías importantes para diseñar soluciones de problemas más complejos: *descomposición funcional* y *diseño orientado a objetos*. Estas metodologías ayudan a crear soluciones que pueden realizarse de manera fácil como programas de C++. Los programas resultantes son legibles, comprensibles y fáciles de depurar y modificar.

La metodología de diseño de software de uso extendido se conoce como **diseño orientado a objetos (DOO).** C++ evolucionó del lenguaje C sobre todo para facilitar el uso de la metodología DOO. En las dos secciones siguientes se presentan los conceptos esenciales del DOO; después, en

Diseño orientado a objetos Técnica para desarrollar software en la cual la solución se expresa en términos de objetos, entidades autocontenidas compuestas de datos y operaciones en esos datos.

Descomposición funcional Técnica para desarrollar software en la cual el problema se divide en subproblemas más fáciles de manejar, cuyas soluciones crean una solución para el problema global.

el libro, se amplía el tratamiento del método. El DOO se emplea con frecuencia junto con otra metodología que se estudia en este capítulo, la **descomposición funcional**.

El DOO se enfoca en entidades (*objetos*) que constan de datos y operaciones en los datos. En el DOO, un problema se resuelve al identificar los componentes que constituyen una solución y al identificar también cómo esos componentes interactúan entre sí a través de operaciones en los datos que contienen. El resultado es un diseño para un conjunto de objetos que pueden ser ensamblados para formar una solución al problema. En contraste, la descomposición funcional ve la solución a un problema como una tarea por realizar. Se centra en la secuencia de operaciones requeridas para completar la tarea. Cuando el problema requiere una secuencia de pasos que es larga o compleja, se divide en subproblemas que son más fáciles de resolver.

La elección de qué metodología usar depende del problema en cuestión. Por ejemplo, un problema grande podría requerir varias fases sucesivas de proceso, como reunir datos y comprobar su exactitud con procesamiento no interactivo, analizar los datos de manera interactiva e imprimir informes no interactivamente en la conclusión del análisis. Este proceso tiene una descomposición funcional natural. Sin embargo, cada una de las fases se podría resolver mejor mediante un conjunto de objetos que representa los datos y las operaciones que se le pueden aplicar. Algunas de las operaciones individuales pueden ser tan complejas que requieren más descomposición, ya sea en una secuencia de operaciones o en otro conjunto de objetos.

Si examina un problema y considera que es natural pensar en él en términos de una colección de partes componentes, entonces debe usar el DOO para resolverlo. Por ejemplo, un problema de transacciones bancarias podría requerir un objeto `checkingAccount` con operaciones relacionadas `OpenAccount`, `WriteCheck`, `MakeDeposit` e `IsOverdrawn`. El objeto `checkingAccount` consta no sólo de datos (el número de cuenta y el saldo actual, por ejemplo) sino también de estas operaciones, todas juntas en una unidad.

Por otro lado, si cree que es natural pensar en la solución del problema como una serie de pasos, entonces debe usar la descomposición funcional. Por ejemplo, al calcular algunas medidas estadísticas en un gran conjunto de números reales, es natural descomponer el problema en una secuencia de pasos que leen un valor, efectúan cálculos y después repiten la secuencia. El lenguaje C++ y la biblioteca estándar proporcionan todas las operaciones necesarias, y sólo se escribe una secuencia de esas operaciones para resolver el problema.

4.7 ¿Qué son los objetos?

Se examinará más de cerca lo que son los objetos y cómo funcionan antes de estudiar más el DOO. Se dijo que un objeto es una colección de datos junto con operaciones relacionadas. Se han creado varios lenguajes de programación, llamados *lenguajes de programación orientados a objetos*, específicamente para apoyar el DOO. Algunos ejemplos son C++, Java, Smalltalk, CLOS, Eiffel y Object-Pascal. En estos lenguajes, una *clase* es un tipo de datos definidos por el programador, de los cuales se crean los objetos. Aunque no se dijo en el momento, se han estado usando clases y objetos para efectuar la entrada y la salida en C++. `cin` es un objeto de un tipo de datos (clase) llamado `istream`, y `cout` es un objeto de clase `ostream`. Como se explicó, el archivo de encabezado `iostream` define las clases `istream` y `ostream`, y también declara `cin` y `cout` como objetos de esas clases:

```
istream cin;
ostream cout;
```

De manera similar, el encabezado de archivo `fstream` define las clases `ifstream` y `ofstream`, de las cuales usted puede declarar sus propios objetos de flujo de archivos de entrada y flujo de archivos de salida.

Otro ejemplo que usted ya ha visto es `string`, una clase definida por el programdor, de la cual usted crea objetos por medio de declaraciones como

```
string lastName;
```

En la figura 4-3 se ilustran los objetos `cin` y `lastName` como entidades que tienen una parte privada y una parte pública. La parte privada incluye datos y funciones a los que el usuario no puede tener acceso y no necesita conocer para usar el objeto. La parte pública, mostrada como óvalos en el lado del objeto, representa la *interfaz* del objeto. La interfaz consta de operaciones disponibles para los programadores que deseen usar el objeto. En C++, las operaciones públicas se escriben como funciones y se conocen como *funciones miembro*. Excepto para operaciones con símbolos como `<<` y `>>`, una función miembro es invocada al dar el nombre del objeto de clase, luego un punto y después el nombre de la función y la lista de argumentos:

```
cin.ignore(100, '\n');
cin.get(someChar);
cin >> someInt;
len = lastName.length();
pos = lastName.find('A');
```

4.8 Diseño orientado a objetos

El primer paso en el DOO es identificar los objetos principales del problema, junto con sus operaciones relacionadas. La solución del problema final se expresa en última instancia en términos de estos objetos y operaciones.

El DOO carga programas que son colecciones de objetos. Cada objeto es responsable de una parte de la solución completa, y los objetos se comunican al tener acceso a cada una de las funciones miembro de los otros. Hay muchas bibliotecas de clases preescritas, incluso la biblioteca estándar C++, bibliotecas públicas (llamadas *freeware* o *shareware*), bibliotecas que son ofrecidas comercialmente y bibliotecas que son desarrolladas por compañías para su uso propio. En muchos casos, es posible consultar una biblioteca, elegir las clases que necesita para solucionar un problema y ensamblarlas para formar una parte sustancial de su programa. Juntar las piezas de este modo es un excelente ejemplo del método de bloques de construcción analizado en el capítulo 1.

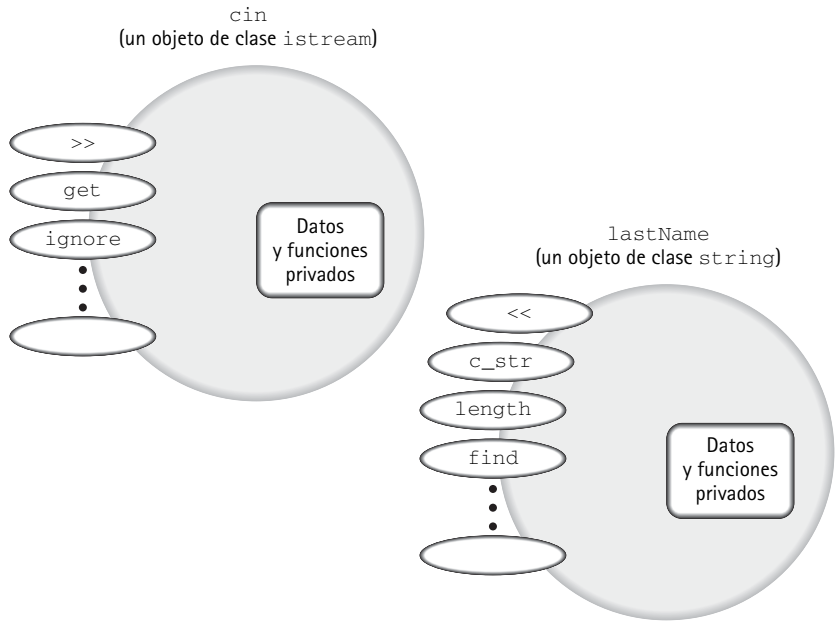


Figura 4-3 *Objetos y sus operaciones*

Cuando no hay una clase disponible en una biblioteca, es necesario definir una nueva clase. En el capítulo 11 se muestra cómo se hace esto. El diseño de una clase nueva comienza con la especificación de su interfaz. Se debe decidir qué operaciones son necesarias en el exterior de la clase para hacer útiles sus objetos. Una vez que se define la interfaz, se puede diseñar la implementación de la clase, incluso todos sus miembros privados.

Uno de los objetivos de diseñar una interfaz es hacerla flexible para que la nueva clase se pueda usar en circunstancias imprevistas. Por ejemplo, se puede proporcionar una función miembro que convierte el valor de un objeto en una cadena, aun cuando no se requiera esta capacidad en el programa. Cuando llega el momento de depurar el programa, puede ser muy útil mostrar valores de este tipo como cadenas.

Las características útiles con frecuencia están ausentes de una interfaz, en ocasiones debido a la falta de previsión y otras veces con el fin de simplificar el diseño. Es muy común descubrir una clase en una biblioteca que es casi correcta para el fin que se persigue, pero falta alguna característica clave. El DOO atiende esta situación con un concepto llamado *herencia*, el cual permite adaptar una clase existente para satisfacer sus necesidades particulares. Se puede usar la herencia para añadir características a una clase (o restringir el uso de características existentes) sin tener que inspeccionar o modificar su código fuente. La herencia se considera parte integral de la programación orientada a objetos en que se usa un término separado, *programación basada en objetos*, para describir la programación con objetos pero no la herencia.

En el capítulo 14 se muestra cómo definir clases que heredan miembros de clases existentes. Juntos, el DOO, las bibliotecas de clase y la herencia, pueden reducir de modo notable el tiempo y el esfuerzo requeridos para diseñar, ejecutar y mantener grandes sistemas de software.

Para resumir el proceso de DOO, se identifican los componentes principales de una solución de problema y cómo interactúan. Luego, en las bibliotecas disponibles se buscan clases que corresponden a los componentes. Cuando se encuentra una clase que es casi correcta, se puede usar la herencia para adaptarla. Cuando no se puede hallar una clase que corresponda a un componente, se debe diseñar una nueva clase. El diseño especifica la interfaz para la clase, y luego se pone en práctica la interfaz con miembros públicos y privados según sea necesario. El DOO no siempre se usa por separado. La descomposición funcional se puede usar al diseñar funciones miembro dentro de una clase o al coordinar las interacciones de objetos.

En esta sección se ha presentado sólo una introducción al DOO. Una descripción más completa requiere conocer temas que se estudian en los capítulos 5 a 10: flujo de control, funciones escritas por el programador y más acerca de tipos de datos. En los capítulos 11 a 13 se aprende cómo escribir nuestras propias clases y se vuelve al DOO en el capítulo 14. Hasta entonces, los programas son relativamente pequeños, de modo que se usa la programación basada en objetos y la descomposición funcional para llegar a la solución de problemas.

4.9 Descomposición funcional

La segunda técnica de diseño que se emplea es la descomposición funcional (conocida también como *diseño estructurado*, *diseño jerárquico*, *refinación por pasos* y *programación modular*). En la descomposición funcional se trabaja de lo abstracto (una lista de pasos principales en la solución) a lo particular (pasos algorítmicos que se pueden traducir directamente en código C++). Esto se puede considerar también como trabajar de una solución de alto nivel, dejando sin especificar los detalles de la implementación, a una solución completamente detallada.

La forma más fácil de resolver un problema es darlo a alguien y decirle: "resuelve este problema". Éste es el nivel más abstracto de la solución de un problema: una solución de una sola sentencia que abarca todo el problema sin especificar los detalles de la implementación. Es en este punto cuando se llama a los programadores. El trabajo es hacer de la solución abstracta una solución concreta: un programa.

Si la solución tiene que ver con una serie de pasos principales, se descompone en piezas. En el proceso, se pasa a un nivel de abstracción menor, es decir, ahora se especifican algunos de los detalles de la implementación (pero no demasiados). Cada uno de los pasos principales se torna un subproblema

independiente que se puede trabajar por separado. En un proyecto muy grande, una persona (el *arquitecto jefe o* un *líder de equipo*) formula los subproblemas y los asigna a otros miembros del equipo de programación, diciendo: "Resuelve este problema". En el caso de un proyecto pequeño, uno mismo se asigna los subproblemas. Después se elige un subproblema a la vez para resolver. Se puede descomponer el subproblema elegido en otra serie de pasos que, a su vez, se vuelven subproblemas más pequeños. O bien, se pueden identificar los componentes que son representados de modo natural como objetos. El proceso continúa hasta que cada subproblema no se puede dividir más o tiene una solución obvia.

¿Por qué se trabaja de esta forma? ¿Por qué no simplemente se escriben todos los detalles? Porque es mucho más fácil enfocarse en un problema a la vez. Por ejemplo, suponga que está trabajando en una parte de un programa para producir ciertos valores y descubre que necesita una fórmula más compleja para calcular un ancho de campo apropiado para imprimir uno de los valores. Calcular los anchos de campo no es el propósito de esta parte del programa. Si cambia su atención al cálculo, tal vez olvide algunos detalles del proceso de salida global. Lo que hace es escribir un paso abstracto —"Calcule el ancho de campo requerido"— y continuar con el problema en cuestión. Una vez que ha escrito los pasos principales, puede volver para resolver el paso que realiza el cálculo.

Al subdividir el problema crea una estructura jerárquica llamada *estructura de árbol*. Cada nivel del árbol es una solución completa para el problema que es menos abstracto (más detallado) que el nivel anterior. En la figura 4-4 se muestra un árbol solución genérico para un problema. Los pasos sombreados tienen suficientes detalles de implementación que se traducirán directamente en sentencias de C++. Éstos son **pasos concretos**. Los no sombreados son **pasos abstractos**; reaparecen como subproblemas en el siguiente nivel descendente. Cada caja

Paso concreto Paso para el cual los detalles de implementación se especifican por completo.

Paso abstracto Paso para el cual algunos detalles de la implementación permanecen sin especificar.

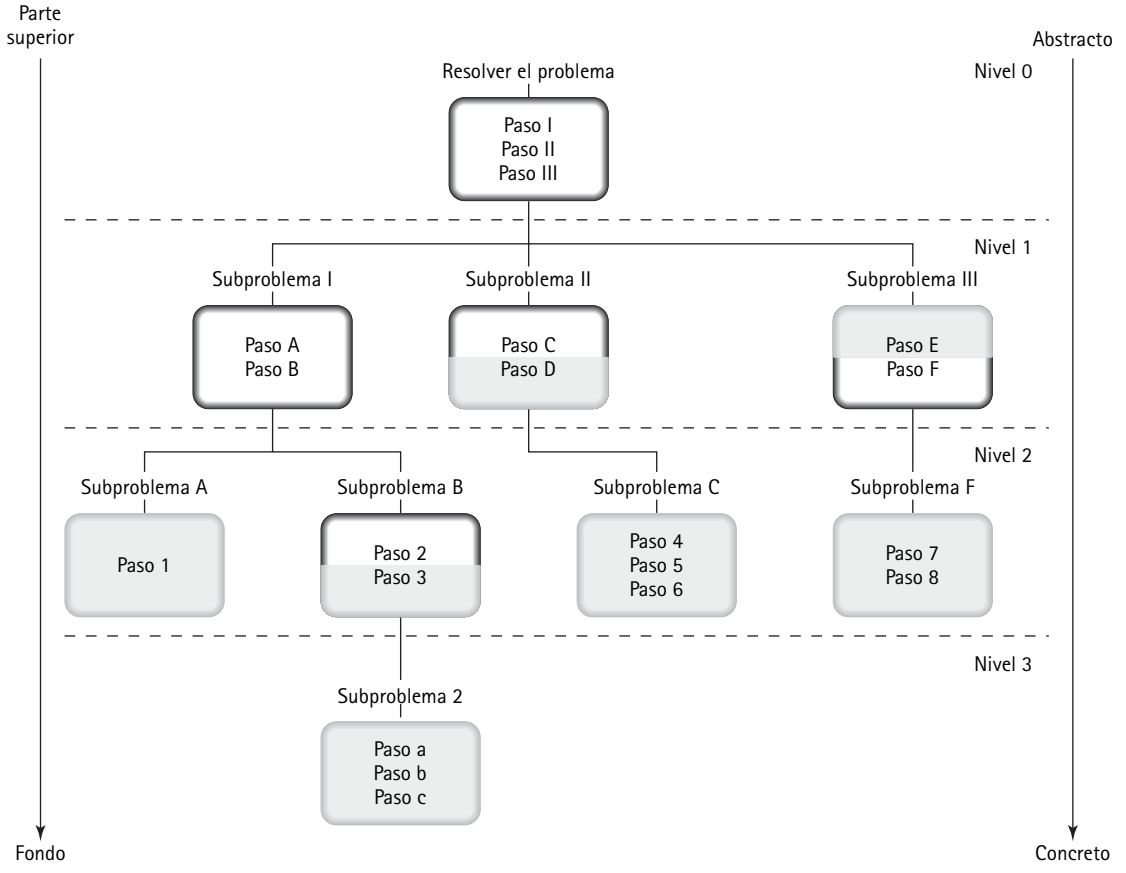


Figura 4-4 *Árbol solución jerárquico*

Módulo Colección de pasos autocontenida que resuelve un problema o subproblema; puede contener pasos concretos y abstractos.

de la figura representa un **módulo**. Éstos son los bloques de construcción básicos en una descomposición funcional. El diagrama de la figura 4-4 se llama también *gráfica de estructura de módulo*.

Al igual que el DOO, la descomposición funcional emplea el método de "divide y vencerás" para la solución de problemas. En ambas técnicas los problemas grandes se descomponen en unidades más pequeñas que son más fáciles de manejar. La diferencia es que en el DOO las unidades son objetos, mientras que las unidades en la descomposición funcional son módulos que representan algoritmos.

Módulos

Un módulo comienza la vida como un paso abstracto en el nivel superior siguiente del árbol solución. Se completa cuando resuelve un subproblema determinado, es decir, cuando se especifica una serie de pasos que hace lo mismo que el paso abstracto del nivel superior. En esta etapa, un módulo es **funcionalmente equivalente** al paso abstracto. (No confunda nuestro uso de *función* con las funciones de C++. Aquí se emplea el término para hacer referencia al papel específico que desempeña el módulo o paso en una solución algorítmica.)

Equivalencia funcional Propiedad de un módulo que efectúa exactamente la misma operación que el paso abstracto que define. Un par de módulos son funcionalmente equivalentes entre sí cuando efectúan exactamente la misma operación.

Cohesión funcional Propiedad de un módulo en que los pasos concretos son dirigidos a resolver sólo un problema, y los subproblemas importantes se escriben como pasos abstractos.

En un módulo escrito de manera apropiada, los únicos pasos que atiende de modo directo el subproblema determinado son los pasos concretos; los pasos abstractos se emplean para nuevos subproblemas significativos. Esto se llama **cohesión funcional**.

La idea detrás de la cohesión funcional es que cada módulo debe hacer una sola cosa y hacerla bien. La cohesión funcional no es una propiedad bien definida; no hay medida cuantitativa de cohesión. Es un producto de la necesidad humana de organizar las cosas en partes nítidas que son fáciles de entender y recordar. Saber qué detalles hacer concretos y cuáles dejar abstractos es un asunto de experiencia, circunstancia y estilo personal. Por ejemplo, usted podría decidir incluir un cálculo de ancho de campo en un módulo de impresión si no hay muchos detalles en el resto del módulo que se vuelva confuso. Por otro lado, si el cálculo se efectúa varias veces, tiene sentido escribirlo como un módulo separado y sólo referirse a él cada vez que lo necesite.

Escribir módulos cohesivos A continuación se presenta un método para escribir módulos cohesivos:

1. Piense en cómo resolver el problema en cuestión.
2. Comience escribiendo los pasos principales.
3. Si un paso es lo suficientemente simple para imaginar cómo ejecutarlo directamente en C++, está en el nivel concreto; no necesita refinamiento adicional.
4. Si tiene que pensar en ejecutar un paso como una serie de pasos más pequeños o como varias sentencias de C++, aún está en un nivel abstracto.
5. Si está tratando de escribir una serie de pasos y comienza a sentirse abrumado por los detalles, es probable que haya omitido uno o más niveles de abstracción. Deténgase a reflexionar y busque piezas que pueda escribir como pasos más abstractos.

Esto se podría denominar "técnica del moroso". Si un paso es difícil, pospóngalo para un nivel inferior; no piense en él hoy, piense en él mañana. Por supuesto, el mañana llega, pero el proceso completo se puede aplicar de nuevo al subproblema. Con frecuencia, un problema parece mucho más simple cuando usted puede centrarse en él. Finalmente, el problema completo se descompone en unidades controlables.

A medida que avanza por el árbol solución, realiza una serie de decisiones de diseño. Si una decisión demuestra ser difícil o equivocada (¡y muchas veces lo es!), puede retroceder (ir por el árbol hasta el módulo de nivel superior) e intentar alguna otra cosa. No tiene que descartar todo su diseño, sólo la pequeña parte con la que está trabajando. Podría haber muchos pasos intermedios y soluciones de prueba antes de llegar al diseño final.

Seudocódigo Encontrará que es más fácil poner en práctica un diseño si escribe los pasos en seudocódigo. El *seudocódigo* es una combinación de sentencias y estructuras de control parecidas a las de

C++ que se pueden traducir fácilmente en C++. (Se ha estado usando seudocódigos en los algoritmos de los Casos prácticos de resolución de problemas.) Cuando un paso concreto se escribe en seudocódigo, debe ser posible reescribirlo directamente como una sentencia de C++ en un programa.

Implementación del diseño

El producto de la descomposición funcional es una solución jerárquica a un problema con múltiples niveles de abstracción. En la figura 4-5 se muestra la descomposición funcional para el programa Hipoteca del capítulo 3. Esta clase de solución constituye la base para la fase de implementación de la programación.

¿Cómo se traduce una descomposición funcional en un programa de C++? Si observa con detenimiento la figura 4-5, verá que los pasos concretos (los que están sombreados) pueden ser ensamblados en un algoritmo completo para resolver el problema. El orden en que se ensamblan se determina por su posición en el árbol. Se empieza en la parte superior del árbol, en el nivel 0, con el primer paso, "Definir constantes". Debido a que esto es abstracto, se debe pasar al siguiente nivel (nivel 1). Ahí se encuentra una serie de pasos concretos que corresponden a este paso; esta serie de pasos se convierte en la primera parte del algoritmo. Debido a que ahora el proceso de conversión es concreto, se puede volver al nivel 0 y continuar con el paso siguiente: "Calcular valores". Dado que esto es abstracto, se pasa al nivel 1 y se encuentra una serie de pasos concretos que corresponden a este paso; esta serie de pasos se convierte en la siguiente parte del algoritmo. Volviendo al nivel 0, se continúa con el siguiente paso: "Producir resultados".

```
Fijar LOAN_AMOUNT = 50000.00
Fijar NUMBER_OF_YEARS = 7
Fijar INTEREST_RATE = 0.0524
Establecer monthlyInteres en YEARLY_INTEREST dividido entre 12
Establecer numberOfPayments en NUMBER_OF_YEARS multiplicado por 12
Establecer payment en (LOAN_AMOUNT*
pow(monthlyInterest+1,numberOfPayments)
  *monthlyInterest)/(pow(monthlyInterest+1, numberOfPayments)-1)
Imprimir "Para una cantidad prestada de" LOAN_AMOUNT "con una tasa de interés de"
  YEARLY_INTEREST "y una " NUMBER_OF_YEARS "hipoteca anual",
Imprimir "sus pagos mensuales son de $" payment "."
```

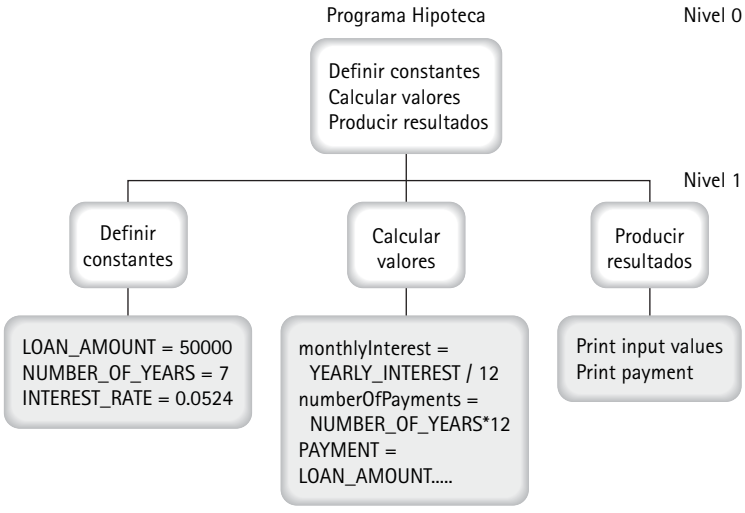


Figura 4-5 *Árbol solución para el programa*

A partir de este algoritmo se puede elaborar una tabla de las constantes y variables requeridas, y luego escribir las declaraciones y sentencias ejecutables del programa.

En la práctica, usted escribe su diseño no como un diagrama de árbol sino como una serie de módulos agrupados por niveles de abstracción.

Principal **Nivel 0**

```
Definir constantes
Calcular valores
Producir resultados
```

Definir constantes

```
Fijar LOAN_AMOUNT = 50000.00
Fijar NUMBER_OF_YEARS = 7
Fijar INTEREST_RATE = 0.05245
```

Calcular valores

```
Establecer monthlyInteres en YEARLY_INTEREST dividido entre 12
Establecer numberOfPayments en NUMBER_OF_YEARS multiplicado por 12
Establecer payment en (LOAN_AMOUNT*pow(monthlyInterest+1,numberOfPayments)
  *monthlyInterest)/(pow(monthlyInterest+1, numberOfPayments)-1)
```

Producir resultados

```
Imprimir "Para una cantidad prestada de" LOAN_AMOUNT "con una tasa de interés de"
  YEARLY_INTEREST "y una " NUMBER_OF_YEARS "hipoteca anual",
Imprimir "sus pagos mensuales son $" payment "."
```

Si examina el programa de C++ para Hipoteca, puede ver que se asemeja mucho a esta solución. La diferencia principal es que en el capítulo 3 no se escribió un "Módulo principal". Se puede ver también que los nombres de los módulos han sido parafraseados como comentarios en el código.

El tipo de implementación que se ha introducido aquí se denomina *plana* o *implementación en línea*. Se está allanando la estructura jerárquica, bidimensional, de la solución al escribir todos los pasos como una secuencia larga. Esta clase de implementación es adecuada cuando una solución es corta y sólo tiene pocos niveles de abstracción. Los programas que produce son claros y fáciles de entender, suponiendo comentarios apropiados y buen estilo.

Los programas más largos, con más niveles de abstracción, son difíciles de resolver como implementaciones planas. En el capítulo 7 se verá que es preferible poner en práctica una solución jerárquica por medio de una *implementación jerárquica*. Ahí se ejecutan muchos de los módulos escribiéndolos como funciones de C++ separadas, y los pasos abstractos en el diseño se remplazan con llamadas a esas funciones.

Una de las ventajas de ejecutar módulos como funciones es que pueden ser llamados desde lugares distintos en un programa. Por ejemplo, si un problema requiere que el volumen de un cilindro sea calculado en varios lugares, se podría escribir una solución función para efectuar el cálculo y simplemente llamarla en cada lugar. Esto da una *implementación semijerárquica*. La implementación no conserva una jerarquía pura porque los pasos abstractos en varios niveles del árbol solución comparten una implementación de un módulo (véase la figura 4-6). Un módulo compartido cae fuera de la jerarquía porque no pertenece a ningún nivel.

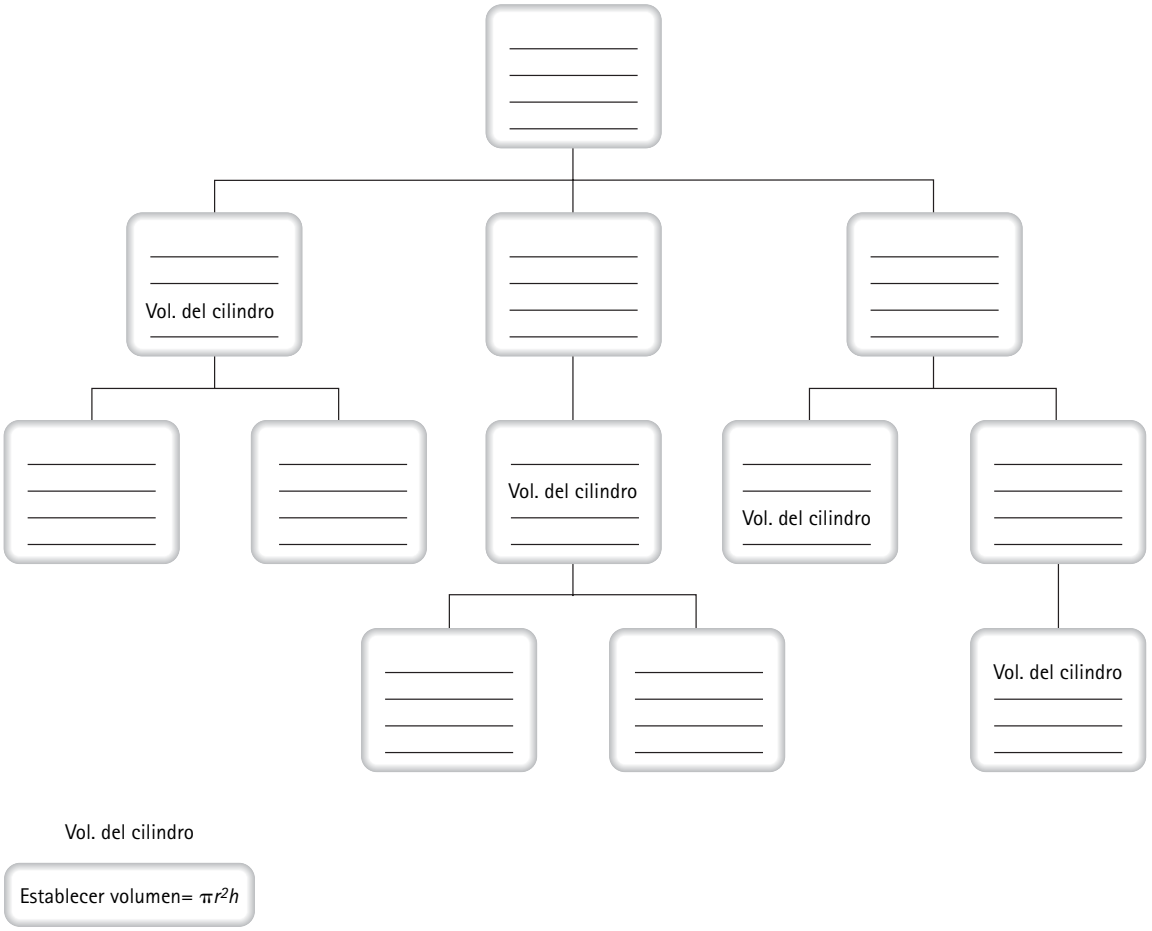


Figura 4-6 *Gráfica de estructura de módulo semijerárquica con un módulo compartido*

Otra ventaja de ejecutar módulos como funciones es la posibilidad de tomarlos y usarlos en otros programas. Con el tiempo, elaborará una biblioteca de sus propias funciones para complementar las que suministra la biblioteca estándar de C++.

Posponemos una descripción detallada de las implementaciones jerárquicas hasta el capítulo 7. Por ahora, los programas son lo necesariamente cortos para que sean suficientes las implementaciones planas. En los capítulos 5 y 6 se examinan temas como el flujo de control, precondiciones y poscondiciones, diseño de interfaz, efectos secundarios y otros que serán requeridos para desarrollar implementaciones jerárquicas.

De ahora en adelante se emplea la descripción siguiente para las descomposiciones funcionales en los casos prácticos, y se recomienda que adopte una descripción similar al resolver sus propios problemas de programación:

Enunciado del problema
Descripción de entrada
Descripción de salida
Análisis
Suposiciones (si existen)
Módulo principal
Módulos restantes por niveles
Gráfica de estructura de módulo

En algunos de nuestros casos prácticos se reorganiza la descripción con las descripciones de entrada y salida después del análisis. En capítulos posteriores se desarrolla también la descripción con secciones adicionales. No piense en esta descripción como una prescripción rígida; es más parecida a una lista de cosas por hacer. Se desea estar seguro de realizar todo lo que está en la lista, pero las circunstancias individuales de cada problema guían el orden en que se realizan.

Una perspectiva sobre el diseño

Se han considerado dos metodologías de diseño: diseño orientado a objetos y descomposición funcional. Hasta que se aprenda lo relacionado con las características adicionales del lenguaje C++ que apoyan el DOO, en los siguientes capítulos se usa la descomposición funcional (y la programación basada en objetos) para proponer soluciones de problemas.

Una perspectiva importante que debe tenerse en mente es que la descomposición funcional y el DOO no son técnicas disjuntas, separadas. El DOO descompone un problema en objetos. Los objetos no sólo contienen datos sino que también tienen operaciones relacionadas. Las operaciones en los objetos requieren algoritmos. En ocasiones los algoritmos son complicados y deben ser descompuestos en subalgoritmos por medio de la descomposición funcional. Los programadores experimentados están familiarizados con ambas metodologías y saben cuándo usar una u otra, o una combinación de ambas.

Recuerde que la fase de resolución del problema del proceso de programación toma tiempo. Si pasa la mayor parte de su tiempo analizando y diseñando una solución, entonces la codificación y ejecución del programa toma relativamente poco tiempo.

Consejo práctico de ingeniería de software

Documentación

Cuando crea su descomposición funcional o diseño orientado a objetos, está desarrollando la documentación para su programa. La *documentación* incluye las especificaciones del problema escritas, el diseño, la historia del desarrollo y el código real de un programa.

La buena documentación ayuda a otros programadores a leer y entender un programa y es invaluable cuando el software está siendo depurado y modificado (en mantenimiento). Si no ha examinado su programa durante seis meses y necesita cambiarlo, estará feliz de haberlo documentado bien. Por supuesto, si alguna otra persona tiene que usar y modificar su programa, la documentación es indispensable.

La documentación es externa e interna para el programa. La documentación externa incluye las especificaciones, la historia del desarrollo y los documentos de diseño. La documentación interna incluye el formato de programa y el **código de autodocumentación**, identificadores y comentarios importantes. Usted puede usar el seudocódigo del proceso de diseño como comentarios en sus programas.

Código de autodocumentación Código de programa que contiene identificadores significativos, así como comentarios explicativos empleados de manera juiciosa.

Esta clase de autodocumentación podría ser suficiente para alguien que lea o dé mantenimiento a sus programas. Sin embargo, si un programa va a ser usado por personas que no son programadores, debe proporcionar también un manual de usuario.

Asegúrese de mantener actualizada la documentación. Indique, en un programa, cualquier cambio que realice en toda la documentación. Use código de autodocumentación para hacer sus programas más legibles.

Ahora se examinará un caso práctico que demuestra la descomposición funcional.

Caso práctico de resolución de problemas

Presentación de un nombre en formatos múltiples

PROBLEMA Usted está comenzando a trabajar en un problema que necesita producir nombres en varios formatos junto con el número de seguro social correspondiente. Como inicio, decide escribir un programa corto en C++ que introduce el número de seguro social y un solo nombre, y los muestra en diferentes formatos de modo que pueda estar seguro de que todas sus expresiones de cadena son correctas.

ENTRADA El número de seguro social y el nombre en tres partes, en el archivo `name.dat`, cada una separada por uno o más espacios en blanco.

SALIDA El nombre se escribirá en cuatro formatos distintos en el archivo `name.out`:

1. Nombre, segundo nombre, apellido, número de seguro social
2. Apellido, nombre, segundo nombre, número de seguro social
3. Apellido, nombre, inicial del segundo nombre, número de seguro social
4. Nombre, inicial del segundo nombre, apellido

ANÁLISIS Fácilmente podría escribir el número de seguro social y el nombre en los cuatro formatos como literales de cadena en el código, pero el propósito de este ejercicio es desarrollar y probar las expresiones en cadena que necesita para un problema más grande. El enunciado del problema no dice en qué orden se introducen las partes del nombre en el archivo, pero dice que van separadas por espacios en blanco. Se supone que el orden es nombre, segundo nombre o inicial y apellido. Debido a que los datos están en un archivo, no necesita solicitar los valores. Una vez que ya tiene el número de seguro social y el nombre, sólo los escribe en los distintos formatos.

SUPOSICIÓN El orden del nombre en el archivo es nombre, segundo nombre y apellido.

Módulo principal **Nivel 0**

```
Abrir archivos
Obtener el número de seguro social
Obtener el nombre
Escribir los datos en formatos apropiados
Cerrar archivos
```

Abrir archivos **Nivel 1**

```
inData.Open("name.dat")
outData.open("name.out")
```

El paso "Obtener el número de seguro social" se puede ejecutar de modo directo mediante lectura hacia la variable de cadena. Así, no se requiere expansión en el nivel 1 del diseño.

Obtener el nombre

```
Obtener el nombre
Obtener el segundo nombre o la inicial
Obtener el apellido
```

¿Hay algún problema al leer el segundo nombre si hay en su lugar una inicial en vez del segundo nombre? En realidad no. Sólo se supone que se introduce el segundo nombre y se extrae la inicial de éste cuando la necesita para los formatos tercero y cuarto. Si se introdujo la inicial para el segundo nombre, entonces las formas de salida tercera y cuarta serán las mismas. ¿Qué hay acerca de la puntuación en la salida? Si el apellido viene primero, debe ir seguido de una coma, y la inicial del segundo nombre debe ir seguida de un punto. Así, si se introduce la inicial en vez del segundo nombre, debe ir seguida de un punto. Esto se debe agregar a las suposiciones.

SUPOSICIÓN El orden del nombre es: nombre, segundo nombre y apellido, y si se introduce la inicial del segundo nombre en lugar del segundo nombre, debe ir seguida de un punto.

Escribir los datos en formatos apropiados

Escribir el nombre, espacio en blanco, segundo nombre, espacio en blanco, apellido, espacio en blanco, número de seguro social
Escribir el apellido, coma, espacio en blanco, nombre, espacio en blanco, apellido, espacio en blanco, número de seguro social
Escribir el apellido, coma, espacio en blanco, nombre, espacio en blanco, inicial del segundo nombre, punto, espacio en blanco, número de seguro social
Escribir el nombre, espacio en blanco, inicial del segundo nombre, punto, espacio en blanco, apellido

Lo único que falta definir es la inicial del segundo nombre. Se puede usar el método `substr` para tener acceso al primer carácter en el segundo nombre.

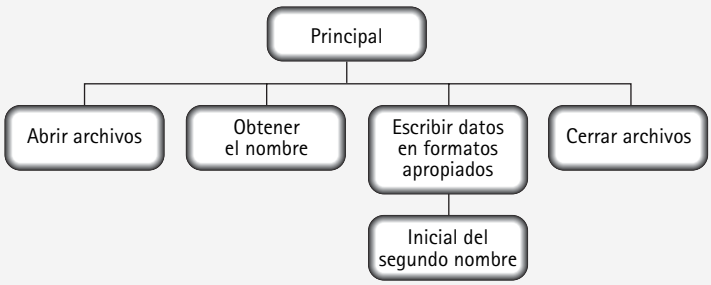
Inicial del segundo nombre **Nivel 2**

Fijar la inicial en middleName.substr(0, 1) + punto

Cerrar archivos

```
inData.close()
outData.close()
```

Gráfica de estructura de módulo



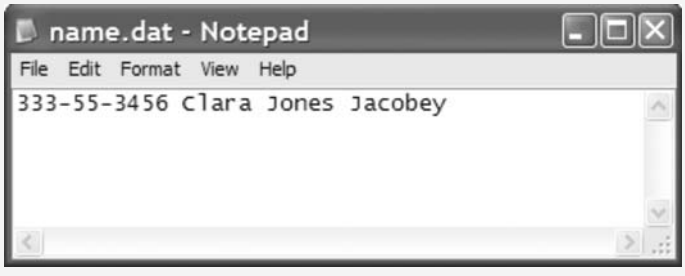
Variables

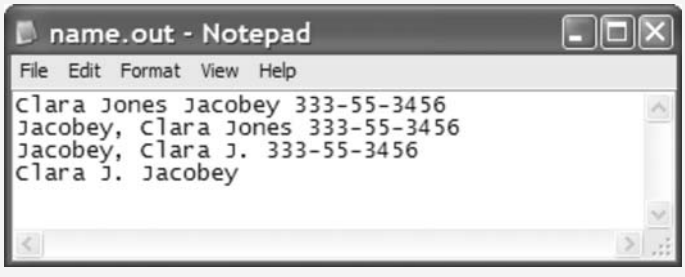
Nombre	Tipo de datos	Descripción
inData	ifstream	Archivo de entrada
outData	ofstream	Archivo de salida
socialNum	string	Número de seguro social
firstName	string	Nombre
lastName	string	Apellido
middleName	string	Segundo nombre
initial	string	Inicial del segundo nombre

```
//*****************************************************************
// Programa Formato de nombres
// Este programa lee un número de seguro social, un nombre, un
// segundo nombre o inicial y el apellido del archivo inData.
// El nombre se escribe en el archivo outData en tres formatos:
//    1. Nombre, segundo nombre, apellido y número de seguro
//    social.
//    2. Apellido, nombre, segundo nombre y número de seguro
//    social
//    3. Apellido, nombre, segundo nombre y número de seguro
//    social
//    4. Apellido, nombre, inicial del segundo nombre y número de
//    seguro social
//*****************************************************************

#include <fstream>                // Access ofstream
#include <string>                 // Access string

using namespace std;

int main()
{
    // Declarar y abrir archivos
    ifstream inData;
    ofstream outData;
    inData.open("name.dat");
    outData.open("name.out");

    // Declarar variables
    string socialNum;             // Número de seguro social
    string firstName;             // Nombre
    string lastName;              // Apellido
    string middleName;            // Segundo nombre
    string initial;               // Inicial del segundo nombre

    // Leer los datos del archivo inData
    inData >> socialNum >> firstName >> middleName >> lastName;

    // Tener acceso a la inicial del segundo nombre y anexar un punto
    initial = middleName.substr(0, 1) + '.';
```

```
    // Producir información en los formatos requeridos
    outData << firstName << ' ' << middleName << ' ' << lastName
            << ' ' << socialNum << endl;
    outData << lastName << ", " << firstName << ' ' << middleName
            << ' ' << socialNum  << endl;
    outData << lastName << ", " << firstName << ' ' << initial
            << ' ' << socialNum  << endl;
    outData << firstName << ' ' << initial << ' ' << lastName;

    // Cerrar archivos
    inData.close();
    outData.close();
    return 0;
}
```

A continuación se muestran ejemplos de archivos de entrada y salida





Las imágenes mostradas corresponden a la salida producida por el programa original, escrito en idioma inglés.

Información básica

Programación a muchas escalas

Para ayudar a poner en contexto los temas de este libro, se describe en términos amplios la manera en que se hace la programación en sus muchas formas en "el mundo real". Obviamente no es posible cubrir cada posibilidad, pero se intentará dar una idea de lo último a este respecto.

La programación de proyectos varía en tamaño desde la escala pequeña, en la cual un estudiante o aficionado a la computación escribe un programa corto para probar algo nuevo, hasta los proyectos de programación multicompañías a gran escala donde participan cientos de personas. Entre estos dos extremos están los esfuerzos de muchos otros tamaños. Hay personas que usan la programación en sus profesiones, aun cuando no es su empleo principal. Por ejemplo, un científico podría escribir un programa con un objetivo particular a fin de analizar datos de un determinado experimento.

(continúa)

Programación a muchas escalas

Incluso entre los programadores profesionales hay muchas áreas de programación especializadas. Un individuo podría tener una especialidad en el procesamiento de datos de negocios, en escribir compiladores o desarrollar procesadores de palabras (una especialidad conocida como "tool making"), en apoyo a la investigación y el desarrollo, en desarrollo de presentación gráfica, en la escritura de software de entretenimiento o en alguna otra, entre muchas áreas. Sin embargo, una persona puede producir sólo programas muy pequeños (unas cuantas decenas de miles de líneas de código en el mejor de los casos). Esta clase de trabajo se llama programación en pequeño.

Una aplicación más amplia, como el desarrollo de un nuevo sistema operativo, podría requerir cientos de miles o incluso millones de líneas de código. Esta clase de proyectos a gran escala requiere equipos de programadores, muchos de ellos especialistas, que deben organizarse de alguna manera o desperdiciarán tiempo valioso intentando comunicarse entre sí.

En general, se establece una organización jerárquica a lo largo de las líneas de la gráfica de estructura de módulo. Una persona, el *arquitecto principal* o *director de proyecto*, determina la estructura básica del programa y luego delega la responsabilidad de poner en práctica los componentes principales. Dichos componentes pueden ser módulos producidos por una descomposición funcional o podrían ser clases y objetos que resultan de un diseño orientado a objetos. En proyectos más pequeños, los componentes pueden ser delegados directamente a programadores. En cada etapa, la persona a cargo debe tener el conocimiento y la experiencia necesarios para definir el nivel que sigue en la jerarquía y estimar los recursos necesarios para ponerlo en práctica. Este tipo de organización se llama *programación en grande*.

Los lenguajes de programación y las herramientas de software pueden ayudar mucho para apoyar la programación en grande. Por ejemplo, si un lenguaje de programación permite a los programadores desarrollar, compilar y probar partes de un programa de manera independiente antes de juntarlas, entonces esto permite que varias personas trabajen en el programa al mismo tiempo. Por supuesto, es difícil apreciar la complejidad de la programación en grande cuando se escribe un programa pequeño para una tarea de clase. Sin embargo, la experiencia que obtiene en este curso será valiosa cuando comience a desarrollar programas más grandes.

Lo siguiente es un ejemplo clásico de lo que sucede cuando se desarrolla un gran programa sin la organización cuidadosa y el apoyo de lenguaje apropiado. En la década de 1960, IBM desarrolló un nuevo sistema operativo importante denominado OS/360, que fue uno de los primeros ejemplos verdaderos de programación en grande. Después de haber escrito el sistema operativo, se encontraron más de 1 000 errores significativos. A pesar de que pasaron años en intentar arreglar estos errores, los programadores nunca los redujeron por debajo de 1 000, y en ocasiones las "composturas" producían más errores que los que eliminaban.

¿Qué condujo a esta situación? El análisis *a posteriori* mostró que el código estaba mal organizado y que las distintas piezas estaban tan interrelacionadas que nadie pudo arreglarlo. Un cambio aparentemente simple en una parte del código ocasionó que fallaran otras partes del sistema. Por último, a un gran costo, se creó un sistema completamente nuevo con mejor organización y herramientas.

En los primeros tiempos de la computación, todos esperaban que ocurrieran errores ocasionales, y aún fue posible realizar trabajo útil con un sistema operativo defectuoso. Sin embargo, en la actualidad las computadoras se emplean cada vez más en aplicaciones críticas —como equipo médico y sistemas de control de aviones—, donde los errores pueden ser fatales. Muchas de estas aplicaciones dependen de la programación en gran escala. Si en este momento estuviera abordando un moderno avión comercial, bien podría hacer una pausa y preguntarse: ¿qué clase de lenguaje y herramientas emplearon cuando escribieron los programas para esta cosa? Por fortuna, la mayoría de los grandes esfuerzos en el desarrollo de software hoy día usan una combinación de buena metodología, lenguaje apropiado y herramientas organizacionales extensas, un método conocido como *ingeniería de software*.

Prueba y depuración

Una parte importante de la implementación de un programa es probarlo (corroborar los resultados). Por ahora debe comprender que no hay nada mágico respecto a la computadora. Es infalible sólo si la persona que escribe las instrucciones e introduce los datos es infalible. No confíe en que ha obtenido respuestas correctas hasta que haya comprobado suficientes a la mano para convencerse de que el programa está funcionando.

De aquí en adelante, estas secciones de Prueba y depuración ofrecen consejos prácticos acerca de cómo probar sus programas y qué hacer si un programa no funciona como se tenía previsto. Pero no espere hasta que haya encontrado un error para leer las secciones de Prueba y depuración. Es mucho más fácil evitar errores que arreglarlos.

Al probar programas que introducen valores de datos desde un archivo, es posible que fallen las operaciones de ingreso. Y cuando falla el ingreso en C++, la computadora no emite un mensaje de advertencia ni termina el programa. El programa simplemente continúa su ejecución e ignora las operaciones de entrada adicionales en ese archivo. Las dos causas más comunes para que falle el ingreso son datos no válidos y el error de fin de archivo (*end-of-file error*).

Un error de fin de archivo ocurre cuando el programa ha leído todos los datos de entrada disponibles en el archivo y necesita más datos para llenar las variables en sus sentencias de entrada. Es posible que el archivo de datos simplemente no haya sido preparado de manera adecuada. Quizá contiene menos datos que los requeridos por el programa. O tal vez el formato de los datos de entrada sea erróneo. Dejar espacios en blanco entre valores numéricos es asegurarse de que habrá problemas. Por ejemplo, es posible que se desee que un archivo de datos contenga tres valores enteros: 25, 16 y 42. Observe lo que sucede con estos datos:

```
2516 42
```

y este código:

```
inFile >> i >> j >> k;
```

Las dos primeras operaciones de entrada consumen los datos del archivo, dejando a la tercera sin datos para leer. El flujo `inFile` entra en estado de falla, así que a `k` no se le asigna un nuevo valor y la computadora continúa con la ejecución de la siguiente sentencia del programa.

Si el archivo de datos se prepara de manera correcta y aún hay un error de fin de archivo, el problema está en la lógica del programa. Por alguna razón el programa intenta muchas operaciones de entrada. Podría ser un simple descuido como especificar demasiadas variables en una sentencia de entrada particular. Podría ser un mal uso de la función ignore, que ocasiona que los valores sean omitidos de manera inadvertida; o podría ser un defecto grave en el algoritmo. Se deben verificar todas estas posibilidades.

La otra fuente principal de falla de entrada, datos no válidos, tiene varias causas posibles. La más común es un error en la preparación o introducción de los datos. Los datos numéricos y de caracteres combinados de modo inapropiado en la entrada pueden ocasionar que falle el flujo de entrada si se supone que lea un valor numérico, pero el marcador de lectura se coloca en el carácter que no está permitido en el número. Otra causa es usar el nombre de variable equivocado (que puede ser del tipo de datos erróneo) en una sentencia de entrada. Declarar una variable del tipo de datos erróneo es una variación del problema. Por último, dejar una variable (o incluir una extra) en una sentencia de entrada puede causar que el marcador de lectura termine posicionado en el tipo de datos equivocado.

Otro descuido, uno que no causa falla de entrada pero ocasiona frustración del programador, es usar `cin` o `cout` en una sentencia I/O cuando su intención es especificar un flujo de archivo. Si usa erróneamente `cin` en lugar de un flujo de archivo de entrada, el programa se detiene y espera la entrada desde el teclado. Si por error emplea `cout` en lugar de un flujo de archivo de salida, se obtiene un resultado inesperado en la pantalla.

Al darle un marco de trabajo que lo ayude a organizar y seguir la pista de los detalles relacionados con el diseño e implementación de un programa, la descomposición funcional (y, después, el diseño orientado a objetos), debe, sobre todo, ayudar a evitar muchos de estos errores.

En capítulos posteriores verá que puede probar módulos por separado. Si se asegura de que cada módulo funciona por sí mismo, su programa debe funcionar cuando reúne todos los módulos. Probar módulos por separado es menos trabajo que intentar probar un programa completo. En una sección más pequeña de código es menos probable que se combinen errores múltiples que produzcan un comportamiento difícil de analizar.

Sugerencias de prueba y depuración

1. Las sentencias de entrada y salida comienzan siempre con el nombre de un objeto de flujo, y los operadores `>>` y `<<` apuntan en la dirección en que van los datos. La sentencia

   ```
   cout << n;
   ```

   envía datos *al* flujo de salida `cout`, y la sentencia

   ```
   cin >> n;
   ```

   envía datos *a* la variable `n`.
2. Cuando un programa introduce desde un archivo o produce hacia él, asegúrese de que cada sentencia I/O desde o hacia el archivo emplee el nombre del flujo de archivo, no `cin` o `cout`.
3. La función `open` relacionada con un objeto `ifstream` u `ofstream` requiere una cadena C como argumento. El argumento no puede ser un objeto `string`. En este punto del libro, el argumento sólo puede ser *a*) una cadena literal o *b*) la cadena C devuelta por la llamada de función `myString.c_str()`, donde `myString` es de tipo `string`.
4. Cuando abra un archivo de datos para ingreso, asegúrese de que el argumento para la función `open` proporcione el nombre correcto del archivo como existe en el disco.
5. Al leer una cadena de caracteres en un objeto `string`, el operador `>>` se detiene en, *pero no consume*, el primer carácter de espacio en blanco posterior.
6. Asegúrese de que cada sentencia de entrada especifique el número correcto de variables, y que cada una de dichas variables sea del tipo de datos correcto.
7. Si sus datos de entrada se combinan (caracteres y valores numéricos), asegúrese de ocuparse de los espacios en blanco intermedios.
8. Imprima por eco los datos de entrada para comprobar que cada valor está donde pertenece y en el formato apropiado. (Esto es muy importante, porque una falla de entrada en C++ no produce un mensaje de error ni termina el programa.)

Resumen

Los programas operan en datos. Si los datos y programas se mantienen separados, los datos están disponibles para uso con otros programas, y el mismo programa se puede ejecutar con diferentes conjuntos de datos de entrada.

El operador de extracción `(>>)` introduce datos desde el teclado o un archivo, y los almacena en la variable especificada como su operando derecho. El operador de extracción omite cualquier carácter de espacio en blanco para hallar el siguiente valor de datos en el flujo de entrada. La función `get` no omite caracteres de espacio en blanco; introduce el siguiente carácter y lo almacena en la variable `char` especificada en su lista de argumentos. Tanto el operador `>>` como la función `get` dejan el marcador de lectura posicionado en el siguiente carácter por leer. La siguiente operación de entrada comienza a leer en el punto indicado por el marcador.

El carácter de nueva línea (denotado por `\n` en un programa de C++) marca el fin de una línea de datos. Cada vez que usted oprime la tecla Return o Enter crea un carácter de nueva línea. Su programa genera una nueva línea cada vez que emplea el manipulador `endl` o produce de modo explícito el carácter `\n`. La línea nueva es un carácter de control; no se imprime. Dicho carácter controla el movimiento del cursor en la pantalla o la posición de una línea en una impresora.

Los programas interactivos alertan al usuario en cada ingreso de datos y le informan de resultados y errores. Diseñar diálogo interactivo es un ejercicio en el arte de la comunicación.

La entrada y salida no interactivas permiten que los datos se preparen antes de la ejecución de un programa y que se ejecute de nuevo con los mismos datos en caso de que surja un problema durante el proceso.

Con frecuencia los archivos de datos se emplean para proceso no interactivo y permiten que el resultado de un programa se utilice como entrada para otro programa. Para usar estos archivos son necesarias cuatro cosas: (1) incluir el archivo de encabezado `fstream`, (2) declarar los flujos de archivos junto con sus otras declaraciones de variables, (3) preparar los archivos para lectura o escritura llamando a la función `open`, y (4) especificar el nombre del flujo de archivo en cada sentencia de entrada o salida que lo emplee.

El diseño orientado a objetos y la descomposición funcional son metodologías para tratar problemas de programación no triviales. El diseño orientado a objetos produce una solución de un problema al enfocarse en objetos y sus operaciones relacionadas. El primer paso es identificar los objetos principales del problema y elegir las operaciones apropiadas en esos objetos. Un objeto es un ejemplo de un tipo de datos llamado clase. Durante el diseño orientado a objetos, las clases pueden ser diseñadas a partir de cero, pueden ser obtenidas de bibliotecas y emplearlas tal como están o ser adaptadas de clases existentes por medio de la técnica de herencia. El resultado del proceso de diseño es un programa que consta de objetos autocontenidos que manejan sus propios datos y se comunican al invocar operaciones entre sí.

La descomposición funcional comienza con una solución abstracta que luego se divide en pasos principales. Cada paso se convierte en un subproblema que se analiza y subdivide aún más. Un paso concreto es el que se puede traducir directamente en C++; los pasos que necesitan más refinación son pasos abstractos. Un módulo es una colección de pasos concretos y abstractos que resuelven un problema. Los programas pueden ser construidos de módulos por medio de una implementación plana, jerárquica o semijerárquica.

La atención cuidadosa en el diseño del programa, formato y documentación produce programas altamente estructurados y legibles.

Comprobación rápida

1. ¿Por qué es necesario usar mensajes para I/O interactivas? (pp. 124-126)
2. ¿Qué condiciones buscaría en un problema para decidir si es apropiada la entrada interactiva o no interactiva? (pp. 124-126)
3. ¿Cuál es el primer paso en el proceso de diseño orientado a objetos? (pp. 135 y 136)
4. Escriba una sentencia de entrada que lea tres valores enteros en variables `a`, `b` y `c`. (pp. 117-119)
5. Si una línea de entrada contiene

   ```
   Jones, Walker Thomas
   ```

   ¿cuáles serán los valores en las variables de cadena (`string`), `first`, `middle` y `last` cuando se ejecuta la siguiente sentencia? (pp. 123 y 124)

   ```
   cin >> first >> middle >> last;
   ```

6. Después de incluir el archivo de encabezado `fstream` y declarar un flujo de archivos, ¿cuál es el siguiente paso al usar una salida o entrada de archivo? (pp. 126-129)
7. ¿Qué caracteriza a un paso concreto en un diseño de descomposición funcional? (pp. 136-137)
8. Si se le proporciona a usted un diseño de descomposición funcional, ¿cómo lo llevaría a la práctica? (pp. 139 y 141)

Respuestas

1. Para indicar al usuario cómo y cuándo introducir valores como entrada. **2.** La cantidad de datos a ingresar y si los datos pueden ser preparados para introducción antes de ejecutar el programa. **3.** Identificar los objetos principales en el problema. **4.** `cin >> a >> b >> c;`

5.
```
first = "Jones,"
middle = "Walker"
last = "Thomas"
```
6. Preparar el archivo para lectura o escritura con la función `open`. **7.** Es un paso que puede ser puesto en práctica de modo directo en un lenguaje de programación. **8.** Identificando todos los pasos concretos, empezando desde la parte superior del árbol y disponiéndolos en el orden apropiado. La secuencia de pasos se convierte después paso a paso en código.

Ejercicios de preparación para examen

1. La sentencia

   ```
   cin >> maximum >> minimum;
   ```

   es equivalente a las dos sentencias:

   ```
   cin >> minimum;
   cin >> maximum;
   ```

   ¿Verdadero o falso?
2. ¿Qué es incorrecto en cada una de las siguientes sentencias?
   a) `cin << score;`
   b) `cout >> maximum;`
   c) `cin >> "Enter data";`
   d) `cin.ignore('Y', 35);`
   e) `getLine(someString, cin);`
3. Si los datos de entrada se introducen como sigue

   ```
   10  20
   30  40
   50  60
   70  80
   ```

   y las sentencias de entrada que lee son

   ```
   cin >> a >> b >> c;
   cin >> d >> e >> f;
   cin >> a >> b;
   ```

   ¿cuál es el contenido de las variables `a`, `b`, `c`, `d`, `e` y `f` después que se han ejecutado las sentencias?
4. Si los datos de entrada se introducen como sigue:

   ```
   10   20   30   40   50   60
   10   20   30   40   50   60
   10   20   30   40   50   60
   70   80
   ```

   y las sentencias de entrada que lee son

   ```
   cin >> a >> b >> c;
   cin.ignore(100, '\n');
   cin.get(ch1);
   cin >> d >> e >> f;
   cin.ignore(100, '\n');
   cin.ignore(100, '\n');
   cin >> a >> b;
   ```

   ¿Cuál es el contenido de las variables `a`, `b`, `c`, `d`, `e` y `f` después de que se han ejecutado las sentencias?

5. Dados los datos de entrada

```
January 25, 2005
```

y la sentencia de entrada

```
cin >> string1 >> string2;
```

   a) ¿Cuál es el contenido de cada una de las variables de cadena después de que se ejecuta la sentencia?
   b) ¿Dónde está el marcador de lectura después de que se ejecuta la sentencia?

6. Dados los datos de entrada

```
January 25, 2005
```

y la sentencia de entrada

```
getline(cin, string1);
```

   a) ¿Cuál es el contenido de la variable de cadena después de que se ejecuta la sentencia?
   b) ¿Dónde está el marcador de lectura después de que se ejecuta la sentencia?

7. Dados los datos de entrada

```
January 25, 2005
```

y la sentencia de entrada (donde el tipo de cada variable está dado por su nombre)

```
cin >> string1 >> int1 >> char1 >> string2;
```

   a) ¿Cuál es el contenido de cada una de las variables después de que se ejecuta la sentencia?
   b) ¿Dónde está el marcador de lectura después de que se ejecuta la sentencia?

8. Si el marcador de lectura está sobre el carácter de nueva línea al final de una línea, y usted llama la función `get`, ¿qué valor devuelve en su argumento?
9. ¿Qué especifican los dos argumentos para la función `ignore`?
10. Usted está escribiendo un programa que introduce una fecha en la forma mes/día/año.
   a) Escriba la sentencia de salida para indicar a un usuario poco experimentado que introduzca una fecha.
   b) Escriba la sentencia de salida para indicar a un usuario experimentado que introduzca una fecha.
11. ¿Cuáles son los cuatro pasos necesarios para usar un archivo para entrada o salida?
12. ¿Cuáles son los dos tipos de datos de flujo de archivos analizados en este capítulo?
13. ¿Cuál es el propósito de las dos sentencias siguientes?

```
ifstream inFile;
inFile.open("datafile.dat");
```

14. Corrija el siguiente segmento de código de modo que abra el archivo cuyo nombre se introduce vía la sentencia de entrada.

```
ifstream inData;
string name;

cout << "Enter the name of the file: ");
cin >> name;
infile.open(name);
```

15. ¿Qué sucede cuando se efectúa una operación de entrada en un archivo que está en el estado de falla?
16. Cuando usted intenta abrir un archivo que no existe, C++ produce un mensaje de error y termina la ejecución del programa. ¿Verdadero o falso?

17. El simple hecho de abrir un flujo de archivos de entrada no puede causar que entre al estado de falla. ¿Verdadero o falso?
18. Usted está escribiendo un programa para una compañía de renta de automóviles que mantiene un registro de los vehículos. ¿Cuáles son los objetos principales del problema?
19. ¿Qué construcción de C++ se emplea para implementar un objeto?
20. Defina los siguientes términos:
    a) Paso concreto
    b) Paso abstracto
    c) Módulo
    d) Equivalencia funcional
    e) Cohesión funcional
21. ¿Qué construcción de C++ se usa para implementar módulos?
22. ¿Cuáles de las siguientes funciones son funciones miembro de la clase `istream`?
    a) `>>`
    b) `ignore`
    c) `get`
    d) `getline`
    e) `cin`
23. La función `open` es un miembro de las clases `ifstream` y `ofstream`. ¿Verdadero o falso?

Ejercicios de preparación para la programación

1. Escriba una sentencia de entrada C++ que lea tres valores enteros hacia las variables `int1`, `int2` e `int3`, en ese orden.
2. Escriba una sentencia de salida indicadora y luego una sentencia de entrada que lea un nombre en tres variables de cadena: `first`, `middle` y `last`. El nombre se introducirá con el formato: nombre, segundo nombre, apellido.
3. Escriba una sentencia de salida indicadora y luego una sentencia de entrada que lea un nombre hacia una sola variable de cadena, `name`. El nombre se introducirá en una línea con el formato: nombre, segundo nombre, apellido.
4. Escriba las sentencias necesarias para solicitar e introducir tres valores de punto flotante y luego producir su promedio. Para este ejercicio, suponga que el usuario no tiene experiencia previa con computadoras y, por tanto, necesita instrucciones muy detalladas. Para ayudar a evitar errores, el usuario debe introducir cada valor por separado.
5. Escriba las sentencias necesarias para solicitar e introducir tres valores de punto flotante y después producir su promedio. Para este ejercicio, suponga que el usuario tiene suficiente experiencia con las computadoras y, por lo mismo, necesita mínimas instrucciones. El usuario debe introducir los tres valores en una línea.
6. Escriba las declaraciones y sentencias de entrada necesarias para leer cada uno de los siguientes conjuntos de valores de datos en variables de los tipos apropiados. Usted elige los nombres de las variables. En algunos casos hay marcas de puntuación o símbolos especiales que deben ser omitidos.
    a) `100 A 98.6`
    b) `February 23 March 19`
    c) `19, 25, 103.876`
    d) `A a B b`
    e) `$56.45`
7. Escriba una sola sentencia de entrada que lea las siguientes líneas de datos hacia las variables `streetNum`, `street1`, `street2`, `town`, `state` y `zip`.

    782 Maple Avenue
    Blithe, CO 56103

8. Escriba las sentencias necesarias para preparar un archivo denominado `"temperatures.dat"` para lectura como un `ifstream` nombrado `temps`.

9. El archivo `"temperatures.dat"` contiene una lista de seis temperaturas, dispuestas una por línea. Suponiendo que el archivo ya ha sido preparado para lectura, como se describió en el ejercicio 8, escriba las sentencias para leer en los datos desde el archivo e imprima la temperatura promedio. También necesitará declarar las variables `float` necesarias para realizar la tarea.
10. Llene los espacios en blanco en el siguiente programa:

```
#include <iostream>
#include ___________

using namespace std;

_______ inData;
_______ outData;
const float PI = 3.14159265;
float radius;
float circumference;
float area;

int main()
{
    _____________("indata.dat");
    _____________("outdata.dat");
    _______ >> radius;
    circumference = radius * 2 * PI;
    area = radius * radius * PI;
    cout << "Para el primer círculo, la circunferencia es "
         << circumference << " y el área es " << area << endl;
    _______ << radius << " " << circumference << " " << area
            << endl;
    _______ >> radius;
    circumference = radius * 2 * PI;
    area = radius * radius * PI;
    cout << "Para el segundo círculo, la circunferencia es "
         << circumference << " y el área es " << area << endl;
    _______ << radius << " " << circumference << " " << area
            << endl;
    return 0;
}
```

11. Modifique el programa del ejercicio 10 de modo que permita al usuario introducir el nombre del archivo de salida en lugar de pedir que el programa use `"outdata.dat"`.
12. El flujo de archivos `inFile` contiene dos valores enteros. Escriba una sentencia de entrada que ocasione que entre en el estado de falla.
13. Escriba un segmento de código que solicite al usuario un nombre de archivo, lea el nombre de archivo hacia una `string` denominada `filename` y abra el flujo de archivos `userFile` usando el nombre suministrado.
14. Use la descomposición funcional para escribir un algoritmo para escritura y envío por correo de una carta de negocios.
15. ¿Cuáles son los objetos del problema descrito en el ejercicio 14?

Problemas de programación

1. Escriba un programa interactivo en C++ que introduzca un nombre desde el usuario en el formato de:

```
last, first middle
```

El programa debe producir después el nombre en el formato de:

```
first middle last
```

El programa tendrá que usar operaciones de cadena para eliminar la coma del final del apellido. Asegúrese de usar el formato y comentarios apropiados en su código. La entrada debe tener una llamada de atención apropiada y el resultado debe ser marcado con claridad y tener un formato nítido.

2. Escriba un programa interactivo en C++ que introduzca una serie de 12 temperaturas desde el usuario. Éste debe escribir en el archivo `"tempdata.dat"` cada temperatura y la diferencia entre la temperatura actual y la precedente. La diferencia no se produce para la primera temperatura que se introduce. Al final del programa, la temperatura promedio se debe mostrar para el usuario vía `cout`. Por ejemplo, dados los siguientes datos de entrada:

```
34.5 38.6 42.4 46.8 51.3 63.1 60.2 55.9 60.3 56.7 50.3 42.2
```

El archivo `tempdata.dat` contendría:

```
34.5
38.6  4.1
42.4  3.8
46.8  4.4
51.3  4.5
63.1  11.8
60.2  -2.9
55.9  -4.3
60.3  4.4
56.7  -3.6
50.3  -6.4
42.2  -8.1
```

Asegúrese de usar el formato y los comentarios apropiados en su código. La entrada debe tener mensajes apropiados y el resultado se debe marcar de manera clara y tener un formato nítido.

3. Escriba un programa interactivo en C++ que calcule y produzca la media y la desviación estándar como un conjunto de valores enteros introducidos por el usuario. (Si solucionó el problema de programación 2 en el capítulo 3, entonces puede volver a usar mucho de ese código aquí.) La media es la suma de los cuatro valores divididos entre 4, y la fórmula de la desviación estándar es

$$s = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n-1}}$$

donde $n = 4$, x_i se refiere a cada uno de los cuatro valores y $\bar{x}$ es la media. Observe que aunque los valores individuales son enteros, los resultados son valores de punto flotante. Asegúrese de usar el formato y los comentarios apropiados en su código. Proporcione los mensajes apropiados al usuario. El resultado se debe marcar de manera clara y tener un formato nítido.

4. Escriba un programa en C++ que lea datos desde un archivo cuyo nombre sea introducido por el usuario, y que produzca la primera palabra después de cada una de las tres primeras comas en el archivo. Por ejemplo, si el archivo contiene el texto de este problema, entonces el programa daría como resultado:

```
y
si
entonces
```

Suponga que aparece una coma en, por lo menos, cada 200 caracteres en el archivo. Asegúrese de usar el formato y comentarios apropiados en su código. Proporcione los mensajes apropiados para el usuario. El resultado se debe marcar con claridad y tener un formato nítido.

5. Escriba un programa C++ que permita al usuario introducir el porcentaje de la cara de la Luna que aparece iluminada y que produzca el área superficial de esa porción de la Luna. La fórmula para el área de superficie de un segmento de una esfera es

   $S = 2R^2\theta$

   donde R es el radio de la esfera (el radio de la Luna es 1738.3 km) y θ es el ángulo de la cuña en radianes. Hay 2π radianes en un círculo, así que la semiesfera de la Luna que vemos representa a lo sumo π radianes. Así, si el usuario introduce 100% (luna llena), el ángulo de la cuña es π y la fórmula se puede evaluar como sigue.

   $S = 2 \times 1738.3^2 \times 3.14159 = 18985818.672$ kilómetros cuadrados

   Si el usuario introduce 50% (cuarto creciente o cuarto menguante), entonces el ángulo de la cuña es $\pi \times 0.5$, y así sucesivamente. Asegúrese de usar el formato y comentarios apropiados en su código. Proporcione mensajes adecuados para el usuario. El resultado se debe leer con claridad y tener un formato nítido (limite la precisión decimal a tres lugares, como en el ejemplo anterior).

Seguimiento de caso práctico

1. Remplace la información de Clara Jones Jacobey con su propio nombre como entrada para el programa Format Names.
2. Cambie este programa de modo que el número de seguro social se escriba en una línea por sí mismo con los distintos formatos de nombre sangrados cinco espacios en las siguientes cuatro líneas.
3. Cambie el programa de modo que el nombre del archivo de entrada se lea desde el teclado.
4. Cambie el programa de modo que los nombres del archivo de entrada y el archivo de salida se lean desde el teclado.
5. En el capítulo 3, el caso práctico fue una calculadora de pago de hipoteca donde los datos se almacenaron como constantes. Reescriba ese programa de modo que la información se introduzca desde el teclado.

CAPÍTULO

5

Condiciones, expresiones lógicas y estructuras de control de selección

Objetivos

Objetivos de conocimiento

- *Entender cómo funcionan los operadores booleanos.*
- *Entender el flujo de control en una sentencia de bifurcación.*
- *Entender el flujo de control en una sentencia de bifurcación anidada.*
- *Conocer qué son las precondiciones o poscondiciones*

Objetivos de habilidades

Ser capaz de:

- *Construir una expresión lógica simple (booleana) para evaluar una condición específica.*
- *Construir una expresión lógica compleja para evaluar una condición dada.*
- *Construir una sentencia If-Then-Else para efectuar una tarea específica.*
- *Construir una sentencia If-Then para efectuar una tarea específica.*
- *Construir un conjunto de sentencias If anidadas para realizar una tarea específica.*
- *Seguir la pista de la ejecución de un programa en C++.*
- *Probar y depurar un programa en C++.*

Hasta el momento, las sentencias en nuestros programas se han ejecutado en su orden físico. Se ejecuta la primera sentencia, después la segunda, y así sucesivamente hasta que todas se han ejecutado. Pero, ¿qué pasa si se desea que la computadora ejecute las sentencias en algún otro orden? Suponga que se quiere comprobar la validez de los datos de entrada y luego realizar un cálculo *o* imprimir un mensaje de error, no ambas cosas. Para esto, se debe poder hacer una pregunta y luego, con base en la respuesta, elegir uno u otro curso de acción.

La sentencia If permite ejecutar sentencias en un orden que es diferente de su orden físico. Se puede hacer una pregunta con ella y hacer una cosa si la respuesta es sí (verdadero) u otra si la respuesta es no (falso). En la primera parte de este capítulo, se estudia la realización de preguntas; en la segunda parte, la sentencia misma If.

5.1 Flujo de control

El orden en el que se ejecutan las sentencias en un programa se llama **flujo de control**. En un sentido, la computadora está bajo control de una sentencia a la vez. Cuando una sentencia ha sido ejecutada, el control se pasa a la siguiente sentencia (como la estafeta que se pasa en una carrera de relevos).

Flujo de control Orden en el que la computadora ejecuta las sentencias en un programa.

Estructura de control Sentencia empleada para alterar el flujo de control normalmente secuencial.

El flujo de control normalmente es secuencial (véase la figura 5-1). Es decir, cuando se termina la ejecución de una sentencia, el control pasa a la siguiente sentencia del programa. Cuando se desea que el flujo de control no sea secuencial, se usan **estructuras de control**, sentencias especiales que transfieren el control a una sentencia distinta a la que viene a continuación físicamente. Las estructuras de control son tan importantes que se centra la atención en ellas en el resto del capítulo y en los cuatro capítulos siguientes.

Selección

Se usa una estructura de control de selección (o bifurcación) cuando se desea que la computadora elija entre acciones alternativas. Se hace una afirmación, que es verdadera o falsa. Si la afirmación es verdadera, la computadora ejecuta una sentencia. Si es falsa, ejecuta otra (véase la figura 5-2). La capacidad de la computadora para resolver problemas prácticos es producto de su capacidad para tomar decisiones y ejecutar diferentes secuencias de instrucciones.

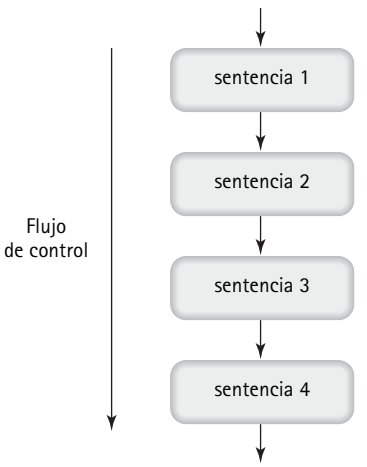
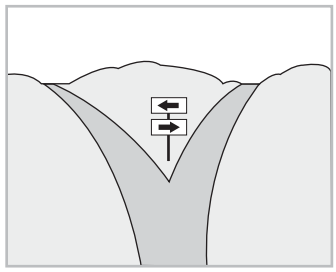


Figura 5–1 *Control secuencial*



En el programa del año bisiesto del capítulo 1 se muestra el proceso de selección en funcionamiento. La computadora debe decidir si un año es bisiesto. Lo hace al probar la afirmación de que el año no es divisible entre 4. Si la afirmación es verdadera, la computadora sigue las instrucciones para devolver falso, indicando que el año no es bisiesto. Si la afirmación es falsa, la computadora continúa con la comprobación de las excepciones a la regla general. Antes de examinar las estructuras de control de selección en C++, se examinará con detenimiento cómo se logra que la computadora tome decisiones.

5.2 Condiciones y expresiones lógicas

Una pregunta en C++, no se expresa como pregunta; se enuncia como una afirmación. Si la afirmación que se hace es verdadera, la respuesta a la pregunta es sí. Si la sentencia no es verdadera, la respuesta a la pregunta es no. Por ejemplo, si se desea preguntar: "¿Vamos a cenar espinacas esta noche?", se diría: "Cenaremos espinacas esta noche". Si la afirmación es verdadera, la respuesta a la pregunta es sí. De lo contrario, la respuesta es no.

Así, hacer preguntas en C++ consiste en hacer una afirmación que es verdadera o falsa. La computadora *evalúa* la afirmación y la comprueba contra alguna condición interna (los valores almacenados en algunas variables, por ejemplo) para ver si es verdadero o falso.

Tipo de datos `bool`

En C++, el tipo de datos `bool` es un tipo integrado que consta de sólo dos valores, las constantes `true` y `false`. La palabra reservada `bool` es una abreviatura para booleano.* Los datos booleanos se emplean para probar condiciones en un programa de modo que la computadora pueda tomar decisiones (con una estructura de control de selección).

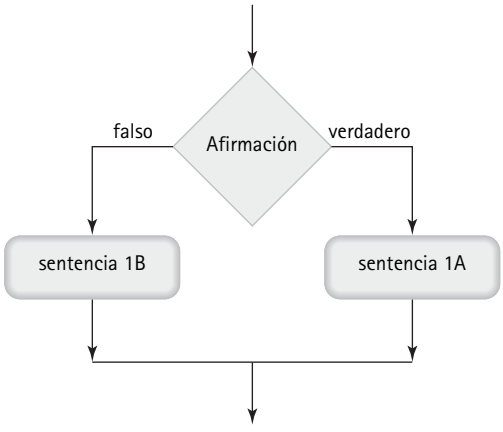


Figura 5–2 *Estructura de control de selección (bifurcación)*

* La palabra *booleano* se debe a George Boole, matemático inglés del siglo XIX quien describió un sistema de lógica usando variables con sólo dos valores: Verdadero y Falso. (Véase el cuadro Conozca a, en la página 203.)

Las variables del tipo `bool` se declaran de la misma forma que las variables de otros tipos, es decir, escribiendo el nombre del tipo de datos y luego un identificador:

```
bool dataOK;     // Verdadero si los datos de entrada son válidos
bool done;       // Verdadero si se realiza el proceso
bool taxable;    // Verdadero si el elemento tiene impuesto
```

Cada variable de tipo `bool` puede contener uno de dos valores: `true` o `false`. Es importante comprender desde el comienzo que `true` y `false` no son nombres de variables ni cadenas. Son constantes especiales en C++ y, de hecho, son palabras reservadas.

Información básica

Antes del tipo `bool`

El lenguaje C no tiene un tipo de datos `bool`, y con anterioridad al estándar del lenguaje C++ ISO/ANSI, tampoco C++. En C y C++ pre-estándar, el valor 0 representa *falso* y cualquier valor no cero representa *verdadero*. En estos lenguajes suele emplearse el tipo `int` para representar datos booleanos:

```
int dataOK;
  ⋮
dataOK = 1;   // Almacenar "true" en dataOK
  ⋮
dataOK = 0;   // Almacenar "false" en dataOK
```

Para hacer más autodocumentado el código, muchos programadores de C y C++ pre-estándar prefieren definir su propio tipo de datos booleanos por medio de una *sentencia Typedef*. Esta sentencia permite introducir un nuevo nombre para un tipo de datos existente:

```
typedef int bool;
```

Todo lo que hace esta sentencia es decir al compilador que sustituya la palabra `int` cada vez que aparezca la palabra `bool` en el resto del programa. Así, cuando el compilador encuentra una sentencia como

```
bool dataOK;
```

traduce la sentencia en

```
int dataOK;
```

Con la sentencia Typedef y las declaraciones de dos constantes nombradas, `true` y `false`, el código al comienzo de esta descripción se convierte en lo siguiente:

```
typedef int bool;
const int true = 1;
const int false = 0;
  ⋮
bool dataOK;
  ⋮
dataOK = true;
  ⋮
dataOK = false;
```

Con C++ estándar, nada de esto es necesario porque `bool` es un tipo integrado. Si está trabajando con C++ pre-estándar, véase la sección D.4, del apéndice D, para más información acerca de cómo definir su propio tipo `bool`, de modo que pueda trabajar con los programas de este libro.

Expresiones lógicas

En los lenguajes de programación, las afirmaciones toman la forma de *expresiones lógicas* (llamadas también *expresiones booleanas*). Así como una expresión aritmética está constituida por valores numéricos y operaciones, una expresión lógica está conformada por valores lógicos y operaciones. Toda expresión lógica tiene uno de dos valores: verdadero o falso.

Aquí se muestran algunos ejemplos de expresiones lógicas:

- Una variable o constante booleana
- Una expresión seguida de un operador relacional seguido de una expresión
- Una expresión lógica seguida de un operador lógico seguido de una expresión lógica

Considérese cada una éstas en detalle.

Variables y constantes booleanas Como se ha visto, una variable boleana es una variable declarada del tipo `bool`, y puede contener un valor `true` (verdadero) o un valor `false` (falso). Por ejemplo, si `dataOK` es una variable booleana, entonces

```
dataOK = true;
```

es una sentencia de asignación válida.

Operadores relacionales Otra forma de asignar un valor a una variable booleana es igualarla al resultado de comparar dos expresiones con un *operador relacional*. Los operadores relacionales prueban una relación entre dos valores.

Considérese un ejemplo. En el siguiente fragmento de programa, `lessThan` es una variable booleana e `i` y `j` son variables `int`:

```
cin >> i >> j;
lessThan = (i < j);  // Compara a i y j con el operador relacional
                     // "less than" y asigna el
                     // valor verdadero a lessThan
```

Al comparar dos valores, se afirma que existe una relación (como “less than”, “menor que”) entre ellos. Si existe la relación, la afirmación es verdadera; de lo contrario, es falsa. Éstas son las relaciones que se pueden probar en C++:

Operador	Relación probada
`==`	Igual a
`!=`	No igual a
`>`	Mayor que
`<`	Menor que
`>=`	Mayor que o igual a
`<=`	Menor que o igual a

Una expresión seguida de un operador relacional seguido de una expresión se llama *expresión relacional*. El resultado de una expresión relacional es de tipo `bool`. Por ejemplo, si `x` es 5 y `y` es 10, las siguientes expresiones tienen el valor `true`:

```
x != y
y > x
x < y
y >= x
x <= y
```

Si `x` es el carácter 'M' y `y` es 'R', los valores de las expresiones aún son `true` porque el operador relacional <, usado con letras, significa "viene antes en el alfabeto", o bien, de modo más adecuado, "viene antes en la secuencia de intercalación del conjunto de caracteres". Por ejemplo, en el conjunto de caracteres ASCII ampliamente usado, las letras mayúsculas están en orden alfabético, al igual que las minúsculas, pero las mayúsculas van antes de las minúsculas. Por tanto,

```
'M' < 'R'
```

y

```
'm' < 'r'
```

tienen el valor `true`, pero

```
'm' < 'R'
```

tienen el valor `false`.

Por supuesto, se debe tener cuidado con los tipos de datos de cosas que se comparan. El método más seguro es comparar siempre `int` con `int`, `float` con `float`, `char` con `char`, etc. Si combina tipos de datos en una comparación, la coerción implícita de tipos toma lugar del mismo modo que en expresiones aritméticas. Si se comparan un valor `int` y un valor `float`, la computadora coerce temporalmente al valor `int` a su equivalente `float` antes de hacer la comparación. Como con las expresiones aritméticas, es sabio usar el moldeo explícito de tipos para dar a conocer sus intenciones:

```
someFloat >= float(someInt)
```

Si compara un valor `bool` con un valor aritmético (quizá por error), el valor `false` es forzado de manera temporal al número 0, y `true` es forzado a 1. Por tanto, si `boolVar` es una variable `bool`, la expresión

```
boolVar < 5
```

produce `true` porque 0 y 1 son menores que 5.

Hasta que aprenda más acerca del tipo `char` en el capítulo 10, sea cuidadoso al comparar valores `char` sólo con otros valores `char`. Por ejemplo, las comparaciones

```
'0' < '9'
```

y

```
0 < 9
```

son apropiadas, pero

```
'0' < 9
```

genera una coerción implícita de tipos y un resultado que tal vez no sea lo que espera.

Se pueden usar operadores relacionales no sólo para comparar variables o constantes, sino también para comparar valores de expresiones aritméticas. En la tabla siguiente se comparan los resultados de sumar 3 a `x` y multiplicar `y` por 10 para diferentes valores de `x` y `y`.

Valor de x	Valor de y	Expresión	Resultado
12	2	`x + 3 <= y * 10`	`true`
20	2	`x + 3 <= y * 10`	`false`
7	1	`x + 3 != y * 10`	`false`
17	2	`x + 3 == y * 10`	`true`
100	5	`x + 3 > y * 10`	`true`

Precaución: es fácil confundir el operador de asignación (=) y el operador relacional ==. Estos dos operadores tienen efectos muy diferentes en un programa. Algunas personas pronuncian el operador relacional como "igual-igual" para recordar la diferencia.

Comparación de cadenas Recuerde, del capítulo 4, que `string` es una clase, un tipo definido por el programador de la cual usted declara variables conocidas más comúnmente como objetos. Contenida dentro de cada objeto, `string` es una cadena de caracteres. La clase `string` se diseña de manera que es posible comparar estas cadenas por medio de operadores relacionales. Desde el punto de vista sintáctico, los operandos de un operador relacional pueden ser dos objetos `string`, como en

```
myString < yourString
```

o un objeto `string` como una cadena de C:

```
myString >= "Johnson"
```

Sin embargo, ambos operandos no pueden ser cadenas de C.

La comparación de cadenas sigue la secuencia de intercalación del conjunto de caracteres de la máquina (ASCII, por ejemplo). Cuando la computadora prueba una relación entre dos cadenas, comienza con el primer carácter de cada una, las compara de acuerdo con la secuencia de intercalación y, si son lo mismo, repite la comparación con el siguiente carácter en cada cadena. La prueba carácter por carácter procede hasta que se encuentra una disparidad o hasta que han sido comparados los caracteres finales y son iguales. Si todos los caracteres son iguales, entonces las dos cadenas son iguales. Si se encuentra una discordancia, entonces la cadena con el carácter que viene antes que el otro es la cadena "menor".

Por ejemplo, dadas las sentencias

```
string word1;
string word2;

word1 = "Tremendous";
word2 = "Small";
```

las expresiones relacionales de la tabla siguiente tienen los valores indicados.

Expresión	Valor	Razón
`word1 == word2`	`false`	Son distintas en el primer carácter.
`word1 > word2`	`true`	'T' viene después de 'S' en la secuencia de intercalación.
`word1 < "Tremble"`	`false`	Los caracteres en la quinta posición no concuerdan, y 'b' viene antes que 'e'.
`word2 == "Small"`	`true`	Son iguales.
`"cat" < "dog"`	Impredecible	Ambos operandos no pueden ser cadenas de `C.*`

* La expresión es correcta desde el punto de vista sintáctico en C++, pero da como resultado una comparación de apuntador, no una comparación de cadena. Los apuntadores se analizan en el capítulo 15.

En la mayoría de los casos, el orden de las cadenas corresponde al orden alfabético. Pero cuando las cadenas tienen letras mayúsculas y minúsculas, se obtienen resultados no alfabéticos. Por ejemplo, en un directorio se espera ver a Macauley antes que MacPherson, pero la secuencia de intercalación ASCII coloca las letras mayúsculas antes que las minúsculas, así que la cadena `"MacPherson"` se compara como menor que `"Macauley"`. Para comparar cadenas para orden alfabético estricto, todos los caracteres deben ser del mismo tipo (mayúsculas o minúsculas). En un capítulo posterior se muestra un algoritmo para cambiar una cadena de mayúsculas a minúsculas o viceversa.

Si se comparan dos cadenas con longitudes distintas y la comparación es igual hasta el final de la cadena más corta, entonces la cadena más corta se compara como menor que la cadena más larga. Por ejemplo, si `word2` contiene `"Small"`, la expresión

```
word2 < "Smaller"
```

produce `true`, porque las cadenas son iguales hasta el quinto carácter (el fin de la cadena de la izquierda), y la cadena de la derecha es más larga.

Operadores lógicos En matemáticas, los *operadores lógicos* (o booleanos) AND, OR y NOT toman expresiones lógicas como operandos. C++ emplea símbolos especiales para los operadores lógicos: `&&` (para AND), `||` (para OR) y ! (para NOT). Al combinar operadores relacionales con operadores lógicos, se pueden hacer afirmaciones más complejas. Por ejemplo, suponga que se quiere determinar si una puntuación final es mayor que 90 *y* una puntuación intermedia es mayor que 70. En C++ la expresión se escribiría de esta forma:

```
finalScore > 90 && midtermScore > 70
```

La operación AND (`&&`) requiere que ambas relaciones sean ciertas para que el resultado global sea verdadero. Si una o ambas relaciones son falsas, el resultado es falso.

La operación OR (`||`) toma dos expresiones lógicas y las combina. Si *una de ellas* o *ambas* son verdaderas, el resultado es verdadero. Ambos valores deben ser falsos para que el resultado sea falso. Ahora se puede determinar si la calificación intermedia es A *o* si la calificación final es A. Si la calificación intermedia o la final es igual a A, la afirmación es verdadera. En C++ la expresión se escribe de la siguiente manera:

```
midtermGrade == 'A' || finalGrade == 'A'
```

Los operadores `&&` y `||` aparecen siempre entre dos expresiones; son operadores binarios (dos operandos). El operador NOT (`!`) es un operador unario (un operando). Precede a una sola expresión lógica y da a su contrario (complemento) como resultado. Si (`grade == 'A'`) es falsa, entonces `!` (`grade == 'A'`) es verdadera. NOT da una forma conveniente de invertir el significado de una afirmación. Por ejemplo,

```
!(hours > 40)
```

es el equivalente de

```
hours <= 40
```

En algunos contextos, la primera forma es más clara; en otras, la segunda es más entendible.

Los siguientes pares de expresiones son equivalentes:

Expresión	Expresión equivalente
`!(a == b)`	`a != b`
`!(a == b || a == c)`	`a != b && a != c`
`!(a == b && c > d)`	`a != b || c <= d`

Examine con detenimiento estas expresiones para asegurarse de que entiende por qué son equivalentes. Pruebe evaluarlas con algunos valores para `a`, `b`, `c` y `d`. Observe el patrón: la expresión de la izquierda es justo la de la derecha con el signo ! agregado y los operadores relacional y lógico invertidos (por ejemplo, `==` en lugar de `!=` y `||` en lugar de `&&`). Recuerde este patrón. Esto le permite reescribir expresiones en la forma más simple.*

Los operadores lógicos pueden aplicarse a los resultados de comparaciones. También se pueden aplicar directamente a variables de tipo `bool`. Por ejemplo, en lugar de escribir

```
isElector = (age >= 18 && district == 23);
```

para asignar un valor a la variable booleana `isElector`, se podrían usar dos variables booleanas intermedias, `isVoter` e `isConstituent`:

```
isVoter = (age >= 18);
isConstituent = (district == 23);
isElector = isVoter && isConstituent;
```

En las dos tablas siguientes se resumen los resultados de aplicar `&&` y `||` a un par de expresiones lógicas (representadas aquí por las variables `x` y `y`).

Valor de `x`	Valor de `y`	Valor de `x && y`
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`false`
`false`	`false`	`false`

Valor de `x`	Valor de `y`	Valor de `x || y`
`true`	`true`	`true`
`true`	`false`	`true`
`false`	`true`	`true`
`false`	`false`	`false`

En la tabla siguiente se resumen los resultados de aplicar el operador `!` a una expresión lógica (representada por la variable booleana `x`).

Valor de `x`	Valor de `!x`
`true`	`false`
`false`	`true`

Técnicamente, los operadores de C++ `!`, `&&` y `||` no necesitan tener expresiones lógicas como operandos. Sus operandos pueden ser cualquier tipo de datos simples, incluso valores de punto flo-

* En álgebra booleana, el patrón se formaliza mediante un teorema llamado *ley de DeMorgan*.

tante. Si un operando no es de tipo `bool`, su valor es coercionado temporalmente al tipo `bool` como sigue: el valor 0 es coercionado a `false`, y cualquier valor no cero es coercionado a `true`. Como ejemplo, en ocasiones se encuentra un código de C++ que se parece a esto:

```
float height;
bool  badData;
  ⋮
cin >> height;
badData = !height;
```

La sentencia de asignación dice establecer `badData` en `true` si el valor coercionado de `height` es `false`. Es decir, la sentencia dice "Establecer `badData` en `true` si `height` es igual a 0.0". Aunque esta sentencia de asignación funciona correctamente de acuerdo con el lenguaje C++, muchos programadores encuentran más legible la sentencia siguiente:

```
badData = (height == 0.0);
```

En este texto los operadores lógicos se aplican *sólo* a expresiones lógicas, no a expresiones aritméticas.

Precaución: es fácil confundir los operadores lógicos `&&` y `||` con otros operadores de C++, `&` y `|`. Aquí no se analizan los operadores `&` y `|`, pero se señala que se emplean para manipular bits individuales dentro de una celda de memoria, una función muy distinta de la de los operadores lógicos. Si de manera accidental usa `&` en lugar de `&&`, o `|` en lugar de `||`, no obtendrá un mensaje de error del compilador, pero es probable que su programa calcule respuestas erróneas. Algunos programadores pronuncian `&&` como "y-y" y `||` como "o-o" para evitar cometer errores.

Evaluación de cortocircuito Considere la expresión lógica

```
i == 1 && j > 2
```

Algunos lenguajes de programación usan *evaluación completa* de expresiones lógicas. Con la evaluación completa, la computadora evalúa primero ambas subexpresiones (tanto `i == 1` como `j > 2`) antes de aplicar el operador `&&` para producir el resultado final.

Evaluación de cortocircuito (condicional) Evaluación de una expresión lógica en orden de izquierda a derecha con detención de la evaluación tan pronto como se determina el valor verdadero final.

En contraste, C++ usa la **evaluación de cortocircuito** (o **condicional**) de expresiones lógicas. La evaluación procede de izquierda a derecha, y la computadora detiene la evaluación de subexpresiones tan pronto como es posible, es decir, tan pronto como conoce el valor verdadero de la expresión completa. ¿Cómo puede saber la computadora si una expresión lógica larga produce `true` o `false` si no examina todas las subexpresiones? Considérese primero la operación AND.

Una operación AND produce el valor `true` sólo si sus dos operadores son `true`. En la expresión anterior, suponga que el valor de `i` es 95. La primera subexpresión produce `false`, así que es innecesario examinar la segunda expresión. La computadora detiene la evaluación y produce el resultado final `false`.

Con la operación OR, la evaluación de izquierda a derecha se detiene tan pronto como se encuentra una subexpresión que produce `true`. Recuerde que OR produce un resultado `true` si uno o los dos operadores son `true`. Dada la expresión:

```
c <= d || e == f
```

si la primera subexpresión es `true`, la evaluación se detiene y el resultado completo es `true`. La computadora no pierde tiempo con una evaluación innecesaria de la segunda subexpresión.

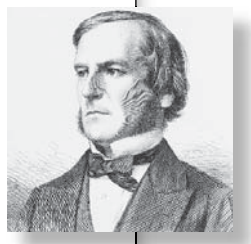
Conozca a

George Boole

El álgebra booleana se nombró así en honor a su inventor, el matemático inglés George Boole, quien nació en 1815. Su padre, un comerciante, comenzó a enseñarle matemáticas a temprana edad. Pero al inicio Boole se interesó más en la literatura clásica, los idiomas y la religión, interés que mantuvo toda su vida. A la edad de 20 años, había aprendido por sí mismo francés, alemán e italiano. Era una persona muy versada en los escritos de Aristóteles, Spinoza, Cicerón y Dante, y escribió varios documentos filosóficos.

Para ayudar a su familia, a los 16 años de edad entró como asistente de enseñanza en una escuela privada. Su trabajo ahí, y un segundo empleo como profesor, le dejaban poco tiempo para estudiar. Pocos años después abrió una escuela y comenzó a aprender matemáticas superiores por sí mismo. Cuando sólo tenía 24 años de edad, a pesar de su falta de capacitación formal, su primer artículo científico fue publicado en el *Cambridge Mathematical Journal*. Boole publicó más de 50 artículos y varios trabajos importantes antes de su muerte, en 1864, en la cumbre de su carrera.

The Mathematical Analysis of Logic, de Boole, se publicó en 1847. Finalmente formaría la base para el desarrollo de las computadoras digitales. En su libro, Boole estableció los axiomas formales de la lógica (muy parecidos a los axiomas de la geometría) sobre los que se construye el campo de la lógica simbólica.



Boole se basó en los símbolos y operaciones del álgebra para crear su sistema de lógica. Relacionó el valor 1 con el conjunto universal (el conjunto que representa todo en el universo) y el valor 0 con el conjunto vacío, y restringió su sistema a estas dos cantidades. Después definió las operaciones que son análogas a la resta, suma y multiplicación. Las variables del sistema tienen valores simbólicos. Por ejemplo, si una variable booleana *P* representa el conjunto de todas las plantas, entonces la expresión 1 – *P* se refiere al conjunto de todas las cosas que no son plantas. La expresión se simplifica al usar –*P* para indicar *"no* plantas". (0 – *P* es simplemente 0 porque no es posible eliminar elementos del conjunto vacío.) El operador de resta en el sistema de Boole corresponde al operador `!` (NOT) en C++. En un programa de C++ se podría establecer el valor de la variable booleana `plant` en `true` cuando se introduce el nombre de una planta, mientras que `!plant` es `true` cuando se introduce el nombre de alguna otra cosa.

La expresión 0 + *P* es lo mismo que *P*. Sin embargo, 0 + *P* + *F*, donde *F* es el conjunto de los alimentos, es el conjunto de todas las cosas que son plantas o alimentos. Así que el operador de suma en el álgebra de Boole es lo mismo que el operador `||` (OR) en C++.

La analogía se puede llevar a la multiplicación: 0 × *P* es 0, y 1 × *P* es *P*. ¿Pero qué es *P* × *F*? Es el conjunto de cosas que son plantas y alimentos. En el sistema de Boole, el operador de multiplicación es lo mismo que el operador `&&` (AND).

En 1854, Boole publicó *An Investigation of the Laws of Thought, on Which Are Founded the Mathematical Theories of Logic and Probabilities*. En el libro, describió teoremas elaborados sobre sus axiomas de lógica y extendió el álgebra para mostrar cómo se podrían calcular las probabilidades en un sistema lógico. Cinco años después, Boole publicó *Treatise on Differential Equations*, luego *Treatise on the Calculus of Finite Differences*. El último es una de las piedras angulares del análisis numérico, que trata con la exactitud de los cálculos. (En el capítulo 10, se examina el papel importante que desempeña el análisis numérico en la programación.)

Boole recibió poco reconocimiento y mínimos honores por su trabajo. Dada la importancia del álgebra booleana en la tecnología moderna, es difícil creer que su sistema de lógica no se haya tomado en serio hasta comienzos del siglo XX. George Boole fue uno de los fundadores de la ciencia de la computación.

Precedencia de operadores

En el capítulo 3 se analizaron las reglas de precedencia, las reglas que gobiernan la evaluación de expresiones aritméticas complejas. Las reglas de precedencia de C++ gobiernan también los operado-

res relacionales y lógicos. A continuación se proporciona una lista que muestra el orden de precedencia para operadores aritméticos, relacionales y lógicos (incluido también el operador de asignación):

```
!  Unario +  Unario -
*   /   %
+   -
<   <=   >   >=
==   !=
&&
||
=
```

Mayor precedencia

↓

Menor precedencia

Los operadores en la misma línea de la lista tienen la misma precedencia. Si una expresión contiene varios operadores con la misma precedencia, la mayoría de los operadores se agrupan (o *asocian*) de izquierda a derecha. Por ejemplo, la expresión

```
a / b * c
```

significa (`a/b`) `* c`, no `a/`(`b * c`). Sin embargo, los operadores unarios (`!`, unario +, unario –) se agrupan de derecha a izquierda. Aunque usted nunca tendría ocasión de usar esta expresión:

```
!!badData
```

el significado de ésta es `!(!badData)` en vez de `(!!)badData`, carente de significado. En el apéndice B, "Precedencia de operadores", se enlista el orden de precedencia para los operadores en C++. En una lectura superficial del apéndice, se puede ver que algunos de los operadores se asocian de derecha a izquierda (por la misma razón que se describió para el operador !).

Los paréntesis se emplean para anular el orden de evaluación en una expresión. Si no está seguro de si los paréntesis son necesarios, empléelos de cualquier manera. El compilador hace caso omiso de paréntesis innecesarios. Así que si aclaran una expresión, utilícelos. Algunos programadores prefieren incluir paréntesis extra al asignar una expresión relacional a una variable booleana.

```
dataInvalid = (inputVal == 0);
```

Los paréntesis no son necesarios; el operador de asignación tiene la menor precedencia de todos los operadores que se han enlistado. Así que la sentencia se podría escribir como

```
dataInvalid = inputVal == 0;
```

pero algunas personas encuentran más legible la expresión con paréntesis.

Un comentario final acerca de los paréntesis: C++, como otros lenguajes de programación, requiere que los paréntesis se empleen siempre en pares. Siempre que escriba una expresión complicada, tómese un minuto para repasar y ver que cada apertura de paréntesis se cierre.

PEANUTS © UFS. Reimpreso con permiso

Consejo práctico de ingeniería de software

Cambiar los enunciados en expresiones lógicas

En la mayoría de los casos se puede escribir una expresión lógica directamente de un enunciado o término matemático en un algoritmo. Pero es necesario observar algunas situaciones delicadas. Recuerde la expresión lógica del ejemplo:

```
midtermGrade == 'A' || finalGrade == 'A'
```

En inglés (o en español), se estaría tentado a escribir: "la calificación intermedia o la calificación final es igual a A". En C++, no puede escribir la expresión como lo haría en inglés o en español. Es decir,

```
midtermGrade || finalGrade == 'A''
```

no funcionaría porque el operador `||` está conectando un valor `char (midtermGrade)` y una expresión lógica `(finalGrade == 'A')`. Los dos operandos de `||` deben ser expresiones lógicas. (Observe que esta expresión es errónea en términos de lógica, pero no es "equivocada" para el compilador de C++. Recuerde que el operador `||` puede conectar lícitamente dos expresiones de cualquier tipo de datos, de modo que este ejemplo no generará un mensaje de error de sintaxis. El programa correrá, pero no funcionará de la manera como usted pretendía.)

Una variación de este error es expresar la afirmación "*i* es igual a 3 o 4" como

```
i == 3 || 4
```

De nuevo, la sintaxis es correcta pero no la semántica. Esta expresión se evalúa siempre como `true`. La primera subexpresión, `i == 3`, puede ser `true` o `false`. Pero la segunda, 4, es distinta de cero y, por tanto, es forzada al valor `true`. Así, la operación `||` ocasiona que toda la expresión sea `true`. De nuevo: use el operador `||` (y el operador `&&`) sólo para conectar dos expresiones lógicas. Aquí está lo que se desea:

```
i == 3 || i == 4
```

En los libros de matemáticas se podría ver una notación como esta:

$$12 < y < 24$$

que significa "*y* está entre 12 y 24". Esta expresión es lícita en C++ pero proporciona un resultado inesperado. Primero se evalúa la relación 12 < *y*, y da el resultado `true` o `false`. La computadora obliga entonces el resultado a 1 o 0 para compararlo con el número 24. Debido a que 1 y 0 son menores que 24, el resultado siempre es `true`. Para escribir esta expresión correctamente en C++, se debe usar el operador `&&` como sigue:

```
12 < y && y < 24
```

Operadores relacionales con tipos de punto flotante

Hasta ahora se ha hablado sólo de comparar valores `int`, `char` y `string`. Aquí se consideran valores `float`.

No comparar números de punto flotante para igualdad. Debido a la posibilidad de que surjan errores pequeños en los lugares decimales de la derecha cuando se efectúan cálculos en números de punto flotante, dos valores `float` rara vez son exactamente iguales. Por ejemplo, considere el código siguiente que usa dos variables `float` llamadas `oneThird` y `x`.

```
oneThird = 1.0 / 3.0;
x = oneThird + oneThird + oneThird;
```

Se esperaría que `x` contuviera el valor 1.0, pero es probable que no. La primera sentencia de asignación almacena una *aproximación* de 1/3 en `oneThird`, quizá 0.333333. La segunda sentencia almacena un valor como 0.999999 en `x`. Si ahora se pide a la computadora que compare `x` con 1.0, la comparación produce `false`.

En lugar de probar números de punto flotante para igualdad, se prueba para *casi* igualdad. Para esto, se calcula la diferencia entre los dos números y se prueba ver si el resultado es menor que alguna diferencia máxima permisible. Por ejemplo, con frecuencia se emplean comparaciones como ésta:

```
fabs(r - s) < 0.00001
```

donde `fabs` es la función valor absoluto de punto flotante de la biblioteca estándar C++. La expresión `fabs(r - s)` calcula el valor absoluto de la diferencia entre dos variables `float r` y `s`. Si la diferencia es menor que 0.00001, los dos números están suficientemente cerca para considerarlos iguales. En el capítulo 10 se analiza con más detalle este problema con la exactitud de punto flotante.

5.3 Sentencia If

Ahora que se ha visto cómo escribir expresiones lógicas, se emplearán para alterar el flujo de control normal en un programa. La *sentencia If* es la estructura de control fundamental que permite bifurcaciones en el flujo de control. Con ésta, se puede hacer una pregunta y elegir un curso de acción: *Si* existe cierta condición, *entonces* se efectúa una acción, *de otro modo* se realiza una acción diferente.

En tiempo de ejecución, la computadora realiza una de las dos acciones, lo cual depende del resultado de la condición que es probada. Sin embargo, se debe incluir el código para *ambas* acciones en el programa. ¿Por qué? Porque, dependiendo de las circunstancias, la computadora puede elegir ejecutar *cualquiera* de ellas. La sentencia If proporciona una manera de incluir ambas acciones en un programa y da a la computadora una posibilidad de decidir cuál acción tomar.

Forma If-Then-Else

En C++, la sentencia If viene en dos formas: la forma *If-Then-Else* y la forma *If-Then*. Considérese primero If-Then-Else. Aquí está su plantilla de sintaxis.

Sentencia If (forma If-Then-Else)

```
if  ( Expresión )
      Sentencia 1A
else
      Sentencia 1B
```

La expresión entre paréntesis puede ser de cualquier tipo de datos simples. Casi sin excepción, ésta será una expresión lógica (booleana); si no, su valor es coercionado implícitamente al tipo `bool`. Al momento de la ejecución, la computadora evalúa la expresión. Si el valor es `true`, la computadora ejecuta la sentencia 1A. Si el valor de la expresión es `false`, se ejecuta la sentencia 1B. La sentencia 1A suele llamarse *cláusula then;* la sentencia 1B es la *cláusula else*. En la figura 5-3 se ilustra el flujo de control de If-Then-Else. En la figura, la sentencia 2 es la siguiente en el programa después de la sentencia completa If.

Observe que una sentencia If en C++ emplea las palabras reservadas `if` y `else`, pero no incluye la palabra *then*. No obstante, se usa el término *If-Then-Else* porque corresponde a cómo se expresan cosas en el lenguaje común: "*Si* algo es verdadero, *entonces* se hace esto, *de otro modo* se hace aquello".

En el fragmento de código siguiente se muestra cómo escribir una sentencia If en un programa. Observe la sangría de la cláusula then y de la cláusula else, que facilita la lectura de la sentencia. Observe el lugar de la sentencia después de la sentencia If.

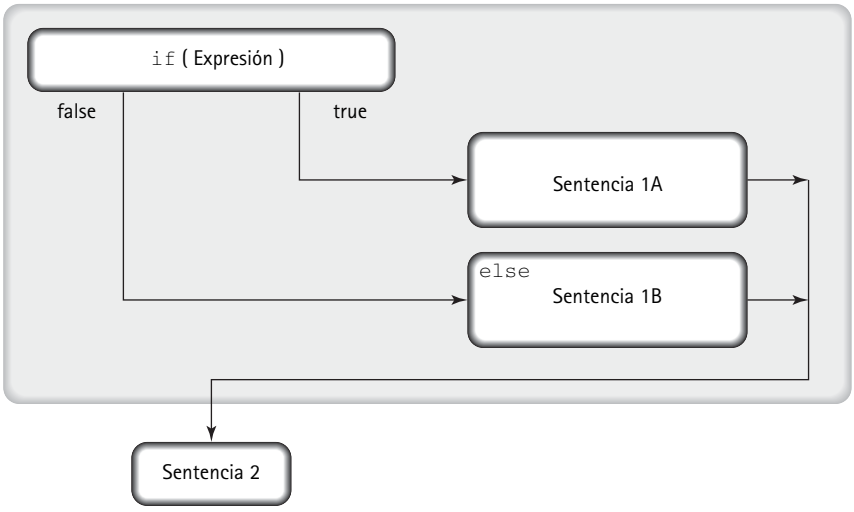


Figura 5-3 *Flujo de control de If-Then-Else*

```
if (hours <= 40.0)
    pay = rate * hours;
else
    pay = rate * (40.0 + (hours - 40.0) * 1.5);
cout << pay;
```

En términos de instrucciones para la computadora, el fragmento de código anterior dice, "Si `hours` es menor que o igual a 40.0, calcule el pago regular y luego continúe con la ejecución de la sentencia de salida. Pero si `hours` es mayor que 40, calcule el pago regular y el pago por tiempo extra, y después prosiga con la ejecución de la sentencia de salida". En la figura 5-4 se muestra el flujo de control de esta sentencia If.

If-Then-Else se emplea con frecuencia para comprobar la validez de la entrada. Por ejemplo, antes de pedir a la computadora que divida entre un valor, se debe estar seguro de que el valor no sea cero. (Incluso las computadoras no pueden dividir entre cero. Si lo intenta, la mayoría de las

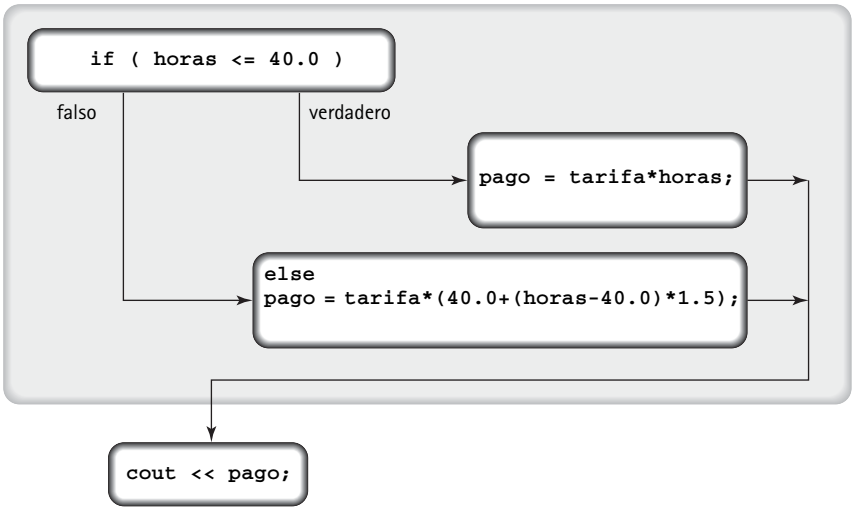


Figura 5-4 *Flujo de control para calcular el pago*

computadoras detienen la ejecución de su programa.) Si el divisor es cero, el programa debe imprimir un mensaje de error. Aquí está el código:

```
if (divisor != 0)
    result = dividend / divisor;
else
    cout << "No se permite la división entre cero." << endl;
```

Como otro ejemplo de If-Then-Else, suponga que se desea determinar dónde se localiza en una variable `string` la primera ocurrencia (si la hay) de la letra A. Recuerde, del capítulo 3, que la clase `string` tiene una función miembro llamada `find`, que devuelve la posición donde se encontró el ítem (o la constante nombrada `string::npos` si no se encontró el ítem). El siguiente código produce el resultado de la búsqueda:

```
string myString;
string::size_type pos;
          ⋮
pos = myString.find('A');
if (pos == string::npos)
    cout << "No se encontró 'A'" << endl;
else
    cout << "Se encontró una 'A' en la posición " << pos << endl;
```

Antes de ver algo más en las sentencias If, examine de nuevo la plantilla de sintaxis de If-Then-Else. De acuerdo con la plantilla, no hay punto y coma al final de una sentencia If. En todos los fragmentos de programa anteriores, los ejemplos del pago del trabajador, la división entre cero y la búsqueda de cadena, parece haber un punto y coma al final de cada sentencia If. Sin embargo, los punto y coma pertenecen a las sentencias de las cláusulas else en esos ejemplos; las sentencias de asignación terminan en punto y coma, así como las sentencias de salida. La sentencia If no tiene su propio punto y coma al final.

Bloques (sentencias compuestas)

En el ejemplo de la división entre cero, suponga que cuando el divisor es igual a cero se desea hacer *dos* cosas: imprimir el mensaje de error *e* igualar la variable llamada `result` a un valor especial como 9999. Serían necesarias dos sentencias en la misma rama, pero al parecer la plantilla de sintaxis limita a una.

Lo que se desea es convertir la cláusula else en una *secuencia* de sentencias. Esto es fácil. Recuerde, del capítulo 2, que el compilador trata el bloque (sentencia compuesta)

```
{
    ⋮
}
```

como una sola sentencia. Si pone un par de llaves `{ }` alrededor de la secuencia de sentencias que desea en una rama de la sentencia If, la secuencia de sentencias se convierte en un solo bloque. Por ejemplo:

```
if (divisor != 0)
    result = dividend / divisor;
else
{
    cout << "No se permite la división entre cero." << endl;
    result = 9999;
}
```

Si el valor del `divisor` es 0, la computadora imprime el mensaje de error y establece el valor de `result` en 9999 antes de continuar con cualquier sentencia después de la sentencia If.

Los bloques se pueden usar en ambas ramas de If-Then-Else. Por ejemplo:

```
if (divisor != 0)
{
    result = dividend / divisor;
    cout << "División efectuada." << endl;
}
else
{
    cout << "No se permite la división entre cero." << endl;
    result = 9999;
}
```

Cuando usa bloques en una sentencia If, hay una regla de sintaxis de C++ por recordar: *nunca use punto y coma después de la llave derecha de un bloque*. Los punto y coma se emplean sólo para terminar sentencias simples como sentencias de asignación, de entrada y salida. Si examina los ejemplos anteriores, no verá un punto y coma después de la llave derecha que señala el fin de cada bloque.

Cuestiones de estilo

Llaves y bloques

Los programadores de C++ emplean estilos diferentes cuando se trata de localizar la llave izquierda de un bloque. La manera como se usa aquí es poner las llaves izquierda y derecha directamente abajo de las palabras `if` y `else`, cada llave en su propia línea:

```
if (n >= 2)
{
    alpha = 5;
    beta = 8;
}
else
{
    alpha = 23;
    beta = 12;
}
```

Otro estilo popular es colocar las llaves izquierdas al final de la línea `if` y la línea `else`; las llaves derechas se alinean directamente debajo de las palabras `if` y `else`. Esta forma de dar formato a la sentencia If se originó con los programadores que usan el lenguaje C, el predecesor de C++.

```
if (n >= 2) {
    alpha = 5;
    beta = 8;
}
else {
    alpha = 23;
    beta = 12;
}
```

Para el compilador de C++ no hay diferencia en cuanto al estilo que usted use (y también hay otros estilos). Es una cuestión de preferencia personal. Sin embargo, cualquiera que sea el estilo que use, debe ser el mismo en todo el programa. La inconsistencia puede confundir a la persona que lee su programa y dar la impresión de descuido.

Forma If-Then

En ocasiones se encuentra una situación donde usted quiere decir: "*Si* existe cierta condición, *entonces* se efectúa alguna acción; de lo contrario, no hacer nada". En otras palabras, desea que la computadora omita una serie de instrucciones si no se satisface cierta condición. Esto se podría hacer al dejar vacía la rama `else` y usar sólo la sentencia nula:

```
if (a <= b)
    c = 20;
else
    ;
```

Mejor todavía, puede simplemente eliminar la parte `else`. La sentencia resultante es la forma If-Then de la sentencia If. Ésta es la plantilla de sintaxis:

Sentencia If (la forma If-Then)

```
if ( Expresión )
     Sentencia
```

A continuación se presenta un ejemplo de una forma If-Then. Observe el sangrado y la colocación de la sentencia después de If-Then.

```
if (age < 18)
    cout << "Not an eligible ";
cout << "voter." << endl;
```

Esta sentencia significa que si `age` es menor que 18, se imprime primero "Not an eligible" y luego se imprime "voter". Si `age` no es menor que 18, se omite la primera sentencia de salida y se pasa directamente a imprimir "voter". En la figura 5-5 se muestra el flujo de control para If-Then.

Al igual que las dos ramas de If-Then-Else, la rama de If-Then puede ser un bloque. Por ejemplo, digamos que usted está escribiendo un programa para calcular impuestos de ingresos. Una de las líneas de la forma de impuestos lee "resta la línea 23 de la línea 27 e introduce el resultado en la línea 24; si el resultado es menor que cero, introduce cero y comprueba la caja 24A". Se puede usar If-Then para hacer esto en C++:

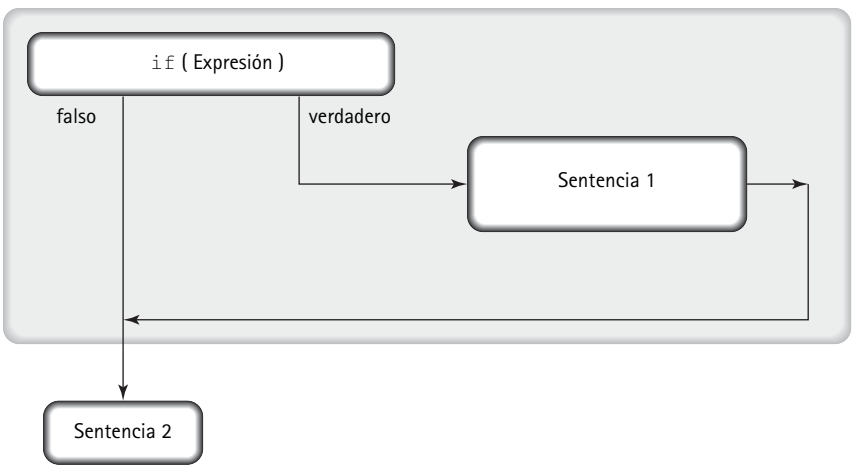


Figura 5-5 *Flujo de control de If-Then*

```
result = line17 - line23;
if (result < 0.0)
{
    cout << "Check box 24A" << endl;
    result = 0.0;
}
line24 = result;
```

Este código hace exactamente lo que dice la forma de impuestos. Calcula el resultado de restar la línea 23 de la línea 17. Luego compara si `result` es menor que 0. Si es así, el fragmento imprime un mensaje que indica al usuario comprobar la caja 24A y luego establece `result` en 0. Por último, el resultado calculado (o 0, si el resultado es menor que 0) se almacena en una variable denominada `line24`.

¿Qué sucede si se omiten las llaves izquierda y derecha en el fragmento de código anterior? Examínese lo siguiente:

```
result = line17 - line23;              // Incorrect version
if (result < 0.0)
    cout << "Comprobar la caja 24A" << endl;
    result = 0.0;
line24 = result;
```

A pesar de la forma como se ha colocado la sangría en el código, el compilador toma la cláusula then como una sola sentencia, la sentencia de salida. Si `result` es menor que 0, la computadora ejecuta la sentencia de salida, luego establece `result` en 0 y después guarda `result` en `line24`. Hasta ahora, todo bien. Pero si `result` es inicialmente mayor que o igual a 0, la computadora omite la cláusula then y procede a la sentencia que sigue a la sentencia If: la sentencia de asignación que fija `result` en 0. ¡El resultado poco afortunado es que `result` termina como 0 sin importar cuál fue su valor inicial! La moraleja aquí es no confiar sólo en la sangría; no se puede engañar al compilador. Si desea una sentencia compuesta para una cláusula then o else, debe incluir las llaves izquierda y derecha.

Un error común

Ya se advirtió sobre la confusión entre el operador `=` y el operador `==`. Aquí se ofrece un ejemplo de un error que, se garantiza, todo programador de C++ cometerá por lo menos una vez en su carrera:

```
cin >> n;
if (n = 3)                        // Wrong
    cout << "n equals 3";
else
    cout << "n doesn't equal 3";
```

Este segmento de código *siempre* imprime

```
n equals 3
```

sin importar qué se introdujo para `n`.

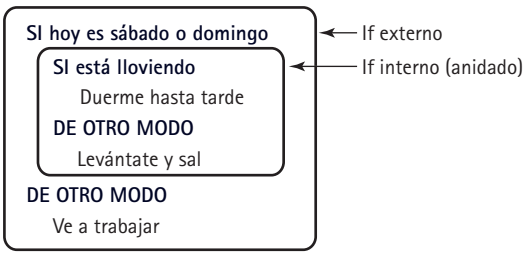
Aquí está la razón: se ha usado el operador equivocado en la prueba If. La expresión `n = 3` no es una expresión lógica; se llama *expresión de asignación*. (Si una asignación se escribe como una sentencia separada que termina con punto y coma, es una *sentencia* de asignación.) Una expresión de asignación tiene un *valor* (en el segmento de código anterior es 3) y un *efecto secundario* (almacenar 3 en `n`). En la sentencia If del ejemplo, la computadora encuentra que el valor de la expresión probada es 3. Porque 3 es un valor distinto de cero y, por tanto, es obligado o coercionado a `true`, se ejecuta la cláusula then, sin importar cuál es el valor de `n`. Peor aún, el efecto secundario de la expresión de asignación es guardar 3 en `n`, destruyendo lo que hubiera ahí.

La intención no es centrarse en expresiones de asignación; su uso se analiza más adelante en este libro. Lo que importa ahora es que vea el efecto de usar = cuando su intención es usar ==. El programa compila de manera correcta, pero su ejecución es incorrecta. Al depurar un programa defectuoso, examine siempre sus sentencias If para ver si ha cometido este error particular.

5.4 Sentencias If anidadas

No hay restricciones en cuanto a cuáles pueden ser las sentencias en un If. Por tanto, un If dentro de un If es correcto. De hecho, un If dentro de un If dentro de un If es lícito. La única limitación aquí es que las personas no pueden seguir una estructura demasiado compleja, y la legibilidad es una de las características de un buen programa.

Cuando se coloca un If dentro de un If, se está creando una *estructura de control anidada*. Las estructuras de control se anidan de forma parecida a como se acomodan los recipientes, con el más pequeño dentro del más grande. A continuación se proporciona un ejemplo, escrito en seudocógido:



En general, cualquier problema relacionado con una *rama multivía* (más de dos cursos de acción opcionales) se puede codificar por medio de sentencias If anidadas. Por ejemplo, para imprimir el nombre de un mes dado su número, se podría usar una secuencia de sentencias If (no anidadas):

```
if (month == 1)
    cout << "January";
if (month == 2)
    cout << "February";
if (month == 3)
    cout << "March";
  ⋮
if (month == 12)
    cout << "December"
```

Pero la estructura If anidada equivalente

```
if (month == 1)
    cout << "January";
else
    if (month == 2)          // If anidado
        cout << "February";
    else
        if (month == 3)          // If anidado
            cout << "March";
        else
            if (month == 4)          // If anidado
                ⋮
```

es más eficaz porque hace menos comparaciones. La primera versión, la secuencia de sentencias If independientes, siempre prueba cada condición (las 12), incluso si se satisface la primera. Por contraste, la solución If anidada omite las comparaciones restantes después de que se ha seleccionado una opción. A pesar de la rapidez de las computadoras modernas, muchas aplicaciones requieren demasiados cálculos a tal grado que los algoritmos ineficientes pierden mucho tiempo de computadora. Siempre esté atento para hacer sus programas más eficientes, siempre y cuando no se dificulte que otros programadores entiendan. En general es mejor sacrificar un poco de eficiencia por legibilidad.

En el ejemplo anterior, observe cómo la sangría de las cláusulas then y else ocasiona que las sentencias se muevan de modo continuo a la derecha. En lugar de eso, se puede usar un estilo de sangrado especial con sentencias If-Then-Else profundamente anidadas para indicar que la estructura compleja elige sólo una de un conjunto de opciones. Esta rama multivía general se conoce como una estructura de control *If-Then-Else*:

```
if (month == 1)
    cout << "January";
else if (month == 2)          // If anidado
    cout << "February";
else if (month == 3)          // If anidado
    cout << "March";
else if (month == 4)          // If anidado
      ⋮
else
    cout << "December";
```

Este estilo evita que la sangría marche de modo continuo a la derecha. Pero, más importante, visualmente conlleva la idea de que se está usando una rama de 12 vías basada en la variable `month`.

Es importante observar la diferencia entre la secuencia de sentencias If y el If anidado: la secuencia de sentencias If puede tomar más de una opción, pero el If anidado sólo puede seleccionar una. Para ver por qué esto es importante, considere la analogía de llenar un cuestionario. Algunas preguntas son como una secuencia de sentencias If, en la que se pide circular los elementos de una lista que le aplican a usted (por ejemplo, sus pasatiempos). En otras preguntas se le pide circular un elemento de una lista (su grupo de edad, por ejemplo) y son, por tanto, como una estructura If anidada. Ambas clases de preguntas ocurren en problemas de programación. Reconocer qué tipo de pregunta se hace permite seleccionar de inmediato la estructura de control apropiada.

Otro uso particularmente útil del If anidado es cuando se desea seleccionar de una serie de intervalos consecutivos de valores. Por ejemplo, suponga que se desea imprimir una actividad apropiada para la temperatura ambiente, dada la tabla siguiente.

Actividad	Temperatura
Nadar	Temperatura > 85
Tenis	70 < temperatura ≤ 85
Golf	32 < temperatura ≤ 70
Esquiar	0 < temperatura ≤ 32
Bailar	Temperatura ≤ 0

A primera vista se podría estar tentado a escribir una sentencia If separada para cada intervalo de temperaturas. Sin embargo, al examinar con detenimiento es evidente que estas condiciones If son independientes. Es decir, si se ejecuta una de las sentencias, ninguna de las otras debe ejecutarse. Se está seleccionando una opción de un conjunto de posibilidades, como el tipo de situación en que se

puede usar una estructura If anidada como una rama multivía. La única diferencia entre este problema y el ejemplo anterior de imprimir el nombre del mes a partir de su número, es que se deben comprobar intervalos de números en las expresiones If de las ramas.

Cuando los intervalos son consecutivos, se puede aprovechar ese hecho para hacer más eficiente el código. Las ramas se arreglan en orden consecutivo por intervalo. Entonces, si se ha alcanzado una rama particular, se sabe que los intervalos precedentes ya no se toman en cuenta. Así, las expresiones If deben comparar la temperatura sólo con el valor mínimo de cada intervalo. Examine el siguiente programa de actividades.

```
//*****************************************************************
// Programa Actividad
// Este programa produce una actividad apropiada
// para una temperatura dada
//*****************************************************************
#include <iostream>

using namespace std;

int main()
{
    int temperature;    // La temperatura exterior

    // Leer e imprimir por eco la temperatura

    cout << "Introducir la temperatura exterior:" << endl;
    cin >> temperature;
    cout << "La temperatura actual es " << temperature << endl;

    // Imprimir actividad

    cout << "La actividad recomendada es ";
    if (temperature > 85)
        cout << "nadar." << endl;
    else if (temperature > 70)
        cout << "tenis." << endl;
    else if (temperature > 32)
        cout << "golf." << endl;
    else if (temperature > 0)
        cout << "esquiar." << endl;
 else
        cout << "bailar." << endl;

    return 0;
}
```

Para ver cómo funciona la estructura If-Then-Else-If en este programa, considere la rama que realiza la prueba para `temperature` mayor que 70. Si se ha alcanzado, se sabe que `temperature` debe ser menor que o igual a 85 porque esa condición ocasiona que se tome esta rama particular else. Así, sólo se necesita probar si `temperature` está arriba del fondo de este intervalo (> 70). Si falla esa prueba, entonces se introduce la siguiente cláusula `else` sabiendo que `temperature` debe ser menor que o igual a 70. Cada rama sucesiva comprueba el fondo de su intervalo hasta que se llega al `else` final, que se encarga de las demás posibilidades.

Observe que si los intervalos no son consecutivos, entonces se debe probar el valor de datos contra el valor máximo y mínimo de cada intervalo. Aún se emplea If-Then-Else-If porque ésta es la mejor estructura para seleccionar una sola rama de múltiples posibilidades, y es posible arreglar los intervalos en orden consecutivo para facilitar la lectura al humano. Pero no hay forma de reducir el número de comparaciones cuando hay espacios entre los intervalos.

`else` suspendido

Cuando las sentencias If están anidadas, es posible confundirse con los pares `if-else`. Es decir, ¿a cuál `if` pertenece un `else`? Por ejemplo, suponga que si el promedio de un alumno está debajo de 60, se desea imprimir "Reprobado"; si está en por lo menos 60 pero debajo de 70, se quiere imprimir "Pasa pero está en el mínimo"; y si su promedio es 70 o mayor no se desea imprimir nada.

Esta información se codifica con un If-Then-Else anidado dentro de un If-Then.

```
if (average < 70.0)
    if (average < 60.0)
        cout << "Reprobado";
    else
        cout << "Pasa pero está en el mínimo";
```

¿Cómo se sabe a qué `if` pertenece el `else`? Aquí está la regla que sigue el compilador de C++: en ausencia de llaves, un `else` forma pareja siempre con el `if` precedente más cercano que aún no tenga un `else` como pareja. El código se escribió con sangrías para reflejar las parejas.

Suponga que el fragmento se escribe como se muestra a continuación:

```
if (average >= 60.0)      // Versión incorrecta
    if (average < 70.0)
        cout << "Pasa pero está en el mínimo";
else
    cout << "Reprobado";
```

Aquí se quiere la rama `else` unida a la sentencia If externa, no a la interna, de modo que se sangra el código como se observa. Pero el sangrado no afecta la ejecución del código. Aunque el `else` se alinea con el primer `if`, el compilador lo relaciona con el segundo `if`. Un `else` que sigue un If-Then anidado se llama else *suspendido*. Desde el punto de vista lógico no pertenece al If anidado, pero el compilador lo relaciona con él.

Para asociar el `else` con el primer `if`, no el segundo, se puede convertir la cláusula then externa en un bloque:

```
if (average >= 60.0)      // Versión correcta
{
    if (average < 70.0)
        cout << "Pasa pero está en el mínimo";
}
else
    cout << "Reprobado";
```

El par de `{ }` indica que la sentencia If interna está completa, así que el `else` debe pertenecer al `if` externo.

5.5 Probar el estado de un flujo I/O

En el capítulo 4 se habló del concepto de flujos de entrada y salida en C++. Se introdujeron las clases `istream`, `ostream`, `ifstream` y `ofstream`. Se dijo que cualquiera de las siguientes acciones puede causar que un flujo de entrada se coloque en el estado de falla:

- Datos de entrada no válidos
- Un intento de leer más allá del fin de un archivo
- Un intento de abrir un archivo no existente para entrada

C++ proporciona una forma de comprobar si un flujo se encuentra en el estado de falla. En una expresión lógica, sólo se usa el nombre del objeto de flujo (por ejemplo, `cin`) como si fuese una variable booleana:

```
if (cin)
  ⋮

if ( !inFile )
  ⋮
```

Probar el estado de un flujo Acto de usar un objeto de flujo de C++ en una expresión lógica como si fuera una variable booleana; el resultado es `true` si tuvo éxito la última operación I/O en ese flujo, y `false` en caso contrario.

Cuando usted hace esto, se dice que se está **probando el estado del flujo.** El resultado de la prueba es `true` (lo cual indica que la última operación I/O en ese flujo tuvo éxito) o `false` (cuyo significado es que falló la última operación I/O).

Conceptualmente, se puede considerar un objeto de flujo en una expresión lógica como una variable booleana con un valor `true` (el estado de flujo es correcto) o `false` (el estado no es correcto).

Observe en la segunda sentencia If anterior que se teclearon espacios alrededor de la expresión `!inFile`. Los espacios no son requeridos por C++ sino que están ahí para legibilidad. Sin los espacios, es más difícil ver el signo de admiración:

```
if (!inFile)
```

En una sentencia If, la forma de enunciar la expresión lógica depende de lo que se desee haga la cláusula then. La sentencia

```
if (inFile)
  ⋮
```

ejecuta la cláusula then si tiene éxito la última operación I/O en `inFile`. La sentencia

```
if ( !inFile )
  ⋮
```

ejecuta la cláusula then si `inFile` está en el estado de falla. (Recuerde que una vez que un flujo está en el estado de falla, permanece así. Cualquier operación adicional I/O en ese flujo es nula.)

A continuación se presenta un ejemplo que muestra cómo comprobar si un archivo de entrada se abrió con éxito:

```
//****************************************************************
// StreamState program
// This program demonstrates testing the state of a stream
//****************************************************************
#include <iostream>
#include <fstream>     // For file I/O

using namespace std;

int main()
```

```
{
    int      height;
    int      width;
    ifstream inFile;

    inFile.open("measures.dat");       // Attempt to open input file
    if ( !inFile )                     // Was it opened?
    {
        cout << "Can't open the input file.";   // No--print message
        return 1;                               // Terminate program
    }
    inFile >> height >> width;
    cout << "For a height of " << height << endl
         << "and a width of " << width << endl
         << "the area is " << height * width << endl;
    return 0;
}
```

En este programa se empieza por intentar abrir el archivo de disco `measures.dat` para ingreso. Se comprueba de inmediato si el intento fue exitoso. Si es así, el valor de la expresión `!inFile` en la sentencia If es `false` y se omite la cláusula then. El programa procede a leer datos del archivo y luego ejecuta un cálculo. Concluye al ejecutar la sentencia

```
return 0;
```

Con esta sentencia, la función `main` devuelve el control al sistema operativo de la computadora. Recuerde que el valor de función devuelto por `main` se conoce como el estado de salida. El valor 0 significa terminación normal del programa. Cualquier otro valor (por lo común 1, 2, 3, ...) indica que algo estuvo mal.

Repasemos de nuevo el programa, suponiendo que no se pudo abrir el archivo de entrada. Como resultado de la función open, el flujo `inFile` se encuentra en el estado de falla. En la sentencia If, el valor de la expresión `!inFile` es `true`. Así, se ejecuta la cláusula then. El programa imprime un mensaje de error para el usuario y luego termina, devolviendo un estado de salida de 1 para informar al sistema operativo de una terminación anormal del programa. (La elección del valor 1 para el estado de salida es puramente arbitraria. Los programadores de sistemas en ocasiones usan varios valores distintos en un programa a fin de señalar razones distintas para la terminación de un programa. Pero la mayoría de las personas sólo emplean el valor 1.)

Siempre que abra un archivo de datos para ingreso, asegúrese de probar el estado de flujo antes de proceder. Si lo olvida, y la computadora no puede abrir el archivo, su programa continúa la ejecución de un modo discreto e ignora cualquier operación de entrada en el archivo.

Caso práctico de resolución de problemas

Calculadora para el IMC

PROBLEMA Se ha hablado mucho acerca del sobrepeso actual de gran parte de la población estadounidense. En casi todas las revistas hay un artículo relacionado con los problemas de salud causados por la obesidad. En lugar de examinar una gráfica que muestra el peso promedio para una estatura particular, se ha hecho popular una medida denominada índice de masa corporal (IMC), que calcula una relación de su peso y estatura para determinar un peso apropiado. La fórmula para valores no métricos es

$$IMC = peso * 703/estatura^2$$

El IMC se correlaciona con la grasa corporal, que puede ser empleado para determinar si un peso es inapropiado para cierta estatura. Aunque el análisis del IMC en los medios es muy reciente, Adolphe Quetelet, un estadístico

belga del siglo XIX, obtuvo la fórmula. Realice una investigación del "índice de masa corporal" en Internet y encontrará más de un millón de sugerencias. En estas referencias, la fórmula es la misma, pero varía la interpretación de los resultados, dependiendo de la edad y el sexo. A continuación se muestra la interpretación más común:

IMC	Interpretación
`< 20`	Debajo del peso normal
`20-25`	Normal
`26-30`	Sobrepeso
`más de 30`	Obeso

Escriba un programa que calcule el IMC dados el peso y la estatura, e imprima un mensaje apropiado.

ENTRADA La sentencia del problema dice que la fórmula es para valores no métricos. Esto significa que el peso debe estar en libras y la estatura en pulgadas. Así, la entrada deben ser dos valores `float`: peso y estatura.

SALIDA
Mensajes para ingreso de los valores
Un mensaje basado en el IMC

ANÁLISIS Para calcular el IMC, usted lee el peso y la estatura y los inserta en la fórmula. Si eleva la estatura al cuadrado, debe incluir `<cmath>` para tener acceso a la función `pow`. Es más efectivo multiplicar la estatura por sí misma.

IMC = peso * 703/(estatura * estatura)

Si estuviera realizando el cálculo a mano, tal vez observaría si el peso o la estatura fueron negativos y lo cuestionaría. Si la semántica de sus datos indica que los valores deben ser no negativos, entonces su programa debe realizar una prueba para asegurarse de que así es. El programa debe probar cada valor y usar una variable booleana para presentar un informe de resultados. Aquí está el módulo principal para este algoritmo.

Principal **Nivel 0**

```
Obtener los datos
Probar los datos
SI los datos son correctos
   Calcular el IMC
   Imprimir mensaje que indica el estado
DE OTRO MODO
   Imprimir "datos no válidos; el peso y la estatura deben ser positivos".
```

¿Cuál de estos pasos requiere expansión? *Obtener datos*, *Probar datos* e *Imprimir mensaje que indique el estado* requieren sentencias múltiples para resolver su subproblema particular. Por otro lado, se puede traducir *Imprimir "Datos no válidos:..."* directamente en una sentencia de salida de C++. ¿Qué hay acerca del paso *Calcular IMC*? Se puede escribir como una sola sentencia de C++, pero hay otro nivel de detalle que se debe llenar, la fórmula que se empleará. Debido a que la fórmula está en un nivel inferior de detalle que el resto del módulo principal, se elige expandir *Calcular IMC* como un módulo de nivel 1.

Obtener datos **Nivel 1**

```
Solicitar el peso
Leer el peso
Solicitar la estatura
Leer la estatura
```

Probar datos

```
SI peso < 0 O estatura < 0
    Establecer en falso dataAreOK
DE OTRO MODO
    Establecer en verdadero dataAreOK
```

Calcular el IMC

```
Establecer el índice de masa corporal en peso * 703/(estatura * estatura)
```

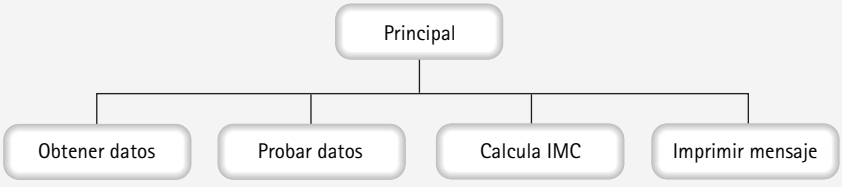
Imprimir mensaje que indica estado

El problema no dice exactamente cuál debe ser el mensaje, sólo reportar el estado. Por qué no hacer un poco cordial el resultado imprimiendo un mensaje junto con el estado.

Estado	Mensaje
Debajo del peso normal	Tome una malteada.
Normal	Tome un vaso de leche.
Sobrepeso	Tome un té helado.
Obeso	Consulte a su médico.

```
Imprimir "Su índice de masa corporal es", bodyMassIndex, '.'
Imprimir "Interpretación e instrucciones."
SI bodyMassIndex < 20
    Imprimir "Debajo del peso normal: tome una malteada."
DE OTRO MODO SI bodyMassIndex <= 25
    Imprimir "Normal: tome un vaso de leche."
DE OTRO MODO SI bodyMassIndex <= 30
    Imprimir "Sobrepeso: tome un vaso de té helado."
DE OTRO MODO
    Imprimir "Obeso: consulte a su médico."
```

Gráfica de estructura de módulo



Constantes

Nombre	Valor	Función
BMI_CONSTANT	703	Constante de fórmula

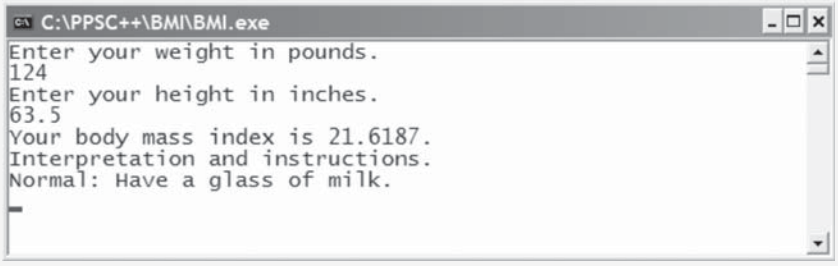
Variables

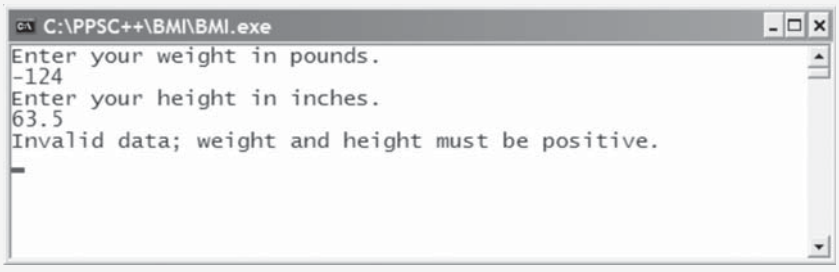
Nombre	Tipo de datos	Descripción
weight	float	Peso en libras
height	float	Estatura en pulgadas
bodyMassIndex	float	Índice de masa corporal
dataAreOK	bool	Informa si los datos son correctos

```
//*************************************************************
// Programa IMC
// Este programa calcula el índice de masa corporal (IMC) dado un peso
// en libras y una estatura en pulgadas e imprime un mensaje de salud
// con base en el IMC. Introduzca los datos en medidas inglesas.
//*************************************************************

#include <iostream>

using namespace std;

int main()
{
    const int BMI_CONSTANT = 703;  // Constante en la fórmula no métrica
    float weight;                  // Peso en peso
    float height;                  // Estatura en estatura
    float bodyMassIndex;           // IMC apropiado
    bool dataAreOK;                // Verdadero si los datos no son negativos

    // Prompt for and input weight and height
    cout << "Introduzca su peso en libras. " << endl;
    cin >> weight;
    cout << "Introduzca su estatura en pulgadas. " << endl;
    cin >> height;

    // Datos de prueba
    if (weight < 0 || height < 0)
        dataAreOK = false;
    else
        dataAreOK = true;

    if ( dataAreOK)
    {
        // Calcular el índice de masa corporal

        bodyMassIndex = weight * BMI_CONSTANT / (height * height);
```

```
        // Imprimir el mensaje que indica el estado

        cout << "Su índice de masa corporal es " << bodyMassIndex
             << ". " << endl;
        cout << "Interpretación e instrucciones. " << endl;
        if (bodyMassIndex < 20)
            cout << "Peso debajo del normal: tome una malteada. " << endl;
        else if (bodyMassIndex <= 25)
            cout << "Normal: tome un vaso de leche. "  << endl;
        else if (bodyMassIndex <=  30)
cout << "Sobrepeso: tome un vaso de té helado. "  << endl;
        else
            cout << "Obeso: consulte a su médico. " << endl;
    }
    else
        cout << "Datos no válidos; el peso y la estatura deben ser
                positivos."
             << endl;
    return 0;
}
```

Aquí se presentan las pantallas con datos buenos y malos





Las imágenes mostradas corresponden a la salida producida por el programa original, escrito en idioma inglés.

En este programa, se usa una estructura If anidada que es fácil entender aunque es un poco ineficiente. Se asigna un valor a `dataAreOK` en una sentencia antes de probarla en el texto. Se podría reducir el código al escribir

```
dataAreOK = !(weight < 0 || height < 0);
```

Si se emplea la ley de DeMorgan, esta sentencia se podría escribir también como:

```
dataAreOK = (weight >= 0 && height >= 0);
```

De hecho, se podría reducir el código aún más al eliminar la variable `dataAreOK` y usar:

```
if (weight => 0 && height >= 0)
  :
```

en lugar de

```
if (dataAreOK)
  :
```

Para convencerse usted mismo de que estas tres variaciones funcionan, inténtelas a mano con algunos datos de prueba. Si todas estas sentencias hacen lo mismo, ¿cómo elige una para usarla? Si su objetivo es la eficiencia, la variación final, la condición compuesta en la sentencia If principal, es mejor. Si su intención es tratar de expresar su código lo más claro posible, la forma más larga mostrada en el programa puede ser mejor. Las otras variaciones están en alguna parte intermedia. (Sin embargo, algunas personas encontrarían que la condición compuesta en la sentencia If principal no es sólo la más eficaz, sino la más fácil de entender.) No hay reglas absolutas para seguir aquí, pero la pauta general es esforzarse por lograr claridad, incluso si debe sacrificar un poco de eficiencia.

Prueba y depuración

En el capítulo 1 se analizaron las fases de resolución de problemas y ejecución de la programación en computadora. Probar es parte integral de ambas fases. Aquí se prueban ambas fases del proceso empleado para desarrollar el programa del IMC. La prueba en la fase de resolución del problema se hace después de que se desarrolló la solución, pero antes de ponerla en práctica. En la fase de implementación se prueba después de que el algoritmo se tradujo en un programa, y de nuevo después de que el programa se ha compilado con éxito. La compilación en sí es otra etapa de prueba que se efectúa de manera automática.

Prueba en la fase de resolución del problema: repaso del algoritmo

Determinar precondiciones y poscondiciones Para probar durante la fase de resolución del problema, se hace un *repaso* del algoritmo. Para cada módulo de la descomposición funcional, se establece una afirmación denominada precondición y otra nombrada poscondición. Una **precondición** es una afirmación que debe ser verdadera antes de ejecutar un módulo, a fin de que el módulo se ejecute de manera correcta. Una **poscondición** es una afirmación que debe ser verdadera después de que se ejecutó el módulo, si ha hecho su trabajo correctamente. Para probar un módulo, se "repasan" los pasos del algoritmo para confirmar que producen la poscondición requerida, dada la precondición estándar.

Precondición Afirmación que debe ser verdadera antes de ejecutar un módulo.

Poscondición Afirmación que debe ser verdadera después de que se ejecutó un módulo.

Nuestro algoritmo tiene cinco módulos: el módulo Principal, Obtener datos, Probar datos, Calcular el IMC e Imprimir un mensaje que indica el estado. En general, no hay precondición para el módulo principal. La poscondición del módulo principal es que produzca los resultados correctos, dada la entrada correcta. De modo más específico, la poscondición para el módulo principal es:

- La computadora ha introducido dos valores reales en `weight` y `height`.
- Si la entrada no es válida, se imprime un mensaje de error; de lo contrario, el índice de masa corporal ha sido calculado y se imprime un mensaje apropiado con base en el resultado.

Debido a que Obtener datos es el módulo principal ejecutado en el algoritmo y a que en él no se supone nada acerca del contenido de las variables que se manejarán, no tiene precondición.

Su poscondición es que `dataAreOK` contiene `true` si los valores en `weight` y `height` son no negativos; de lo contrario, `dataAreOK` contiene `false`.

La precondición para el módulo Calcular el IMC es que `weight` y `height` contienen valores significativos. Su poscondición es que la variable llamada `bodyMassIndex` contiene la evaluación de la fórmula para el IMC (peso * 703 /(estatura * estatura)).

La precondición para el módulo Imprimir mensaje que indica el estado es que `bodyMassIndex` contiene el resultado de evaluar la fórmula del IMC. Su poscondición es que se han impreso la docu-

mentación apropiada y el valor de `bodyMassIndex`, junto con los mensajes: "Peso debajo del normal: tome una malteada.", si el valor del IMC es menor que 20; "Normal: tome un vaso de leche.", si el valor es menor que o igual a 26; "Sobrepeso: tome un vaso de té helado.", si el valor es menor que o igual a 30, y " Obeso: consulte a su médico.", si el valor es mayor que 30.

A continuación se resumen las precondiciones y poscondiciones del módulo en forma tabular. En la tabla se usa *AND* con su significado usual en una afirmación, la operación lógica AND. Asimismo, una frase como "`someVariable` está asignada" es una forma abreviada de afirmar que a `someVariable` ya se le asignó un valor significativo.

Módulo	Precondición	Poscondición
Principal	—	Dos valores float han sido introducidos AND si la entrada es válida, se calcula la fórmula del IMC y se imprime el valor con un mensaje apropiado; de lo contrario, se ha impreso un mensaje de error.
Obtener datos	—	se han introducido `weight` y `height`
Probar datos	`weight` y `height` son valores asignados	`dataAreOK` contiene `true` si `weight` y `height` son no negativos; de otro modo, `dataAreOK` contiene `false`.
Calcular IMC	`weight y height` son valores asignados	`bodyMassIndex` contiene la evaluación de la fórmula del IMC.
Imprimir mensaje Indicar estado	`bodyMassIndex` contiene la evaluación de la fórmula del IMC	El valor de `bodyMassIndex` ha sido impreso, junto con un mensaje que interpreta el valor.

Efectuar el repaso del algoritmo Ahora que se han establecido las precondiciones y poscondiciones, se repasa el módulo principal. En este punto, sólo interesan los pasos del módulo principal, así que por el momento se supone que cada módulo de nivel inferior se ejecuta de modo correcto. En cada paso se deben determinar las condiciones actuales. Si el paso es una referencia a otro módulo, se debe comprobar que las condiciones actuales satisfacen la precondición de ese módulo.

Se empieza con la primera sentencia del módulo principal. Obtener datos no tiene una precondición, y se supone que Obtener datos satisface su poscondición de que introduce de manera correcta dos valores reales en `weight` y `height`.

La precondición para el módulo Probar datos es que `weight` y `height` son valores asignados. Éste debe ser el caso si la poscondición de Obtener datos es verdadera. De nuevo, debido a que sólo interesa el paso a nivel 0, se supone que Probar datos satisface su poscondición de que `dataAreOK` contiene `true` o `false`, dependiendo de los valores de entrada.

A continuación, la sentencia If comprueba si `dataAreOK` es `true`. Si es así, el algoritmo ejecuta la cláusula then. Suponiendo que Calcular IMC evalúa de manera correcta la fórmula del IMC y que Imprimir mensaje indicando el estado imprime el resultado y el mensaje apropiado (recuerde: por ahora se está suponiendo que los módulos de nivel inferior son correctos), entonces la cláusula then de la sentencia If es correcta. Si el valor en `dataAreOK` es `false`, el algoritmo ejecuta la cláusula else e imprime un mensaje de error.

Ahora se ha comprobado que el módulo principal (nivel 0) es correcto, suponiendo que los módulos del nivel 1 son correctos. El siguiente paso es examinar cada módulo en el nivel 1 y contestar esta pregunta: si se supone que los módulos (si existen) del nivel 2 son correctos, ¿este módulo de nivel 1 hace lo que se supone tiene que hacer? Simplemente se repite el proceso de repaso para cada módulo, empezando con su precondición particular. En este ejemplo, no hay módulos de nivel 2, así que deben estar completos los módulos del nivel 1.

Obtener datos lee correctamente dos valores —`weight` y `height`—, así que satisfacen su poscondición. (El siguiente refinamiento es codificar la instrucción en C++. Si está codificada o no de ma-

nera correcta, *no* es asunto de esta fase; se trata con el código cuando se efectúa la prueba en la fase de implementación.)

Probar datos comprueba si ambas variables contienen valores no negativos. La condición If usa correctamente operadores OR para combinar las expresiones relacionales de modo que si cualquiera de ellas es `true`, se ejecuta la cláusula then. Así, se asigna `false` a `dataAreOK` si alguno de los números es negativo; de otro modo, se asigna `true`. Por tanto, el módulo satisface su poscondición.

Calcular IMC evalúa la fórmula del IMC: peso * 703 /(estatura * estatura). Por consiguiente, la poscondición requerida es verdadera. ¿Pero qué pasa si el valor de la estatura es 0? Se comprobó que las entradas sean no negativas, pero se olvidó que la estatura se usa como divisor y no puede ser cero. Será necesario arreglar este problema antes de liberar este programa para uso general.

Imprimir mensaje que indique el estado produce el valor en `bodyMassIndex` con documentación apropiada. Luego compara el resultado con los estándares e imprime la interpretación apropiada. Se imprime "Peso debajo del normal: tome una malteada.", si el valor es menor que 20; "Normal: tome un vaso de leche.", si el valor es menor o igual a 26; "Sobrepeso: tome un vaso de té helado.", si el valor es menor que o igual a 30, y "Obeso: consulte a su médico.", si el valor es mayor que 30. Así, el módulo satisface su poscondición.

Una vez completado el repaso del algoritmo, se tiene que corregir cualquier discrepancia y repetir el proceso. Cuando se sabe que los módulos hacen lo que se supone tienen que hacer, se empieza a traducir el algoritmo en el lenguaje de programación.

Una poscondición estándar para cualquier programa es que el usuario ha sido notificado de datos no válidos. Se debe *validar* todo valor de entrada para el cual aplica cualquier restricción. Una sentencia If de validación de datos prueba un valor de entrada y produce un mensaje de error si el valor no es aceptable. (Se validan los datos cuando se prueban valores negativos en el programa del IMC.) El mejor lugar para validar los datos es inmediatamente después de su ingreso. Para satisfacer la poscondición de validación de datos, el algoritmo debe probar también los valores de entrada para asegurar que no son demasiado grandes o demasiado pequeños.

Con la autorización de Johny Hart and Creators Syndicate, Inc.

Prueba en la fase de implementación

Ahora que se ha hablado acerca de la prueba en la fase de resolución de problemas, se dirige la atención a la prueba en la fase de implementación. En esta fase es necesario probar en varios puntos.

Repaso del código Después de escribir el código, se debe revisar línea por línea para asegurarse de que el algoritmo se reprodujo fielmente, un proceso conocido como *repaso del código*. En una situa-

ción de programación en equipo, se pide a otros miembros del equipo que repasen el algoritmo y el código junto con usted para así realizar una doble comprobación del diseño y el código.

Seguimiento de la ejecución Se deben tomar algunos valores reales y realizar cálculos manuales de lo que debe ser el resultado mediante un *seguimiento de la ejecución* (o *seguimiento manual*). Cuando se ejecuta el programa, se pueden usar estos mismos valores como entrada y comprobar los resultados.

La computadora es un dispositivo muy literal, hace exactamente lo que se le indica, que puede ser o no lo que se desea que haga. Se trata de asegurarse de que un programa haga lo que se quiere al seguir la ejecución de las sentencias.

A continuación se usa un programa sin sentido para demostrar esta técnica. Se hace un seguimiento de los valores de las variables del programa del lado derecho. Las variables con valores indefinidos se indican con un guión. Cuando se asigna un valor a una variable, ese valor se enlista en la columna apropiada.

Sentencia	Valor de a	b	c
`const int x = 5;`			
`int main()`			
`{`			
`int a, b, c;`	—	—	—
`b = 1;`	—	1	—
`c = x + b;`	—	1	6
`a = x + 4;`	9	1	6
`a = c;`	6	1	6
`b = c;`	6	6	6
`a = a + b + c;`	18	6	6
`c = c % x;`	18	6	1
`c = c * a;`	18	6	18
`a = a % b;`	0	6	18
`cout << a << b << c;`	0	6	18
`return 0;`	0	6	18
`}`			

Ahora que se ha visto cómo funciona la técnica, se aplicará al programa del IMC. Aquí se enlista sólo el conjunto de sentencias ejecutables. Los valores de entrada son 124 y 63.5.

La cláusula then de la primera sentencia If no se ejecuta para estos datos de entrada, así que no se llena ninguna de las columnas de variables a su derecha. Se ejecuta la cláusula then de la segunda sentencia If y, por tanto, no se ejecuta la cláusula `else`. Se ejecuta la cláusula else de la tercera sentencia If, que es otra sentencia If. La cláusula then se ejecuta aquí, y se deja el resto del código sin ejecutar. Se crean siempre columnas para todas las variables, incluso si se sabe que algunas permanecerán vacías. ¿Por qué? Porque es posible que después se encuentre una referencia errónea para una variable vacía; tener una columna para la variable es un recordatorio para comprobar tal error.

Sentencia	peso	estatura	IMC	dataIsOK
`cout << "Introduzca su peso en libras. " << endl;`	—	—	—	—
`cin >> weight;`	124	—	—	—
`cout << "Introduzca su estatura en pulgadas. " << endl;`	124	—	—	—
`cin >> height;`	124	63.5	—	—
`if (weight < 0 || height < 0)`	124	63.5	—	—
`dataAreOK = false;`				
`else`				
`dataAreOK = true;`	124	63.5	—	true
`if ( dataAreOK)`	124	63.5	—	true
`{`				
`bodyMassIndex = weight * BMI_CONSTANT /`				
`(height * height);`	124	63.5	21.087	true
`cout << "Su índice de masa corporal es "`				
`<< bodyMassIndex << ". " << endl;`	124	63.5	21.087	true
`cout << "Interpretación e instrucciones. "`				
`<< endl;`	124	63.5	21.087	true
`if (bodyMassIndex < 20)`	124	63.5	21.087	true
`cout << "Peso debajo del normal: tome una malteada. "`				
`<< endl;`				
`else if (bodyMassIndex <= 25)`	124	63.5	21.087	true
`cout << "Normal: tome un vaso de leche. "`				
`<< endl;`	124	63.5	21.087	true
`else if (bodyMassIndex <= 30)`				
`cout <<`				
`"Sobrepeso: tome un vaso de té helado. "`				
`<< endl;`				
`else`				
`cout << "Obeso: consulte a su médico. "`				
`<< endl;`				
`}`				
`else`				
`cout << "Datos no válidos; el peso "`				
`<< "y la estatura deben ser positivos."`				
`<< endl;`				
`return 0;`	124	63.5	21.087	true

Cuando un programa contiene bifurcaciones, es una buena idea volver a seguir su ejecución con diferentes datos de entrada, de modo que cada rama (bifurcación) se siga por lo menos una vez. En la sección siguiente se describe cómo desarrollar conjuntos de datos que prueban cada una de las bifurcaciones de un programa.

Probar las estructuras de control de selección Para probar un programa con bifurcaciones, es necesario ejecutar cada rama por lo menos una vez y comprobar los resultados. Por ejemplo, en el programa del IMC hay cinco sentencias If-Then-else (véase la figura 5-6). Se necesita una serie de conjuntos de datos para probar las distintas bifurcaciones. Por ejemplo, los siguientes conjuntos de valores de entrada para `weight` y `height` causan que se ejecuten todas las ramas:

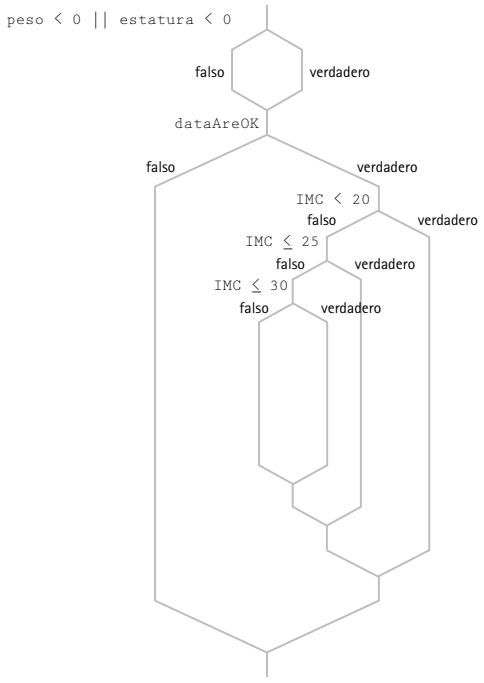


Figura 5–6 *Estructura de ramificación para el programa del IMC*

Conjunto de datos	Peso en libras	Estatura en pulgadas	Estado
1	110.0	67.5	Debajo del peso normal
2	120.0	63.0	Normal
3	145.0	62.0	Sobrepeso
4	176.6	60.0	Obeso
5	-100	65.0	Mensaje de error

En la figura 5-7 se muestra el flujo de control por la estructura de ramificación del programa IMC para cada uno de estos conjuntos de datos. Cada rama en el programa se ejecuta por lo menos una vez a través de esta serie de ejecuciones de prueba; eliminar cualquiera de los conjuntos de datos de prueba dejaría al menos una rama sin probar. Esta serie de conjuntos de datos proporciona lo que se llama *cobertura completa mínima* de la estructura de ramificación del programa. Siempre que pruebe un programa con bifurcaciones, debe diseñar una serie de pruebas que cubra todas las bifurcaciones. Podría ser de ayuda dibujar diagramas como los de la figura 5-7, de modo que pueda ver qué ramificaciones se están ejecutando.

Debido a que una acción en una rama de un programa suele afectar el proceso en una rama posterior, es necesario probar a través de un programa tantas *combinaciones de rama*, o trayectorias, como sea posible. De este modo se puede estar seguro de que no hay relaciones de elementos que pudieran causar problemas. Por supuesto, podría ser imposible seguir algunas combinaciones de rama. Por ejemplo, si se toma el `else` en la primera rama del programa del IMC, no se puede tomar el `else` de la segunda rama. ¿Se deben probar todas las trayectorias posibles? Sí, en teoría sí se debe. Sin embargo, el número de trayectorias, incluso en un programa pequeño, puede ser muy grande.

El método de prueba que se ha empleado aquí se llama *cobertura de código* porque los datos de prueba se diseñan considerando el código del programa. La cobertura de código también se llama *prueba de caja blanca* (o *caja clara*) porque se permite ver el código mientras se diseñan las pruebas. Otro método de prueba, *cobertura de datos*, intenta probar tantos datos permisibles como sea posible

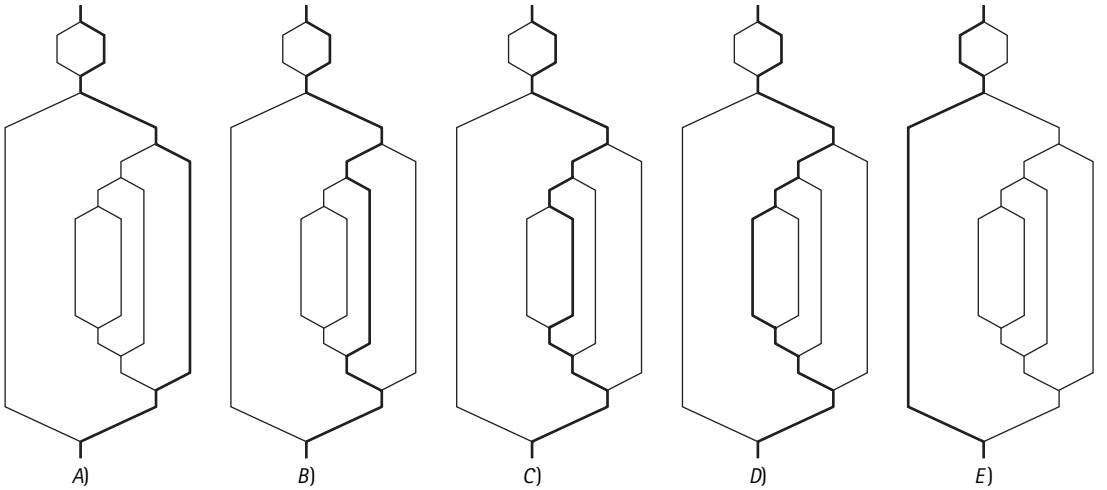


Figura 5-7 *Flujo de control por el programa IMC para cada uno de cinco conjuntos de datos*

sin considerar el código de programa. Debido a que es innecesario ver el código en esta forma de prueba, también se le conoce como *prueba de caja negra*: se diseñaría el mismo conjunto de pruebas, incluso si el código estuviera oculto en una caja negra.

La cobertura completa de datos es tan poco práctica como la de código para muchos programas. Por ejemplo, si un programa tiene cuatro valores de entrada `int`, hay aproximadamente (`2 * INT_MAX`)[4] entradas posibles. (`INT_MAX` e `INT_MIN` son constantes declaradas en el archivo de encabezado `climits`. Representan los valores `int` más grandes y más pequeños posibles, respectivamente, en su computadora y compilador de C++ particulares.)

Con frecuencia, la prueba es una combinación de estas dos estrategias. En lugar de probar cada dato (cobertura de datos), se examina el código (cobertura de código) y se buscan intervalos de valores para los cuales el proceso es idéntico. Después, se prueban los valores en las fronteras y, en ocasiones, un valor en medio de cada intervalo. Por ejemplo, una condición simple como

```
alpha < 0
```

divide a los enteros en dos intervalos:

1. `INT_MIN through -1`
2. `0 through INT_MAX`

Así, se deben probar los cuatro valores `INT_MIN`, -1, 0 e `INT_MAX`. Una condición como

```
alpha >= 0 && alpha <= 100
```

divide los enteros en tres intervalos:

1. `INT_MIN through -1`
2. `0 through 100`
3. `101 through INT_MAX`

Así, se tienen seis valores para prueba. Además, para comprobar que los operadores relacionales son correctos, se deben probar valores de 1 (> 0) y 99 (< 100).

Las bifurcaciones condicionales son sólo un factor al desarrollar una estrategia de prueba. En capítulos posteriores se consideran más de estos factores.

Plan de prueba

Hemos examinado estrategias y técnicas para probar programas, pero, ¿cómo se trata la prueba de un programa específico? Esto se hace mediante el diseño y la implementación de un **plan de prueba**, un documento que especifica los casos de prueba que se deben probar, la razón para cada caso se prueba y el resultado es el esperado. **Implementar un plan de prueba** requiere correr el programa con los datos especificados por los casos de prueba en el plan, y comprobar y registrar los resultados.

Plan de prueba Documento que especifica cómo se probará un programa.

Implantación del plan de prueba Usar los casos de prueba especificados en un plan de prueba para comprobar que un programa produce los resultados predichos.

El plan de prueba se debe desarrollar junto con la descomposición funcional. Cuando usted crea cada módulo, escribe su precondición y poscondición y observa los datos de prueba requeridos para comprobarlos. Considere la cobertura de código y la cobertura de datos para ver si ha dejado pruebas para algunos aspectos del programa (si ha olvidado algo, tal vez indica que una precondición o poscondición está incompleta).

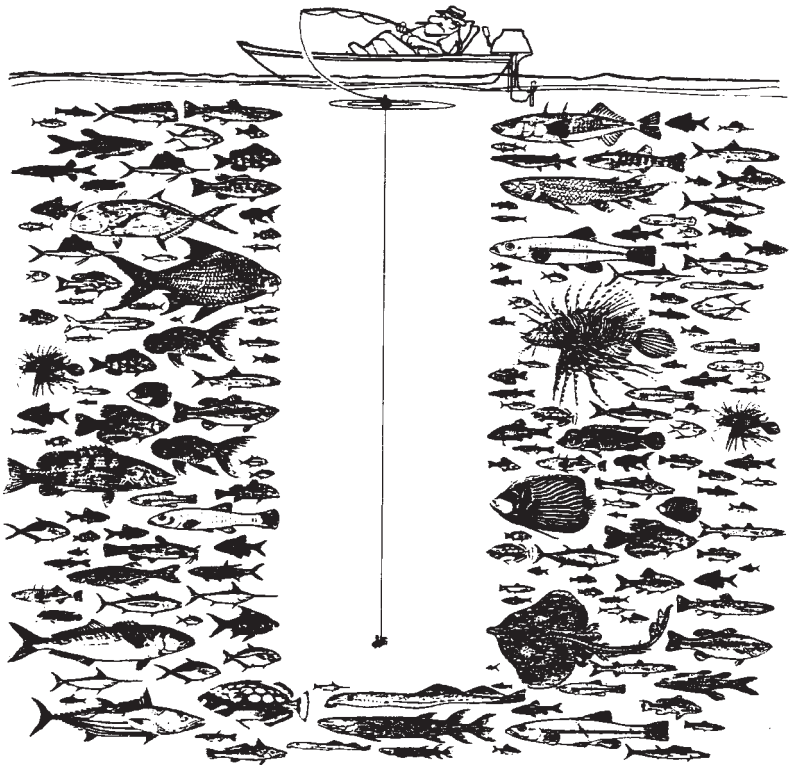
En la tabla siguiente se muestra un plan de prueba parcial para el programa IMC. Tiene seis casos de prueba. Los cuatro primeros prueban las distintas trayectorias por un programa para datos válidos. Dos casos de prueba adicionales comprueban que el peso y la estatura sean validados de manera apropiada al introducir por separado un valor inválido para cada uno.

Se debe probar el programa en casos extremos; es decir, donde el IMC es exactamente 20, 25 y 30. Debido a que se calcula el IMC sin introducir ningún valor, es difícil sugerir los valores de entrada apropiados. En los Ejercicios de seguimiento de caso práctico se pide examinar este problema, completar el plan de prueba e implementarlo.

Plan de prueba para el programa IMC

Razón para el caso de prueba	Valores de entrada	Resultado esperado	Resultado observado
Caso con peso menor al normal	110, 67.5	Su índice de masa corporal es 16.9723. Interpretación e instrucciones. Peso debajo del normal: tome una malteada.	
Caso normal	120, 63	Su índice de masa corporal es 21.2547. Interpretación e instrucciones. Normal: tome un vaso de leche.	
Caso con sobrepeso	145, 62	Su índice de masa corporal es 26.518. Interpretación e instrucciones. Sobrepeso: tome un vaso de té helado.	
Caso con obesidad	176.6, 60	Su índice de masa corporal es 34.4861. Interpretación e instrucciones. Obeso: consulte a su médico.	
Peso negativo	–120, 63	Datos no válidos; el peso y la estatura deben ser positivos.	
Estatura negativa	120, –63	Datos no válidos; el peso y la estatura deben ser positivos.	

Poner en práctica un plan de prueba no garantiza que el programa sea del todo correcto. Significa que una prueba sistemática, cuidadosa del programa no demostró ningún error. La situación mostrada en la figura 5-8 es análoga a tratar de probar un programa sin un plan; dependiendo sólo de la suerte, se podría perder por completo la posibilidad que tiene un programa de detectar numerosos errores. Por otro lado, desarrollar e implementar un plan de prueba escrito lanza una amplia red en la que es mucho más probable hallar errores.



Reimpreso con permiso de Jeff Griffin.

Figura 5-8 *Cuando prueba un programa sin un plan, nunca sabe lo que podría estar pescando*

Pruebas efectuadas automáticamente durante la compilación y ejecución

Una vez que un programa está codificado y se han preparado los datos de prueba, está listo para compilar. El compilador tiene dos responsabilidades: reportar cualquier error y (si no hay errores) traducir el programa en código de objeto.

Los errores pueden ser de sintaxis o de semántica. El compilador encuentra errores sintácticos. Por ejemplo, el compilador le advierte cuando las palabras reservadas están mal escritas, no se han declarado identificadores, faltan punto y coma, y no concuerdan los tipos de operando. Pero no encontrará todos sus errores de mecanografía. Si escribe > en lugar de <, no obtendrá un mensaje de error; en lugar de eso obtiene resultados erróneos cuando prueba el programa. Corresponde a usted diseñar un plan de prueba y comprobar minuciosamente el código para detectar errores de este tipo.

Los errores semánticos (conocidos también como *errores lógicos*) son errores que dan la respuesta equivocada. Son más difíciles de localizar que los errores sintáticos y en general surgen al ejecutar un programa. C++ detecta sólo los errores semánticos más obvios, los que dan como resultado una operación no válida (dividir entre cero, por ejemplo). Aunque los errores semánticos en ocasiones son causados por errores de mecanografía, con más frecuencia son producto de un diseño de algoritmo defectuoso. La falta de comprobación de si la estatura es 0, hallada en el repaso del algoritmo para el problema del IMC, es un error semántico común.

Revisar el algoritmo y el código, seguir la ejecución del programa y desarrollar una estrategia de prueba minuciosa permiten evitar, o por lo menos localizar con rapidez, errores semánticos en sus programas.

En la figura 5-9 se ilustra el proceso de prueba que se ha estado analizando. En la figura se muestra dónde ocurren los errores de sintaxis y semántica, y en qué fase se pueden corregir.

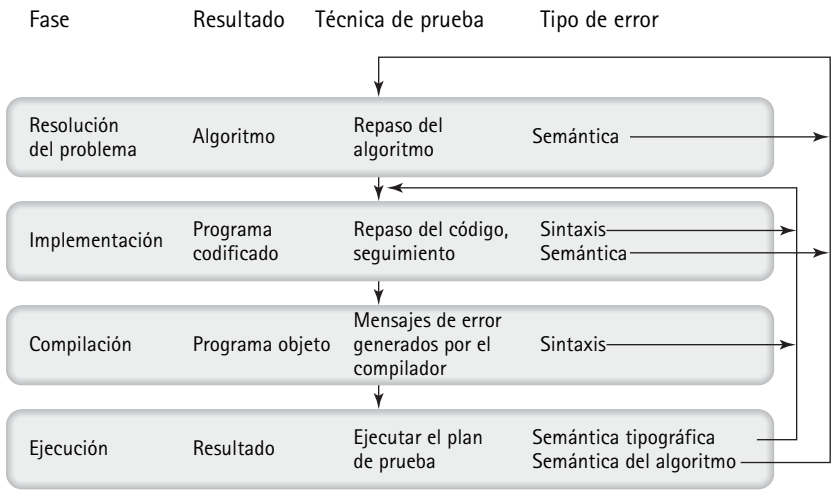
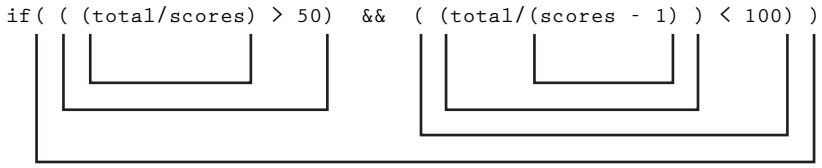


Figura 5-9 *Proceso de prueba*

Prueba y sugerencias de depurado

1. C++ tiene tres pares de operadores que son similares en apariencia, pero muy diferentes en efecto: == y =, `&&` y `&` y `||` y `|`. Realice una doble comprobación de sus expresiones lógicas para asegurarse de que está usando los operadores "igual-igual", "y-y" y "o-o".
2. Si emplea paréntesis extra para claridad, asegúrese de que correspondan los paréntesis de apertura y cierre. Para comprobar que los paréntesis tienen su pareja correspondiente, empiece con el par interno y dibuje una línea que los una. Haga lo mismo para los otros, y finalice con el par externo. Por ejemplo,



   A continuación se describe una forma rápida para saber si tiene igual número de paréntesis de apertura y cierre. El esquema emplea un solo número (el "número mágico"), cuyo valor inicialmente es 0. Explore la expresión de izquierda a derecha. En cada paréntesis de apertura, sume 1 al número mágico; en cada paréntesis de cierre, reste 1. En los paréntesis de cierre finales, el número mágico debe ser 0. Por ejemplo,

```
if (((total/scores) > 50) && ((total/(scores - 1)) < 100))
0  123            2     1    23      4          32      10
```

3. No use =< para indicar "menor que o igual a"; sólo funciona el símbolo <=. De igual manera, => no es válido para "mayor que o igual a "; se debe usar >= para esta operación.
4. En una sentencia If, recuerde usar un par de llaves `{ }` si la cláusula then o else es una secuencia de sentencias. Asegúrese de no escribir punto y coma después de la llave derecha.
5. Imprima por eco todos los datos de entrada. De este modo usted sabe que sus valores de entrada son lo que se supone tienen que ser.
6. Pruebe datos malos. Si un valor de datos debe ser positivo, use una sentencia If para probar el valor. Si el valor es negativo o cero, se debe imprimir un mensaje de error; de otro modo, el proceso debe continuar. Por ejemplo, el siguiente segmento de código prueba si tres puntuaciones de prueba son menores que 0 o mayores que 100.

```
dataOK = true;
if (test1 < 0 || test2 < 0 || test3 < 0)
{
    cout << "Datos no válidos: puntuación(es) menor que cero. "
    << endl; dataOK = false;
}
if (test1 > 100 || test2 > 100 || test3 > 100)
{
    cout << "Datos no válidos: puntuación(es) mayor que 100. "
    << endl; dataOK = false;
}
```

Estas sentencias If prueban los límites de puntuaciones razonables, y el resto del programa continúa sólo si los valores de datos son razonables.

7. Tome algunos valores de muestra y pruébelos a mano, como se hizo para el programa IMC. (En el capítulo 6 hay más acerca de este método.)
8. Si su programa lee datos de un archivo de entrada, éste debe comprobar que el archivo se abrió con éxito. Inmediatamente después de la llamada para la función `open`, una sentencia If debe probar el estado del flujo de archivos.
9. Si su programa produce una respuesta que no concuerda con el valor que calculó a mano, intente estas sugerencias:

   a) Realice de nuevo sus cálculos aritméticos.
   b) Vuelva a comprobar sus datos de entrada.
   c) Revise con cuidado la sección de código que hace el cálculo. Si duda respecto al orden en el que se ejecutan las operaciones, inserte paréntesis aclarativos.
   d) Compruebe el desbordamiento de enteros. El valor de una variable `int` pudo haber excedido `INT_MAX` a la mitad del cálculo. Algunos sistemas dan un mensaje de error cuando esto sucede, pero la gran mayoría no.
   e) Compruebe las condiciones en las sentencias de ramificación para asegurarse de que se toma la rama correcta en todas las circunstancias.

Resumen

Usar expresiones lógicas es una manera de hacer preguntas mientras un programa está en ejecución. El programa evalúa cada expresión lógica, y produce el valor `true` si la expresión es verdadera o el valor `false` si la expresión es falsa.

La sentencia If permite tomar trayectorias distintas a través de un programa basado en el valor de una expresión lógica. Se emplea If-Then-Else para elegir entre dos cursos de acción; If-Then se usa para elegir si se toma o no un curso de acción particular. Las bifurcaciones de If-Then o If-Then-Else pueden ser cualquier sentencia, simple o compuesta. Incluso pueden ser otras sentencias If.

El algoritmo de repaso requiere definir una precondición y una poscondición para cada módulo en un algoritmo. Después es necesario comprobar que esas afirmaciones son ciertas al comienzo y al final de cada módulo. Al probar el diseño en la fase de resolución de problemas, es posible eliminar errores que pueden ser más difíciles de detectar en la fase de implementación.

Un seguimiento de ejecución es una forma de hallar errores de programa una vez que se entró en la fase de implementación. Es una buena idea realizar el seguimiento de un programa antes de correrlo, de modo que se cuente con algunos resultados de muestra contra los cuales comprobar el resultado del programa. Un plan de prueba escrito es una parte esencial de cualquier esfuerzo de desarrollo de programa.

Comprobación rápida

1. ¿Cuáles son los dos valores que son la base para la lógica booleana? (p. 159)
2. ¿Qué permite una rama hacer a la computadora? (pp. 159 y 160)
3. ¿A qué propósito sirve una rama anidada? (pp. 158 y 159)
4. ¿Qué nos dice una precondición de un módulo? (pp. 186 y 188)
5. Escriba una expresión booleana que sea verdadera cuando el valor de la variable `temperature` es mayor que 32. (pp. 161-163)
6. Escriba una expresión booleana que sea verdadera cuando el valor de la variable `temperature` está en el intervalo de 33 a 211 grados y la variable `bool`, `fahrenheit`, es verdadera. (pp. 163-166)
7. Escriba una sentencia If-Then-Else que usa la prueba de la pregunta 5 para producir `"Arriba de la congelación."` o `"En el punto de congelación o debajo."` (pp. 170 y 171)
8. Escriba una sentencia If-Then que produzca "Dentro del intervalo." cuando la expresión booleana de la pregunta 6 es verdadera. (pp. 174-175)
9. Escriba sentencias If anidadas para imprimir mensajes que indican si una temperatura está debajo de la congelación, en el punto de congelación, arriba de la congelación pero no en ebullición, o en la ebullición y arriba. (pp. 176-177)
10. Al realizar el seguimiento de un programa a mano se escribe una tabla. ¿Qué se escribe en las columnas de la tabla y qué representan los renglones? (pp. 188-192)
11. Si un programa nunca parece tomar una rama de una sentencia If, ¿cómo procedería con la depuración? (pp. 192-193)

Respuestas

1. Verdadero y falso. **2.** Permite a la computadora elegir entre cursos de acción alternativos, dependiendo de una prueba de ciertas condiciones. **3.** Permite a la computadora seleccionar entre cualquier número de cursos de acción alternativos. **4.** Una precondición indica de manera precisa qué condiciones deben existir en la entrada a un módulo, para que el módulo se pueda ejecutar de modo correcto. **5.** `temperature > 32` **6.** `temperature > 212 && temperature < 121 && fahrenheit`

7.
```
if (temperature > 32)
      cout << "Arriba del punto de congelación.";
  else
      cout << "En el punto de congelación o debajo.";
```

8.
```
if (temperature > 32 && temperature < 212 && fahrenheit)
      cout << "Dentro del intervalo.";
```

9.
```
if (temperature < 32)
      cout << "Debajo del punto de congelación.";
  else if (temperature = = 32)
      cout << "Congelación.";
  else if (temperature < 212)
     cout << "Arriba del punto de congelación y debajo del punto de ebullición.";
  else
      cout << "Ebullición y arriba.";
```

10. Las columnas son los valores de las variables y los renglones corresponden al resultado de ejecutar cada sentencia durante el seguimiento. **11.** Pruebe diferentes entradas para asegurarse de que hay un problema; después compruebe que la prueba condicional está escrita correctamente. Trabajando hacia atrás desde allí, compruebe que cada variable en la prueba tiene asignado correctamente un valor. Compruebe que todas las precondiciones de cada módulo relacionado se satisfacen antes de que se ejecute.

Ejercicios de preparación para examen

1. Defina el término "flujo de control".
2. Los valores `true` y `false` son palabras clave (reservadas) en C++. ¿Verdadero o falso?
3. El operador "igual o mayor" en C++ se escribe =>. ¿Verdadero o falso?
4. ¿Por qué es que `'A' < 'B'` y `'a' < 'b'` son `true`, pero `'a' < 'B'` es `false`?

5. Si `int1` tiene el valor 12, `int2` tiene el valor 18 e `int3` tiene el valor 21, ¿cuál es el resultado de cada una de las siguientes expresiones booleanas?
   a) `int1 < int2 && int2 < int3`
   b) `int1 < int3 || int3 < int2`
   c) `int1 <= int2 - 6`
   d) `int2 <= int1 + 5 || int3 >= int2 + 5`
   e) `!(int1 < 30)`
   f) `!(int2 == int1 && int3 == int1)`
   g) `!(int1 > 25) && !(int2 < 17)`
6. Si `string1` tiene el valor `"miniscule"`, `string2` tiene el valor `"minimum"` y `string3` tiene el valor `"miniature"`, ¿cuál es el resultado de cada una de las siguientes expresiones?
   a) `string1 > string2`
   b) `string1 > string2 && string2 > string3`
   c) `string1.substr(0, 4) == string2.substr(0, 4)`
   d) `string1 > "maximum"`
   e) `string3.substr(0, 4) == "mini" || string1 == string2`
   f) `string3.length() > string1.length() && string1 > string3`
   g)
```
!((string1.substr(8, 1) == string3.substr(8, 1)) &&
   string1.length() == 9)
```
7. ¿Por qué la siguiente expresión no da como resultado un error de división entre cero cuando `someInt` tiene el valor 0?

```
someInt != 0 && 5/someInt > 10
```

8. Los operadores `bool` tienen menor precedencia que los operadores aritméticos, con excepción del operador `!`, que tiene la misma precedencia que el menos unario. ¿Verdadero o falso?
9. La expresión lógica en una sentencia If se encierra entre paréntesis sólo para hacer el código más legible. C++ no requiere los paréntesis. ¿Verdadero o falso?
10. ¿Qué hace la siguiente sentencia If cuando el valor en someInt es 77?

```
if (someInt <= 44) || (someInt - 37 < 40)
    cout << "The data is within range.";
else
    cout << "The data doesn't make sense.";
```

11. ¿Qué hace la siguiente sentencia If cuando el valor en `string1` es `"The"`?

```
if (string1.length() == 3 && string1.substr(0, 1) = "T")
    cout << "The word may be \"The\"";
else
{
    string1 = "The";
    cout << "The word is now \"The\"";
}
```

12. ¿Qué hace la siguiente sentencia If cuando el valor en `float1` es `3.15`?

```
if (fabs(float1 - 3.14) < 0.00000001)
{
    cout << "The area of the circle of radius 6.0 is
                    approximately:"
         << endl;
    cout << 6.0 * 6.0 * float1;
}
```

13. ¿Por qué la siguiente sentencia If produce siempre `"false"` sin importar el valor en `someInt`?

```
if (someInt = 0)
    cout << "true";
else
    cout << "false";
```

14. ¿Cuál es el resultado del siguiente segmento de código cuando `score` tiene el valor `85`?

```
if (score < 50)
    cout << "Failing";
else if (score < 60)
    cout << "Below average";
else if (score < 70)
    cout << "Average";
else if (score < 80)
    cout << "Above average";
else if (score < 90)
    cout << "Very good";
else if (score < 100)
    cout << "Excellent";
```

15. ¿Cuál es el resultado del siguiente segmento de código cuando `score` tiene el valor `85`?

```
if (score < 50)
    cout << "Failing";
if (score < 60)
    cout << "Below average";
if (score < 70)
    cout << "Average";
if (score < 80)
    cout << "Above average";
if (score < 90)
    cout << "Very good";
if (score < 100)
    cout << "Excellent";
```

16. ¿Cómo arregla una sentencia If anidada que tiene un else suspendido?
17. ¿Cómo escribiría una expresión booleana en una sentencia If si desea que la rama de las sentencias Then se ejecuten cuando el archivo `inData` está en estado de falla?
18. ¿Hay un límite respecto a cuán profundamente es posible anidar las sentencias If?

Ejercicios de calentamiento para programación

1. Escriba una expresión booleana que es verdadera cuando la variable `bool`, `moon`, tiene el valor `"blue"` o el valor `"Blue"`.
2. Escriba una expresión booleana que es verdadera cuando `inFile1` e `inFile2` están en estado de falla.
3. Escriba una sentencia de bifurcación que lee hacia una variable string denominada `someString`, desde un archivo nombrado `inFile` si el archivo no está en estado de falla.
4. Escriba una sentencia de bifurcación que prueba si una fecha viene antes que otra. Las fechas se almacenan como enteros que representan el mes, día y año. Así que las variables para las dos fechas se llaman `month1`, `day1`, `year1`, `month2`, `day2` y `year2`. La sentencia debe producir un mensaje apropiado dependiendo del resultado de la prueba. Por ejemplo:

```
12/21/01 viene antes que 1/27/05
```

o bien,

```
7/14/04 no viene antes que 7/14/04
```

5. Cambie la sentencia de bifurcación que escribió para el ejercicio 4 de modo que cuando la primera fecha no viene antes que la segunda, establezca la primera fecha igual a la segunda además de imprimir el mensaje.
6. Escriba una expresión booleana que sea `true` cuando algunas de las variables `bool`, `bool1` o `bool2` sea `true`, pero `false` siempre que ambas sean `true` o ninguna de ellas sea `true`.
7. Escriba una sentencia de bifurcación que pruebe si `score` está en el intervalo de 0 a 100 y que produzca un mensaje de error si no está dentro de ese intervalo.
8. Cambie la sentencia de bifurcación que escribió para el ejercicio 7 de modo que, cuando `score` está en el intervalo apropiado, sume `score` a una variable total en ejecución denominada `scoreTotal` e incremente un contador llamado `scoreCount`.
9. Escriba un segmento de código que lea un valor `int` de cada uno de dos archivos, `infile1` e `infile2`, y si ningún archivo está en el estado de falla, que escriba el valor menor de los dos en un archivo denominado `outfile` y lea otro valor del archivo que tuvo el valor menor. Si cualquiera de los archivos está en el estado de falla, entonces el valor del que no está en estado de falla se escribe en `outfile`. Si ambos archivos están en estado de falla, entonces se produce un mensaje de error para `cout`. Los valores `int` se pueden introducir en variables `value1` y `value2`.
10. Cambie la siguiente serie de sentencias If-Then en una estructura anidada If-Else-If.

```
if (score > 100)
    cout << "Incompetente.";
if (score <= 100 && score > 80)
    cout << "Fin de semana regular.";
if (score <= 80 && score > 72)
    cout << "Jugador competitivo. ";
if (score <= 72 && score > 68)
    cout << "¡Se volvió profesional!";
if (score <= 68)
    cout << "¡Hora de participar en un torneo!";
```

11. Escriba una estructura If que produzca por lo menos tres valores, `count1`, `count2` y `count3`. Si hay más de una variable con el valor mínimo, entonces produzca el valor tantas veces como se encuentre en esas variables.
12. Se supone que el siguiente segmento de programa no imprime nada; sin embargo, produce el primer mensaje de error, `"Error en el máximo: 100"`. ¿Qué sucede y cómo se corregiría esto? ¿Por qué no produce ambos mensajes de error?

```
maximum = 75;
minimum = 25;
if (maximum = 100)
    cout << "Error en el máximo: " << maximum << endl;
if (minimum = 0)
    cout << "Error en el mínimo: " << minimum << endl;
```

13. Escriba una sentencia If que toma la raíz cuadrada de la variable `area` sólo cuando su valor es no negativo. De otro modo la sentencia debe fijar `area` igual a su valor absoluto y luego tomar la raíz cuadrada del nuevo valor. El resultado se debe asignar a la variable `root`.
14. Escriba un plan de prueba para la siguiente estructura de ramificación.

```
cout << "El agua es un ";
if (temp >= 212)
    cout << "gas.";
else if (temp > 32)
    cout << "líquido.";
else
    cout << "sólido.";
```

15. Escriba un plan de prueba para la siguiente estructura de ramificación. (Observe que la prueba para el año bisiesto dada aquí no incluye las reglas especiales para años de siglo.)

```
if (month == 2 && day > 28)
    if (year%4 != 0)
        cout << "Fecha errónea. No es un año bisiesto. "
    else
        if (day > 29)
            cout << "Fecha errónea. Día inadecuado para febrero. "
```

Problemas de programación

1. Use la descomposición funcional para escribir un programa en C++ que introduzca una letra y produzca la palabra correspondiente del alfabeto de la International Civil Aviation Organization (éstas son las palabras que usan los pilotos cuando necesitan deletrear algo en un canal de radio con interferencia). El alfabeto es como sigue:

   A Alpha
   B Bravo
   C Charlie
   D Delta
   E Echo
   F Foxtrot
   G Golf
   H Hotel
   I India
   J Juliet
   K Kilo
   L Lima
   M Mike
   N November
   O Oscar
   P Papa
   Q Quebec
   R Romeo
   S Sierra
   T Tango
   U Uniform
   V Victor
   W Whiskey
   X X-ray
   Y Yankee
   Z Zulu

   Asegúrese de usar el formato y los comentarios apropiados en su código. Proporcione los mensajes apropiados al usuario. El resultado debe marcarse con claridad y tener un formato nítido.
2. Use la descomposición funcional para escribir un programa en C++ que pida al usuario que introduzca su peso y el nombre de un planeta. El programa produce entonces la cifra del peso del usuario en ese planeta. En la tabla siguiente se proporciona el factor por el que se debe multiplicar el peso para cada planeta. El programa debe producir un mensaje de error si el usuario escribe mal el nombre del planeta. El indicador y el mensaje de error deben aclarar al usuario cómo se debe introducir el nombre de un planeta. Asegúrese de usar el formato y los comentarios apropiados en su código. El resultado debe ser marcado con claridad y tener un formato nítido.

Mercurio	0.4155
Venus	0.8975
Tierra	1.0
Luna	0.166
Marte	0.3507
Júpiter	2.5374
Saturno	1.0677
Urano	0.8947
Neptuno	1.1794
Plutón	0.0899

3. Use la descomposición funcional para escribir un programa en C++ que tome un número en el intervalo de 1 a 12 como entrada y produzca el mes correspondiente del año, donde 1 es enero, y así sucesivamente. El programa debe producir un mensaje de error si el número introducido no está en el intervalo requerido. El indicador y el mensaje de error deben aclarar al usuario cómo se debe introducir un número de mes. Asegúrese de usar el formato y los comentarios apropiados en su código. El resultado debe marcarse con claridad y tener un formato nítido.
4. Use la descomposición funcional para escribir un programa en C++ que tome un número en el intervalo de 0 a 6 y un segundo número en el intervalo de 1 a 366 como entrada. El primer número representa el día de la semana en la cual comienza el año, donde 0 es domingo, etcétera. El segundo número indica el día del año. El programa produce entonces el nombre del día de la semana que corresponde al día del año. El número del día de la semana se puede calcular como sigue:

   (día de inicio + día del año – 1)%7

   El programa debe producir un mensaje de error si los números introducidos no están en los intervalos requeridos. El indicador y el mensaje de error deben aclarar al usuario cómo se introducen los números. Asegúrese de usar el formato y comentarios apropiados en su código. El resultado se debe marcar con claridad y tener un formato nítido.
5. Use la descomposición funcional para escribir un programa en C++ que tome un número en el intervalo de 1 a 365 como entrada. El número representa el día del año. El programa produce después el nombre del mes (suponga que el año no es bisiesto). Esto se puede hacer comparando el día del año con el número de días en el año que preceden el inicio de cada mes. Por ejemplo, 59 días preceden a marzo, que tiene 31 días. Así que si el día del año está en el intervalo de 60 a 91, entonces su programa produciría marzo. El programa debe producir un mensaje de error si el número introducido no está en el intervalo requerido. El indicador y el mensaje de error deben aclarar al usuario cómo se debe introducir el número. Asegúrese de usar el formato y comentarios apropiados en su código. El resultado se debe marcar con claridad y tener un formato nítido.
6. Use la descomposición funcional para escribir un programa en C++ que tome como entrada tres números que representan el número de pinos que tira un jugador de boliche en tres lanzamientos. Las reglas del boliche son que si el primer lanzamiento es una chuza (diez bolos tirados), entonces la puntuación es igual a esos 10 puntos más el número de pinos tirados en los siguientes dos lanzamientos. La puntuación máxima (tres chuzas) es, por tanto, treinta. Si en el primer lanzamiento son derribados menos de diez pinos, pero en el segundo cae el resto de los pinos (un *blow*), entonces la puntuación es sólo el número total de pinos tirados en los primeros dos lanzamientos. Su programa debe producir la puntuación calculada, y también debe comprobar una introducción errónea. Por ejemplo, un lanzamiento puede estar en el intervalo de 0 a 10 pinos, y el total de los dos primeros lanzamientos debe ser menor o igual a 10, excepto cuando el primer lanzamiento es una chuza. Asegúrese de usar el formato y comentarios apropiados en su código. El resultado se debe marcar con claridad y tener un formato nítido, y los mensajes de error deben ser informativos.
7. Use la descomposición funcional para escribir un programa en C++ que calcule la puntuación de una competencia de danza. Hay cuatro jueces que califican a los bailarines en el intervalo de 0

a 10, y la puntuación global es el promedio de las tres puntuaciones más altas (se excluye la puntuación mínima). Su programa debe producir un mensaje de error, en lugar del promedio, si alguna de las puntuaciones no está en el intervalo correcto. Asegúrese de usar el formato y comentarios apropiados en su código. El resultado se debe marcar con claridad y tener un formato nítido, y el mensaje de error debe indicar de modo claro qué puntuación no fue válida.

8. Use la descomposición funcional para escribir un programa en C++ que determine la mediana de tres números de entrada. La mediana es el número medio cuando los tres están dispuestos en orden. Sin embargo, el usuario puede introducir valores en cualquier orden, de modo que su programa debe determinar cuál valor está entre los otros dos. Por ejemplo, si el usuario introduce

```
41.52 27.18 96.03
```

entonces el programa produciría:

```
La mediana de 41.52, 27.18 y 96.03 es 41.52.
```

Una vez que funcione el caso de tres números, amplíe el programa para manejar cinco números. Asegúrese de usar el formato y comentarios apropiados en su código. El resultado se debe marcar con claridad y tener un formato nítido.

Seguimiento de caso práctico

1. ¿Cómo podría elegir valores para el peso y la estatura de modo que los valores del IMC de 20, 25 y 30 pudieran probarse?
2. Cambie el programa de modo que se redondee el IMC. Elija casos apropiados para probar las condiciones extremas. Ponga en práctica su plan de prueba.
3. Reescriba este programa para tomar valores métricos del peso y la estatura. Recurra a Internet para hallar la fórmula correcta.
4. Cambie el programa original de modo que la estatura se introduzca en pies y pulgadas. Mediante una indicación haga saber que se introduzcan por separado.
5. En el programa del IMC se consideró como una condición de error un valor de entrada negativo para el peso y la estatura. ¿Hay otros valores para el peso o la estatura que deban ser considerados como condiciones de error? Explique.

CAPÍTULO

6

Ciclos

Objetivos

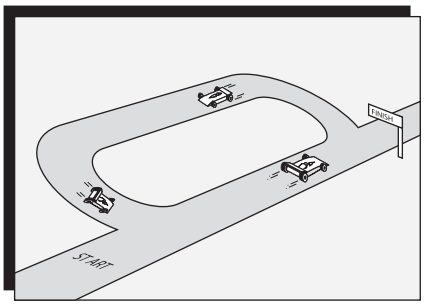
Objetivos de conocimiento

- *Entender el flujo de control en un ciclo.*
- *Entender las diferencias entre ciclos controlados por conteo, controlados por suceso y controlados por bandera.*
- *Entender las operaciones de conteo y suma dentro de un ciclo.*
- *Saber cómo elegir el tipo correcto de ciclo para un problema dado.*

Objetivos de habilidades

Ser capaz de:

- *Construir ciclos While correctos desde el punto de vista sintáctico.*
- *Construir ciclos controlados por conteo, controlados por suceso y controlados por bandera por medio de la sentencia While.*
- *Usar la condición final de archivo para controlar el ingreso de datos.*
- *Construir ciclos de conteo y suma con una sentencia While.*
- *Construir ciclos While anidados.*
- *Elegir conjuntos de datos que prueban un programa de iteración de un modo exhaustivo.*



En el capítulo 5 se dijo que el flujo de control en un programa puede diferir del orden físico de las sentencias. El *orden físico* es el orden en el cual aparecen las sentencias en un programa; el orden en el que se desea que se ejecuten las sentencias se llama *orden lógico*.

Ciclo Estructura de control que causa que una sentencia o grupo de sentencias se ejecute de manera repetida.

La sentencia If es una forma de hacer el orden lógico distinto del orden físico. Las estructuras de control de ciclo son otras. Un **ciclo** ejecuta la misma sentencia (simple o compuesta) una y otra vez, siempre que se satisfaga una condición o conjunto de condiciones.

En este capítulo se estudian distintas clases de ciclos y cómo se construyen por medio de la sentencia While. Asimismo, se analizan los *ciclos anidados* (ciclos que contienen otros ciclos) y se introduce una notación para comparar la cantidad de trabajo realizado por diferentes algoritmos.

6.1 La sentencia While

La sentencia While, como la sentencia If, prueba una condición. En seguida se muestra la plantilla de sintaxis para la sentencia While:

WhileStatement

```
while ( Expression )
    Statement
```

y éste es un ejemplo de una:

```
while (inputVal != 25)
    cin >> inputVal;
```

La sentencia While es una estructura de control de ciclo. La sentencia que se ejecutará cada vez por el ciclo se llama *cuerpo* del ciclo. En el ejemplo anterior, el cuerpo del ciclo es la sentencia de entrada que lee en un valor para `inputVal`. Esta sentencia While dice que se ejecute el cuerpo de manera repetida siempre que el valor no sea igual a 25. La sentencia While se completa (y, por tanto, se detiene) cuando `inputVal` es igual a 25. El efecto de este ciclo, entonces, es consumir e ignorar los valores en el flujo de entrada hasta que se lea el número 25.

Del mismo modo que la condición en una sentencia If, la condición en una sentencia While puede ser una expresión de cualquier tipo de datos simple. Casi siempre es una expresión lógica (booleana); si no, su valor se coerciona implícitamente al tipo `bool` (recuerde que un valor cero se coerciona a `false`, y cualquier valor distinto de cero se coerciona a `true`). La sentencia While dice, "Si el valor de la expresión es `true`, ejecuta el cuerpo y luego regresa y prueba de nuevo la expresión. Si el valor de la expresión es `false`, omite el cuerpo." Así, el cuerpo del ciclo se ejecuta una y otra vez siempre que la expresión sea `true` cuando se prueba. Cuando la expresión es `false`, el programa

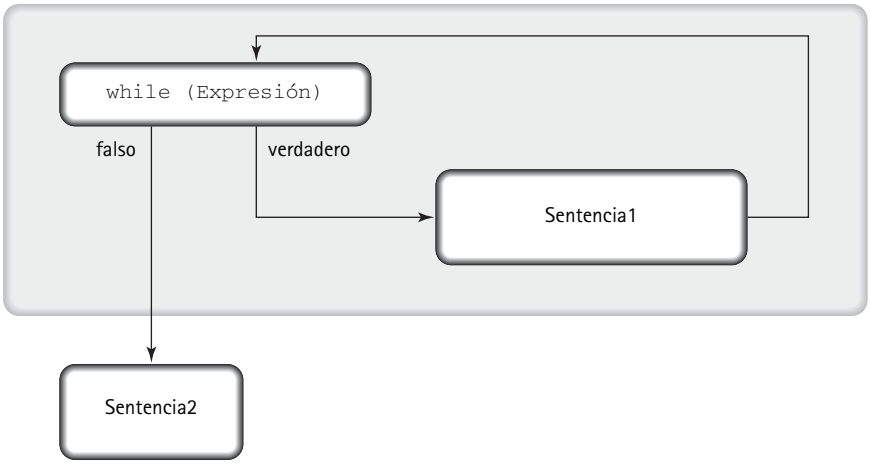


Figura 6-1 *Flujo de control de la sentencia Whil*

omite el cuerpo y la ejecución continúa en la sentencia inmediatamente después del ciclo. Por supuesto, si para empezar la expresión es `false`, el cuerpo no se ejecuta. En la figura 6-1 se muestra el flujo de control de la sentencia While, donde Sentencia1 es el cuerpo del ciclo y Sentencia2 es la sentencia después del ciclo.

El cuerpo de un ciclo puede ser una sentencia compuesta (bloque), que permite ejecutar cualquier grupo de sentencias de modo repetido. Con frecuencia se emplean ciclos While en la siguiente forma:

```
while (Expression)
{
  ⋮
}
```

En esta estructura, si la expresión es `true`, se ejecuta toda la secuencia de sentencias en el bloque, y luego la expresión se comprueba de nuevo. Si aún es `true`, las sentencias se ejecutan de nuevo. El ciclo continúa hasta que la expresión se vuelve `false`.

Aunque en algunas formas las sentencias If y While son parecidas, hay diferencias fundamentales entre ellas (véase la figura 6-2). En la estructura If, Sentencia1 se omite o se ejecuta exactamente una vez. En la estructura While, Sentencia1 se puede omitir, ejecutar una vez o ejecutar una y otra vez. La estructura If se usa para *elegir* un curso de acción; la While se usa para *repetir* un curso de acción.

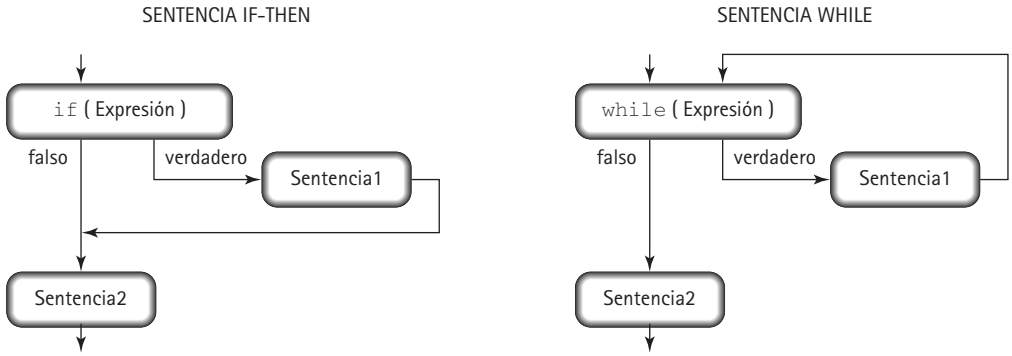


Figura 6-2 *Una comparación de If y While*

6.2 Fases de ejecución del ciclo

El cuerpo de un ciclo se ejecuta en varias fases:

Entrada del ciclo Punto en el cual el flujo de control alcanza la primera sentencia dentro de un ciclo.

Iteración Paso por, o repetición de, el cuerpo del ciclo.

Prueba del ciclo Punto en el cual se evalúa la expresión While y se toma la decisión, ya sea para comenzar una nueva iteración o pasar a la sentencia inmediatamente después del ciclo.

Salida del ciclo Punto en el cual termina la repetición del cuerpo del ciclo y el control pasa a la primera sentencia después del ciclo.

Condición de terminación Condición mediante la cual se sale de un ciclo.

- El momento en que el flujo llega a la primera sentencia dentro del cuerpo del ciclo es la **entrada del ciclo**.
- Cada vez que se ejecuta el cuerpo del ciclo, se realiza un paso por el ciclo. Este paso se llama **iteración**.
- Antes de cada iteración, el control se transfiere a la **prueba del ciclo** al comienzo del ciclo.
- Cuando se completa la última iteración y el flujo de control ha pasado a la primera sentencia después del ciclo, el programa ha **salido del ciclo**. La condición que causa que se salga de un ciclo es la condición de terminación. En el caso de un ciclo While, la **condición de terminación** es que la expresión While se vuelva `false`.

Observe que la salida de ciclo ocurre sólo en un punto: cuando se efectúa la prueba de ciclo. Aun cuando se puede satisfacer la condición de terminación a la mitad de la ejecución del ciclo, la iteración actual se completa antes de que la computadora compruebe de nuevo la expresión.

El concepto de iteración es fundamental para la programación. En este capítulo se dedica cierto tiempo a examinar las clases comunes de ciclos y formas de ponerlos en práctica con la sentencia While. Estas situaciones de iteración se presentan una y otra vez al analizar problemas y algoritmos de diseño.

6.3 Ciclos con la sentencia While

Ciclo controlado por conteo Un ciclo que se ejecuta un número específico de veces.

Ciclo controlado por suceso Un ciclo que termina cuando sucede algo dentro del cuerpo del bucle para señalar que éste debe terminar.

En la resolución de problemas se hallarán dos tipos principales de ciclos: **ciclos controlados por conteo**, que se repiten un número específico de veces, y **ciclos controlados por suceso**, que se repiten hasta que algo sucede dentro del ciclo.

Si está preparando un pastel y la receta dice "Bata la mezcla 300 veces", usted está ejecutando un ciclo controlado por conteo. Si está preparando una base para pay y la receta dice "Corte con una mezcladora de repostería hasta que la mezcla adquiera una apariencia áspera", está ejecutando un ciclo controlado por suceso; no conoce por adelantado el tiempo del número exacto de repeticiones del ciclo.

Ciclos controlados por conteo

Un ciclo controlado por conteo emplea una variable a la que se denomina *variable de control del ciclo* en la prueba de ciclo. Antes de entrar a un ciclo controlado por conteo, es necesario *inicializar* (establecer el valor inicial de) la variable de control del ciclo y después probarla. Luego, como parte de cada iteración del ciclo, se debe *incrementar* (en 1) la variable de control del ciclo. El siguiente es un ejemplo de un programa que produce repetidamente "¡Hola!" en la pantalla:

```
//*****************************************************************
// Programa Hola
// Este programa demuestra un ciclo controlado por conteo
//*****************************************************************
#include <iostream>
```

```
using namespace std;

int main()
{
    int loopCount;   // Variable de control de ciclo

    loopCount = 1;                    // Inicialización
    while (loopCount <= 10)           // Prueba
    {
        cout << "¡Hola!" << endl;
        loopCount = loopCount + 1;    // Incremento
    }
    return 0;
}
```

En el programa Hola, `loopCount` es la variable de control del ciclo. Se establece en 1 antes de la entrada del ciclo. La sentencia While prueba la expresión

```
loopCount <= 10
```

y ejecuta el cuerpo del ciclo siempre que la expresión sea `true`. Dentro del cuerpo del ciclo, la acción principal que se desea repetir es la sentencia de salida. La última sentencia en el cuerpo del ciclo incrementa `loopCount` al sumarle 1.

Examine la sentencia en la cual se incrementa la variable de control del ciclo. Observe su forma:

```
variable = variable + 1;
```

Esta sentencia suma 1 al valor actual de la variable, y el resultado reemplaza la antigua variable. Las variables que se emplean de esta manera se denominan *contadores*. En el programa Hola, `loopCount` se incrementa con cada iteración del ciclo; se emplea para contar las iteraciones. La variable de control del ciclo de un ciclo controlado por conteo es siempre un contador.

Se ha encontrado otra forma de incrementar una variable en C++. El operador de incremento (++) aumenta la variable que es su operando. La sentencia

```
loopCount++;
```

tiene precisamente el mismo efecto que la sentencia de asignación

```
loopCount = loopCount + 1;
```

De aquí en adelante se emplea de ordinario el operador ++, como lo hacen la mayoría de los programadores de C++.

Al diseñar ciclos, es responsabilidad del programador ver que la condición por probar se establece (inicializa) correctamente antes de que comience la sentencia While. El programador debe asegurarse también de que la condición cambia dentro del ciclo, de manera que finalmente se vuelva `false`; de otro modo, nunca se sale del ciclo.

```
loopCount = 1;                          ←Se debe inicializar la variable loopCount
while (loopCount <= 10)
{
        ⋮
    loopCount++;                        ←Se debe incrementar loopCount
}
```

Un ciclo que nunca termina se llama ciclo infinito porque, en teoría, el ciclo se ejecuta por siempre. En el código anterior, omitir el incremento de `loopCount` en el fondo del ciclo origina un ciclo

infinito; la expresión While siempre es `true` porque el valor de `loopCount` es por siempre 1. Si su programa continúa ejecutándose más de lo esperado, es posible que haya creado un ciclo infinito. Tal vez sea necesario recurrir a un comando del sistema operativo para detener el programa.

¿Cuántas veces se ejecuta el ciclo del programa Hola, 9 o 10? Para determinar esto, es necesario examinar el valor inicial de la variable de control del ciclo y luego probar cuál es su valor final. Aquí se ha inicializado `loopCount` en 1, y la prueba indica que el cuerpo del ciclo se ejecuta para cada valor de `loopCount` hasta 10. Si `loopCount` comienza en 1 y llega hasta 10, el cuerpo del ciclo se ejecuta 10 veces. Si se desea que el ciclo se ejecute 11 veces, es necesario inicializar `loopCount` en 0 o cambiar la prueba a

```
loopCount <= 11
```

Ciclos controlados por suceso

Hay varias clases de ciclos controlados por suceso: controlados por centinela, controlados por final de archivo, y controlados por bandera. En todos estos ciclos, la condición de terminación depende de algún suceso que ocurra mientras el cuerpo del ciclo se está ejecutando.

Ciclos controlados por centinela Los ciclos se emplean con frecuencia para leer y procesar listas largas de datos. Cada vez que se ejecuta el cuerpo del ciclo, se lee y procesa un nuevo dato. Es común usar un valor de datos especial, llamado *centinela* o *valor final* para indicar al programa que no hay más datos qué procesar. La iteración continúa siempre y cuando el valor leído *no* sea el centinela; el ciclo se detiene cuando el programa reconoce al centinela. En otras palabras, leer el valor centinela es el suceso que controla el proceso de iteración.

Un valor centinela debe ser algo que nunca se muestra en la entrada normal de un programa. Por ejemplo, si un programa lee fechas de calendario, se podría usar el 31 de febrero como valor centinela:

```
// Este código es incorrecto:

while ( !(month == 2 && day == 31) )
{
    cin >> month >> day;                // Obtener una fecha
     ⋮                                  // Procesarla
}
```

Hay un problema en el ciclo del ejemplo anterior. Los valores de month y day no están definidos antes del primer paso por el ciclo. De algún modo es necesario inicializar estas variables. Se podría asignar valores arbitrarios, pero entonces se correría el riesgo de que los primeros valores introducidos sean valores centinela, que entonces se procesarían como datos. Asimismo, es ineficiente inicializar variables con valores que nunca son usados.

El problema se resuelve leyendo el primer conjunto de datos *antes* de entrar al ciclo. Esto se llama *lectura principal*. (La idea es similar a cebar una bomba introduciéndole agua en el mecanismo antes de encenderla.) Sumemos la lectura principal al ciclo:

```
// Esto aún es incorrecto:

cin >> month >> day;                  // Obtener una fecha--lectura principal
while ( !(month == 2 && day == 31) )
{
    cin >> month >> day;              // Obtener una fecha
     ⋮                                // Procesarla
}
```

Con la lectura principal, si los primeros valores introducidos son valores centinela, entonces el ciclo no los procesa correctamente. Se ha resuelto un problema, pero se tiene otro cuando los primeros valores introducidos son datos válidos. Observe que lo primero que hace el programa dentro del ciclo es obtener una fecha y destruir los valores obtenidos en la lectura principal. Así, la primera fecha en la lista de datos nunca se procesa. Dada la lectura principal, lo *primero* que el ciclo debe hacer es procesar los datos que ya se han leído. ¿Pero en qué punto se lee el siguiente conjunto de datos? Esto se hace *al final* del ciclo. De esta manera, la condición While se aplica al siguiente conjunto de datos antes de procesarse. A continuación se ilustra cuál es su apariencia:

```
// Esta versión es correcta:

cin >> month >> day;                  // Obtener una fecha--lectura principal
while ( !(month == 2 && day == 31) )
{
     ⋮                                // Procesarla
    cin >> month >> day;              // Obtener la siguiente fecha
}
```

Este segmento funciona bien. Se lee el primer conjunto; si no es el centinela, se procesa. Al final del ciclo se lee el siguiente conjunto de datos y se vuelve al comienzo del ciclo. Si el nuevo conjunto de datos no es el centinela, se procesa igual que el primero. Cuando se lee el valor del centinela, la expresión While se vuelve `false` y se sale del ciclo (*sin* procesar el centinela).

Muchas veces el problema dicta el valor del centinela. Por ejemplo, si el problema no permite valores de datos de 0, entonces el valor centinela debe ser 0. Algunas veces no es válida una combinación de valores. La combinación de febrero y 31 como fecha es un caso de este tipo. En ocasiones un intervalo de valores (números negativos, por ejemplo) es el centinela. Y cuando se procesan datos `char` una línea de entrada a la vez, el carácter de nueva línea `('\n')` suele servir como centinela. A continuación se muestra un programa que lee e imprime todos los caracteres de una línea de un archivo de entrada:

```
//*****************************************************************
// Programa EchoLine
// Este programa lee y repite los caracteres de una línea
// de un archivo de entrada
//*****************************************************************
#include <iostream>
#include <fstream>                    // Para archivo I/O

using namespace std;

int main()
{
    char     inChar;                  // Un carácter de entrada
    ifstream inFile;                  // Archivo de datos

    inFile.open("text.dat");          // Intentar abrir el archivo de entrada
    if ( !inFile )                    // ¿Se abrió?
    {
        cout << "No puedo obtener el archivo de entrada.";
                                                // No--imprimir mensaje
        return 1;                               // Terminar programa
    }
```

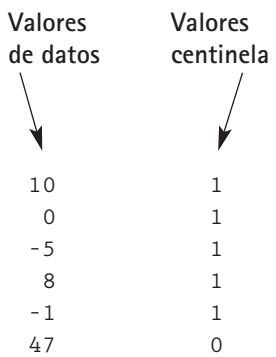
```
    inFile.get(inChar);                // Obtener el primer carácter
    while (inChar != '\n')
    {
        cout << inChar;                // Repítelo
        inFile.get(inChar);            // Obtener el siguiente carácter
    }
    cout << endl;
    return 0;
}
```

(Observe que para esta tarea particular se usa la función `get`, no el operador `>>`, para ingresar un carácter. Recuerde que el operador `>>` omite caracteres de espacio en blanco, incluso caracteres de espaciado y nuevas líneas, para hallar el valor de datos siguiente en el flujo de entrada. En este programa se desea introducir *todo* carácter, incluso un carácter de espaciado y en particular el carácter de nueva línea.)

Al elegir un valor para usar como centinela, ¿qué sucede si ninguno de los valores de datos es válido? Entonces es posible que tenga que introducir un valor extra en cada iteración, un valor cuyo único propósito sea señalar el fin de los datos. Por ejemplo, considere este segmento de código:

```
cin >> dataValue >> sentinel;        // Obtener el primer valor de datos
while (sentinel == 1)
{
          ⋮                          // Procesarlo
    cin >> dataValue >> sentinel;    // Obtener el siguiente valor de datos
}
```

El segundo valor en cada línea del siguiente conjunto de datos se usa para indicar si hay más datos o no. En este conjunto de datos, cuando el valor centinela es 0, no hay más datos; cuando es 1, hay más datos.



¿Qué sucede si olvidó meter el valor centinela? En un programa interactivo, el ciclo se ejecuta de nuevo y solicita una entrada. En ese punto, se puede introducir el valor centinela, pero la lógica de su programa puede ser equivocada si ya introdujo lo que consideró fue el valor centinela. Si la entrada para el programa viene de un archivo, una vez que se han leído los datos del archivo, el cuerpo del ciclo se ejecuta de nuevo. Sin embargo, no queda ningún dato, porque la computadora ha llegado al final del archivo, de modo que el flujo de archivos entra en estado de falla. En la siguiente sección se describe una forma de usar la situación de final de archivo como una alternativa al uso de un centinela.

Antes de continuar, se menciona una cuestión que no tiene relación con el diseño de ciclos, sino con el uso del lenguaje C++. En el capítulo 5, se habló del error común de usar el operador de asignación (`=`) en lugar del operador relacional (`==`) en una condición If. Este mismo error puede suceder cuando escribe sentencias While. Vea lo que sucede cuando se usa el operador erróneo en el ejemplo previo:

```
cin >> dataValue >> sentinel;
while (sentinel = 1)                    // ¡Vaya!
{
    ⋮
    cin >> dataValue >> sentinel;
}
```

Este error crea un ciclo infinito. La expresión While es ahora una expresión de asignación, no relacional. El valor de la expresión 1 (interpretado en la prueba de ciclo como `true` porque no es cero), y su efecto secundario es almacenar el valor 1 en `sentinel`, remplazando el valor que se introdujo en la variable. Debido a que la expresión While siempre es `true`, el ciclo nunca se detiene.

Ciclos controlados por final de archivo Ya aprendió que un flujo de entrada (como `cin` o un flujo de archivos de entrada) entra en el estado de falla *a*) si encuentra datos de entrada inaceptables, *b*) si el programa intenta abrir un archivo de entrada inexistente, o *c*) si el programa intenta leer después del fin de un archivo de entrada. Considérese la tercera de estas posibilidades.

Después de que un programa ha leído el último dato de un archivo de entrada, la computadora está al final del archivo. En este momento, el estado del flujo es correcto. Pero si se intenta introducir un dato más, el flujo entra en estado de falla. Se puede aprovechar esta situación. Para escribir un ciclo que introduce un número desconocido de datos, se puede usar la falla del flujo de datos como una forma de centinela.

En el capítulo 5 se describe cómo probar el estado de un flujo I/O. En una expresión lógica, se usa el nombre del flujo como si fuera una variable booleana:

```
if (inFile)
    ⋮
```

En una prueba como ésta, el resultado es `true` si la operación I/O más reciente tiene éxito, o `false` si falla. En una sentencia While, probar el estado de un flujo funciona de la misma manera. Suponga que se tiene un archivo de datos que contiene valores enteros. Si `inData` es el nombre del flujo de archivos en el programa, aquí se muestra un ciclo que lee y repite los valores de datos del archivo:

```
inData >> intVal;                       // Obtener el primer valor
while (inData)                          // Mientras la entrada tiene éxito...
{
    cout << intVal << endl;             // Repetirlo
    inData >> intVal;                   // Obtener el siguiente valor
}
```

Se procede a realizar un seguimiento de este código y se supone que hay tres valores en el archivo: 10, 20 y 30. La lectura principal introduce el valor 10. La condición While es `true` porque la entrada tiene éxito. Por tanto, la computadora ejecuta el cuerpo del ciclo. Primero, el cuerpo imprime el valor 10 y luego introduce el segundo valor de datos, 20. Al volver a la prueba del ciclo, la expresión `inData` es `true` porque la entrada tuvo éxito. El cuerpo se ejecuta de nuevo, imprime el valor 20 y lee el valor 30 del archivo. De regreso a la prueba, la expresión es `true`. Incluso cuando se está al final del archivo, el estado del flujo es aún OK; la operación de entrada previa tuvo éxito. El cuerpo se ejecuta una tercera vez, imprime el valor 30 y ejecuta la sentencia de entrada. Esta vez falla la sentencia de entrada; se está intentando leer más allá del final del archivo. El flujo `inData` entra en estado de falla. Volviendo a la prueba de ciclo, el valor de la expresión es `false` y se sale del ciclo.

Cuando se escriben ciclos controlados por final de archivo como el anterior, se espera que el final del archivo sea la razón para la falla de flujo. Pero no hay que olvidar que *cualquier* error de entrada ocasiona la falla del flujo. El ciclo anterior termina, por ejemplo, si la entrada falla como resultado de caracteres nulos en los datos de entrada. Este hecho destaca de nuevo la importancia de la impresión por eco. Ayuda a comprobar que todos los datos se leyeron correctamente antes de encontrar el final de archivo (EOF).

Los ciclos controlados por final de archivo (EOF) son similares a los ciclos controlados por centinela en que el programa no sabe por adelantado cuántos datos se introducirán. En el caso de los ciclos controlados por centinela, el programa lee hasta que encuentra el valor centinela. Con los ciclos controlados por final de archivo, lee hasta que llega al final del archivo.

¿Es posible usar un ciclo controlado por final de archivo cuando se lee desde el dispositivo de entrada estándar (vía el flujo `cin`) en lugar de un archivo de datos? En muchos sistemas sí. Con el sistema operativo UNIX, se puede teclear Ctrl-D (es decir, se mantiene oprimida la tecla Ctrl y se golpea la tecla D) para indicar el final de archivo durante la entrada interactiva. Con el sistema operativo MS-DOS y el IDE CodeWarrior, las teclas de final de archivo son Ctrl-Z (o a veces Ctrl-D). Otros sistemas usan teclas similares. El siguiente es un segmento de programa que prueba el final de archivo en el flujo `cin` en UNIX:

```
cout << "Introducir un entero (o Ctrl-D para salir): ";
cin >> someInt;
while (cin)
{
    cout << someInt << " el doble es " << 2 * someInt << endl;
    cout << "Siguiente número (o Ctrl-D para salir): ";
    cin >> someInt;
}
```

Flujos controlados por bandera Una bandera es una variable booleana que se usa para controlar el flujo lógico de un programa. Se puede establecer una variable booleana en `true` antes de un ciclo While; entonces, cuando se desea detener la ejecución del ciclo, se restablece en `false`. Es decir, se puede usar la variable booleana para registrar si ha ocurrido o no el suceso que controla el proceso. Por ejemplo, el siguiente segmento de código lee y suma valores hasta que el valor de entrada es negativo. (`nonNegative` es la bandera booleana; las demás variables son del tipo `int`.)

```
sum = 0;
nonNegative = true;              // Inicializar la bandera
while (nonNegative)
{
    cin >> number;
    if (number < 0)              // Probar el valor de entrada
        nonNegative = false;     // Establecer la bandera si ocurrió el suceso
    else
        sum = sum + number;
}
```

Observe que con banderas es posible codificar ciclos controlados por centinela. De hecho, este código emplea un valor negativo como centinela.

No es necesario inicializar banderas en `true`; es posible inicializarlas en `false`. Si lo hace, debe usar el operador NOT (!) en la expresión While y restablecer la bandera en `true` cuando ocurre el suceso. Compare el segmento de código anterior con el siguiente; ambos realizan la misma tarea. (Suponga que `negative` es una variable booleana.)

```
sum = 0;
negative = false;                // Inicializar la bandera
while ( !negative )
{
    cin >> number;
    if (number < 0)              // Probar el valor de entrada
        negative = true;         // Establecer la bandera si ocurrió el suceso
    else
        sum = sum + number;
}
```

Subtareas de ciclo

Se han estado considerando formas de usar ciclos para afectar el flujo de control en programas. Pero el ciclo por sí mismo no hace nada. El cuerpo del ciclo debe realizar una tarea a fin de que el ciclo lleve a cabo algo. En esta sección se examinan tres tareas (contar, sumar y seguir la pista de un valor previo) que se emplean con frecuencia en ciclos.

Conteo Tarea común en un ciclo es seguir la pista del número de veces que se ha ejecutado el ciclo. Por ejemplo, el siguiente fragmento de programa lee y cuenta los caracteres de entrada hasta que llega a un punto. (`inChar` es de tipo `char`; `count` es de tipo `int`.) El ciclo en este ejemplo tiene una variable contadora, pero el ciclo no es controlado por conteo porque la variable se está usando como una variable de control de ciclo.

```
count = 0;                              // Inicializar el contador
cin.get(inChar);                        // Leer el primer carácter
while (inChar != '.')
{
     count++;                           // Incrementar el contador
     cin.get(inChar);                   // Obtener el siguiente carácter
}
```

El ciclo continúa hasta que se lee un punto. Después de que ha finalizado el ciclo, `count` contiene uno menos que el número de caracteres leídos. Es decir, cuenta el número de caracteres hasta, pero sin incluir, el valor centinela (el punto). Observe que si un punto es el primer carácter, no se introduce el cuerpo del ciclo y `count` contiene 0, como debe ser. Se usa una lectura inicial aquí porque el ciclo se controla por centinela.

La variable contadora de este ejemplo tiene el nombre de **contador de iteración** porque su valor es igual al número de iteraciones en el ciclo.

Contador de iteración Variable contadora que se incrementa con cada iteración de un ciclo.

Según la definición, la variable de control de ciclo de un ciclo controlado por conteo es un contador de iteración. Sin embargo, como se vio, no todos los contadores de iteración son variables de control de ciclo.

Suma Otra tarea común de ciclo es sumar un conjunto de valores de datos. En el siguiente ejemplo, observe que la operación de sumar se escribe de la misma forma, sin importar cómo se controla el ciclo.

```
sum = 0;                                // Inicializar la suma
count = 1;
while (count <= 10)
{
    cin >> number;                      // Introducir un valor
    sum = sum + number;                 // Agregar un valor a la suma
    count++;
}
```

Se inicializa `sum` en 0 antes de que empiece el ciclo, para que la primera vez que se ejecute el cuerpo del ciclo, la sentencia

```
sum = sum + number;
```

sume el valor actual de `sum` (0) a `number` para formar el nuevo valor de `sum`. Después de que se ha ejecutado todo el fragmento de código, `sum` contiene el total de los diez valores leídos, `count` contiene 11 y `number` contiene el último valor leído.

Aquí `count` se incrementa en cada iteración. Para cada nuevo valor de `count`, hay un nuevo valor para `number`. ¿Esto significa que se podría disminuir `count` en 1 e inspeccionar el valor previo de `number`? No. Una vez que se ha leído un nuevo valor en `number`, el valor previo desaparece por siempre a menos que se haya guardado en otra variable. En la siguiente sección se verá cómo hacer eso.

Considérese otro ejemplo. Se desea contar y sumar los primeros diez números impares en un conjunto de datos. Se requiere probar cada número para ver si es par o impar. (Para indagar se puede usar el operador de módulo. Si `number` % 2 es igual a 1, `number` es impar; de lo contrario, es par.) Si el valor de entrada es par, no se hace nada. Si es impar, se incrementa el contador y se agrega el valor a la suma. Se emplea una bandera para controlar el ciclo porque no es un ciclo normal controlado por conteo. En el siguiente segmento de código, todas las variables son del tipo `int`, excepto la bandera booleana, `lessThanTen`.

```
count = 0;                                  // Inicializar el contador de sucesos
sum = 0;                                    // Inicializar la suma
lessThanTen = true;                         // Inicializar la bandera de control
                                               de ciclo

while (lessThanTen)
{
    cin >> number;                          // Obtener el siguiente valor
    if (number % 2 == 1)                    // ¿El valor es impar?
    {
        count++;                            // Sí—Incrementa el contador
        sum = sum + number;                 // Agregar un valor a la suma
        lessThanTen = (count < 10);         // Actualizar la bandera de control
                                               de ciclo
    }
```

En este ejemplo, no hay relación entre el valor de la variable contadora y el número de veces que se ejecuta el ciclo. La expresión While se pudo haber escrito de esta manera:

```
while (count < 10)
```

pero esto podría llevar al lector a pensar que el ciclo se controla por conteo en la forma normal. Así, en cambio, se controla el ciclo con la bandera `lessThanTen` para destacar que `count` se incrementa sólo si lee un número impar. El contador de este ejemplo es un **contador de suceso**; se inicializa en 0 y se incrementa sólo cuando cierto suceso ocurre. El contador del ejemplo previo fue un contador de iteración; se inicializó en 1 y se incrementó durante cada iteración del ciclo.

Contador de suceso Variable que se incremente cada vez que ocurre un suceso particular.

Seguir la pista de un valor previo Algunas veces se desea recordar el valor previo de una variable. Suponga que se desea escribir un programa que cuenta el número de operadores no igual (!=) en un archivo que contiene un programa en C++. Se puede hacer contando sólo el número de veces que aparece un signo de cierre de admiración (!) seguido de un signo igual (=) en la entrada. Una manera de hacer esto es leer en el archivo de entrada un carácter a la vez, siguiendo la pista de los dos caracteres más recientes, el valor actual y el valor previo. En cada iteración del ciclo, se lee un nuevo valor actual y el valor anterior se convierte en el valor previo. Cuando se llega al final del archivo, el ciclo termina. A continuación se presenta un programa que cuenta operadores no igual de esta manera:

```
//**************************************************************************
// Programa NotEqualCount
// Este programa cuenta las ocurrencias de "!=" en un archivo de datos
//**************************************************************************
```

```
#include <iostream>
#include <fstream>          // Para archivo I/O

using namespace std;

int main()
{
    int      count;         // Número de operadores !=
    char     prevChar;      // Último carácter leído
    char     currChar;      // Carácter leído en esta iteración de ciclo
    ifstream inFile;        // Archivo de datos

    inFile.open("myfile.dat");         // Intentar abrir el archivo de entrada
    if ( !inFile )                     // ¿Se abrió?
    {
        cout << "** No puedo abrir el archivo de entrada **"  // No--imprimir
                                                                mensaje
             << endl;
        return 1;              // Terminar programa
    }
    count = 0;                 // Inicializar el contador
    inFile.get(prevChar);      // Inicializar el valor previo
    inFile.get(currChar);      // Inicializar el valor actual
    while (inFile)             // Mientras la entrada previa tiene éxito ...
    {
        if (currChar == '=' && // Probar el suceso
            prevChar == '!')
            count++;           // Incrementar el contador
        prevChar = currChar;   // Remplazar el valor previo
                               //   con el valor actual
        inFile.get(currChar);  // Obtener el siguiente valor
    }
    cout << count << " != se encontraron operadores. " << endl;
    return 0;
}
```

Estudie este ciclo cuidadosamente. Va a ser de mucha utilidad. Hay muchos problemas en los que se debe seguir la pista del último valor leído además del valor actual.

6.4 Cómo diseñar ciclos

Una cosa es entender cómo funciona un ciclo cuando lo examina y otra diseñar un ciclo que resuelve un problema determinado. En esta sección se estudia cómo diseñar ciclos. Se puede dividir el proceso de diseño en dos tareas: diseñar el flujo de control y diseñar el proceso que tiene lugar en el ciclo. A su vez se puede descomponer cada tarea en tres fases: la tarea en sí, inicialización y actualización. Es importante especificar el estado del programa cuando sale del ciclo, debido a que un ciclo que deja en desorden variables y archivos no está bien diseñado.

Hay siete puntos distintos a considerar en el diseño de un ciclo:

1. ¿Cuál es la condición que termina el ciclo?
2. ¿Cómo se debe inicializar la condición?
3. ¿Cómo se debe actualizar la condición?
4. ¿Cuál es el proceso que se repite?

5. ¿Cómo se debe inicializar el proceso?
6. ¿Cómo se debe actualizar el proceso?
7. ¿Cuál es el estado del programa al salir del ciclo?

Estas preguntas se usan como lista de comprobación. Las primeras tres ayudan a diseñar partes del ciclo que controlan su ejecución. Las tres siguientes ayudan a diseñar el proceso dentro del ciclo. La última pregunta es un recordatorio para asegurarse de que el ciclo sale de una manera apropiada.

Diseñar el flujo de control

El paso más importante en el diseño de ciclo es decidir qué debe hacer que se detenga el ciclo. Si la condición de terminación no está bien pensada, existe la posibilidad de ciclos infinitos y otros errores. Así que aquí está la primera pregunta.

- ¿Cuál es la condición que termina el ciclo?

Esta pregunta normalmente se puede contestar al examinar con cuidado el enunciado del problema. En la tabla siguiente se listan algunos ejemplos.

Frase clave en el enunciado del problema	Condición de terminación
"Sume 365 temperaturas"	El ciclo termina cuando un contador llega a 365 (ciclo controlado por conteo).
"Procese todos los datos del archivo"	El ciclo termina cuando ocurre el final de archivo (ciclo controlado por final de archivo).
"Procese hasta que se hayan leído diez enteros"	El ciclo termina cuando han sido introducidos diez números impares (contador de suceso).
"El final de los datos se indica como una puntuación de prueba negativa"	El ciclo termina cuando se encuentra un valor de entrada negativo (ciclo controlado por centinela)

.

Ahora se necesitan sentencias que aseguren que el ciclo se inicie correctamente y sentencias que permitan que el ciclo alcance la condición de terminación. Así, es necesario hacer las dos preguntas siguientes:

- ¿Cómo se debe inicializar la condición?
- ¿Cómo se debe actualizar la condición?

Las respuestas a estas preguntas dependen del tipo de condición de terminación.

Ciclos controlados por conteo Si el ciclo se controla por conteo, se inicializa la condición al dar un valor inicial a la variable de control de ciclo. Para ciclos controlados por conteo en los que la variable de control de ciclo es también un contador de iteración, el valor inicial es por lo general 1. Si el proceso requiere que el contador se ejecute en un intervalo específico de valores, el valor inicial debe ser el valor mínimo en ese intervalo.

La condición se actualiza incrementando el valor del contador en 1 para cada iteración. (Ocasionalmente, es posible hallar un problema que requiere un contador para contar de algún valor *bajo* a un valor aún inferior. En este caso, el valor inicial es el valor mayor, y el contador se reduce en 1 para cada iteración.) Así, para ciclos controlados por conteo que usan un contador de iteración, éstas son las respuestas a las preguntas:

- Inicializar el contador de iteración en 1.
- Incrementar el contador de iteración al final de cada iteración.

Si el ciclo se controla mediante una variable que está contando un suceso dentro del ciclo, la variable de control normalmente se inicializa en 0 y se incrementa cada vez que ocurre el suceso. Para ciclos controlados por conteo que usan un contador de suceso, éstas son las respuestas a las preguntas:

- Inicializar el contador de suceso en 0.
- Incrementar el contador de suceso cada vez que ocurre el suceso.

Ciclos controlados por centinela En los ciclos controlados por centinela, una lectura principal puede ser la única inicialización necesaria. Si la fuente de entrada es un archivo en vez del teclado, también podría ser necesario abrir el archivo en preparación para lectura. Para actualizar la condición, se lee un nuevo valor al final de cada iteración. Así, para ciclos controlados por centinela, se contestan las preguntas de esta manera:

- Abrir el archivo, si es necesario, e introducir un valor antes de entrar al ciclo (lectura principal).
- Introducir un nuevo valor para proceso al final de cada iteración.

Ciclos controlados por final de archivo Esta clase de ciclos requiere la misma inicialización que los ciclos controlados por centinela. Se debe abrir el archivo, si es necesario, y efectuar una lectura principal. La actualización de la condición del ciclo sucede de modo explícito; el estado de flujo se actualiza para reflejar el éxito o fracaso cada vez que se introduce un valor. Sin embargo, si el ciclo no lee ningún dato, nunca llega al final del archivo, así que actualizar la condición de ciclo significa que el ciclo debe seguir leyendo datos.

Ciclos controlados por bandera En este tipo de ciclos la variable de bandera booleana debe inicializarse en `true` o `false` y luego actualizarse cuando cambia la condición.

- Inicializar la variable de bandera en `true` o `false` según convenga.
- Actualizar la variable de bandera tan pronto como cambie la condición.

En un ciclo controlado por bandera, la variable de bandera permanece sin cambio hasta que llega el momento de terminar el ciclo; entonces el código detecta alguna condición que se repite dentro del proceso que cambia el valor de la bandera (a través de una sentencia de asignación). Debido a que la actualización depende de lo que hace el proceso, a veces éste se tiene que diseñar antes de decidir cómo actualizar la condición.

Diseño del proceso dentro del ciclo

Una vez determinada la estructura de iteración, se pueden llenar los detalles del proceso. Al diseñar el proceso, se debe decidir primero qué se desea que haga una sola iteración. Suponga por un momento que el proceso se va a ejecutar sólo una vez. ¿Qué tareas debe efectuar el proceso?

- ¿Cuál es el proceso que se repite?

Para contestar a esta pregunta, se tiene que echar otro vistazo al enunciado del problema. La definición del problema puede requerir que el proceso sume valores de datos o mantenga una cuenta de valores de datos que satisfagan alguna prueba. Por ejemplo:

Contar el número de enteros en el archivo `howMany`.

Esta sentencia indica que el proceso a repetir es una operación de conteo.

Aquí está otro ejemplo:

Leer un valor de la bolsa para cada día hábil en una semana y calcular el precio promedio.

En este caso, parte del proceso requiere leer un valor de datos. Se tiene que concluir de nuestro conocimiento de cómo se calcula un promedio que el proceso requiere también sumar los valores de datos.

Además de contar y sumar, otro proceso de iteración común es leer datos, efectuar un cálculo y escribir el resultado. Pueden aparecer muchas otras operaciones en un proceso de iteración. Aquí se ha mencionado sólo la más simple; más adelante se consideran algunos otros procesos.

Después de determinar las operaciones por realizar si el proceso se ejecuta una vez, se diseñan las partes del proceso necesarias para que se repita correctamente. Con frecuencia se tienen que añadir algunos pasos para tomar en cuenta que el ciclo se ejecuta más de una vez. Esta parte del diseño requiere inicializar ciertas variables antes del ciclo y luego reinicializarlas o actualizarlas antes de cada iteración posterior.

- ¿Cómo se debe inicializar el proceso?
- ¿Cómo se debe actualizar el proceso?

Por ejemplo, si el proceso dentro de un ciclo requiere que se efectúen varias operaciones y sumas diferentes, cada una debe tener sus propias sentencias para inicializar variables, incrementar variables de conteo o agregar valores a sumas. Sólo trate cada cuenta u operación de suma por sí sola; es decir, primero escriba la sentencia de inicialización, y luego escriba la sentencia de incremento o suma. Una vez que ha hecho esto para una operación, continúe con la siguiente.

Salida del ciclo

Cuando ocurre la condición de terminación y el flujo de control pasa a la sentencia después del ciclo, las variables empleadas en el ciclo aún contienen valores. Y si se ha utilizado el flujo `cin`, el marcador de posición quedó en alguna posición en el flujo. O tal vez un archivo de salida tiene nuevo contenido. Si estas variables o archivos se usan después en el programa, el ciclo debe dejarlos en un estado apropiado. Así, el paso final al diseñar un ciclo es contestar a la pregunta:

- ¿Cuál es el estado del programa al salir del ciclo?

Ahora se tienen que considerar las consecuencias del diseño y realizar una doble comprobación de su validez. Por ejemplo, suponga que se ha usado un contador de sucesos y que el proceso posterior depende del número de sucesos. Es importante asegurarse (con un repaso de algoritmo) de que el valor dejado en el contador es el número exacto de sucesos, y no se ha incrementado o decrementado por 1.

Considere este segmento de código:

```
commaCount = 1;            // Este código es incorrecto
cin.get(inChar);
while (inChar != '\n')
{
    if (inChar == ',')
        commaCount++;
    cin.get(inChar);
}
cout << commaCount << endl;
```

Este ciclo lee caracteres de una línea de entrada y cuenta el número de comas en la línea. Sin embargo, cuando el ciclo termina, `commaCount` es igual al número real de comas más 1 porque el ciclo inicializa el contador de sucesos en 1 antes de tener lugar cualquier suceso. Al determinar el estado de `commaCount` en la salida del ciclo, se ha detectado un error en la inicialización. `commaCount` debe inicializarse en 0.

Diseñar ciclos correctos depende tanto de la experiencia como de la aplicación de la metodología del diseño. En este punto, tal vez quiera leer el Caso práctico de resolución de problemas, al final del capítulo, para ver cómo se aplica el proceso de diseño a un problema real.

6.5 Lógica anidada

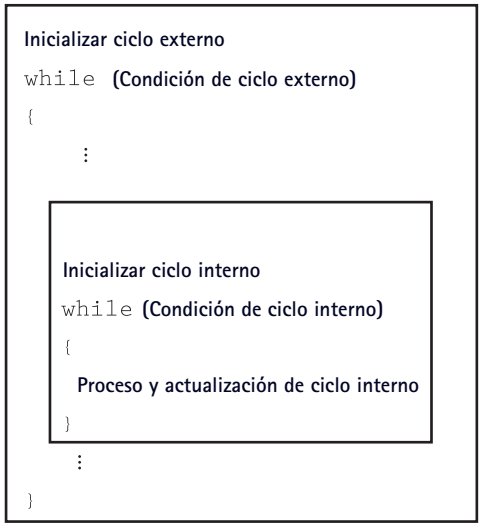
En el capítulo 5 se describieron las sentencias anidadas If. También es posible anidar sentencias While. Tanto las sentencias While como las If contienen sentencias y, por sí mismas, son sentencias. Así que el cuerpo de una sentencia While o la rama de una sentencia If pueden contener otras sentencias While e If. Mediante la anidación se pueden crear estructuras de control complejas.

Suponga que se desea ampliar el código para contar comas en una línea, repitiéndolo para todas las líneas de un archivo. Se escribe un ciclo controlado por final de archivo alrededor de él:

```
cin.get(inChar);                // Inicializar el ciclo externo
while (cin)                     // Prueba de ciclo externo
{
    commaCount = 0;             // Inicializar el ciclo interno
                                //   (La lectura principal del ciclo externo
                                //    se encarga de la lectura principal)
    while (inChar != '\n')      // Prueba de ciclo interno
    {
        if (inChar == ',')
            commaCount++;
        cin.get(inChar);        // Actualizar la condición de terminación
                                   interna
    }
    cout << commaCount << endl;
    cin.get(inChar);            // Actualizar la condición de terminación
                                   externa
}
```

En este código, observe que se ha omitido la lectura principal para el ciclo interno. La lectura principal para el ciclo externo ya ha "preparado el asunto". Sería un error incluir otra lectura principal justo antes del ciclo interno; el carácter leído por la lectura principal externa se destruiría antes de que se pudiera probar.

Examinemos el patrón general de un ciclo anidado simple. Los puntos representan lugares donde el proceso y actualización pueden tener lugar en el ciclo externo.



Observe que cada ciclo tiene su propia inicialización, prueba y actualización. Es posible que un ciclo externo no realice otro proceso que ejecutar repetidamente el ciclo interno. Por otro lado, el

ciclo interno podría ser sólo una pequeña parte del proceso hecho por el ciclo externo; podría haber muchas sentencias antes o después del ciclo interno.

Considérese otro ejemplo. Para ciclos anidados controlados por conteo, el patrón se parece a esto (donde `outCount` es el contador para el ciclo externo, `inCount` es el contador para el ciclo interno, y `limit1` y `limit2` son el número de veces que se debe ejecutar cada ciclo):

```
outCount = 1;                    // Inicializar el contador de ciclo externo
while (outCount <= limit1)
{
    ⋮
    inCount = 1;                 // Inicializar el contador de ciclo interno
    while (inCount <= limit2)
    {
        ⋮
        inCount++;               // Incrementar el contador de ciclo interno
    }
    ⋮
    outCount++;                  // Incrementar el contador de ciclo externo
}
```

Aquí, tanto el ciclo interno como el externo son ciclos controlados por conteo, pero el patrón se puede usar con cualquier combinación de ciclos.

En el siguiente fragmento de programa se muestra un ciclo controlado por conteo anidado dentro de un ciclo controlado por final de archivo. El ciclo externo introduce un valor entero que indica cuántos asteriscos imprimir en un renglón de la pantalla. (Los números de la derecha del código se emplean para seguir la ejecución del programa.)

```
cin >> starCount;                        1
while (cin)                              2
{
    loopCount = 1;                       3
    while (loopCount <= starCount)       4
    {
        cout << '*';                     5
        loopCount++;                     6
    }
    cout << endl;                        7
    cin >> starCount;                    8
}
cout << "Hasta luego" << endl;           9
```

Para ver cómo funciona este código, se seguirá su ejecución con estos valores de datos (<EOF> denota las teclas final de archivo presionadas por el usuario):

```
3
1
<EOF>
```

Se seguirá la pista de las variables `starCount` y `loopCount`, así como de las expresiones lógicas. Para hacer esto, se ha numerado cada línea (excepto las que contienen sólo una llave izquierda o derecha). Conforme se sigue el programa, la primera ejecución de la línea 3 se indica por 3.1, la segunda por 3.2, etcétera. Cada iteración de ciclo se encierra mediante una llave grande, y `true` (verdadero) y `false` (falso) se abrevian como T y F (véase la tabla 6-1).

Tabla 6-1 Code trace

Sentencia	Variables		Expresiones lógicas		Salida
	starCount	loopCount	cin	loopCount <= starCount	
1.1	3	—	—	—	—
2.1	3	—	T	—	—
3.1	3	1	—	—	—
4.1	3	1	—	T	—
5.1	3	1	—	—	*
6.1	3	2	—	—	—
4.2	3	2	—	T	—
5.2	3	2	—	—	*
6.2	3	3	—	—	—
4.3	3	3	—	T	—
5.3	3	3	—	—	*
6.3	3	4	—	—	—
4.4	3	4	—	F	—
7.1	3	4	—	—	\n (nueva línea)
8.1	1	4	—	—	—
2.2	1	4	T	—	—
3.2	1	1	—	—	—
4.5	1	1	—	T	—
5.4	1	1	—	—	*
6.4	1	2	—	—	—
4.6	1	2	—	F	—
7.2	1	2	—	—	\n (nueva línea)
8.2	1	2	—	—	—
	(operación nula)				
2.3	1	2	F	—	—
9.1	1	2	—	—	Hasta luego

Aquí se presenta una ejecución muestra del programa. La entrada del usuario está resaltada. De nuevo, el símbolo <EOF> denota las teclas final de archivo que pulsa el usuario (el símbolo no aparecería en la pantalla).

```
3
***
1
*
<EOF>
Goodbye
```

Debido a que `starCount` y `loopCount` son variables, sus valores son los mismos hasta que se cambian de forma explícita, como indican los valores repetidos de la tabla 6-1. Los valores de las expresiones lógicas `cin` y `loopCount <= starCount` existen sólo cuando se hace la prueba. Este hecho se indica con guiones en esas columnas en todos los otros casos.

Diseño de ciclos anidados

Para diseñar un ciclo anidado, se comienza con el ciclo externo. El proceso que se repite incluye el ciclo anidado como uno de sus pasos. Debido a que ese paso es más complejo que una sola sentencia, la metodología de descomposición funcional indica hacerlo en módulo separado. Se puede volver a él después y diseñar el ciclo anidado del mismo modo que se diseñaría cualquier otro ciclo.

Por ejemplo, aquí está el proceso de diseño para el segmento de código precedente:

1. *¿Cuál es la condición que termina el ciclo?* En la entrada se llega al final del archivo.
2. *¿Cómo se debe inicializar la condición?* Se debe ejecutar una lectura principal antes de que comience el ciclo.
3. *¿Cómo se debe actualizar la condición?* Al final de cada iteración debe ocurrir una sentencia de entrada.
4. *¿Cuál es el proceso que se repite?* Al usar el valor del entero de entrada actual, el código debe imprimir los asteriscos en una línea de salida.
5. *¿Cómo se debe inicializar el proceso?* No es necesaria ninguna inicialización.
6. *¿Cómo se debe actualizar el proceso?* Se produce una secuencia de asteriscos y luego un carácter de nueva línea. No hay variables contadoras o sumas qué actualizar.
7. *¿Cuál es el estado del programa al salir del ciclo?* El flujo `cin` está en estado de falla (porque el programa intentó leer después del final de archivo), `starCount` contiene el último entero leído del flujo de entrada, y los renglones de asteriscos han sido impresos junto con un mensaje de conclusión.

De las respuestas a estas preguntas, se puede escribir lo siguiente del algoritmo:

```
Leer starCount
MIENTRAS NO SE LLEGUE AL FINAL DEL ARCHIVO
    Imprimir los asteriscos de starCount
    Producir una nueva línea
    Leer starCount
Imprimir "Hasta luego"
```

Después de diseñar el ciclo externo, es evidente que el proceso en su conjunto (imprimir una secuencia de asteriscos) es un paso complejo que requiere diseñar un ciclo interno. Así que se repite la metodología para el módulo de nivel inferior correspondiente.

1. *¿Cuál es la condición que termina el ciclo?* Una condición de iteración excede el valor de `starCount`.
2. *¿Cómo se debe inicializar la condición?* El contador de iteración se debe inicializar en 1.
3. *¿Cómo se debe actualizar la condición*? El contador de iteración se incrementa al final de cada iteración.
4. *¿Cuál es el proceso que se repite?* El código debe imprimir un solo asterisco en el dispositivo de salida estándar.
5. *¿Cómo se debe inicializar el proceso?* No es necesaria ninguna inicialización.
6. *¿Cómo se debe actualizar el proceso*? No se requiere ninguna actualización.
7. *¿Cuál es el estado del programa al salir del ciclo?* Se ha impreso un solo renglón de asteriscos, el marcador de escritura está al final de la línea de salida actual y `loopCount` contiene un valor 1 mayor que el valor actual de `starCount`.

Ahora se puede escribir el algoritmo:

```
Leer starCount
MIENTRAS NO SE LLEGUE AL FINAL DEL ARCHIVO
    Establecer loopCount = 1
    MIENTRAS loopCount <= starCount
        Imprimir '*'
        Incrementar loopCount
    Producir una nueva línea
    Leer starCount
Imprimir "Hasta luego"
```

Por supuesto, los ciclos anidados por sí mismos pueden contener ciclos anidados (llamados *ciclos doblemente anidados*), que pueden contener ciclos anidados (*ciclos triplemente anidados*), etcétera. Este diseño de proceso se puede usar para cualquier número de niveles de anidación. El truco es diferir detalles por medio de la metodología de descomposición funcional, es decir, centrar la atención primero en el ciclo externo y tratar cada nuevo nivel de ciclo anidado como un módulo dentro del ciclo que lo contiene.

También es posible que el proceso dentro de un ciclo incluya más de un ciclo. Por ejemplo, aquí está un algoritmo que lee e imprime los nombres de personas desde un archivo, omitiendo el apellido paterno en la salida:

```
Leer e imprimir el nombre (termina con una coma)
MIENTRAS NO SE LLEGUE AL FINAL DEL ARCHIVO
    Leer y eliminar los caracteres del segundo nombre (termina con una coma)
    Leer e imprimir el apellido (termina en una nueva línea)
    Producir una nueva línea
    Leer e imprimir un nombre (termina con una coma)
```

Los pasos para leer nombre, segundo nombre y apellido requiere diseñar tres ciclos separados. Todos estos ciclos son controlados por centinela.

Esta clase de estructura de control compleja sería difícil de leer si se escribiera completa. Hay simplemente muchas variables, condiciones y pasos qué recordar en un momento. En los dos capítulos siguientes se examina la estructura de control que descompone los programas en trozos más manejables: el subprograma.

Bases teóricas

Análisis de algoritmos

Si para limpiar una habitación tiene que elegir entre un cepillo de dientes y una escoba, es probable que elija la escoba. Usar una escoba representa menos trabajo que usar un cepillo de dientes. Cierto, si la habitación fuese una casa de muñecas, podría ser más fácil usar el cepillo de dientes, pero en general usar una escoba es la forma más fácil de limpiar. Si se le diera a elegir entre lápiz y papel y una calculadora para sumar números, es probable que elija la calculadora porque de ordinario representa menos trabajo. Si tuviera que elegir entre caminar o manejar para llegar a una reunión, tal vez elegiría conducir; al parecer es más fácil.

(continúa) ▼

Análisis de algoritmos

¿Qué tienen en común estos ejemplos? ¿Qué tiene que ver con la computación? En cada una de las situaciones mencionadas, una de las elecciones parece requerir menos trabajo. Medir con exactitud la cantidad de trabajo es difícil en cada caso porque hay incógnitas. ¿Qué tan grande es la habitación? ¿Cuántos números hay? ¿Qué tan lejos es la reunión? En cada caso, la información desconocida se relaciona con el tamaño del problema. Si el problema es especialmente pequeño (por ejemplo, sumar 2 más 2), la estimación original de cuál método elegir (usar la calculadora) podría ser equivocado. Sin embargo, la intuición suele ser correcta porque la mayoría de los problemas son razonablemente grandes.

En computación se requiere una forma de medir la cantidad de trabajo hecho por un algoritmo en relación con el tamaño de un problema, porque hay más de un algoritmo que resuelve cualquier problema dado. Con frecuencia se debe elegir el algoritmo más eficiente, el algoritmo que hace el mínimo trabajo para un problema de un determinado tamaño.

La cantidad de trabajo requerida para ejecutar un algoritmo en relación con el tamaño del problema se denomina **complejidad** del algoritmo. Sería deseable poder examinar un algoritmo y determinar su complejidad. Después se podrían tomar dos algoritmos que realizan la misma tarea y determinar cuál la completa más rápido (requiere menos trabajo).

Complejidad Medida del esfuerzo que realiza la computadora para efectuar un cálculo, en relación con el tamaño del mismo.

¿Cómo se mide la cantidad requerida de trabajo para ejecutar un algoritmo? Se usa el número total de *pasos* ejecutados como medida de trabajo. Una sentencia, lo mismo que una asignación, podría requerir sólo un paso; otra, como un ciclo, podría necesitar muchos pasos. Se define un paso como cualquier operación aproximadamente equivalente en complejidad a una comparación, una operación I/O o una asignación.

Dado un algoritmo con sólo una secuencia de sentencias simples (ninguna bifurcación o ciclo), el número de pasos llevados a cabo se relaciona de modo directo con el número de sentencias. Cuando se introducen bifurcaciones, se posibilita omitir algunas sentencias del algoritmo. Las bifurcaciones permiten restar pasos sin eliminarlos físicamente del algoritmo porque sólo se ejecuta una rama a la vez. Pero debido a que, por lo común, se quiere expresar trabajo en términos del escenario del peor de los casos, se usa el número de pasos de la rama más larga.

Ahora considere el efecto de un ciclo. Si el ciclo repite una secuencia de 15 sentencias simples 10 veces, éste efectúa 150 pasos. Los ciclos permiten multiplicar el trabajo hecho en un algoritmo sin añadir sentencias.

Ahora que se tiene una medida para el trabajo hecho en un algoritmo, se pueden comparar algoritmos. Por ejemplo, si un algoritmo A ejecuta siempre 3124 pasos y el algoritmo B hace siempre la misma tarea en 1321 pasos, entonces se puede decir que el algoritmo B es más eficiente; es decir, son menos los pasos para realizar la misma tarea.

Si un algoritmo, de una ejecución a otra, siempre toma el mismo número de pasos o menos, se dice que se ejecuta en una cantidad de tiempo acotada por una constante. Se dice que este tipo de algoritmos tiene complejidad de *tiempo constante*. Tenga cuidado: tiempo constante no significa pequeño; significa que la cantidad de trabajo no excede cierta cantidad de una ejecución a otra.

Si un ciclo se ejecuta un número fijo de veces, el trabajo hecho es mayor que el número físico de sentencias, pero aún es constante. ¿Qué sucede si el número de interacciones de ciclo cambia de una ejecución a la siguiente? Suponga que un archivo de datos contiene *N* valores de datos que serán procesados en un ciclo. Si el ciclo lee y procesa un valor durante cada iteración, entonces el ciclo ejecuta *N* iteraciones. La cantidad de trabajo realizado depende de una variable: el número de valores de datos. La variable *N* determina el tamaño del problema en este ejemplo.

Si se tiene un ciclo que se ejecuta *N* veces, el número de pasos por ejecutar es algún factor multiplicado por *N*. El factor es el número de pasos realizados dentro de una sola iteración del ciclo.

(continúa) ▼

Análisis de algoritmos

Específicamente, el trabajo realizado por un algoritmo con un ciclo dependiente de datos está dado por la expresión

$$\overbrace{S_1 \times N}^{\text{Pasos que efectúa el ciclo}} + \underbrace{S_0}_{\text{Pasos efectuados fuera del ciclo}}$$

donde S_1 es el número de pasos en el cuerpo del ciclo (una constante para un determinado ciclo simple), N es el número de iteraciones (una variable que representa el tamaño del problema) y S_0 es el número de pasos fuera del ciclo. Los matemáticos nombran lineales a esta forma de expresiones; por consiguiente, se dice que algoritmos como éste tienen complejidad de tiempo lineal. Observe que si N se hace muy grande, el término $S_1 \times N$ domina al tiempo de ejecución. Es decir, S_0 se vuelve una parte insignificante del tiempo de ejecución total. Por ejemplo, si S_0 y S_1 constan de 20 pasos cada uno, y N es 1 000 000, entonces el número total de pasos es 20 000 020. Los 20 pasos con los que contribuye S_0 son una pequeña fracción del total.

¿Qué hay acerca de un ciclo dependiente de datos que contiene un ciclo anidado? El número de pasos en el ciclo interno, S_2, y el número de iteraciones efectuadas por el ciclo interno, L, se debe multiplicar por el número de iteraciones en el ciclo externo:

$$\overbrace{(S_2 \times L \times N)}^{\text{Pasos efectuados por el ciclo anidado}} + \overbrace{(S_1 \times N)}^{\text{Pasos realizados por el ciclo externo}} + \overbrace{S_0}^{\text{Pasos que lleva a cabo el ciclo externo}}$$

Por sí mismo, el ciclo interno lleva a cabo $S_2 \times L$ pasos, pero debido a que el ciclo externo lo repite N veces, representa un total de $S_2 \times L \times N$ pasos. Si L es una constante, entonces el algoritmo aún se ejecuta en tiempo lineal.

Ahora, suponga que para cada una de las N iteraciones del ciclo externo, el ciclo interno efectúa N pasos ($L = N$). Por consiguiente, la fórmula para el número total de pasos es

$$(S_2 \times N \times N) + (S_1 \times N) + S_0$$

o bien,

$$(S_2 \times N^2) + (S_1 \times N) + S_0$$

Debido a que N^2 crece mucho más rápido que N (para valores grandes de N), el término del ciclo interno ($S_2 \times N^2$) representa la mayor parte de los pasos ejecutados y del trabajo realizado. Así, el tiempo de ejecución correspondiente es en esencia proporcional a N^2. Los matemáticos llaman a este tipo de fórmula cuadrática. Si se tiene un ciclo doblemente anidado en el cual cada ciclo depende de N, entonces la expresión es

$$(S_3 \times N^3) + (S_2 \times N^2) + (S_1 \times N) + S_0$$

(continúa) ▼

Análisis de algoritmos

y el trabajo y el tiempo son proporcionales a N^3, siempre que N sea razonablemente grande. Tal fórmula se llama *cúbica*.

En la tabla siguiente se muestra el número de pasos requeridos para cada incremento del exponente de N, donde N es un factor de tamaño para el problema, como el número de valores de entrada.

N	N^0 (Constante)	N^1 (Lineal)	N^2 (Cuadrática)	N^3 (Cúbica)
1	1	1	1	1
10	1	10	100	1 000
100	1	100	10 000	1 000 000
1 000	1	1 000	1 000 000	1 000 000 000
10 000	1	10 000	100 000 000	1 000 000 000 000
100 000	1	100 000	10 000 000 000	1 000 000 000 000 000

Como puede ver, cada vez que el exponente se incrementa en 1, el número de pasos se multiplica por orden de magnitud adicional (factor de 10). Es decir, si N se hace 10 veces mayor, el trabajo requerido en un algoritmo N^2 se incrementa por un factor de 100, y el trabajo requerido en un algoritmo N^3 crece por un factor de 1 000. Para poner esto en términos más concretos, un algoritmo con un ciclo doblemente anidado en el que cada ciclo depende del número de valores de datos toma 1 000 pasos para 10 valores de entrada y 1 000 billones de pasos para 100 000 valores. En una computadora que ejecuta 1 000 millones de instrucciones por segundo, la ejecución del último caso tomaría cerca de 12 días.

En la tabla se muestra también que los pasos fuera del ciclo interno representan una porción significativa del número total de pasos cuando N crece. Debido a que el ciclo interno domina el tiempo total, se clasifica la complejidad de un algoritmo de acuerdo con el orden más alto de N que aparece en su expresión de complejidad, llamado *orden de magnitud*, o simplemente *orden*, de esa expresión. Así, se habla de algoritmos que tienen "complejidad de orden N cuadrada" (o cúbica, etcétera) o se describen con los que se denomina *notación O mayúscula*. La complejidad se expresa al escribir entre paréntesis el término de orden máximo con una O mayúscula enfrente. Por ejemplo, $O(1)$ es tiempo constante, $O(N)$ es tiempo lineal, $O(N^2)$ es tiempo cuadrático y $O(N^3)$ es tiempo cúbico.

Determinar las complejidades de distintos algoritmos permite comparar el trabajo que requieren sin tener que programarlos y ejecutarlos. Por ejemplo, si usted tuviera un algoritmo $O(N^2)$ y un algoritmo lineal que realiza la misma tarea, tal vez elegiría el algoritmo lineal. Se dice *probablemente* porque un algoritmo $O(N^2)$ podría ejecutar en realidad menos pasos que un algoritmo $O(N)$ para valores pequeños de N. Recuerde que si el factor de tamaño N es pequeño, las constantes y los términos de orden menor en la expresión de complejidad pueden ser significativos.

Considérese un ejemplo. Suponga que el algoritmo A es $O(N^2)$ y que el algoritmo B es $O(N)$. Para valores grandes de N, normalmente se elegiría el algoritmo B porque requiere menos trabajo que A. Pero suponga que en el algoritmo B, $S_0 = 1\,000$ y $S_1 = 1\,000$. Si $N = 1$, entonces la ejecución del algoritmo B requiere 2 000 pasos. Ahora suponga que para el algoritmo A, $S_0 = 10$, $S_1 = 10$ y $S_2 = 10$. Si $N = 1$, entonces el algoritmo A requiere sólo 30 pasos. A continuación se muestra una tabla en la que se compara el número de pasos que requieren estos dos algoritmos para distintos valores de N.

(continúa) ▼

Análisis de algoritmos

N	Algoritmo A	Algoritno B
1	30	2 000
2	70	3 000
3	130	4 000
10	1 110	11 000
20	4 210	21 000
30	9 310	31 000
50	25 510	51 000
100	101 010	101 000
1 000	10 010 010	1 001 000
10 000	1 000 100 010	10 001 000

De esta tabla se puede ver que el algoritmo A en $O(N^2)$ es en realidad más rápido que el algoritmo B de $O(N)$, hasta el punto en que N es igual a 100. Más allá de ese punto, el algoritmo B se hace más eficiente. Así, si se sabe que N es siempre menor que 100 en un problema particular, se elegiría el algoritmo A. Por ejemplo, si el factor de tamaño N es el número de puntuaciones de prueba en un examen y el tamaño de clase está limitado a 30 alumnos, el algoritmo A sería más eficiente. Por otro lado, si N es el número de puntaciones en una universidad con 25 000 alumnos, se elegiría el algoritmo B.

Las expresiones constante, lineal, cuadrática y cúbica son ejemplos de expresiones *polinomiales*. Por consiguiente, se dice que los algoritmos cuya complejidad se caracteriza por esta clase de expresiones se ejecutan en *tiempo polinomial* y forman una amplia clase de algoritmos que abarcan todo lo que se ha explicado hasta aquí.

Además de los algoritmos de tiempo polinomial, en el capítulo 13 se encuentra un algoritmo de tiempo logarítmico. Hay también algoritmos de clase factorial ($O(N!)$), exponencial ($O(N^N)$) e hiperexponencial ($O(N^{N^N})$), cuya ejecución puede requerir vastas cantidades de tiempo y están fuera del alcance de este libro. Por ahora, el punto importante a recordar es que diferentes algoritmos que resuelven el mismo problema pueden variar de modo importante en la cantidad de trabajo que realizan.

Caso práctico de resolución de problemas

Diseño de estudio de grabación

PROBLEMA Usted ha entrado a trabajar a una empresa consultora que se especializa en convertir habitaciones en estudios de grabación. La empresa le ha pedido que escriba un programa que introduzca un conjunto de mediciones de sonoridad para una habitación y que imprima estadísticas básicas. Las mediciones se hacen al tocar una serie de 12 tonos distintos y registrar las lecturas de un medidor de nivel de sonido en un archivo. Las lecturas del medidor varían de 50 a 126 decibeles (una medida de sonoridad). Sin embargo, su programa producirá las mediciones en relación con el primer tono; es decir, mostrará cuánto difiere cada lectura de la primera. Una vez que se han leído todos los datos, el programa imprimirá las lecturas máxima y mínima.

ENTRADA Doce números reales, que representan las lecturas del medidor, en el archivo "acoustic.dat".

SALIDA Los 12 valores de entrada (impresos por eco) y sus valores respecto a la primera lectura. Al final del programa, el valor real, valor relativo y número de secuencia de la lectura máxima y la mínima.

ANÁLISIS Esto es fácil de hacer a mano. Simplemente se explora la lista, restando el primer valor de cada valor de la lista. Conforme se explora la lista, se sigue la pista de cuál valor es el máximo y cuál es el mínimo.

¿Cómo se traduce este proceso en un algoritmo? Considérese con más detenimiento lo que se está haciendo. Para hallar el número más grande en una lista, se comparan los números primero y segundo y se recuerda el más grande. Luego, se compara ese número con el tercero, recordando el más grande. El proceso se repite para todos los números, y el que se recuerda al final es el mayor. Se emplea el mismo proceso para hallar el número menor, sólo que se recuerda el número más pequeño en lugar del más grande. Considere un conjunto de datos de muestra:

Número de lectura	Lectura real	Lectura relativa
1	86.0	0.0
2	86.5	0.5
3	88.0	2.0
4	83.5	–2.5
5	88.3	2.3
6	89.6	3.6
7	80.1	–5.9
8	84.0	–2.0
9	86.7	0.7
10	79.3	–6.7
11	74.0	–12.0
12	73.5	–12.5

La lectura máxima fue la número 6 en 89.6 decibeles. La mínima fue la lectura 12 en 73.5 decibeles.

"Explorar la lista" en el algoritmo a mano se traduce en un ciclo. Ahora que se entiende el proceso, se diseñará el algoritmo de iteración con la lista de comprobación.

1. *¿Cuál es la condición en que termina el ciclo?* Debido a que hay exactamente 12 valores en un conjunto de lecturas, se usa un contador para controlar el ciclo. Cuando pasa de 12, termina el ciclo.
2. *¿Cómo se debe inicializar la condición?* El primer valor será introducido antes del ciclo, debido a que es un caso especial; es el valor que se resta de los otros para obtener sus valores relativos. También, su valor relativo es automáticamente 0.0. Así, la primera iteración del ciclo obtiene el segundo valor, de modo que el contador comenzará en 2.
3. *¿Cómo se debe actualizar la condición?* El contador se debe incrementar al final de cada iteración.
4. *¿Cuál es el proceso que se repite?* El proceso lee un valor, lo imprime por eco, resta el primer valor del nuevo, imprime el resultado y comprueba si el nuevo valor debe remplazar al valor alto o bajo actual.
5. *¿Cómo se debe inicializar el proceso?* Se debe leer el primer número. Su valor relativo se imprime automáticamente como 0.0. Es el valor inicial alto y bajo, y también se guarda como la lectura base. El número de secuencia para los valores alto y bajo se establecerá en 1 y sus valores relativos serán 0.0.
6. *¿Cómo se debe actualizar el proceso?* En cada iteración, se introduce una nueva lectura actual. Si la lectura actual es mayor que el valor alto (high), ésta remplaza al valor alto actual. Si la lectura actual es menor que el valor bajo (low) actual, entonces se convierte en el nuevo valor bajo. Se debe guardar también el número de lectura de los valores máximo y mínimo y sus valores relativos.

7. *¿Cuál es el estado del programa al salir del ciclo?* Se han introducido e impreso por eco 12 lecturas junto con 12 valores relativos. La variable de control de ciclo es igual a 13. high contiene el valor máximo leído, highNumber el número de ese valor, y highRelative el valor relativo para esa lectura; low contiene el valor mínimo, lowNumber contiene el número de esa lectura, y lowRelative tiene el valor relativo correspondiente.

SUPOSICIONES Se introducirán por lo menos 12 números reales y todos estarán dentro del intervalo apropiado.

Módulo principal **Nivel 0**

```
Inicializar el proceso
Inicializar la condición de terminación del ciclo
WHILE readingNumber <=12 DO
    Actualizar el proceso
    Actualizar la condición de terminación
Imprimir las lecturas alta y baja
Cerrar el archivo
```

Inicializar el proceso **Nivel 1**

```
Abrir el archivo de entrada
Si no se abre de manera correcta
    Escribir un mensaje de error
    Devolver 1
Imprimir el encabezado para la salida
Obtener el valor base (baseValue)
Imprimir readingNumbe1, baseValue, relativeValue 0.0
Fijar el valor alto en valor base(baseValue)
Fijar el valor alto (highNumber) en 1
Fijar el valor relativo alto (highRelative) en 0.0
Fijar el valor bajo en valor base (baseValue)
Fijar el valor bajo (lowNumber) en 1
Fijar el valor relativo bajo (lowRelative) en 0.0
```

Inicializar la condición de terminación del ciclo

```
Fijar el número de lectura (readingNumber) en 2
```

Actualizar el proceso

```
Obtener la lectura actual
Fijar el valor relativo en valor base actual (current-baseValue)
Imprimir readingNumber, current, relative
Comprobar el nuevo valor alto
Comprobar el nuevo valor bajo
```

Actualizar la condición de terminación del ciclo

```
Incrementar el número de lectura (readingNumber)
```

Imprimir las lecturas alta y baja

```
Imprimir 'El número de lectura máxima es', highNumber
Imprimir 'La lectura máxima es', high
Imprimir 'El valor relativo máximo es', highRelative
Imprimir 'El número de lectura mínima es', lowNumber
Imprimir 'La lectura mínima es', low
Imprimir 'El valor relativo mínimo es', lowRelative
```

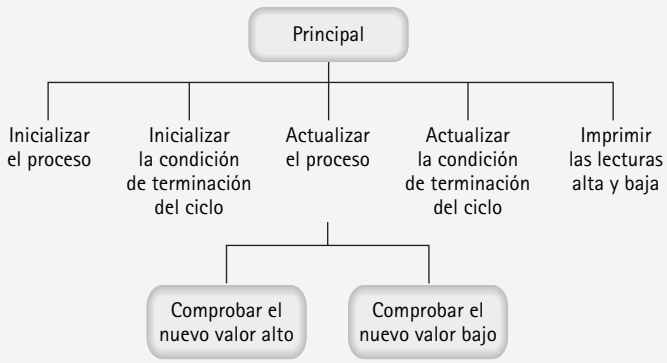
Comprobar el nuevo valor alto (high) **Nivel 2**

```
IF current > high
    Fijar high en current
    Fijar highNumber en readingNumber
    Fijar highRelative en relative
```

Comprobar el nuevo valor bajo

```
If current < low
    Establecer low en current
    Establecer lowNumber en readingNumber
    Establecer lowRelative en relative
```

DIAGRAMA DE ESTRUCTURA DEL MÓDULO

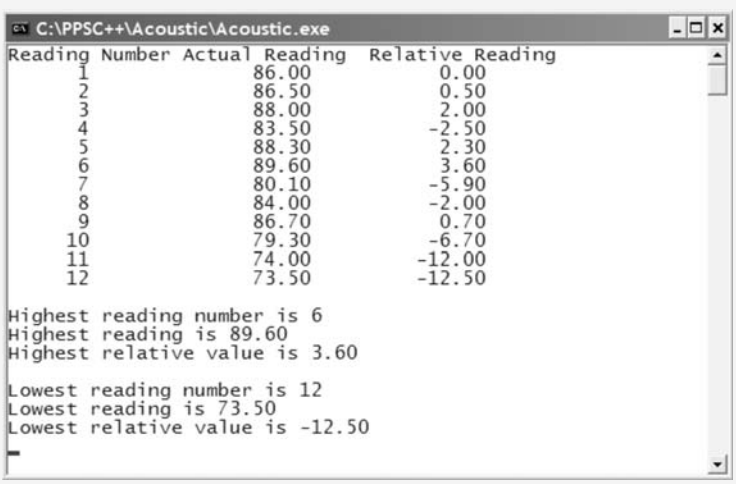



```
//*************************************************************************
// Programa Acoustic
// Este programa introduce 12 lecturas de nivel de sonido, tomadas en una
// habitación a diferentes frecuencias. La primera lectura se emplea como un
// valor base. Para cada lectura, se calcula e imprime un valor relativo
// a la base. El programa termina al imprimir las
// lecturas máxima y mínima.
//*************************************************************************

#include <iostream>
#include <fstream>
#include <iomanip>
```

```
using namespace std;

int main()
{
    // Declarar variables
    float baseValue;          // Primera lectura
    float current;            // Introducir durante cada iteración
    float relative;           // Valor actual menos valor base
    float high;               // Ingreso del valor máximo
    float highRelative;       // Valor alto menos valor base
    float low;                // Ingreso del valor mínimo
    float lowRelative;        // Valor bajo menos valor base
    int highNumber;           // Número de secuencia del valor alto
    int lowNumber;            // Número de secuencia del valor mínimo
    int readingNumber;        // Número de secuencia de la lectura actual

    // Declarar y abrir el archivo de entrada
    ifstream inData;                    // Archivo de entrada de lecturas
    inData.open("acoustic.dat");
    if ( !inData )                      // ¿Se abrió correctamente el archivo
                                           de entrada?
    {   // no
        cout << "No se puede abrir el archivo de entrada." << endl;
        return 1;                       // Terminar el programa
    }

    // Inicializar variables y resultado
    readingNumber = 1;
    relative = 0.0;
    cout << setw(14) << "Número de lectura" << setw(15)
         << "Lectura real" << setw(18) << "Lectura relativa"
         << endl;

    inData >> baseValue;                // Introducir el valor base

    // Escribir la primera línea del resultado
    cout << fixed << showpoint << setprecision(2) << setw(7)
         << readingNumber << setw(19) << baseValue << setw(15)
         << relative << endl;

    // Inicializar el proceso
    high = baseValue;
    highNumber = 1;
    highRelative = 0.0;
    low = baseValue;
    lowNumber = 1;
    lowRelative = 0.0;
    readingNumber = 2;                // Inicializar la terminación del ciclo

    while (readingNumber <= 12)
    {
        inData >> current;                    // Introducir la nueva lectura
        relative = current - baseValue;       // Calcular el nuevo valor
                                                 relativo
```

```
            cout << setw(7) << readingNumber << setw(19) << current
                 << setw(15) << relative << endl;

            if (current > high)             // Comprobar el nuevo valor alto
            {
                high = current;
                highNumber = readingNumber;
                highRelative = relative;
            }

           if (current < low)               // Comprobar el nuevo valor bajo
           {
               low = current;
               lowNumber = readingNumber;
               lowRelative = relative;
           }
           readingNumber++;       // Incrementar el número de lectura

       }

        // Imprimir las lecturas máxima y mínima
        cout << endl;
        cout << "El número de lectura máxima es " << highNumber << endl;
        cout << "La lectura máxima es "  << high << endl;
        cout << "El valor relativo máximo es " << highRelative << endl;
        cout << endl;
        cout << "El número de lectura mínima es " << lowNumber << endl;
        cout << "La lectura mínima es " << low << endl;
        cout << "El valor relativo mínimo es " << lowRelative << endl;

        inData.close();
        return 0;
}
```

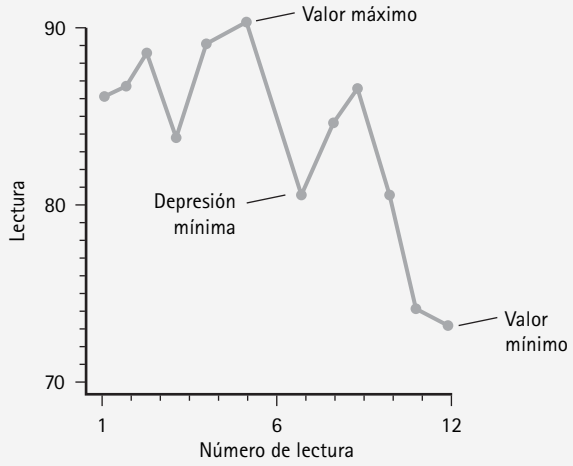
A continuación se muestra una pantalla del resultado.



La imagen mostrada corresponde a la salida producida por el programa original, escrito en idioma inglés.

Ahora que ha escrito el programa, llama al presidente de la empresa consultora y le envía los resultados. Después de estudiarlos, guarda un momento de silencio y dice: "¡Oh! Correcto". "Lo siento —dice—, pero te informé que realizaras un seguimiento del valor erróneo. No nos interesa el valor mínimo, sino la *depresión más baja en las lecturas*. Si te das cuenta el último valor casi siempre es el mínimo porque es el tono mínimo, y sabemos que la mayoría de las habitaciones responden de modo deficiente a las notas graves. No hay nada qué hacer al respecto. Pero la depresión más baja en las lecturas suele ocurrir con un tono mayor, y eso es, por lo común, un signo de un problema que se puede arreglar. ¿Puedes cambiar el programa para que produzca la depresión más baja?" Usted confirma que una depresión es una lectura seguida de una lectura mayor.

Ahora tiene que averiguar cómo reconocer una depresión o disminución en las lecturas. Usted traza algunas gráficas con líneas aleatorias y selecciona a mano las depresiones más bajas. Después conforma algunos conjuntos de datos que generarían las gráficas. Encuentra que una depresión es, en general, una serie de valores decrecientes seguida de valores crecientes. Pero también puede ser sólo una serie de valores iguales seguidos de valores mayores. El fondo de una depresión es sólo el valor mínimo antes de que los valores empiecen a crecer. Usted sabe cómo seguir la pista de un valor mínimo, ¿pero cómo indica si va seguido de un valor mayor?



La respuesta es que usted lo comprueba después de que se introduce el siguiente valor. Es decir, cuando lee un nuevo valor, comprueba si el valor precedente fue menor. Si así fue, entonces comprueba si fue el valor mínimo introducido hasta aquí. Ya ha visto el algoritmo para seguir la pista de un valor previo, y ahora sólo tiene que añadirlo al programa.

Ahora puede volver a la lista de comprobación y ver qué debe cambiar. El control del ciclo es el mismo, sólo cambia el proceso. Así, es posible omitir las tres primeras preguntas.

4. *¿Cuál es el proceso que se repite?* Es el mismo que antes, excepto donde se comprueba el valor mínimo. En cambio, se comprueba primero si el valor precedente es menor que el valor actual. Si es así, entonces se comprueba si el valor precedente debe remplazar al mínimo hasta aquí. Al remplazar el valor mínimo, se debe asignar readingNumber menos uno a lowNumber, porque la depresión ocurrió con la lectura previa. También se podría seguir la pista del valor previo de readingNumber, pero es más fácil calcularlo.
5. *¿Cómo se debe inicializar el proceso?* El cambio aquí es que se debe inicializar el valor precedente. Se puede igualar con el primer valor. También se tiene que inicializar en 0.0 el valor precedingRelative.
6. *¿Cómo se debe actualizar el proceso?* Se agregan pasos para igualar el valor precedente con el actual al final del ciclo, y guardar el valor relativo en precedingRelative para uso en la siguiente iteración.
7. *¿Cuál es el estado del programa al salir del ciclo?* En este punto, el valor precedente contiene el último valor introducido y precedingRelative contiene el último valor calculado; low contiene el

valor de la depresión más baja, lowNumber contiene el número de lectura de la depresión más baja, y lowRelative es la diferencia entre la depresión más baja y el primer valor.

Ahora se examinarán los módulos que han cambiado. Se usarán cursivas para indicar los pasos que son diferentes:

Inicializar el proceso **Nivel 1**

```
Abrir el archivo de entrada
Si no se abre de manera correcta
    Escribir un mensaje de error
    Devolver 1
Imprimir el encabezado para la salida
Obtener el valor base (baseValue)
Imprimir readingNumber 1, baseValue, relativeValue 0.0
Establecer el valor precedente en baseValue
Establecer precedingRelative en 0.0
Fijar high en baseValue
Fijar highNumber en 1
Fijar highRelative en 0.0
Fijar low en baseValue
Fijar lowNumber en 1
Fijar lowRelative en 0.0
```

Actualizar el proceso

```
Solicitar la lectura actual
Obtener la lectura actual
Fijar relative en current-baseValue
Imprimir readingNumber, current, relative
Comprobar el nuevo valor alto
Comprobar el nuevo valor bajo
Establecer el valor precedente en actual
Establecer precedingRelative en relative
```

Imprimir las lecturas alta y baja

```
Imprimir 'El número de lectura máxima es', highNumber
Imprimir 'La lectura máxima es', high
Imprimir 'El valor relativo máximo es', highRelative
Imprimir 'La depresión más baja es el número', lowNumber
Imprimir 'La lectura de depresión más baja es', low
Imprimir 'La depresión relativa más baja es', lowRelative
```

Comprobar el nuevo valor bajo

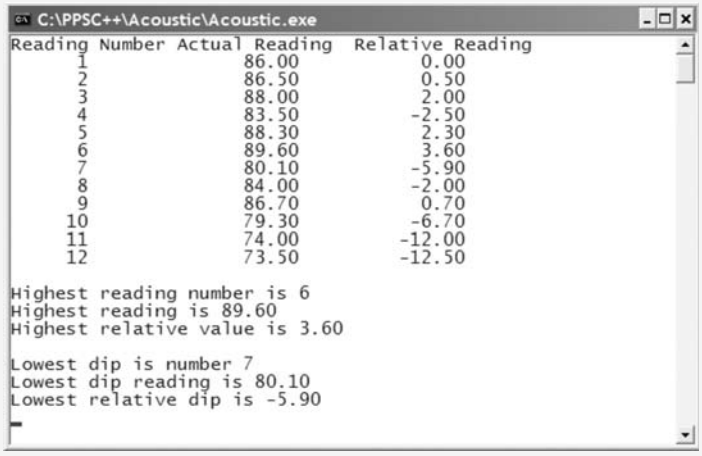
```
If current > preceding
    If preceding < low
Establecer low en preceding
        Establecer lowNumber en readingNumber – 1
        Establecer lowRelative en precedingRelative
```

Aquí se muestra una nueva versión del programa.

```
//**************************************************************************
// Programa Acoustic
// Este programa introduce 12 lecturas de nivel de sonido, tomadas en una
// habitación a diferentes frecuencias. La primera lectura se emplea como un
// valor base. Para cada lectura, se calcula e imprime un valor relativo
// a la base. El programa termina al imprimir la depresión más baja
// en las lecturas, donde una depresión se define como una lectura
// seguida de una lectura mayor
//**************************************************************************

#include <iostream>
#include <fstream>
#include <iomanip>

using namespace std;

int main()
{
    // Declarar variables
    float baseValue;          // Primera lectura
    float preceding;          // Lectura precedente actual
    float precedingRelative; // Valor relativo precedente actual
    float current;            // Introducir durante cada iteración
    float relative;           // Valor actual menos valor base
    float high;               // Valor máximo introducido
    float highRelative;       // Valor alto menos valor base
    float low;                // Depresión más baja en las lecturas
    float lowRelative;        // Valor relativo de low
    int highNumber;           // Número de secuencia de high
    int lowNumber;            // Número de secuencia de la depresión más baja
    int readingNumber;        // Número de secuencia de la lectura actual

    // Declarar y abrir el archivo de entrada
    ifstream inData;                    // Archivo de entrada de lecturas
    inData.open("acoustic.dat");
    if ( !inData )                      // ¿Se abrió correctamente el archivo
                                           de entrada?
    {   // no
        cout << "No es posible abrir el archivo de entrada." << endl;
        return 1;                       // Terminar el programa
    }

    // Inicializar variables y resultado
    cout << setw(14) << "Número de lectura" << setw(15)
         << "Lectura real" << setw(18) << " Lectura relativa"
         << endl;
    inData >> baseValue;
    preceding = baseValue;
    precedingRelative = 0.0;
    highNumber = 1;
    lowNumber = 1;
    high = baseValue;
```

```
    low = baseValue;
    highRelative = 0.0;
    lowRelative = 0.0;
    readingNumber = 1;
    relative = 0.0;

    // Escribir la primera línea de salida
    cout << fixed << showpoint << setprecision(2) << setw(7)
         << readingNumber << setw(19)
         << baseValue << setw(15) << relative << endl;

    readingNumber = 2;                    // Inicializar la terminación del
                                             ciclo
    while (readingNumber <= 12)
    {
        inData >> current;                // Introducir la nueva lectura
        relative = current - baseValue;   // Calcular el nuevo valor relativo
        cout << setw(7) << readingNumber << setw(19) << current
             << setw(15) << relative << endl;

        if (current > high)               // Comprobar el nuevo valor alto
        {
            high = current;
            highNumber = readingNumber;
            highRelative = relative;
        }

        if (current > preceding)          // Comprobar el nuevo valor bajo
        {
            if (preceding < low)
            {
                low = preceding;
                lowNumber = readingNumber - 1;
                lowRelative = precedingRelative;
            }
        }
        preceding = current;
        precedingRelative = relative;
        readingNumber++;
    }

    // Imprimir las lecturas alta y baja
    cout << endl;
    cout << "El número de lectura máxima es " << highNumber << endl;
    cout << "La lectura máxima es "  << high << endl;
    cout << "El valor relativo máximo es " << highRelative << endl;
    cout << endl;
    cout << "El número de depresión más baja es " << lowNumber << endl;
    cout << "La depresión más baja es " << low << endl;
    cout << "La depresión relativa más baja es " << lowRelative << endl;

    inData.close();
    return 0;
}
```

A continuación se muestra la pantalla de salida.



La imagen mostrada corresponde a la salida producida por el programa original, escrito en idioma inglés.

Prueba y depuración

Estrategia de prueba de ciclo

Aun si un ciclo se ha diseñado y comprobado de manera apropiada, es importante probarlo rigurosamente debido a que siempre está presente la posibilidad de que se introduzca un error durante la fase de ejecución. Debido a que los ciclos permiten introducir muchos conjuntos de datos en una ejecución y a que cada iteración puede ser afectada por las precedentes, los datos de prueba para un programa de iteración son, en general, más extensos que para un programa con sólo sentencias secuenciales o de ramificación. Para probar por completo un ciclo, se debe comprobar la ejecución adecuada de una sola iteración e iteraciones múltiples.

Recuerde que un ciclo tiene siete partes (correspondientes a las siete preguntas de la lista de comprobación). Una estrategia de prueba debe corroborar cada parte. Aunque las siete partes no se ejecutan por separado en un ciclo, la lista de comprobación es un recordatorio de que algunas operaciones de iteración sirven para múltiples propósitos, cada uno de los cuales se debe probar. Por ejemplo, la sentencia de incremento en un ciclo controlado por conteo puede estar actualizando tanto el proceso como la condición de terminación, así que es importante comprobar que realice ambas acciones de modo apropiado con respecto al resto del ciclo.

Para probar un ciclo, se intenta diseñar conjuntos de datos que puedan ocasionar que las variables salgan del intervalo o dejen los archivos en estados inadecuados que violan la poscondición de ciclo (una afirmación que debe ser verdadera inmediatamente después de la salida del ciclo) o la poscondición del módulo que contiene el ciclo.

También es buena práctica probar un ciclo para cuatro casos especiales: 1) cuando el ciclo se omite por completo, 2) cuando el cuerpo del ciclo se ejecuta sólo una vez, 3) cuando el ciclo se ejecuta cierto número normal de veces y 4) cuando no se sale del ciclo.

Las sentencias que siguen un ciclo dependen por lo común de su proceso. Si se puede omitir un ciclo, esas sentencias podrían no ejecutarse de manera correcta. Si es posible ejecutar una sola iteración de un ciclo, los resultados pueden mostrar si el cuerpo se ejecuta de modo correcto en ausencia de los efectos de iteraciones previas, lo cual puede ser muy útil cuando se pretende aislar la fuente de error. Obviamente, es importante probar un ciclo en condiciones normales, con una amplia variedad de entradas. Si es posible, se debe probar el ciclo con datos reales además de simular conjuntos de datos. Los ciclos controlados por conteo se deben probar para asegurar que se ejecutan exactamente el número correcto de veces. Por último, si hay posibilidad de que nunca finalice un ciclo, sus datos de prueba deben tratar de que eso suceda.

Probar un programa puede ser tan interesante como escribirlo. Para probar un programa es necesario retroceder, examinar con nuevos ojos lo que se ha escrito y luego atacarlo en toda forma posible para hacerlo fallar. Esto no siempre es fácil de hacer, pero es necesario si pretende que sus programas sean confiables. (Un *programa confiable* es el que funciona de modo consistente y sin errores, sin importar si los datos de entrada son válidos o no.)

Planes de prueba relacionados con ciclos

En el capítulo 5 se introdujeron planes de prueba formales y se analizó la prueba de bifurcaciones. Esas guías aún se aplican a programas con ciclos, pero aquí se proporcionan algunas normas adicionales que son específicas para ciclos.

Infortunadamente, cuando un ciclo está incluido en un programa más grande, en ocasiones es difícil controlarlo y observar las condiciones en que se ejecuta el ciclo con sólo datos de prueba y resultado. En algunos casos se deben usar pruebas directas. Por ejemplo, si un ciclo lee valores de punto flotante de un archivo e imprime su promedio sin imprimirlos por eco, no se puede decir que el ciclo procesa todos los datos; si los valores de datos en el archivo son los mismos, entonces el promedio aparece correcto siempre que incluso uno de ellos se procese. Es necesario construir el archivo de entrada para que el promedio sea un valor único al que se puede llegar sólo procesando todos los datos.

Para simplificar la prueba de dichos ciclos, sería bueno observar los valores de las variables relacionadas con el ciclo al comienzo de cada iteración. ¿Cómo es posible observar los valores de las variables mientras está en ejecución un programa? Dos técnicas comunes son el uso del programa *depurador* del sistema y el uso de sentencias de salida extra diseñadas sólo para fines de depuración. Estas técnicas se analizan en la siguiente sección, Sugerencias de prueba y depuración.

Ahora se examinarán algunos casos de prueba que son específicos para los distintos tipos de ciclos que se han visto en este capítulo.

Ciclos controlados por conteo Cuando un ciclo es controlado por conteo, se debe incluir un caso de prueba que especifique el resultado para todas las iteraciones. Podría ser útil añadir una columna extra para el plan de prueba que lista el número de iteración. Si el ciclo lee datos y produce un resultado, entonces cada valor de entrada debe producir una salida diferente para facilitar la detección de errores. Por ejemplo, en un ciclo que se supone lee e imprime 100 valores de datos, es más fácil decir que el ciclo ejecuta el número correcto de iteraciones cuando los valores son 1, 2, 3, ..., 100 que si todos son los mismos.

Si el programa introduce la cuenta de iteración para el ciclo, necesita probar los casos en que se introduce una cuenta no válida, como un número negativo (debe aparecer un mensaje de error y se debe omitir el ciclo), una cuenta de 0 (se debe omitir el ciclo), una cuenta de 1 (el ciclo se debe ejecutar una vez) y algún número representativo de iteraciones (el ciclo se debe ejecutar el número especificado de veces).

Ciclos controlados por suceso En un ciclo controlado por suceso, debe probar la situación en la que el suceso ocurre antes que el ciclo, en la primera iteración y en un número representativo de iteraciones. Por ejemplo, si el suceso es que ocurre el final del archivo, entonces pruebe un archivo vacío, un archivo con un conjunto de datos, y otro con varios conjuntos de datos. Si su prueba requiere leer archivos de prueba, debe adjuntar las copias impresas de los archivos al plan de prueba e identificar cada uno de alguna manera, de modo que el plan pueda referirse a ellos. También es útil identificar dónde comienza cada iteración en las columnas Entrada y Salida esperada del plan de prueba.

Cuando el suceso es la entrada de un valor centinela, son necesarios los siguientes casos de prueba: el centinela es el único conjunto de datos, el centinela sigue a un conjunto de datos, y el centinela sigue a un número representativo de conjuntos de datos. Dado que los ciclos controlados por centinela requieren una lectura principal, es muy importante comprobar que los conjuntos de datos primero y último sean procesados de modo adecuado.

Sugerencias de prueba y depuración

1. Planee con cuidado sus datos de prueba para probar todas las secciones de un programa.
2. Tenga cuidado con los ciclos infinitos, en los que la expresión de la sentencia While nunca se vuelve `false`. El síntoma: el programa no se detiene. Si está en un sistema que monitorea el tiempo de ejecución de un programa, es posible que vea un mensaje como "TIME LIMITED EXCEEDED".

   Si ha creado un ciclo infinito, compruebe su lógica y la sintaxis de sus ciclos. Asegúrese de que no hay punto y coma inmediatamente después del paréntesis derecho de la condición While:

   ```
   while (Expresión);          // Erróneo
     Sentencia
   ```

   Este punto y coma causa un ciclo infinito en la mayoría de los casos; el compilador piensa que el contenido del ciclo es la sentencia nula (la sentencia no hace nada, compuesta sólo de un punto y coma). En un ciclo controlado por bandera, asegúrese de que la bandera finalmente cambia.

   Y, como siempre, tenga cuidado del problema = versus == en las condiciones While, así como en las condiciones If. La línea

   ```
   while (someVar = 5)       // Wrong (should be ==)
   ```

   produce un ciclo infinito. El valor de la expresión de asignación (no relacional) es siempre 5, que se interpreta como `true`.
3. Compruebe de manera cuidadosa la condición de terminación del ciclo y asegúrese de que algo en el ciclo causa que se cumpla. Observe con detenimiento los valores que causan una iteración demasiadas o pocas veces (síndrome de "off-by-1").
4. Recuerde usar la función `get` en vez del operador >> en ciclos que son controlados por detección de un carácter de nueva línea.
5. Efectúe un repaso de algoritmo para verificar que ocurren todas las precondiciones y poscondiciones apropiadas en los lugares correctos.
6. Realice a mano un seguimiento de la ejecución del ciclo con un repaso de código. Simule los primeros y los últimos pasos con mucho cuidado para ver cómo se comporta en realidad el ciclo.
7. Use un *depurador* si su sistema lo proporciona. Un depurador es un programa que ejecuta su programa en "movimiento lento", permitiéndole ejecutar una instrucción a la vez y examinar el contenido de las variables a medida que cambian. Si no lo ha hecho, compruebe si su sistema cuenta con un depurador.
8. Si todo lo demás falla, use *sentencias de salida depuradoras*, sentencias de salida insertadas en un programa para ayudar a depurarlo. Ellas producen mensajes que indican el flujo de ejecución en el programa o producen un informe de los valores de las variables en ciertos puntos del programa.

   Por ejemplo, si desea saber el valor de la variable beta en cierto punto de un programa, podría insertar esta sentencia:

   ```
   cout << "beta = " << beta << endl;
   ```

   Si esta sentencia de salida está en un ciclo, obtendrá tantos valores de beta como iteraciones haya en el cuerpo del ciclo.

   Una vez que ha depurado su programa, puede eliminar las sentencias de salida depuradoras o sólo precederlas con // para que sean tratadas como comentarios. (Esta práctica se conoce como

comentar una pieza de código.) Puede eliminar las barras inclinadas dobles si necesita usar de nuevo las sentencias.

9. Una onza de prevención vale una libra de depuración. Use las preguntas de la lista de comprobación para diseñar su ciclo correctamente al inicio. Podría parecer trabajo extra, pero a la larga tiene su recompensa.

Resumen

La sentencia While es una construcción de ciclo que permite al programa repetir una sentencia siempre que el valor de una expresión sea `true`. Cuando el valor de la expresión es `false`, se omite el cuerpo del ciclo y la ejecución continúa con la primera sentencia después del ciclo.

Con la sentencia While, usted construye varios tipos de ciclos que usará una y otra vez. Estos tipos de ciclos caen en dos categorías: ciclos controlados por conteo y ciclos controlados por suceso.

En un ciclo controlado por conteo, el cuerpo del ciclo se repite un número especificado de veces. Se inicializa una variable contadora antes de la sentencia While. Esta variable es la variable de control del ciclo. La variable de control se prueba contra el límite en la expresión While. La última sentencia del cuerpo del ciclo incrementa la variable de control.

Los ciclos controlados por suceso continúan ejecutándose hasta que algo dentro del cuerpo señala que se debe detener el proceso de iteración. Los ciclos controlados por suceso incluyen que prueban un valor centinela en los datos, el final del archivo o un cambio en la variable de bandera.

Los ciclos controlados por centinela son ciclos de entrada que continúan con el ingreso (y proceso) de valores de datos hasta que se terminan. Para ponerlos en práctica con la sentencia While, se debe probar el estado del flujo de entrada usando el nombre del objeto de flujo como si fuera una variable booleana. La prueba produce `false` cuando no hay más valores de datos. Una bandera es una variable que se fija en una parte del programa y se prueba en otra. En un ciclo controlado por bandera, se debe establecer la bandera antes de que comience el ciclo, probarla en la expresión While y cambiarla en alguna parte del cuerpo del ciclo.

Contar es una operación iterativa que sigue la pista de cuántas veces se repite un ciclo o cuántas veces ocurre algún suceso. Esta cuenta se puede usar en cálculos o para controlar el ciclo. Un contador es una variable que se emplea para contar. Puede ser la variable de control del ciclo en un ciclo controlado por conteo, un contador de iteración en un ciclo de conteo, o un contador de suceso que cuenta el número de veces que ocurre una condición particular en un ciclo.

Sumar es una operación de iteración que mantiene un total de ejecución de ciertos valores. Es como contar en el sentido de que la variable que contiene la suma se inicializa fuera del ciclo. Sin embargo, la operación de suma agrega valores desconocidos; la operación de conteo añade una constante (1) al contador cada vez.

Cuando diseña un ciclo, hay siete puntos a considerar: cómo se inicializa, prueba y actualiza la condición de terminación; cómo se inicializa, efectúa y actualiza el proceso en el ciclo; y el estado del programa al salir del ciclo. Al contestar las preguntas de la lista de comprobación, es posible destacar cada uno de estos puntos.

Para diseñar una estructura anidada, comience con el ciclo externo. Cuando llegue a donde debe aparecer el ciclo interno, construya un módulo separado y vuelva después a su diseño.

El proceso de probar un ciclo se basa en las respuestas a las preguntas de la lista de comprobación y los patrones que podría encontrar el ciclo (como ejecutar una sola iteración, múltiples iteraciones, una cantidad infinita de iteraciones, o ninguna en absoluto).

Comprobación rápida

1. ¿Cuáles son las cuatro fases de ejecución del ciclo? (pp. 208)
2. ¿En qué se asemeja un ciclo controlado por bandera y uno controlado por suceso, y en qué son diferentes? (pp. 210-214).
3. ¿Promediar un conjunto de valores introducidos por un ciclo implica sumar, contar, o ambas cosas? (pp. 215-217)

4. ¿Qué tipo de ciclo usaría para leer un archivo de números de licencia de controladores hasta que se introduce un número específico? (pp. 217-219)
5. Escriba una sentencia While que aparece cuando la variable `int`, `count`, es mayor que 10 o la bandera booleana `found` es `true`. (pp. 208-212)
6. Escriba las partes de inicialización y actualización del ciclo del ejercicio 5. La variable `found` se hace verdadera cuando la variable `int`, `inData`, contiene cero. La cuenta empieza en 1 y se incrementa en cada iteración. (pp. 217-219)
7. Añada una prueba para final de archivo en `cin` para el ciclo del ejercicio 6, y agregue sentencias al ciclo que efectúa una lectura principal y una lectura de actualización de `inData` desde `cin`. (pp. 212-214)
8. Añada operaciones de inicialización y actualización al ciclo del ejercicio 7 que causan que cuente los resultados del valor 1 que se introducen a la variable `inData`, y cuente todos los valores leídos en `inData`. (pp. 215-217)
9. ¿Cómo ampliaría el ciclo del ejercicio 8 para que se repita en su totalidad cinco veces? (pp. 220-225)
10. ¿Cómo probaría un programa iterativo que lee temperaturas cada hora de un archivo y produce el promedio diario? (pp. 239-240)

Respuestas

1. Entrada, iteración, prueba, salida. **2.** Ambos salen cuando ocurre un suceso. Pero el ciclo controlado por suceso prueba el ciclo del suceso al comienzo de cada iteración, mientras que el ciclo controlado por bandera comprueba el suceso a la mitad de la iteración y fija una bandera que se probará al comienzo de la siguiente iteración. **3.** Ambas. **4.** Un ciclo controlado por suceso. (Los sucesos serían la introducción del número, o llegar al final del archivo.) **5.** `while (count <= 10 && !found)`

6.
```
count = 1;
found = false;
while (count <= 10 && !found)
{
    count++;
    found = inData == 0;
}
```

7.
```
count = 1;
found = false;
cin >> inData;
while (count <= 10 && !found && cin)
{
    count++;
    found = inData == 0;
    cin >> inData;
}
```

8.
```
onesCount = 0;
sum = 0;
count = 1;
found = false;
cin >> inData;
while (count <= 10 && !found && cin)
{
```

```
    if (inData == 1)
        onesCount++;
    sum = sum + inData;
    count++;
    found = inData == 0;
    cin >> inData;
}
```

9. Anidarlo en un ciclo de conteo que cuenta de 1 a 5. **10.** Correr el programa con un archivo vacío, un archivo con 24 valores de entrada, un archivo con menos de 24 valores de entrada, y uno con más de 24 valores de entrada. Probar el programa con 24 valores de entrada, todos iguales, y con 24 valores que difieran, de modo que omitir alguno de ellos dé como resultado un promedio diferente del promedio de los 24.

Ejercicios de preparación para examen

1. La sentencia While termina cuando la condición de terminación se vuelve verdadera. ¿Verdadero o falso?
2. El cuerpo de la sentencia While se ejecuta siempre por lo menos una vez. ¿Verdadero o falso?
3. Usar un bloque como el cuerpo de la sentencia While permite colocar cualquier número de sentencias dentro de un ciclo. ¿Verdadero o falso?
4. Compare la siguiente lista de términos con las definiciones dadas a continuación.
   - ***a***) Entrada de ciclo.
   - ***b***) Iteración.
   - ***c***) Prueba de ciclo.
   - ***d***) Terminación del ciclo.
   - ***e***) Condición de terminación.
   - ***f***) Ciclo controlado por conteo.
   - ***g***) Ciclo controlado por suceso.
   - ***h***) Contador de iteración.
   - ***i***) Contador de suceso.
     - ***i***) Un ciclo que se ejecuta un número específico de veces.
     - ***ii***) Cuando el flujo de control alcanza la primera sentencia dentro de un ciclo.
     - ***iii***) Una variable que se incrementa cada vez que se encuentra una condición particular.
     - ***iv***) Cuando se toma la decisión de si comenzar una nueva iteración o salir.
     - ***v***) Un ciclo que termina cuando se encuentra una condición específica.
     - ***vi***) La condición ocasiona la terminación del ciclo.
     - ***vii***) Una variable que se incrementa con cada iteración de un ciclo.
     - ***viii***) Cuando el control pasa a la sentencia después del ciclo.
     - ***ix***) Un solo paso por un ciclo.
5. ¿Cuántas veces se ejecuta el siguiente ciclo y cuál es su resultado?

```
count = 1;
while (count <= 12)
{
    cout << count << endl;
    count++;
}
```

6. ¿Cuántas veces se ejecuta el siguiente ciclo y cuál es su resultado?

```
count = 0;
while (count <= 11)
{
    cout << count << ", ";
    count++;
}
```

7. ¿Cuántas veces se ejecuta el siguiente ciclo y cuál es su resultado?

```
count = 1;
while (count < 13)
{
    cout << "$" << count << ".00" << endl;
    count++;
}
```

8. ¿Qué produce la siguiente estructura de ciclo anidado?

```
count = 1;
while (count <= 11)
{
    innerCount = 1
    while (innerCount <= (12 - count) / 2)
    {
        cout << " ";
        innerCount++;
    }
    innerCount = 1;
    while (innerCount <= count)
    {
        cout << "@";
        innerCount++;
    }
    cout << endl;
    count++;
}
```

9. ¿Qué produce la siguiente estructura de ciclo anidado?

```
count = 1;
while (count <= 10)
{
    innerCount = 1;
    while (innerCount <= 10)
    {
        cout << setw(5) << count * innercount;
        innerCount++;
    }
    cout << endl;
    count++;
}
```

10. Se supone que el siguiente ciclo suma todos los valores de entrada en el archivo `indata`. ¿Cuál es el error en él? Cambie el código para que funcione de manera correcta.

```
sum = 0;
indata >> number;
while (indata)
{
    indata >> number;
    sum = sum + number;
}
```

11. ¿Qué valor centinela elegiría para un programa que lee nombres como cadenas?
12. Se supone que el siguiente segmento de código escribe los números impares del 1 al 19. ¿Qué produce en realidad? Cambie el código para que funcione de modo correcto.

```
number = 1;
while (number < 10)
{
    number++;
    cout << number * 2 - 1 << " ";
}
```

13. Las lecturas principales no son necesarias cuando el ciclo es controlado por un valor centinela. ¿Verdadero o falso?
14. ¿Cuáles son las siete preguntas que deben contestarse a fin de diseñar un ciclo?
15. Se supone que el siguiente segmento de código produce el promedio de los cinco números en cada línea de entrada, para todas las líneas del archivo. En cambio, produce sólo la suma de todos los números del archivo. ¿Qué sucede con el segmento de código? Cambie el código para que funcione correctamente.

```
sum = 0;
while (indata)
{
    count = 1;
    while (count <= 5 && indata)
    {
        cin >> number;
        sum = sum + number;
    }
    cout << sum / count << endl;
}
```

Ejercicios de preparación para la programación

1. Escriba un segmento de código usando un ciclo While que produzca los números de –10 a 10.
2. Escriba un segmento de código con un ciclo While que sume los enteros, contando desde 1, y se detenga cuando la suma sea mayor que 10 000, imprimiendo el entero que se añadió recientemente a la suma.
3. Escriba un segmento de código iterativo que introduzca hasta 20 puntuaciones (calificaciones) enteras desde el archivo inData y produzca su promedio. Si el archivo contiene menos de 20 puntuaciones, el segmento debe producir aún el promedio correcto. Si el archivo contiene más de 20 puntuaciones, los números adicionales se deben ignorar. Asegúrese de considerar lo que sucede si el archivo está vacío.
4. Escriba un segmento de código que lee líneas de texto del archivo `chapter6` y produce el número de líneas en el archivo que contienen la cadena `"code segment"`.
5. Escriba un segmento de código que lea una cadena de `cin`. La cadena debe ser una de `"Yes"`, `"No"`, `"yes"` o `"no"`. Si no es así, el usuario debe solicitar que se introduzca una respuesta apropiada, y el proceso se debe repetir. Una vez que se recibe una respuesta válida, la variable `bool`, yes debe asignarse a `true` si la respuesta es `"Yes"` o `"yes"` y a `false` si la respuesta es `"No"` o `"no"`.
6. Escriba un segmento de código que imprima los días de un mes en formato de calendario. El día de la semana en que comienza el mes se representa por una variable `int`, `starDay`. Cuando `starDay` es cero, el mes comienza en un domingo. La variable `int`, `days`, contiene el número de días en un mes. Imprimir un encabezado con los días de la semana como la primera línea de salida. Los números de día deben alinearse de manera nítida debajo de estos encabezados de columna.
7. Se podría extender el código del ejercicio 6 para imprimir un calendario para un año anidándolo dentro de un ciclo que repita el código 12 veces.
   a) ¿Qué fórmula usaría para calcular el nuevo día de inicio para el siguiente mes?
   b) ¿Qué información adicional necesitaría para imprimir el calendario para cada mes?

8. Escriba un segmento de código que lea los caracteres del archivo `textData`, y luego produzca el porcentaje de los caracteres que son la letra "z".
9. Cambie el segmento de código del ejercicio 8 para que detenga la lectura al final del archivo después de que se han impreso 10 000 caracteres.
10. Escriba un segmento de código que dé como resultado los números de Fibonacci que son menores que 30 000. Cada número de Fibonacci es la suma de sus dos predecesores. Los primeros dos números de Fibonacci son 1 y 1. Así, la secuencia comienza con:

    1, 1, 2, 3, 5, 8, 13, 21, 34, . . .

    Produzca cada número en una línea separada.
11. Modifique el segmento de código del ejercicio 10 para que también produzca la posición del número de Fibonacci en la secuencia. Por ejemplo:

    ```
    1       1
    2       1
    3       2
    4       3
    5       5
    6       8
    7       13
    ```

12. Escriba un segmento de código que introduce un entero desde `cin` y luego produce un renglón de ese número de estrellas en `cout`.
13. ¿Cómo contestó las preguntas de la lista de comprobación para el ejercicio 12?
14. Cambie el segmento de código del ejercicio 12 para leer una serie de números del archivo `indata` e imprimir un renglón de estrellas en `cout` para lectura de número. (Esto se puede considerar como imprimir una gráfica de barras de un archivo de datos.)
15. ¿Qué datos de prueba usaría para probar el segmento de código del ejercicio 14?

Problemas de programación

1. Diseñe y escriba un programa de C++ que introduce una serie de 24 temperaturas de cada hora desde un archivo y produce una gráfica de barras (usando estrellas) de las temperaturas para el día. La temperatura debe imprimirse a la izquierda de la barra correspondiente y debe haber un encabezado que dé la escala de la gráfica. El intervalo de temperaturas debe ser de –30 a 120. Debido a que es difícil mostrar 150 caracteres en la pantalla, se debe pedir que cada estrella represente un intervalo de 3 grados. De esa manera, las barras serán a lo sumo de 50 caracteres de ancho. Enseguida se muestra un ejemplo parcial, que presenta el encabezado, el resultado para una temperatura negativa y la salida para varias temperaturas positivas. Observe cómo se redondean las temperaturas al número de estrellas apropiado.

```
Temperaturas para 24 horas:
    -30         0        30        60        90        120
 -20     *******|
   0            |
   1            |
   2            |*
   3            |*
   4            |*
   5            |**
  10            |***
  50            |*****************
 100            |*********************************
```

Use nombres de variables significativos, sangrado apropiado y comentarios convenientes.

2. La desviación estándar de un conjunto de datos da un sentido de la dispersión de valores dentro de su intervalo. Por ejemplo, un conjunto de calificaciones de prueba con una desviación estándar pequeña indica que la mayoría de las calificaciones de las personas estuvieron muy cerca del promedio. Así, algunos profesores emplean la desviación estándar como una manera de determinar el intervalo de valores para asignar una calificación particular.

   Diseñe y escriba un programa en C++ que lea el conjunto de calificaciones del archivo `scores.dat` y produzca su media y desviación estándar en `cout`. La fórmula para la desviación estándar es

$$s = \sqrt{\frac{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}{n(n-1)}}$$

   donde n es el número de valores y x_i representa cada uno de los valores. Así, para calcular la desviación estándar, se deben sumar los cuadrados de cada uno de los valores y también elevar al cuadrado la suma de los valores. Toda esta lectura y suma se puede hacer con un solo ciclo, después del cual se calculan la media y la desviación estándar. Asegúrese de marcar de modo apropiado el resultado. Use nombres de variables significativos, sangrado apropiado y comentarios adecuados. Pruebe por completo el programa con sus propios conjuntos de datos.
3. Usted está quemando algunos discos compactos de música para una fiesta. Ha arreglado una lista de canciones en el orden en que desea sean tocadas. Sin embargo, le gustaría aprovechar al máximo el espacio en el CD, que tiene una capacidad para 80 minutos de música, así que quiere averiguar el tiempo total para un grupo de canciones y ver qué tan bien se ajustan. Escriba un diseño y un programa en C++ como ayuda para hacer esto. Los datos están en el archivo `songs.dat`. El tiempo se introduce en segundos. Por ejemplo, si una canción dura 7 minutos y 42 segundos, los datos introducidos para esa canción serían

```
462
```

   Después de que se han leído todos los datos, la aplicación debe imprimir un mensaje que indica el tiempo restante en el CD.

   El resultado debe ser en la forma de una tabla con columnas y encabezados escritos en un archivo. Por ejemplo,

```
Número de  Tiempo de  la canción      Tiempo total
canción     Minutos     Segundos    Minutos   Segundos
---------   --------   ---------   --------  ---------
1            5          10          5         10
2            7          42         12         52
5            4          19         17         11
3            4          33         21         44
4           10          27         32         11
6            8          55         41          6
7            5           0         46          6

Hay 33 minutos y 54 segundos de espacio restantes en el CD de 80 minutos.
```

   Observe que el resultado convierte la entrada de segundos a minutos y segundos. Use nombres de variables significativos, sangrado apropiado y comentarios convenientes.
4. Un palíndromo es una frase que dice lo mismo de izquierda a derecha que de derecha a izquierda. Escriba un programa en C++ que lea una línea de `cin`, imprima sus caracteres en orden inverso en `cout`, y luego enuncie un juicio sobre si la línea de entrada es un palíndromo. Por ejemplo, aquí hay dos ejecuciones del programa:

```
Cadena de entrada: able was I ere I saw elba
able was I ere I saw elba

es un palíndromo.

Cadena de entrada: madam I'm adam
mada m'I madam

no es un palíndromo
```

Sugerencia: use la función substr dentro de un ciclo para extraer e imprimir caracteres uno por uno de la cadena, empezando en el extremo final de la cadena, y al mismo tiempo extraer el carácter correspondiente comenzando en el inicio de la cadena para compararla.

Use un mensaje de persuasión, nombres de variables significativos, sangrado apropiado y comentarios convenientes. Pruebe por completo el programa con sus propios conjuntos de datos.

5. Usted trabaja para una compañía que está elaborando una lista de correos electrónicos a partir de mensajes de correo. La compañía desea que usted escriba un programa que lea un archivo llamado `mail.dat`, y que produzca toda cadena que contenga el signo @ en el archivo `addresses.dat`. Para el propósito de este proyecto, una cadena se define tal como es por el lector de flujo de C++, una cadena contigua de caracteres que no son espacios en blanco. Dados los datos:

```
From:  sharon@marzipan.edu
Date: Wed, 13 Aug 2003 17:12:33 EDT
Subject: Re: hi
To:  john@meringue.com

John,

Dave's email is dave_smith@icing.org.

ttyl,

sharon
```

Entonces el programa produciría en el archivo `addresses.dat`:

```
sharon@marzipan.edu
john@meringue.com
dave_smith@icing.org.
```

Use un mensaje de persuasión, nombres de variables significativos, sangrado apropiado y comentarios convenientes. Pruebe por completo el programa con sus propios conjuntos de datos.

Seguimiento de caso práctico

1. La primera versión del programa Acoustic recuerda el número de lectura de la lectura más alta. Si hay varias lecturas con el mismo valor, recuerda sólo la primera lectura. Cambie el programa de modo que recuerde el último de estos valores. Sugerencia: sólo necesita agregar un carácter al programa.
2. La segunda versión del programa Acoustic hace un seguimiento de la depresión más baja en las lecturas. Si hay dos depresiones iguales, recuerda la primera. Cambie el programa de modo que recuerde la última.
3. ¿El ciclo del programa Acoustic usa una lectura principal?

4. ¿Qué tipo de ciclo, controlado por suceso o por conteo, se emplea en el programa Acoustic?
5. El programa expresa que las lecturas están en un archivo. De igual manera podrían haberse introducido desde el teclado. Cambie el programa revisado de modo que los valores se introduzcan en tiempo real.
6. ¿Cómo podría determinar si es mejor introducir datos desde un archivo o desde el teclado?
7. Analice cómo podría diseñar un plan de prueba para el programa Acoustic.

CAPÍTULO

7

Funciones

Objetivos

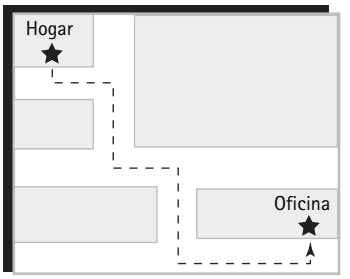
Objetivos de conocimiento

- *Conocer cómo es posible usar las funciones para reflejar la estructura de una descomposición funcional.*
- *Entender la diferencia entre parámetros por valor y por referencia.*
- *Saber cómo usar argumentos y parámetros.*

Objetivos de habilidades

Ser capaz de:

- *Escribir un módulo de su propio diseño como una función void.*
- *Diseñar la lista de parámetros para cada módulo de una descomposición funcional.*
- *Codificar un programa por medio de funciones.*
- *Definir y usar variables correctamente.*
- *Escribir un programa que emplee múltiples llamadas para una sola función.*



Usted ha estado usando funciones de C++ desde que se introdujeron las rutinas de biblioteca estándar como `sqrt` y `abs` en el capítulo 3. Por ahora, debe estar muy cómodo con la idea de llamar estos subprogramas para efectuar una tarea. Hasta aquí, no se ha considerado cómo el programador puede crear sus propias funciones que no sean `main`. Éste es el tema de este capítulo y el siguiente.

Quizá se pregunte por qué hubo que esperar hasta ahora para examinar subprogramas definidos por el usuario. La razón, y el propósito principal para usar subprogramas, es que se escriben las propias funciones de devolución y funciones void para ayudar a organizar y simplificar programas más grandes. Hasta ahora, los programas han sido relativamente pequeños y simples, así que no era necesario escribir subprogramas. Ahora, una vez cubiertas las estructuras de control básicas, se está listo para introducir subprogramas de modo que sea posible escribir programas más grandes y complejos.

7.1 Descomposición funcional con funciones void

Como un breve recordatorio, revisaremos los dos tipos de subprogramas con que funciona el lenguaje C++: funciones de devolución de valor y funciones void (no devuelven valor). Una función de devolución de valor recibe algunos datos a través de su lista de argumentos, calcula un solo valor de función y devuelve este valor de función al código de llamada. El invocador invoca (llama) a una función de devolución de valor por medio de su nombre y una lista de argumentos en una expresión:

```
y = 3.8 * sqrt(x);
```

En contraste, una función void (*procedimiento*, en algunos lenguajes) no devuelve un valor de función. Tampoco es llamada desde dentro de una expresión. En cambio, la llamada de función aparece como una sentencia independiente completa. Un ejemplo es la función `get` relacionada con las clases `istream` e `ifstream`:

```
cin.get(inputChar);
```

En este capítulo, se centra la atención de manera exclusiva en crear nuestras propias funciones void. En el capítulo 8 se examina cómo escribir funciones de devolución de valor.

Desde los primeros capítulos usted ha estado diseñando sus programas como colecciones de módulos. Muchos de estos módulos se ponen en práctica como *funciones void definidas por el usuario*. Ahora se considera cómo cambiar los módulos de sus algoritmos en funciones void definidas por el usuario.

Cuándo usar funciones

En general, se puede codificar cualquier módulo como una función, aunque algunos son tan simples que esto resulta innecesario. Al diseñar un programa, con frecuencia se requiere decidir qué módulos deben ejecutarse como funciones. La decisión se debe basar en si el programa global es más fácil de entender como un resultado. Otros factores pueden afectar esta decisión, pero por ahora éste es el método heurístico (estrategia) a usar.

Si un módulo es de una sola línea, por lo común es mejor escribirlo directamente en el programa. Convertirlo en una función sólo complica el programa global, lo cual frustra el propósito de usar subprogramas. Por otro lado, si un módulo consta de muchas líneas, es más fácil entender el programa si el módulo se convierte en una función.

Recuerde que si elige codificar un módulo como una función o no afecta la legibilidad del programa, puede hacer que sea más o menos conveniente cambiar el programa después. Su elección no afecta el funcionamiento correcto del programa.

Escritura de módulos como funciones void

Es muy simple convertir un módulo en una función void en C++. Básicamente, una función void se parece a la función `main`, excepto que el encabezado de función emplea `void` en lugar de `int` como el tipo de datos de la función. Además, el cuerpo de una función void no contiene una sentencia como

```
return 0;
```

como lo hace `main`. Una función void no devuelve un valor de función a su invocador.

Considérese un programa con funciones void. Un amigo suyo vuelve de un largo viaje y usted quiere escribir un programa que imprima el siguiente mensaje:

```
********************
********************
 ¡Bienvenido a casa!
********************
********************
********************
********************
```

Aquí está un diseño para el programa.

Main **Nivel 0**

Imprimir dos líneas de asteriscos
Imprimir "¡Bienvenido a casa!"
Imprimir cuatro líneas de asteriscos

Imprimir dos líneas **Nivel 1**

Imprimir"**********"
Imprimir"**********"

Imprimir cuatro líneas

```
Imprimir"***************"
Imprimir"***************"
Imprimir"***************"
Imprimir"***************"
```

Si se escriben dos módulos de primer nivel como funciones void, la función `main` es simplemente

```
int main()
{
    Print2Lines();
    cout << "¡Bienvenido a casa!" << endl;
    Print4Lines();
    return 0;
}
```

Observe cuán similar a este código es el módulo principal de la descomposición funcional. Contiene dos llamadas de función, una para una función nombrada `Print2Lines` y otra para la función denominada `Print4Lines`. Ambas funciones tienen listas de argumentos vacías.

El siguiente código debe parecerle familiar, pero observe con atención el encabezado de la función.

```
void Print2Lines()                          // Encabezado de función
{
    cout << "***************" << endl;
    cout << "***************" << endl;
}
```

Este segmento es una *definición de función*. Una definición de función es el código que va del encabezado de la función al final del bloque que es el cuerpo de la función. El encabezado de la función comienza con la palabra `void`, que indica al compilador que no es una función de devolución de valor. El cuerpo de la función ejecuta algunas sentencias ordinarias y *no* termina con una sentencia `return` para devolver un valor de función.

Ahora observe de nuevo el encabezado de función. Al igual que cualquier otro identificador en C++, el nombre de una función no puede incluir espacios en blanco, aunque sea en los nombres de módulo de papel y lápiz. Después del nombre de la función está una lista de argumentos vacía; es decir, no hay nada entre los paréntesis. Más adelante se ve lo que va dentro de los paréntesis si una función emplea argumentos. Ahora se juntará `main` y las otras dos funciones para formar un programa completo.

```
//*********************************************************************
// Programa Bienvenido
// Este programa imprime un mensaje "¡Bienvenido a casa!"
//*********************************************************************
#include <iostream>

using namespace std;

void Print2Lines();                         // Prototipos de función
void Print4Lines();
```

```
int main()
{
    Print2Lines();                          // Llamada de función
    cout << "¡Bienvenido a casa!" << endl;
    Print4Lines();                          // Llamada de función
    return 0;
}

//*************************************************************

void Print2Lines()                          // Encabezado de función

// Esta función imprime dos líneas de asteriscos

{
    cout << "***************" << endl;
    cout << "***************" << endl;
}

//*************************************************************

void Print4Lines()                          // Encabezado de función

// Esta función imprime cuatro líneas de asteriscos

{
    cout << "***************" << endl;
    cout << "***************" << endl;
    cout << "***************" << endl;
    cout << "***************" << endl;
}
```

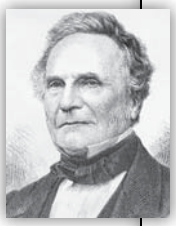
Las definiciones de función de C++ pueden aparecer en cualquier orden. Se podría haber elegido colocar la función `main` al último en vez de en primer lugar, pero los programadores de C++ colocan, por lo común, primero `main` y después cualquier función de apoyo.

En el programa Bienvenido, las dos sentencias antes de la función `main` se denominan *prototipos de función*. Estas declaraciones son necesarias debido a la regla de C++ requerida para declarar un identificador antes de poder usarlo. La función `main` utiliza dos identificadores `Print2Lines` y `Print4Lines`, pero las definiciones de estas funciones no aparecen sino hasta después. Se deben suministrar los prototipos de función para informar por adelantado al compilador que `Print2Lines` y `Print4Lines` son nombres de funciones, que no devuelven valores de función y que no tienen argumentos. Más adelante, en este capítulo, se amplía la información acerca de los prototipos de función.

Debido a que el programa Bienvenido es simple en principio, podría parecer más complicado con sus módulos escritos como funciones. Sin embargo, es evidente que cada vez se asemeja más a la descomposición funcional. Esto es especialmente cierto para la función `main`. Si usted maneja este código para alguien, la persona podría examinar la función `main` (la cual, como se dijo, normalmente aparece primero) y decirle de inmediato lo que hace el programa, imprime dos líneas de algo, imprime "¡Bienvenido a casa!" e imprime cuatro líneas de algo. Si pide a la persona que sea más específica, podría revisar los detalles de las otras definiciones de función. La persona puede comenzar con una revisión superficial del programa y luego estudiar los módulos de nivel inferior, según sea necesario, sin tener que leer todo el programa o examinar una gráfica de estructura de módulos. A medida que los programas crecen e incluyen muchos módulos anidados en varios niveles, la capacidad de leer un programa de la misma manera que una descomposición funcional ayuda en gran medida en el desarrollo y el proceso de depuración.

Conozca a

Charles Babbage



En general, se atribuye al matemático británico Charles Babbage (1791-1871) el diseño de la primera computadora del mundo. Sin embargo, a diferencia de las computadoras electrónicas actuales, la máquina de Babbage fue mecánica. Estaba hecha con engranes y palancas, la tecnología predominante de las décadas de 1820 y 1830.

Babbage en realidad diseñó dos máquinas diferentes. La primera, denominada Máquina de diferencias, se usaba para calcular tablas matemáticas. Por ejemplo, la máquina de diferencias podía producir una tabla de cuadrados:

x	x^2
1	1
2	4
3	9
4	16
⋮	⋮

Básicamente era una calculadora compleja que no podía ser programada. La máquina de diferencias de Babbage fue diseñada para mejorar la exactitud del cálculo de tablas, no la velocidad. En este entonces, todas las tablas se elaboraban a mano, un trabajo tedioso propenso a errores. Debido a que gran parte de la ciencia y la ingeniería dependían mucho de la información de tablas exactas, un error podía tener consecuencias graves. Aunque la máquina de diferencias podía efectuar los cálculos sólo un poco más rápido de lo que un humano era capaz, lo hacía sin errores. De hecho, una de sus características más importantes era que estamparía su resultado directamente en placas de cobre, que podían ser colocadas entonces en una prensa de impresión y, por tanto, evitar errores tipográficos.

En 1833, el proyecto de fabricar la máquina de diferencias se encontró con problemas financieros. El ingeniero que contrató Babbage para fabricarla era deshonesto y había alargado el proyecto lo más que pudo para cobrar la mayor cantidad posible de dinero de los patrocinadores de Babbage en el gobierno británico. Finalmente, los patrocinadores se cansaron de esperar y retiraron su apoyo. Al mismo tiempo, Babbaje perdió interés en el proyecto porque había desarrollado la idea para una máquina más eficiente, a la que denominó Máquina analítica, una computadora verdaderamente programable.

La idea para la máquina analítica llegó a Babbage cuando viajó a Europa a estudiar la mejor tecnología de la época en la preparación para fabricar la máquina de diferencias. Una de las tecnologías que vio fue el telar automático de Jacquard, en el que una serie de tarjetas de papel con orificios perforados alimentaba a la máquina para producir un patrón de tela tejido. El patrón de agujeros era un programa para el telar y permitía tejer de manera automática patrones de cualquier grado de complejidad. De hecho, su inventor tenía incluso un retrato detallado de él mismo que había sido tejido en una de sus máquinas.

Babbage comprendió que esta clase de dispositivo podía usarse para controlar la operación de una máquina de cómputo. En lugar de calcular sólo un tipo de fórmula, la máquina podría ser programada para efectuar cálculos de complejidad mucho mayor, incluso el manejo de símbolos algebraicos. Como lo dijo de modo elegante su asociada, Ada Lovelace (la primera programadora del mundo), "Se puede decir acertadamente que la máquina analítica teje patrones algebraicos". Es evidente que Babbage y Lovelace comprendían todo el poder de una computadora programable e incluso contemplaron la noción de que algún día esta clase de máquinas podrían llegar a pensar de modo artificial.z

(continúa) ▼

Charles Babbage

Desafortunadamente, Babbage nunca completó la fabricación de ninguna de sus máquinas. Algunos historiadores creen que nunca las terminó porque la tecnología de su época no podía apoyar maquinaria tan compleja. Pero la mayoría considera que la falla de Babbage fue su propio proceder. Él era brillante y un tanto excéntrico (se sabe, por ejemplo, que tenía temor al sonido de los organillos italianos). En consecuencia, tenía la tendencia de dejar inconclusos los proyectos para concentrarse en nuevas ideas. Siempre creyó que sus nuevos métodos le permitirían completar una máquina en menos tiempo de lo que sería con sus antiguas ideas.

Cuando murió, Babbage tenía muchas piezas de máquinas calculadoras y dibujos parciales de diseños, pero ninguno de los planes era lo suficientemente completo para producir una sola computadora práctica. Después de su muerte, sus ideas fueron desechadas y sus inventos ignorados. Sólo después de que se desarrollaron las computadoras modernas los historiadores reconocieron la verdadera importancia de sus contribuciones. Babbage reconoció el potencial de la computadora un siglo antes de que se desarrollara por completo. En la actualidad, sólo se puede imaginar cuán diferente sería el mundo si hubiera logrado construir su máquina analítica.

7.2 Resumen de las funciones definidas por el usuario

Ahora que se ha visto un ejemplo de cómo se escribe un programa con funciones, se considerará de manera breve e informal algunos de los puntos más importantes de la construcción y uso de funciones.

Flujo de control en llamadas de función

Se dijo que las definiciones de función de C++ se pueden disponer en cualquier orden, aunque por lo común `main` aparece primero. Durante la compilación, las funciones se traducen en el orden en que aparecen físicamente. Cuando se ejecuta el programa el control comienza en la primera sentencia de la función `main`, y el programa procede en secuencia lógica. Cuando se encuentra una llamada de función, el control lógico se pasa a la primera sentencia en el cuerpo de esa función. Las sentencias de la función se ejecutan en orden lógico. Después de que se ejecutó la última, el control vuelve al punto inmediatamente después de la llamada de función. Debido a que las llamadas de función alteran el orden lógico de ejecución, las funciones son consideradas estructuras de control. En la figura 7-1 se ilustra el orden físico contra el orden lógico de las funciones. En la figura, las funciones A, B y C se escriben en el orden físico A, B, C, pero se ejecutan en el orden C, B, A.

En el programa Bienvenido, la ejecución comienza con la primera sentencia ejecutable de la función `main` (la llamada para la función `Print2Lines`). Cuando se llama a `Print2Lines`, el control pasa a su primera sentencia y sentencias posteriores en su cuerpo. Después que se ha ejecutado la última sentencia en `Print2Lines`, el control vuelve a la función `main` en el punto que sigue a la llamada (la sentencia que imprime "¡Bienvenido a casa!").

Parámetros de función

Al examinar el programa Bienvenido es posible observar que `Print2Lines` y `Print4Lines` son funciones muy similares. Difieren sólo en el número de líneas que imprimen. ¿Se requieren en realidad dos funciones diferentes en este programa? Quizá se debe escribir sólo una función que imprima *cualquier* número de líneas, donde el invocador (`main`) pasa como argumento la expresión "cualquier número de líneas". Aquí está una segunda versión del programa, que usa sólo una función para hacer la impresión. Se le llama Nueva bienvenida.

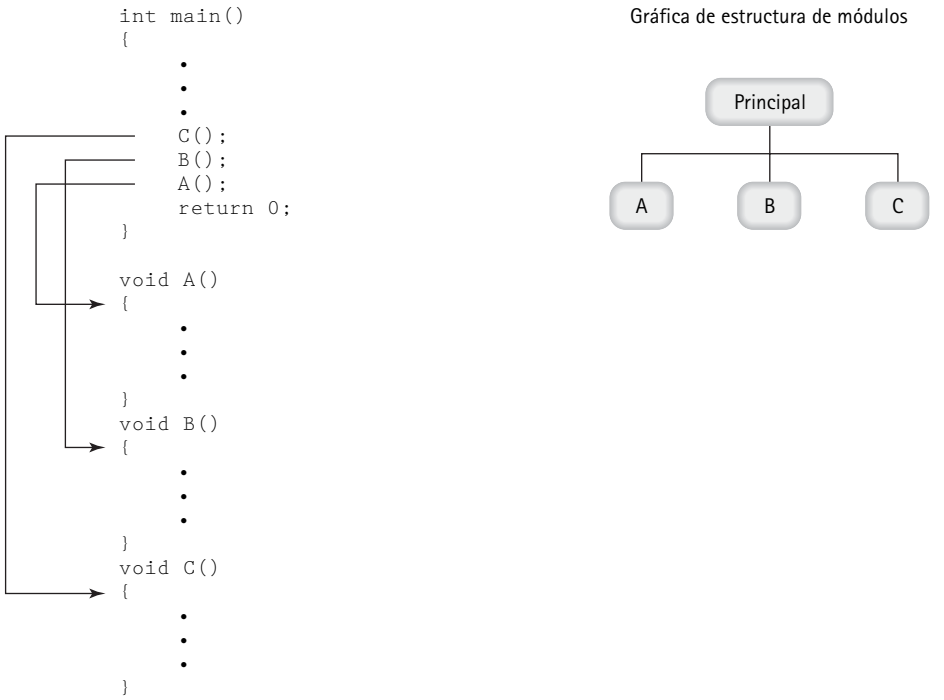


Figura 7-1 *Orden físico contra orden lógico de funciones*

```
//*****************************************************************
// Programa Nueva bienvenida
// Este programa imprime un mensaje "Bienvenido a casa"
//*****************************************************************
#include <iostream>

using namespace std;

void PrintLines( int );                    // Prototipo de función

int main()
{
    PrintLines(2);
    cout << "¡Bienvenido a casa!" << endl;
    PrintLines(4);
    return 0;
}
//*****************************************************************

void PrintLines( int numLines )

// Esta función imprime líneas de asteriscos, donde
// numLines especifica cuántas líneas se imprimen

{
    int count;      // Variable de control de ciclo

    count = 1;
```

```
    while (count <= numLines)
    {
        cout << "***************" << endl;
        count++;
    }
}
```

En el encabezado de función de `PrintLines`, se ve cierto código entre los paréntesis que se parece a una declaración de variables. Ésta es una *declaración de parámetro*. Como se analizó en capítulos anteriores, los argumentos representan una forma para que dos funciones se comuniquen entre sí. Los argumentos permiten que la función invocadora introduzca (pase) valores a otra función que usará en su proceso y, en algunos casos, para permitir que la función invocada produzca (devuelva) resultados para el invocador. Los elementos listados en la llamada para una función son los **argumentos**. Las variables declaradas en el encabezado de función son los **parámetros**. (Algunos programadores usan el par de términos *argumento real* y *argumento formal* en lugar de *argumento* y *parámetro*. Otros usan el término *parámetro real* en lugar de *argumento* y *parámetro formal* en lugar de *parámetro*.) Observe que la función `main` en el código anterior es una función sin parámetros.

Argumento Variable o expresión listada en una llamada para una función; conocida también como *argumento real* o *parámetro real*.

Parámetro Variable declarada en un encabezado de función; llamada también *argumento formal* o *parámetro formal*.

En el programa Nueva bienvenida, los argumentos en las dos llamadas de función son las constantes 2 y 4, y el parámetro en la función `PrintLines` se denomina `numLines`. La función `main` llama primero a `PrintLines` con un argumento de 2. Cuando se pasa el control a `PrintLines`, el parámetro `numLines` se inicializa en 2. Dentro de `PrintLines`, el ciclo controlado por conteo se ejecuta dos veces y regresa la función. La segunda vez que se llama a `PrintLines`, el parámetro `numLines` se inicializa en el valor del argumento, 4. El ciclo se ejecuta cuatro veces, después de lo cual vuelve la función.

Aunque no hay beneficio al proceder de esta forma, se podría escribir la función `main` de esta manera:

```
int main()
{
    int lineCount;
    lineCount = 2;
    PrintLines(lineCount);
    cout << "¡Bienvenido a casa!" << endl;
    lineCount = 4;
    PrintLines(lineCount);
    return 0;
}
```

En esta versión, el argumento en cada llamada para `PrintLines` es una variable y no una constante. Cada vez que `main` llama a `PrintLines`, se pasa una copia del valor del argumento a la función para inicializar el parámetro `numLines`. Esta versión muestra que cuando usted pasa una variable como argumento, tanto el argumento como el parámetro pueden tener nombres distintos.

El programa Nueva bienvenida trae una segunda razón importante para usar funciones; a saber, una función puede ser llamada de muchos lugares en la función `main` (o desde otras funciones). El uso de múltiples llamadas puede ahorrar gran cantidad de esfuerzo al codificar muchas soluciones de problema. Si se debe hacer una tarea en más de un lugar en un programa, se puede evitar la codificación repetitiva escribiéndola como una función y llamarla después cuando sea necesario.

Si se pasa más de un argumento a una función, los argumentos y parámetros se comparan por sus posiciones relativas en las dos listas. Por ejemplo, si desea que `PrintLines` imprima líneas que

constan de algún carácter seleccionado, no sólo asteriscos, se podría escribir la función de modo que su encabezado sea

```
void PrintLines( int  numLines,
                 char whichChar )
```

y una llamada para la función podría ser similar a esto:

```
PrintLines(3, '#');
```

El primer argumento, 3, coincide con `numLines` porque `numLines` es el primer parámetro. De igual manera, el segundo argumento, '#', concuerda con el segundo parámetro, `whichChar`.

7.3 Sintaxis y semántica de funciones void

Llamada de función (invocación)

Llamada de función (para una función void) Sentencia que transfiere el control a una función void. En C++, esta sentencia es el nombre de la función, seguido de una lista de argumentos.

Para llamar (o invocar) una función void, se usa su nombre como una sentencia, con los argumentos entre paréntesis después del nombre. Una **llamada de función** en un programa da como resultado la ejecución del cuerpo de la función invocada. Enseguida se muestra la plantilla de sintaxis de una llamada de función a una función void:

Llamada de función (a una función void)

```
Nombre de la función ( Listadeargumentos ) ;
```

De acuerdo con la plantilla de sintaxis para una llamada de función, la lista de argumentos es opcional. No es necesario que una función tenga argumentos. Sin embargo, como muestra también la plantilla de sintaxis, los paréntesis son requeridos incluso si la lista de argumentos está vacía.

Si hay dos o más argumentos en la lista, es necesario separarlos con comas. La siguiente plantilla de sintaxis es para la lista de argumentos:

Lista de argumentos

```
Expresión , Expresión . . .
```

Cuando se ejecuta una llamada de función, los argumentos se pasan a los parámetros de acuerdo con sus posiciones, izquierda a derecha, y el control se transfiere después a la primera sentencia ejecutable en el cuerpo de la función. Cuando se ha ejecutado la última sentencia de la función, el control vuelve al punto desde el cual se llamó a la función.

Declaraciones y definiciones de función

En C++ es necesario declarar todo identificador antes de poder usarlo. En el caso de funciones, una declaración de función debe preceder físicamente a cualquier llamada de función.

Una declaración de función anuncia al compilador el nombre de la función, el tipo de datos del valor de devolución de la función (ya sea `void` o un tipo de datos como `int` o `float`) y los tipos de datos de parámetros que emplea. En el programa Nueva bienvenida se muestra un total de tres declaraciones de función. La primera (la sentencia marcada como "Prototipo de función") no incluye el cuerpo de la función. Las dos declaraciones restantes, para `main` y `PrintLines`, incluyen los cuerpos de las funciones.

En la terminología de C++, una declaración de función que omite el cuerpo se llama **prototipo de función**, y una declaración que incluye el cuerpo es una **definición de función**. Se puede usar un diagrama de Venn para ilustrar que todas las definiciones son declaraciones, pero no todas las declaraciones son definiciones:

Prototipo de función Declaración de función sin el cuerpo de la función.

Definición de función Declaración de función que incluye el cuerpo de la función.

Si se está hablando de funciones o variables, la idea general en C++ es que una declaración se convierte en una definición si también asigna espacio de memoria para el elemento. (Hay excepciones a esta regla empírica, pero por ahora no se les presta atención.) Por ejemplo, un prototipo de función es sólo una declaración; es decir, especifica las propiedades de una función: nombre, tipo de datos y tipos de datos de sus parámetros. Pero una definición de función hace más; ocasiona que el compilador asigne memoria para las instrucciones del cuerpo de la función. (Desde el punto de vista técnico, las declaraciones de variable que se han empleado hasta aquí han sido *definiciones* de variable así como declaraciones, asignan memoria para la variable. En el capítulo 8 se presentan ejemplos de declaraciones de variable que no son definiciones de variable.)

La regla en C++ es que se puede declarar un elemento tantas veces como se desee, pero sólo se puede definir una vez. En el programa Nueva bienvenida se podrían incluir muchos prototipos de función para `PrintLines` (aunque no hay razón para hacerlo), pero sólo se permite una definición de función.

Prototipos de función Se mencionó que la definición de la función `main` aparece por lo común al comienzo de un programa, seguida de las definiciones de las otras funciones. Para satisfacer el requisito de que los requerimientos sean declarados antes de ser utilizados, los programadores de C++ colocan en general todos los prototipos de función cerca de la parte superior del programa, antes de la definición de `main`.

Un prototipo de función (conocido como una *declaración directa* en algunos lenguajes) especifica por adelantado el tipo de datos del valor de función que será devuelto (o la palabra void) y los tipos de datos de los parámetros. Un prototipo para una función void tiene la siguiente forma:

Prototipo de función (para una función void)

```
void Nombre de la función ( Listadeparámetro );
```

Como puede observar en la plantilla de sintaxis, no se incluye ningún cuerpo para la función, y un punto y coma termina la declaración. La lista de parámetros es opcional para considerar las funciones sin parámetros. Si está presente una lista de parámetros, tiene la siguiente forma:

Lista de parámetros (en un prototipo de función)

```
Tipo de datos & Nombre de la variable, Tipo de datos & Nombre de la variable...
```

El símbolo `&` unido al nombre de un tipo de datos es opcional y tiene una importancia especial que se atiende más adelante en este capítulo.

En un prototipo de función, la lista de parámetros debe especificar los tipos de datos de los parámetros, pero sus nombres son opcionales. Se podría escribir

```
void DoSomething( int, float );>
```

o bien,

```
void DoSomething( int velocity, float angle );
```

En ocasiones es útil, para propósitos de documentación, proporcionar nombres para los parámetros, pero el compilador los ignora.

Definiciones de función En el capítulo 2 se estudió que una definición de función consta de dos partes: encabezado de la función y cuerpo, que desde el punto de vista sintáctico es un bloque (sentencia compuesta). Aquí se muestra la plantilla de sintaxis para una definición de función; en particular, para una función void:

Definición de función (para una función void)

```
void Nombre de la función ( Lista de parámetros )
{
     Sentencia
        ⋮

}
```

Observe que el encabezado de la función *no* termina en punto y coma como lo hace un prototipo de función. Es un error de sintaxis común escribir un punto y coma al final de la línea.

La sintaxis de la lista de parámetros difiere un poco de la de un prototipo de función en que se *deben* especificar los nombres de todos los parámetros. También es preferencia de estilo (pero no un requerimiento de lenguaje) declarar cada parámetro en una línea separada:

Lista de parámetros (en una definición de función)

```
Tipo de datos & Nombre de la variable ,
Tipo de datos & Nombre de la variable
  ⋮
```

Variables locales

Variable local Una variable declarada dentro de un bloque y no accesible fuera de ese bloque.

Debido a que un cuerpo de función es un bloque, cualquier función, no sólo `main`, puede incluir declaraciones de variables dentro de su cuerpo. Estas variables se llaman **variables locales** porque son accesibles sólo dentro del bloque en que son declaradas. Respecto al código invocador, no existen. Si intenta imprimir el contenido de una variable local desde otra función, ocurrirá un error en tiempo de compilación como "UNDECLARED IDENTIFIER". En el programa Nueva bienvenida usted vio un ejemplo de una variable local: la variable `count` declarada dentro de la función `PrintLines`.

En contraste con las variables locales, las variables declaradas fuera de las funciones en un programa se denominan *variables globales*. En el capítulo 8 se retoma el tema de las variables globales.

Las variables locales ocupan memoria sólo mientras se está ejecutando la función. Al momento de llamar la función, se crea espacio de memoria para sus variables locales. Cuando vuelve la función, se destruyen sus variables locales.* Por tanto, cada vez que se llama a la función, sus va-

* En el capítulo siguiente se verá una excepción a esta regla.

riables locales comienzan con sus valores indefinidos. Debido a que toda llamada para una función es independiente de cualquier otra llamada para esa misma función, es necesario inicializar las variables locales dentro de la función misma. Y debido a que las variables locales se destruyen cuando regresa la función, no es posible usarlas para guardar valores entre llamadas para la función.

En el segmento siguiente de código se ilustra cada una de las partes de la declaración de función y el mecanismo de llamada que se ha descrito.

```
#include <iostream>

using namespace std;

void TryThis( int, int, float );        // Prototipo de función

int main()                              // Definición de función
{
    int   int1;                         // Variables locales para main
    int   int2;
    float someFloat;
    .
    .
    .
    TryThis(int1, int2, someFloat);     // Llamada de función con tres
                                        //   argumentos
    .
    .
    .
}

void TryThis( int   param1,             // Definición de función con
              int   param2,             //   tres parámetros
              float param3 )
{
    int   i;                            // Variables locales para TryThis
    float x;
    .
    .
    .
}
```

Sentencia return

La función `main` usa la sentencia

```
return 0;
```

para devolver un valor 0 (o 1, o algún otro valor) a su invocador, el sistema operativo. Toda función de devolución de valor debe regresar su valor de función de esta manera.

Una función void no devuelve un valor de función. El control regresa de la función cuando "llega" al final del cuerpo; es decir, después que se ha ejecutado la sentencia final. Como se vio en el programa Nueva bienvenida, la función `PrintLines` sólo imprime líneas de asteriscos y luego regresa.

Por otro lado, hay una segunda forma de la sentencia Return. Se parece a esto:

```
return;
```

Esta sentencia es válida *sólo* para funciones void. Puede aparecer en cualquier parte en el cuerpo de la función; ocasiona que el control salga de inmediato de la función y vuelva al invocador. He aquí un ejemplo:

```
void SomeFunc( int n )
{
    if (n > 50)
    {
        cout << "El valor está fuera del intervalo.";
        return;
    }
    n = 412 * n;
    cout << n;
}
```

En este ejemplo (sin sentido), hay dos formas para que el control salga de la función. En la entrada de la función se prueba el valor de `n`. Si es mayor que 50, la función imprime un mensaje y regresa de inmediato sin ejecutar otra sentencia. Si `n` es menor que o igual a 50, se omite la cláusula then de la sentencia If y el control avanza a la sentencia de asignación. Después de la última sentencia, el control vuelve al invocador.

Otra forma de escribir la función anterior es utilizando una estructura If-Then-Else:

```
void SomeFunc( int n )
{
    if (n > 50)
        cout << "El valor está fuera del intervalo.";
    else
    {
        n = 412 * n;
        cout << n;
    }
}
```

Si preguntara a diferentes programadores acerca de estas dos versiones de la función, obtendría opiniones distintas. Algunos prefieren la primera versión, con el argumento de que es más directo usar sentencias Return siempre que tenga sentido. Otros insisten en el método de *única entrada, única salida* en la segunda versión. Con esta manera de pensar, el control introduce una función en un punto solamente (la primera sentencia ejecutable) y sale en un punto solamente (el final del cuerpo). Argumentan que las múltiples salidas de una función desde una función dificultan seguir la lógica del programa y la depuración. Otros programadores asumen una posición entre estos dos modos de pensar, lo que permite el uso ocasional de la sentencia Return cuando la lógica es clara. Nuestra recomendación es usar return con moderación; el uso excesivo da lugar a que se confunda el código.

Cuestiones de estilo

Nombrar funciones void

Cuando elija un nombre para una función void, no olvide la apariencia que tendrán las llamadas para éste. Una llamada se escribe como una sentencia; por tanto, debe parecer una orden o instrucción para la computadora. Por esta razón, es buena idea elegir un nombre que sea un verbo imperativo o tenga un verbo imperativo como parte de él. (En español, un verbo imperativo es el que representa una orden: *¡Escucha! ¡Observa! ¡Haz algo!*) Por ejemplo, la sentencia

(continúa) ▼

Nombrar funciones void

```
Lines(3);
```

no tiene verbo para sugerir que es una instrucción. Añadir el verbo *Print* hace que el mismo nombre parezca una acción:

```
PrintLines(3);
```

Cuando esté seleccionando un nombre para una función void, escriba llamadas de muestra con nombres distintos hasta que proponga una que parezca una orden para la computadora.

Archivos de encabezado

Desde el comienzo se han usado directivas `#include` que solicitan al preprocesador de C++ que inserte el contenido de los archivos de encabezado en los programas:

```
#include <iostream>
#include <cmath>      // Para sqrt() y fabs()
#include <fstream>    // Para archivo I/O
#include <climits>    // Para INT_MAX e INT_MIN
```

¿Exactamente qué contienen estos archivos de encabezado?

Resulta que no hay nada mágico en los archivos de encabezado. Su contenido consta de una serie de declaraciones de C++. Hay tantas declaraciones de elementos como constantes (`INT_MAX, INT_MIN`), clases (`istream, ostream, string`) y objetos (`cin`, `cout`) nombrados. Pero la mayoría de los elementos de un archivo de encabezado son prototipos de función. Suponga que su programa necesita usar la función de biblioteca `sqrt` en una sentencia como la siguiente:

```
y = sqrt(x);
```

Todo identificador debe ser declarado antes de que pueda ser utilizado. Si olvida incluir (`#include`) el archivo de encabezado `cmath`, el compilador produce un mensaje de error "UNDECLARED IDENTIFIER". El archivo `cmath` contiene prototipos de función para `sqrt` y otras funciones de biblioteca orientadas a las matemáticas. Con este encabezado de archivo incluido en su programa, el compilador no sólo sabe que el identificador `sqrt` es el nombre de una función; también puede comprobar que su llamada de función es correcta respecto al número de argumentos y sus tipos de datos.

Los archivos de encabezado le ahorran el problema de escribir todos los prototipos de funciones de biblioteca al comienzo de su programa. Con sólo una línea, la directiva `#include`, puede causar que el preprocesador salga y encuentre el archivo de encabezado e inserte los prototipos en su programa. En capítulos posteriores se indicará la manera de crear sus propios archivos de encabezado que contengan declaraciones específicas para los programas.

7.4 Parámetros

Cuando se ejecuta una función, ésta usa los argumentos que se le proporcionan en la llamada de función. ¿Cómo se hace esto? La respuesta a esta pregunta depende de la naturaleza de los parámetros. C++ apoya dos clases de parámetros: **parámetros por valor** y **parámetros por referencia**. Con un parámetro por valor, que se declara sin el símbolo `&` al final del nombre de tipo de datos, la fun-

Parámetro por valor Parámetro que recibe una copia del valor del argumento correspondiente.

Parámetro por referencia Parámetro que recibe la ubicación (dirección de memoria) del argumento del invocador.

ción recibe una copia del valor del argumento. Con un parámetro por referencia, que se declara al añadir el símbolo `(&)` al nombre del tipo de datos, la función recibe la ubicación (dirección de memoria) del argumento del invocador. Antes de examinar en detalle la diferencia entre estas dos clases de parámetros, consideremos un ejemplo de un encabezado de función con una mezcla de declaraciones de parámetros por referencia y por valor.

```
void Example( int&  param1,       // Un parámetro por referencia
              int   param2,       // Un parámetro por valor
              float param3 )      // Otro parámetro por valor
```

Con tipos de datos simples —`int, char, float,` etcétera—, un parámetro por valor es la clase por omisión (supuesta) del parámetro. En otras palabras, si usted no hace nada especial (añadir un símbolo `&`), se supone que un parámetro es un parámetro por valor. Para especificar un parámetro por referencia, tiene que salir de su camino para hacer algo extra (anexar el carácter `&`).

Se examinarán ambas clases de parámetros, empezando por los parámetros por valor.

Parámetros por valor

En el programa Nueva bienvenida, el encabezado de la función `PrintLines` es

```
void PrintLines( int numLines )
```

El parámetro `numLines` es un parámetro por valor porque su tipo de datos no termina en `&`. Si la función se llama por medio de un argumento `lineCount`,

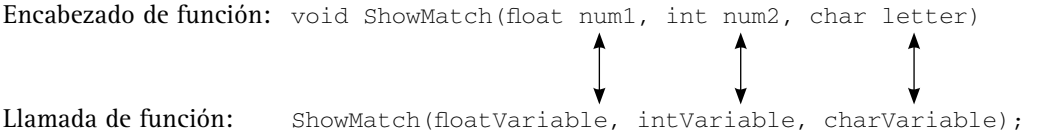
```
PrintLines(lineCount);
```

entonces el parámetro `numLines` recibe una copia del valor de `lineCount`. En este momento hay dos copias de los datos, una en el argumento `lineCount` y otra en el parámetro `numLines`. Si una sentencia dentro de la función `PrintLines` fuera a cambiar el valor de `numLines`, este cambio no afectaría el argumento `lineCount` (recuerde: hay dos copias de los datos). Usar parámetros por valor ayuda, por tanto, a evitar cambios no intencionales en los argumentos.

Debido a que a los parámetros por valor se les transmiten copias de sus argumentos, cualquier cosa que tenga un valor se podría pasar a un parámetro por valor. Esto incluye constantes, variables e incluso expresiones arbitrariamente complicadas. (La expresión simplemente se evalúa y se envía una copia del resultado al parámetro por valor correspondiente.) Para la función `PrintLines` son válidas las siguientes llamadas de función:

```
PrintLines(3);
PrintLines(lineCount);
PrintLines(2 * abs(10 - someInt));
```

Debe haber el mismo número de argumentos en una llamada de función que de parámetros en el encabezado de la función.* También, cada argumento debe tener el mismo tipo de datos que el parámetro en la misma posición (el tipo de datos de cada argumento mostrado a continuación es lo que se supondría a partir de su nombre):



Si los elementos comparados no son del mismo tipo de datos, tiene lugar la coerción implícita de tipos. Por ejemplo, si un parámetro es de tipo `int`, un argumento que sea una expresión `float` se coerciona como un valor `int` antes de pasar a la función. Como es usual en C++, puede evitar la

* Esta sentencia no es la verdad completa. C++ tiene una característica de lenguaje especial, parámetros por omisión, que permiten llamar una función con menos argumentos que parámetros. En este libro no se estudian los parámetros por omisión.

coerción de tipos no planeada por medio de una conversión explícita de tipos, o mejor aún, no combinando tipos de datos en absoluto.

Según se señaló, un parámetro por valor recibe una copia de un argumento y, por tanto, no se puede tener acceso directo al argumento del invocador o cambiarlo. Cuando regresa una función, se destruye el contenido de sus parámetros por valor, junto con el contenido de sus variables locales. La diferencia entre parámetros por valor y variables locales es que los valores de las variables locales no están definidos cuando comienza a ejecutarse una función, mientras que los parámetros por valor se inicializan de manera automática en los valores de los argumentos correspondientes.

Debido a que el contenido de los parámetros por valor se destruye cuando regresa la función, no pueden emplearse para devolver información al código invocador. ¿Qué pasa si se desea devolver información modificando los argumentos del invocador? Se debe usar la segunda clase de parámetro disponible en C++: parámetros por referencia. A continuación se considerarán éstos.

Parámetros por referencia

Un parámetro por referencia es el que se declara al unir un carácter al nombre de su tipo de datos. Se llama parámetro por referencia porque la función llamada puede referirse directamente al argumento correspondiente. En particular, se permite que la función inspeccione *y modifique* el argumento del invocador.

Cuando se invoca una función por medio de un parámetro por referencia, es la *ubicación* (dirección de memoria) del argumento, no su valor, lo que se pasa a la función. Sólo hay una copia de la información, y la utilizan tanto el invocador como la función llamada. Cuando se llama una función, el argumento y el parámetro se vuelven sinónimos para la misma ubicación en la memoria. Cualquiera que sea el valor que deje la función llamada en este lugar es el valor que el invocador encontrará allí. Por tanto, se debe tener cuidado al usar un parámetro por referencia porque cualquier cambio que se le haga afecta al argumento en el código invocador. Considérese un ejemplo.

En el capítulo 5 se elaboró el programa Actividad que lee una temperatura dada por el usuario e imprime la actividad recomendada. Aquí se muestra su diseño.

Principal **Nivel 0**

```
Obtener la temperatura
Imprimir actividad
```

Obtener la temperatura **Nivel 1**

```
Solicitar al usuario la temperatura
Leer la temperatura
Imprimir por eco la temperatura
```

Imprimir actividad

```
Imprimir "La actividad recomendada es"
SI la temperatura > 85
    Imprimir "nadar."
DE OTRO MODO SI la temperatura > 70
    Imprimir "tenis."
DE OTRO MODO SI la temperatura > 32
    Imprimir "golf."
DE OTRO MODO SI la temperatura > 0
    Imprimir "esquiar."
DE OTRO MODO
    Imprimir "bailar."
```

Se escribirán los dos módulos de nivel 1 como funciones void, `GetTemp` y `PrintActivity`, de modo que la función `main` tenga la apariencia del módulo principal de la descomposición funcional. Aquí se presenta el resultado del programa.

```
//**************************************************************************
// Programa Actividad
// Este programa produce una actividad apropiada
// para una temperatura específica
//**************************************************************************
#include <iostream>

using namespace std;

void GetTemp( int& );                          // Prototipos de función
void PrintActivity( int );

int main()
{
    int temperature;    // Temperatura exterior

    GetTemp(temperature);                      // Llamada de función
    PrintActivity(temperature);                // Llamada de función
    return 0;
}

//**************************************************************************

void GetTemp( int& temp )                      // Parámetro por referencia

// Esta función solicita una temperatura que será introducida,
// lee el valor de entrada en temp y la imprime por eco
{
    cout << "Introduzca la temperatura exterior:" << endl;
    cin >> temp;
    cout << "La temperatura actual es " << temp << endl;
}

//**************************************************************************

void PrintActivity( int temp )                 // Parámetro por valor

// Dado el valor de temp, esta función imprime un mensaje
// que indica una actividad apropiada

{
    cout << "La actividad recomendada es ";
    if (temp > 85)
        cout << "nadar." << endl;
    else if (temp > 70)
        cout << "tenis." << endl;
    else if (temp > 32)
        cout << "golf." << endl;
    else if (temp > 0)
```

```
        cout << "esquiar." << endl;
    else
        cout << "bailar." << endl;
}
```

En el programa Actividad, los dos argumentos de las llamadas de función se denominan `temperature`. El parámetro en `GetTemp` es un parámetro por referencia nombrado `temp`. El parámetro en `PrintActivity` es un parámetro por valor, denominado `temp`.

La función `main` dice a `GetTemp` dónde dejar la temperatura, dándole la ubicación de la variable `temperature` cuando hace la llamada de función. Aquí se *debe* usar un parámetro por referencia de modo que `GetTemp` sepa dónde depositar el resultado. En un sentido, el parámetro `temp` es sólo un marcador de campo en la definición de función. Cuando se llama a `GetTemp` con temperature como su argumento, todas las referencias a `temp` dentro de la función se hacen en realidad a `temperature`. Si se fuera a llamar de nuevo a la función con una variable distinta como argumento, todas las referencias a `temp` harían referencia en realidad a la otra variable hasta que la función devolviera el control a `main`.

En contraste, el parámetro `PrintActivity` es un parámetro por valor. Cuando se llama a `PrintActivity`, `main` envía una copia del valor de `temperature` para que trabaje la función. Es apropiado usar un parámetro por valor en este caso porque se supone que `PrintActivity` no modifica el argumento `temperature`.

Debido a que los argumentos y parámetros pueden tener nombres distintos, es posible llamar una función en distintos tiempos con diferentes argumentos. Suponga que se desea cambiar el programa Actividad a fin de que imprima una actividad para las temperaturas externa e interna. Sería posible declarar variables enteras en la función `main` denominada `indoorTemp` y `outdoorTemp`; luego, escribir el cuerpo de `main` como sigue:

```
GetTemp(indoorTemp);
PrintActivity(indoorTemp);
GetTemp(outdoorTemp);
PrintActivity(outdoorTemp)
return 0;
```

En `GetTemp` y `PrintActivity`, los parámetros recibirían valores de o pasarían valores a, `indoorTemp` u `outdoorTemp`.

En la tabla siguiente se resume el uso de argumentos y parámetros.

Elemento	Uso
Argumento	Aparece en una *llamada* de función. El parámetro correspondiente podría ser un parámetro por referencia o valor.
Parámetro por valor	Aparece en un *encabezado* de función. Recibe una *copia* del valor del argumento correspondiente.
Parámetro por referencia	Aparece en un *encabezado* de función. Recibe la *dirección* del argumento correspondiente.

Una analogía

Antes de hablar más acerca de pasar parámetros, se examinará una analogía de la vida cotidiana. Usted está en la sala de exposición local de catálogos de descuento para comprar el regalo del día del padre. Para hacer su pedido llenará una solicitud. Ésta tiene áreas para escribir la cantidad de cada artículo y su número de producto, y zonas donde el encargado de los pedidos escribirá los precios.

Usted escribe lo que quiere y entrega la forma al encargado. Espera que la persona compruebe si los artículos están en existencia y calcula el costo. El encargado le devuelve la solicitud y usted ve que los artículos están disponibles y que el precio es de $48.50. Usted paga al encargado y prosigue con sus actividades.

Esto ilustra la manera en que trabajan las llamadas de función. El encargado es como una función void. Usted, actuando como la función `main`, le pide que haga algún trabajo. Usted le proporciona cierta información: los números de los artículos y las cantidades. Éstos son los parámetros de entrada del encargado. Usted espera hasta que le devuelve cierta información: la disponibilidad de los artículos y sus precios. Éstos son los parámetros de salida del encargado. El encargado realiza esta tarea todo el día con diferentes valores de entrada. Cada solicitud activa el mismo proceso. El comprador espera hasta que el encargado devuelve información con base en la entrada específica.

La solicitud es análoga a los argumentos de una llamada de función. Los espacios en la forma representan las variables de la función `main`. Cuando usted entrega la solicitud al encargado, algunas de las áreas contienen información y otras están vacías. El encargado conserva la solicitud mientras hace su trabajo de modo que pueda llenar las áreas en blanco. Dichas áreas en blanco corresponden a parámetros por referencia; usted espera que el encargado le devuelva los resultados en los espacios.

Cuando la función `main` llama a otra función, los parámetros por referencia permiten que la función llamada tenga acceso a las variables y las cambie en la lista de argumentos. Cuando la función llamada termina, `main` continúa, haciendo uso de cualquier información nueva que deje la función llamada en las variables.

La lista de parámetros es como el conjunto de términos taquigráficos o jerga que usa el encargado para describir los espacios de la solicitud. Por ejemplo, él podría pensar en términos de "unidades", "códigos" y "recibos". Éstos son sus términos (parámetros) que en la solicitud corresponden a "cantidad", "número de catálogo" y "precio" (los argumentos). Pero no pierde tiempo leyendo los nombres en la solicitud cada vez; él sabe que el primer elemento son las unidades (cantidad), el segundo es el código (número de producto), etcétera. En otras palabras, observa sólo la posición de cada espacio de la solicitud. Así es como los argumentos se comparan con los parámetros, por sus posiciones relativas en las dos listas.

Comparación de argumentos con parámetros

Se mencionó que con los parámetros por referencia el argumento y el parámetro se convierten en sinónimos para la misma ubicación de memoria. Cuando una función devuelve el control a su invocador, se rompe el enlace entre el argumento y el parámetro. Son sinónimos sólo durante una llamada particular a la función. La única evidencia de que alguna vez ocurrió una coincidencia entre los dos es que puede haber cambiado el contenido del argumento (véase la figura 7-2).

Sólo se puede pasar una variable como argumento a un parámetro por referencia porque una función puede asignar un nuevo valor al argumento. (En contraste, recuerde que se puede pasar una expresión arbitrariamente complicada a un parámetro por valor.) Suponga que se tiene una función con el siguiente encabezado:

```
void DoThis( float val,        // Parámetro por valor
             int&  count )     // Parámetro por referencia
```

Entonces las siguientes llamadas de función son válidas.

```
DoThis(someFloat, someInt);
DoThis(9.83, intCounter);
DoThis(4.9 * sqrt(y), myInt);
```

En la función `DoThis`, el primer parámetro es de valor, así que se permite cualquier expresión como argumento. El segundo parámetro es de referencia, de modo que el argumento *debe* ser un nombre de variable. La sentencia

```
DoThis(y, 3);
```

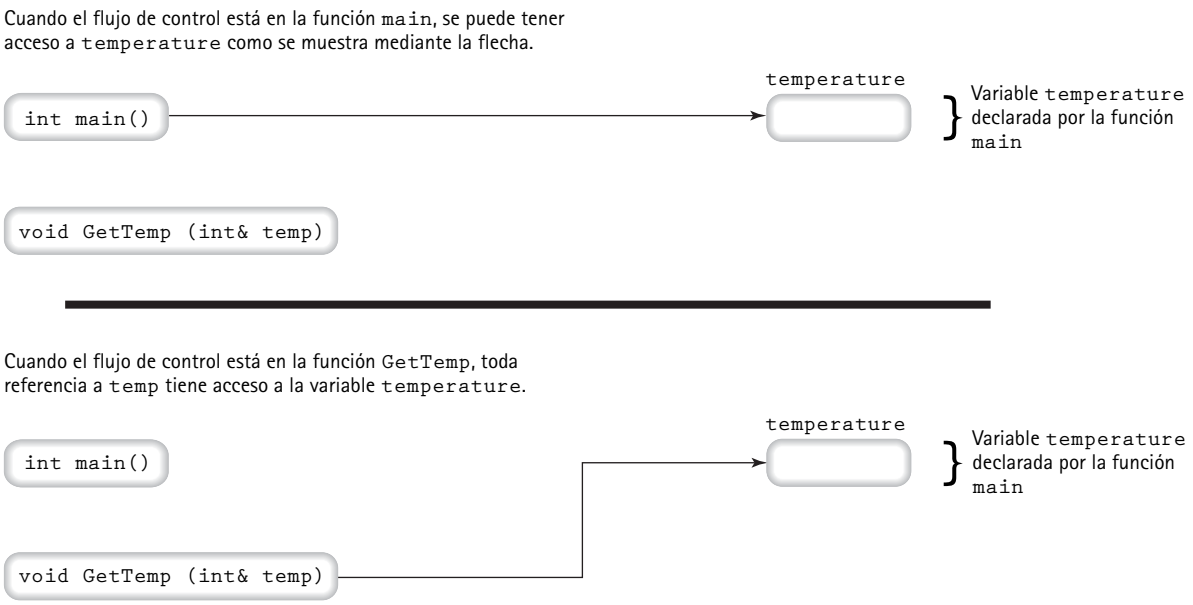


Figura 7-2 *Uso de un parámetro por referencia para tener acceso a un argumento*

genera un error en tiempo de compilación porque el segundo argumento no es un nombre de variable. Se dijo que la plantilla de sintaxis para una lista de argumentos es

Lista de argumentos

```
Expresión , Expresión . . .
```

Pero se debe recordar que Expresión está restringida a un nombre de variable si el parámetro correspondiente es un parámetro por referencia.

Hay otra diferencia importante entre parámetros por valor y de referencia cuando se tienen que comparar argumentos con parámetros. Con parámetros por valor, se dice que ocurre coerción implícita de tipos si los elementos comparados tienen diferentes tipos de datos (el valor del argumento se coerciona, si es posible, a tipo de datos del parámetro). En contraste, con los parámetros por referencia, los elementos comparados *deben* tener exactamente el mismo tipo de datos.

En la tabla siguiente se resumen las formas apropiadas de los argumentos.

Parámetro	Argumento
Parámetro por valor	Una variable, constante o expresión arbitraria (podría haber coerción de tipo)
Parámetro por referencia	Sólo *una* variable, exactamente del mismo tipo de datos que el parámetro

Por último, es responsabilidad del programador asegurarse de que la lista de argumentos y la lista de parámetros correspondan tanto desde el punto de vista semántico como del sintáctico. Por ejemplo, suponga que se ha escrito la modificación interior/exterior para el programa Actividad como sigue:

```
int main()
{
    ⋮
    GetTemp(indoorTemp);
    PrintActivity(indoorTemp);
    GetTemp(outdoorTemp);
    PrintActivity(indoorTemp)    // Argumento erróneo
    return 0;
}
```

La lista de argumentos de la última llamada de función concuerda con la lista de parámetros en su número y tipo de argumentos, así que no se señalaría ningún error de sintaxis. Sin embargo, el resultado sería erróneo porque el argumento es el valor de temperatura erróneo. De manera similar, si una función tiene dos parámetros del mismo tipo, se debe cuidar que los argumentos estén en el orden correcto. Si no es así, no se producirá ningún error de sintaxis, pero las respuestas estarán equivocadas.

Bases teóricas

Mecanismos de paso de argumentos

Hay tres formas importantes de pasar argumentos hacia y desde subprogramas. C++ apoya sólo dos de dichos mecanismos; sin embargo, es útil conocer los tres en caso de que sea necesario usarlos en otro lenguaje.

Los parámetros por referencia de C++ emplean un mecanismo denominado *paso por dirección o paso por ubicación*. Una dirección de memoria se pasa a la función. Otro nombre para esto es paso por referencia, porque la función puede referir directamente a la variable del invocador que se especifica en la lista de argumentos.

Los parámetros por valores de C++ son un ejemplo de *paso por valor*. La función recibe una copia del valor del argumento del invocador. Pasar por valor puede ser menos eficiente que pasar por dirección porque el valor de un argumento podría ocupar muchos lugares de memoria (como se ve en el capítulo 11), mientras que una dirección normalmente ocupa sólo una ubicación. Para tipos de datos simples `int`, `char`, `bool` y `float`, la eficiencia de cualquier mecanismo es casi la misma.

Un tercer método de pasar argumentos se denomina *paso por nombre*. El argumento se pasa a la función como una cadena de caracteres que debe ser interpretada mediante software de soporte en tiempo de ejecución (llamado *thunk*) provisto por el compilador. Por ejemplo, si se pasa el nombre de una variable a una función, el interpretador de tiempo de ejecución busca el nombre del argumento en una tabla de declaraciones para hallar la dirección de la variable. Pasar por nombre puede tener resultados inesperados. Si un argumento tiene la misma ortografía que una variable local en la función, esta última se referirá a la versión local de la variable en lugar de a la variable en el código invocador.

Algunas versiones de paso por nombre permiten que una expresión, o incluso segmento de código, se pase a una función. Cada vez que la función se refiere al parámetro, un intérprete efectúa la acción especificada por el parámetro. Un intérprete es similar a un compilador, y casi con la misma complejidad. Así, pasar por nombre es el menos eficiente de los tres mecanismos de paso de argumentos. Los lenguajes de programación ALGOL y LISP apoyan el paso por nombre, pero no C++.

Hay dos formas distintas de comparar argumentos con parámetros, aunque C++ apoya sólo una de ellas. La mayoría de los lenguajes de programación, C++ entre ellos, compara argumentos y parámetros por sus posiciones relativas en las listas de argumentos y parámetros. Esto se denomina *correspondencia posicional*, *correspondencia relativa* o *correspondencia implícita*. Algunos lenguajes, como Ada, apoyan también la *correspondencia explícita* o *nombrada*. En la correspondencia implícita, la lista de argumentos especifica el nombre del parámetro que se relacionará con cada argumento. La correspondencia explícita permite que los argumentos se escriban en cualquier orden en la llamada de función. La ventaja real es que cada llamada documenta con precisión qué valores se pasan a qué parámetros.

7.5 Diseño de funciones

Se han examinado algunos ejemplos de funciones y definido la sintaxis de prototipos de función y definiciones de función. Pero ¿cómo se diseñan las funciones? Primero, es necesario ser más específicos acerca de qué hacen las funciones. Se ha dicho que permiten organizar más sus programas como descomposiciones funcionales, pero, ¿cuál es la ventaja de hacer eso?

El cuerpo de una función es como cualquier otro segmento de código, excepto que está contenido en un bloque separado dentro del programa. Aislar un segmento de código en un bloque separado significa que sus detalles de ejecución pueden ocultarse. Siempre y cuando sepa cómo llamar y cuál es su propósito, puede usarla sin examinar el código dentro del cuerpo de la función. Por ejemplo, usted no sabe cómo se escribe el código para una función de biblioteca como `sqrt` (no se ve su ejecución); no obstante, puede usarlo de manera efectiva.

La especificación de lo que hace una función y cómo se invoca define su **interfaz** (véase la figura 7-3). Al ocultar una ejecución de módulo, o **encapsular** el módulo, se le pueden hacer cambios sin variar la función `main`, siempre que la interfaz sea la misma. Por ejemplo, se podría reescribir el cuerpo de una función por medio de un algoritmo más eficiente.

Interfaz Enlace de conexión en un límite compartido que permite que sistemas independientes se satisfagan y actúen o se comuniquen entre sí. También es la descripción formal del propósito de un subprograma y el mecanismo para comunicarse con él.

Encapsulación Ocultar la ejecución de un módulo en un bloque separado con una interfaz formalmente especificada.

La encapsulación es lo que se hace en el proceso de descomposición funcional cuando se pospone la solución de un subproblema difícil. Se escribe su propósito, precondición y poscondición y la información que toma y devuelve, y luego se escribe el resto del diseño como si ya se hubiera resuelto el subproblema. Se podría pasar esta especificación de interfaz a otra persona, y ésta podría desarrollar una función para nosotros que resuelva el subproblema. No es necesario saber cómo funciona, siempre y cuando cumpla con la especificación de interfaz. Las interfaces y la encapsulación son la base para la *programación en equipo*, en la cual un grupo de programadores trabaja para resolver un problema grande.

Así, el diseño de una función se puede (y debe) dividir en dos tareas: diseñar la interfaz y diseñar la implementación. Ya se sabe cómo diseñar una interfaz: es un segmento de código que corresponde a un algoritmo. Para diseñar la interfaz, se centra la atención en el *qué*, no en el *cómo*. Se debe definir el comportamiento de la función (qué hace) y el mecanismo para comunicarse con ella.

Usted sabe cómo especificar de modo formal el comportamiento de una función. Debido a que la función corresponde a un módulo, su comportamiento se define mediante la precondición y poscondición del módulo. Lo que falta es definir el mecanismo para comunicarse con la función. Para hacerlo, elabore una lista de los siguientes elementos:

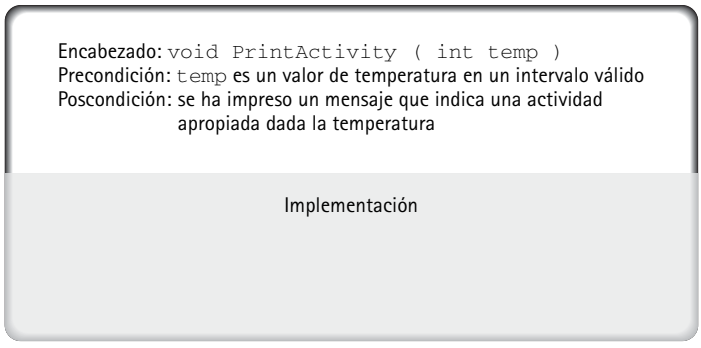


Figura 7-3 *Interfaz de función (visible) y ejecución (oculta)*

1. *Valores entrantes* que la función recibe del invocador.
2. *Valores salientes* que la función produce y devuelve al invocador.
3. *Valores entrantes y salientes:* valores que el invocador quiere que la función cambie (reciba y devuelva).

Ahora decida qué identificadores dentro del módulo corresponden con los valores de esta lista. Estos identificadores se convierten en las variables en la lista de parámetros para la función. Luego, los parámetros se declaran en el encabezado de la función. Las otras variables que necesita la función son locales y deben ser declaradas dentro del cuerpo de la función. Este proceso se repite para todos los módulos en cada nivel.

Examínese más de cerca el diseño de la interfaz. Primero se analizan las precondiciones y poscondiciones de función. Después, se considera con más detalle el concepto de parámetros entrantes, salientes, y entrantes y salientes.

Escritura de afirmaciones como comentarios de programa

Se han estado escribiendo las precondiciones y poscondiciones de módulo como afirmaciones informales. En adelante, se incluyen precondiciones y poscondiciones como comentarios para documentar las interfaces de función C++. Aquí está un ejemplo:

```
void PrintAverage( float sum,
                   int   count )

// Precondición:
//     se asigna la suma  &&  count > 0
// Poscondición:
//     La suma/cuenta promedio ha sido producida en una línea

{
    cout << "El promedio es " << sum / float(count) << endl;
}
```

La precondición es una afirmación que describe todo lo que requiere la función para ser verdadera al momento en que el invocador invoca a la función. La poscondición describe el estado del programa al momento en que termina la ejecución de la función.

Se puede considerar como un contrato a la precondición y la poscondición. El contrato afirma que si la precondición es verdadera en la entrada de función, entonces la poscondición debe ser verdadera en la salida de función. El *invocador* se encarga de asegurar la precondición, y el *código de función* debe asegurar la poscondición. Si el invocador no satisface esta parte del contrato (la precondición), este último se anula; la función no puede garantizar que la poscondición sea verdadera.

Arriba, la precondición advierte al invocador que asegure que a `sum` se le ha asignado un valor significativo y que esté seguro de que `count` es positiva. Si la precondición es verdadera, la función garantiza que ésta satisfará la poscondición. Si `count` no es positiva, cuando se invoca `PrintAverage`, el efecto de la función está indefinido. (Por ejemplo, si `count` es igual a 0, no se satisface la poscondición, y por tanto el programa falla.)

En ocasiones el invocador no necesita satisfacer ninguna precondición antes de llamar a la función. En este caso, la precondición se puede escribir como el valor `true` o simplemente se omite. En el ejemplo siguiente, no es necesaria ninguna precondición:

```
void Get2Ints( int& int1,
               int& int2 )

// Poscondición:
//     Se ha solicitado al usuario que introduzca dos enteros
```

```
//  && int1 == primer valor de entrada
//  && int2 == segundo valor de entrada

{
    cout << "Por favor introduzca dos enteros: ";
    cin >> int1 >> int2;
}
```

En afirmaciones escritas como comentarios de C++, se usa `&&` o AND para denotar el operador lógico AND, o bien `||` u `OR` para denotar un OR lógico; ! o NOT para denotar un NOT lógico, y == para denotar "igual a". (Observe que *no* se usa = para denotar "igual a". Aun cuando se escriben comentarios de programa, se desea mantener el operador == de C++ distinto del operador de asignación.)

Hay una notación final que se usa cuando se expresan afirmaciones como comentarios de programa. Las precondiciones se refieren de modo implícito a valores de variables al momento en que se invoca la función. Las poscondiciones se refieren implícitamente a valores en el momento en que regresa la función. Pero en ocasiones es necesario escribir una poscondición que se refiere a valores de parámetro que existieron al momento en que se invocó la función. Para indicar "en el momento de entrada a la función", se añade el símbolo `@entry` al final del nombre de la variable. A continuación se muestra un ejemplo del uso de esta notación. La función `Swap` intercambia o alterna el contenido de sus dos parámetros.

```
void Swap( int& firstInt,
           int& secondInt )

// Precondición:
//     se asignan firstInt y secondInt
// Poscondición:
//     firstInt == secondInt@entry
//  && secondInt == firstInt@entry

{
    int temporaryInt;

    temporaryInt = firstInt;
    firstInt = secondInt;
    secondInt = temporaryInt;
}
```

Cuestiones de estilo

Precondiciones y poscondiciones de función

Las precondiciones y poscondiciones, cuando se escriben bien, son una descripción concisa pero exacta del comportamiento de una función. Una persona que lea su programa debe poder ver —a simple vista— cómo usar la función examinando sólo su interfaz (el encabezado de la función, y la precondición y la poscondición). El lector nunca debe tener que analizar el código del cuerpo de la función para entender el propósito de la función o cómo usarla.

Una interfaz de función describe lo que hace la función, no los detalles de *cómo* lo hace. Por esta razón, la poscondición debe mencionar (por nombre) cada parámetro saliente y su valor, pero no debe mencionar ninguna variable local. Las variables locales son detalles de implementación; son irrelevantes para la interfaz de la función.

Documentar la dirección del flujo de datos

Flujo de datos Flujo de información del código de llamada a una función y de la función al código de llamada.

Otra pieza útil de documentación en una interfaz de función es la dirección del **flujo de datos** para cada parámetro de la lista. El flujo de datos es el flujo de información entre la función y su invocador. Se ha dicho que cada parámetro se puede clasificar como parámetro *entrante*, *saliente* o *entrante* y *saliente*. (Algunos programadores se refieren a éstos como parámetros de *entrada*, parámetros de *salida* y parámetros de *entrada* y *salida*.)

Para un parámetro entrante, la dirección del flujo de datos es unidireccional, hacia la función. La función inspecciona y usa el valor actual del parámetro, pero no lo modifica. En el encabezado de la función se añade el comentario

```
/* in */
```

a la declaración del parámetro. (Recuerde que los comentarios de C++ vienen en dos formas. La primera, que se usa con más frecuencia, comienza con dos barras inclinadas y se extiende hasta el final de la línea. La segunda forma encierra un comentario entre `/*` y `*/` y permite insertar un comentario dentro de una línea de código.) Enseguida se presenta la función `PrintAverage`, con comentarios añadidos a las declaraciones de parámetro:

```
void PrintAverage( /* in */ float sum,
                   /* in */ int   count )

// Precondición:
//     se asigna la suma  &&  count > 0
// Poscondición:
//     La suma/cuenta promedio ha sido producida en una línea

{
    cout << "El promedio es " << sum / float(count) << endl;
}
```

El paso por valor es apropiado para cada parámetro que es sólo entrante. Como puede ver en el cuerpo de la función, `PrintAverage` no modifica los valores de los parámetros `sum` y `count`. Solamente usa sus valores actuales. La dirección del flujo de datos es unidireccional hacia la función.

El flujo de datos para un parámetro saliente es unidireccional; sale de la función. La función produce un nuevo valor para el parámetro sin usar el valor anterior de ninguna manera. El comentario `/* out */` identifica un parámetro saliente. Aquí se han añadido comentarios al encabezado de función `Get2Ints`:

```
void Get2Ints( /* out */ int& int1,
               /* out */ int& int2 )
```

El paso por referencia se debe usar para un parámetro saliente. Si examina de nuevo el cuerpo de `Get2Ints`, verá que la función almacena nuevos valores en las dos variables (por medio de la sentencia de entrada) y remplaza cualquier valor contenido originalmente.

Por último, el flujo de datos para un parámetro entrante/saliente es bidireccional, hacia adentro y hacia afuera de la función. La función emplea el valor anterior y también produce un nuevo valor para el parámetro. Se usa `/* inout */` para documentar esta dirección de dos sentidos del flujo de datos. Enseguida se muestra un ejemplo de una función que usa dos parámetros, uno de ellos sólo entrante y el otro entrante y saliente:

```
void Calc( /* in */    int  alpha,
           /* inout */ int& beta  )
```

```
// Precondición:
//    se asignan alfa y beta
// Poscondición:
//    beta == beta@entry * 7 - alpha

{
    beta = beta * 7 - alpha;
}
```

La función inspecciona primero el valor entrante de beta para que pueda evaluar la expresión a la derecha del signo igual. Luego almacena un nuevo valor beta al usar la operación de asignación. El flujo de datos para beta es considerado, por tanto, un flujo de información bidireccional. Un paso por valor es apropiado para alpha (sólo es entrante), pero se requiere un paso por referencia para beta (es un parámetro entrante y saliente).

Cuestiones de estilo

Formato de encabezados de función

De aquí en adelante se sigue un estilo específico al codificar los encabezados de función. Los comentarios aparecen junto a los parámetros para explicar cómo se usa cada parámetro. Asimismo, los comentarios insertados indican a cuál de las tres categorías de flujo de datos pertenece cada parámetro (In, Out, Inout).

```
void Print( /* in */    float val,       // Valor que será impreso
            /* inout */ int&  count )   // Número de líneas impresas
                                        //   hasta el momento
```

Observe que el primer parámetro de arriba es un parámetro por valor. El segundo es un parámetro por referencia, tal vez porque la función cambia el valor del contador.

Se usan comentarios en la forma de renglones de asteriscos (o guiones o algún otro carácter) antes y después de una función para que sobresalga del código circundante. Cada función tiene también su propio bloque de comentarios introductorios, como los del inicio del programa, así como su precondición y poscondición.

Es importante tener mucho cuidado al documentar cada función como lo haría en la documentación al comienzo del programa.

En la tabla siguiente se resume la correspondencia entre flujo de datos de parámetro y el mecanismo apropiado de paso de argumentos.

Flujo de datos para un parámetro	Mecanismo de paso de argumentos
Entrante	Paso por valor
Saliente	Paso por referencia
Entrante y saliente	Paso por referencia

Hay excepciones a las directrices de esta tabla. C++ requiere que los objetos de flujo I/O sean pasados por referencia debido a la forma como se ponen en práctica los flujos y archivos. En el capítulo 12 se encuentra otra excepción.

Consejo práctico de ingeniería de software

Ocultación conceptual y ocultación física de una implementación de función

En muchos lenguajes de programación, la encapsulación de una implementación es puramente conceptual. Si se desea conocer cómo se ejecuta una función, se examina el cuerpo de la función. Sin embargo, C++ permite que las implementaciones de función se escriban y guarden separadas de la función `main`.

Los programas de C++ más grandes se dividen y se guardan en archivos separados en un disco. Un archivo podría contener sólo el código fuente para la función `main`; otro, el código fuente para una o dos funciones invocadas por `main`, etcétera. Esta organización se llama *programa multiarchivo*. Para traducir el código fuente en código objeto, se invoca el compilador para cada archivo sin importar los otros, tal vez en tiempos distintos. Un programa denominado *ligador*, reúne todo el código objeto resultante en un solo programa ejecutable.

Cuando escribe un programa que invoca una función localizada en otro archivo, ésta no es necesaria para que el código fuente de la función esté disponible. Todo lo que se requiere es que usted incluya un prototipo de función de modo que el compilador pueda comprobar la sintaxis de la llamada para la función. Después de que termina el compilador, el ligador encuentra el código objeto para esa función y lo liga con el código objeto de la función. Este tipo de cosas se hacen todo el tiempo cuando se invocan funciones de biblioteca. Los sistemas de C++ suministran sólo el código objeto, no el código fuente, para funciones de biblioteca como `sqrt`. El código fuente para sus ejecuciones no es físicamente visible.

Una ventaja de la ocultación física es que ayuda al programador a evitar la tentación de aprovechar cualquier característica inusual de la implementación de una función. Por ejemplo, suponga que se quiere cambiar el programa Actividad para leer temperaturas y producir actividades de manera repetida. Sabiendo que la función `GetTemp` no efectúa la comprobación de intervalo en el valor de entrada, se podría estar tentado a usar –1 000 como un centinela para el ciclo:

```
int main()
{
    int temperature;

    GetTemp(temperature);
    while (temperature != -1000)
    {
        PrintActivity(temperature);
        GetTemp(temperature);
    }
    return 0;
}
```

Este código funciona bien por ahora, pero después otro programador decide mejorar `GetTemp` para que compruebe un intervalo de temperatura válido (como debe ser):

```
void GetTemp( /* out */ int& temp )

// Esta función solicita que se introduzca una temperatura, lee
// el valor de entrada, comprueba que esté en un intervalo
// de temperatura válido y lo imprime por eco
```

(continúa) ▼

Ocultación conceptual y ocultación física de una implementación de función

```
// Poscondición:
//     Se ha solicitado al usuario un valor de temperatura (temp)
//  && Se han impreso mensajes de error y avisos adicionales
//     en respuesta a datos no válidos
//  && Si (IF) no se hallaron datos válidos antes del final del archivo
//         El valor de temp no está definido
//     ELSE
//         Se ha impreso -50 <= temp <= 30 && temp

{
    cout << "Introduzca la temperatura exterior (-50 hasta 130): ";
    cin >> temp;
    while (cin &&                          // Mientras no se llegue al final
                                              del archivo y
           (temp < -50 || temp > 130))     //   la temperatura sea no
                                              válida ...
    {
        cout << "La temperatura debe ser"
             << " -50 hasta 130." << endl;
        cout << "Introduzca la temperatura exterior: ";
        cin >> temp;
    }
    if (cin)                               // Si no se llega al final del
                                              archivo
        cout << "La temperatura actual es "
             << temp << endl;
}
```

Desafortunadamente, esta mejora ocasiona que la función `main` se inserte en un ciclo infinito porque `GetTemp` no permitirá introducir el valor centinela –1 000. Si la ejecución original de `GetTemp` se hubiera ocultado físicamente, el resultado no hubiera sido confiable sin la comprobación de errores. En cambio, la función `main` se habría escrito de manera que la afectara la mejora para `GetTemp`:

```
int main()
{
    int temperature;

    GetTemp(temperature);
    while (cin)                // Mientras no se llegue al final del archivo
    {
        PrintActivity(temperature);
        GetTemp(temperature);
    }
    return 0;
}
```

Más adelante en el libro aprenderá cómo escribir programas multiarchivo y ocultar físicamente las ejecuciones. Mientras tanto, evite escribir algún código que dependa de los trabajos internos de una función.

Caso práctico de resolución de problemas

Costo total de hipoteca

PROBLEMA En el capítulo 3 usted escribió un programa de calculadora para sus padres con el fin de ayudarlos a decidir si refinanciaban su hipoteca. Este ejercicio lo hizo pensar. Todos los anuncios, desde automóviles hasta aparatos para amueblar las casas, afirman que usted paga "sólo tanto al mes". ¿Cuánto está pagando en realidad por una casa o cualquier producto comprado a plazos? Decide insertar su calculadora en un programa que le dice cuánto en realidad está pagando sin importar lo que esté financiando. Puesto que ahora sabe usar los ciclos, decide hacer el proceso dentro de un ciclo, para que el usuario pueda ver cómo un cambio en la tasa de interés o duración del contrato afecta lo que paga en realidad.

ENTRADA Al final del capítulo 4, en el ejercicio 5 de Seguimiento de caso práctico, tuvo que cambiar su programa de calculadora para permitir que el usuario introduzca los distintos valores. Estos valores se introducen al programa.

Cantidad total del préstamo (flotante)
Tasa de interés anual (flotante)
Número de años (entero)

SALIDA Los valores de entrada se imprimen por eco en una tabla que también muestra la cantidad pagada realmente al término del préstamo.

ANÁLISIS Este programa es un ciclo simple controlado por suceso, dentro del cual se pide al usuario que introduzca la cantidad prestada, la tasa de interés y el número de años. Usted decide usar aquí un ciclo controlado por centinela, con un valor negativo para la cantidad prestada como el centinela. Esta decisión requiere que usted use una lectura principal, leer la primera cantidad prestada fuera del ciclo y las siguientes cantidades prestadas al final del ciclo.

Principal **Nivel 0**

```
Abrir el archivo para salida
Si el archivo no se abre correctamente
    Escribir un mensaje de error
    Devolver 1
Imprimir encabezado
Solicitar y leer la cantidad prestada
MIENTRAS la cantidad prestada no sea negativa
    Obtener el resto de los datos
    Determine el pago
    Imprimir los resultados
    Solicitar y leer la cantidad prestada
Cerrar el archivo de salida
```

Abrir el archivo para salida **Nivel 1**

```
dataOut.open("mortgage.out")
```

Imprimir encabezado

```
Imprimir "Cantidad del préstamo"
Imprimir "Núm. de años"
Imprimir "Pago"
Imprimir "Total pagado"
```

Solicitar y leer la cantidad prestada

```
Solicitar al usuario la cantidad del préstamo
Obtener loanAmount (cantidad prestada)
```

Obtener el resto de los datos

```
Solicitar la tasa de interés
Obtener yearlyInterest (interés anual)
Solicitar el número de años
Obtener numberOfYears (número de años)
```

Determinar el pago

```
Fijar la tasa de interés mensual en yearlyInterest/1200
Fijar el número de pagos en numberOfYears * 12
Fijar el pago en (loanAmount*pow(1 + monthlyRate ,
                 numberOfPayments)*monthlyRate )/
                 (pow(1 + monthlyRate , numberOfPayments) – 1)
```

Imprimir resultados

```
Imprimir la cantidad prestada
Imprimir el número de años
Imprimir la tasa de interés
Imprimir el pago
Imprimir (número de años*12*pago)
```

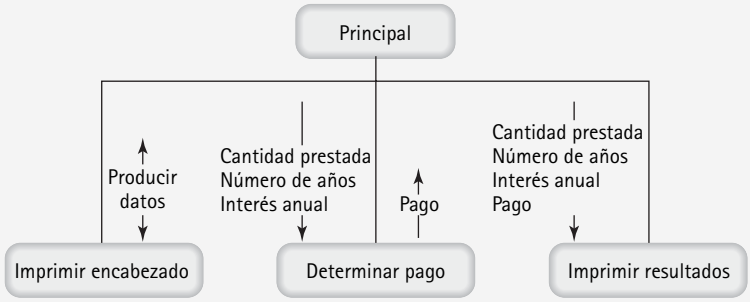
Cerrar el archivo de salida

```
dataOut.close()
```

Esto parece un diseño muy complicado para un problema tan simple. ¿En realidad es necesario representar cada módulo como una función de C++? La respuesta depende de su propio estilo. Algunos módulos tal vez no deben ser funciones; por ejemplo, OpenOutputFile y CloseOutputFile. Los módulos de operación de archivo son sólo una línea de código cada uno, así que no es necesario encapsularlos en una función de C++. Si el problema expresa que los nombres de los archivos de entrada o salida, o ambos, se leerán desde el teclado, entonces este proceso sería parte del módulo de apertura, lo cual permite representar el módulo en una función.

Si los otros módulos se ejecutan como funciones es una cuestión de estilo. Como se dijo, la pregunta es si usar una función o no hace que el programa sea más fácil de entender. La expresión "más fácil de entender" está abierta a interpretación individual. Aquí se elige hacer las funciones PrintHeading y PrintResults, pero no los otros módulos. En el ejercicio 1 de seguimiento de caso práctico se pide que ejecute los otros módulos como funciones y determine cuál es más fácil de entender para usted.

GRÁFICA DE ESTRUCTURA DE MÓDULO

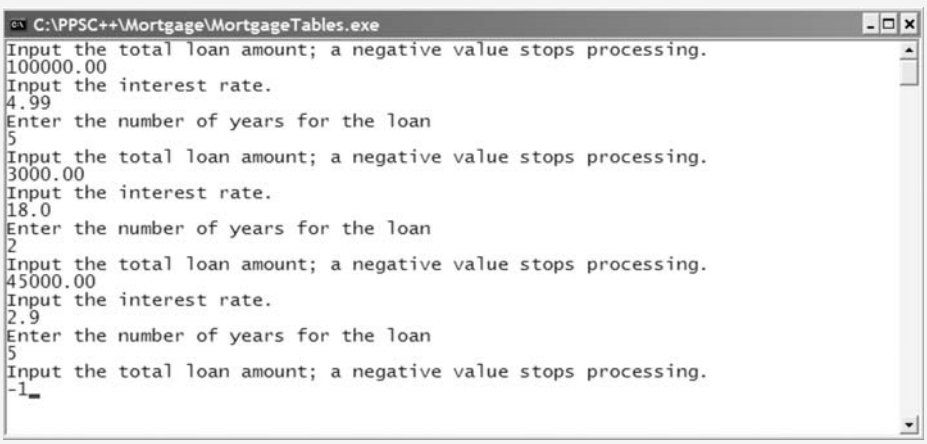



```
//*****************************************************************
// Programa Tablas de pago de hipoteca
// Este programa imprime una tabla que muestra la cantidad prestada,
// la tasa de interés, la duración del préstamo, los pagos mensuales
// y el costo total de la hipoteca.
//*****************************************************************

#include <iostream>
#include <cmath>
#include <iomanip>
#include <fstream>

using namespace std;

// Prototipos de función
void PrintHeading(ofstream&);
void DeterminePayment(float, int, float, float&);
void PrintResults(ofstream&, float, int, float, float);

int main()
{
    // Variables de entrada
    float loanAmount;
    float yearlyInterest;
    int numberOfYears;
    float payment;

    // Declarar y abrir el archivo de entrada
    ofstream dataOut;
    dataOut.open("mortgage.out");
    if ( !dataOut )
    { // no
        cout << "No se puede abrir el archivo de salida." << endl;
        return 1;
    }

    PrintHeading(dataOut);

    // Solicitar y leer la cantidad prestada
    cout << "Introducir la cantidad total prestada; "
```

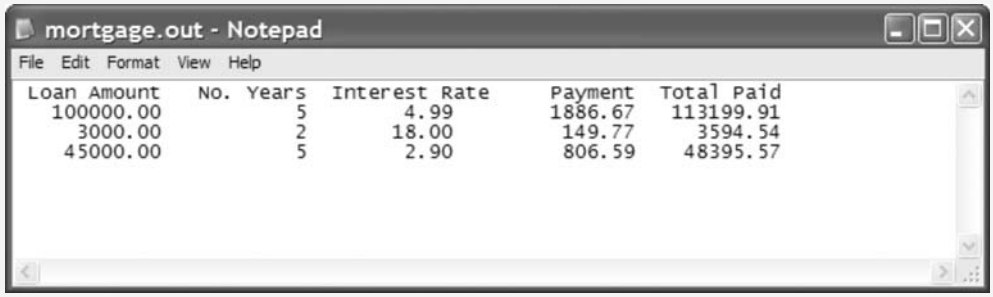
```
         << "un valor negativo detiene el proceso. " << endl;
    cin >> loanAmount;

    // Ciclo que calcula los pagos mensuales
    while (loanAmount >= 0.0)
    {
        // Solicitar y leer la tasa de interés y el número de años.
        cout << "Introducir la tasa de interés." << endl;
        cin >> yearlyInterest;
        cout << "Introducir el número de años del préstamo" << endl;
        cin >> numberOfYears;

        DeterminePayment(loanAmount, numberOfYears, yearlyInterest,
            payment);
        PrintResults(dataOut, loanAmount, numberOfYears,
            yearlyInterest, payment);
        // Solicitar y leer la cantidad prestada
        cout << "Introducir la cantidad total prestada; "
             << "un valor negativo detiene el proceso. "  << endl;
        cin >> loanAmount;
    }

    dataOut.close();
    return 0;
}
//****************************************************************

void DeterminePayment
    ( /* in */ float loanAmount,       // Cantidad prestada
      /* in */ int numberOfYears,     // Plazo del préstamo
      /* in */ float yearlyInterest,  // Tasa de interés
      /* inout */ float& payment )    // Pago mensual

// Calcula el pago mensual para una cantidad prestada con la
// fórmula para el interés compuesto.
// Precondición:
//     Se han asignado valores a los argumentos
// Poscondición:
//     el pago contiene los pagos mensuales calculados
//     mediante la fórmula del interés compuesto

{
    // variables locales
    float monthlyRate;
    int numberOfPayments;

    monthlyRate = yearlyInterest / 1200;
    numberOfPayments = numberOfYears * 12;
    payment = (loanAmount * pow(1 + monthlyRate, numberOfPayments)
               * monthlyRate) /
               (pow(1 + monthlyRate, numberOfPayments) - 1);
}

//****************************************************************
```

```
void PrintResults( /* inout */ ofstream& dataOut,     // Archivo de salida
                   /* in */    float loanAmount,      // Cantidad prestada
                   /* in */    int numberOfYears,     // Plazo del préstamo
                   /* in */    float yearlyInterest,  // Tasa de interés
                   /* in */    float payment )        // Pago

// Imprime la cantidad prestada, el número de años, la tasa de interés anual,
// y la cantidad del pago en el archivo dataOut
// Precondición:
//     El archivo dataOut ha sido abierto con éxito &&
//     Se han asignado valores a todos los argumentos
// Poscondición:
//     Se ha impreso la cantidad prestada, el número de años,
//     la tasa de interés anual y el pago en dataOut
//     con la documentación apropiada.
{
    dataOut << fixed << setprecision(2) << setw(12) << loanAmount
            << setw(12) << numberOfYears << setw(12)
            << yearlyInterest << setw(15) << payment
            << setw(12) << numberOfYears*12*payment << endl;
}

//*************************************************************************

void PrintHeading( /* inout */ ofstream& dataOut )  // Output file

// Imprime el encabezado en el archivo dataOut para cada columna de la tabla.
// Precondición:
//   El archivo dataOut se ha abierto con éxito
// Poscondición:
//   "Cantidad prestada", "Núm. de años", "Tasa de interés", "Pago",
//   El "Total pagado" se ha escrito en el archivo dataOut

{
    dataOut << fixed << setprecision(2) << setw(12) << "Cantidad prestada"
            << setw(12) << "Núm. de años" << setw(15)
            << "Tasa de interés" << setw(12) << "Pago"
            << setw(12) << "Total pagado" << endl;
}
```

Aquí se presenta una ejecución muestra y el resultado.



La imagen mostrada corresponde a la salida producida por el programa original, escrito en idioma inglés



La imagen mostrada corresponde a la salida producida por el programa original, escrito en idioma inglés

Prueba Hay tres entradas numéricas para este programa. Una cantidad de préstamo negativa termina el proceso. La ejecución de muestra usó distintos valores para la cantidad prestada, el interés y la duración del plazo. En el ejercicio 2 del Seguimiento de caso práctico se pide examinar lo que sucede si el interés o el número de años, o ambos, es cero o negativo.

Prueba y depuración

Los parámetros declarados por una función y los argumentos que el invocador pasa a la función deben satisfacer la interfaz para la función. Los errores que ocurren en las funciones se deben con frecuencia a un uso incorrecto de la interfaz entre el código invocador y la función llamada.

Una fuente de errores son las listas de argumentos comparados y las listas de parámetros. El compilador de C++ asegura que las listas tengan el mismo número de elementos y que sean compatibles en tipo. Sin embargo, es responsabilidad del programador comprobar que cada lista de argumentos contenga los elementos correctos. Esto es cuestión de comparar las declaraciones de parámetro con la lista de argumentos en cada llamada para la función. Este trabajo es mucho más fácil si el encabezado de función proporciona a cada parámetro un nombre distinto y describe su propósito en un comentario. Es posible evitar errores al escribir una lista de argumentos por medio de nombres de variables descriptivos en el código invocador para indicar exactamente qué información se pasa a la función.

Otra fuente de error consiste en no asegurarse de que la precondición para una función se satisfaga antes de que sea llamada. Por ejemplo, si una función asume que el archivo de entrada no está al final del archivo cuando se llama, entonces el código invocador debe asegurarse de que esto es verdadero antes de hacer la llamada a la función. Si una función se comporta de modo incorrecto, revise su precondición, luego siga la ejecución del programa hasta el punto de la llamada para verificar la precondición. Se puede pasar mucho tiempo tratando de localizar un error en una función incorrecta cuando el error en realidad está en la parte del programa antes de la llamada.

Si los argumentos corresponden con los parámetros y la precondición se estableció de modo correcto, entonces es más seguro que la fuente del error se localice en la función misma. Siga la función para comprobar que transforma la precondición en la poscondición apropiada. Compruebe que las variables locales se inicializan de manera adecuada. Los parámetros que se supone devuelven datos al invocador deben ser declarados parámetros por referencia (con un símbolo `&` unido al nombre del tipo de datos).

Una técnica importante para depurar una función es usar el programa depurador de su sistema, si hay uno disponible, para recorrer la ejecución de la función. Si no se cuenta con un depurador, es posible insertar sentencias de salida depuradoras para imprimir los valores de los argumentos inmediatamente antes y después de las llamadas para la función. También puede ser útil imprimir los

valores de las variables locales al final de la función. Esta información proporciona una instantánea de la función (una fotografía de su estado en un determinado momento) en sus dos puntos más críticos, lo cual es útil para comprobar seguimientos a mano.

Para probar por completo una función, debe arreglar los valores entrantes para que la precondición se lleve hasta sus límites; luego, se debe comprobar la poscondición. Por ejemplo, si una función requiere que un parámetro se localice dentro de cierto intervalo, intente llamar la función con los valores de la parte media de ese intervalo y en sus extremos.

La función de biblioteca `assert`

Se ha descrito cuán útiles son las precondiciones y poscondiciones para depurar (comprobando que la precondición de cada función es verdadera antes de una llamada de función, y verificando que cada función transforma de modo correcto la precondición en la poscondición) y para probar (al llevar la precondición hasta sus límites, e incluso violarla). Para expresar las precondiciones y poscondiciones para las funciones, se han escrito las afirmaciones como comentarios de programa:

```
// Precondición:
//     studentCount > 0
```

Los comentarios, por supuesto, son ignorados por el compilador. No son sentencias ejecutables; son para que las examinen los humanos.

Por otro lado, la biblioteca estándar de C++ proporciona un formato para escribir *afirmaciones ejecutables*. Por medio del archivo de encabezado `cassert`, la biblioteca ofrece una función void denominada `assert`. Esta función toma una expresión lógica (booleana) como un argumento y detiene el programa si la expresión es falsa. Aquí se presenta un ejemplo:

```
#include <cassert>
  ⋮
assert(studentCount > 0);
average = sumOfScores / studentCount;
```

El argumento para la función `assert` debe ser una expresión lógica válida de C++. Si su valor es `true`, nada sucede; la ejecución continúa en la siguiente sentencia. Si su valor es `false`, la ejecución del programa termina de inmediato con un mensaje que expresa: *a*) la afirmación como aparece en la lista de argumentos; *b*) el nombre del archivo que contiene el código fuente del programa, y *c*) el número de línea en el programa. En el ejemplo anterior, si el valor de `studentCout` es menor que o igual a 0, el programa se detiene después de imprimir un mensaje como el siguiente:

```
Assertion failed: studentCount > 0, file myprog.cpp, line 48
```

(Este mensaje es potencialmente confuso. No significa que `studentCount` ***sea*** mayor que 0. De hecho, es exactamente lo contrario. El mensaje indica que la afirmación `studentCount` > 0 es *falsa*.)

Las afirmaciones ejecutables tienen una ventaja importante sobre afirmaciones expresadas como comentarios: el efecto de una afirmación falsa es muy evidente (el programa termina con un mensaje de error). La función `assert` es, por tanto, valiosa en la prueba de software. Un programa que está en desarrollo podría llenarse con llamadas para la función `assert` con el fin de ayudar a identificar dónde están ocurriendo los errores. Si una afirmación es falsa, el mensaje de error proporciona el número de línea preciso de la afirmación fallida.

Además, hay una forma de "eliminar" las afirmaciones sin quitarlas en realidad. Si usa la directiva de preprocesador `#define` NDEBUG antes de incluir el archivo de encabezado `cassert`, como esto:

```
#define NDEBUG
#include <cassert>
  ⋮
```

entonces las llamadas para la función `assert` son ignoradas cuando usted ejecuta el programa. (NDEBUG representa "No depurar", y una directiva `#define` es una característica de preprocesador que por ahora no se analiza.) Después de que un programa prueba y depura, los programadores suelen "desactivar" las sentencias de depuración dejándolas presentes físicamente en el código fuente en caso de que pudieran necesitarse después. Insertar la línea `#define NDEBUG` desactiva la comprobación de afirmaciones sin tener que eliminarlas.

A pesar de ser tan útil, la función `assert`, tiene dos limitaciones. Primera, el argumento para la función se debe expresar como una expresión lógica de C++. Un comentario como

```
//  0.0 <= yearlyInterest && numberOfYears <= 30
```

se puede convertir en una afirmación ejecutable con la sentencia

```
assert(0.0 <= yearlyInterest && numberOfYears <= 30);
```

Pero no hay una manera fácil de convertir el comentario

```
//  El archivo contiene la cantidad prestada,
//  la tasa de interés y el número de año
```

en una expresión lógica de C++.

La segunda limitación es que la función `assert` es apropiada sólo para probar un programa que está en desarrollo. Un programa de producción (uno ya terminado y liberado al público) debe ser robusto y proporcionar mensajes de error útiles al usuario del programa. Ya se puede imaginar cuán desconcertado estaría un usuario si de repente terminara el programa y mostrara un mensaje de error como

```
Assertion failed: sysRes <= resCount, file newproj.cpp, line 298
```

A pesar de sus limitaciones, debe considerar la función `assert` como una herramienta regular para probar y depurar sus programas.

Sugerencias de prueba y depuración

1. Siga con cuidado las directrices de documentación al escribir funciones (véase el apéndice F). A medida que sus programas son más complejos y, por tanto, más propensos a errores, se hace cada vez más importante adherirse a estándares de documentación y formato. Incluso si el nombre de la función parece reflejar el proceso que se realiza, describa dicho proceso con comentarios. Incluya comentarios que expresen la precondición de función (si hay alguna) y la poscondición para completar la interfaz de función. Use comentarios para explicar los propósitos de los parámetros y variables locales cuyos papeles no son obvios.
2. Proporcione un prototipo de función cerca de la parte superior de su programa para cada función que haya escrito. Asegúrese de que el prototipo y su encabezado de función correspondiente coincidan *exactamente* (excepto por la ausencia de nombres de parámetro en el prototipo).
3. Asegúrese de poner un punto y coma al final de un prototipo de función. Pero *no* ponga punto y coma al final del encabezado de función en una definición de función. Dado que los prototipos de función se parecen mucho a los encabezados de función, es común ver que uno de ellos es erróneo.
4. Asegúrese que la lista de parámetros proporcione el tipo de datos de cada parámetro.
5. Use parámetros por valores a menos que un parámetro devuelva un resultado. Los parámetros por referencia pueden cambiar el contenido del argumento del invocador; los parámetros por valor no.
6. En una lista de parámetros, asegúrese que el tipo de datos de cada parámetro por referencia termina con un símbolo `&`. Sin este carácter, el parámetro es un parámetro por valor.

7. Asegúrese que la lista de argumentos de toda llamada de función corresponde con la lista de parámetros en número y orden de elementos, y sea muy cuidadoso con sus tipos de datos. El compilador identificará cualquier incompatibilidad en el número de argumentos. Pero si hay una diferencia en los tipos de datos, es posible que no haya error al momento de compilar. En particular, con paso por valor, una incongruencia de tipos puede originar coerción implícita de tipos y no un error en tiempo de compilación.
8. Recuerde que un argumento que coincide con un parámetro por referencia *debe* ser una variable, mientras que un argumento que corresponde a un parámetro por valor puede ser cualquier expresión que proporciona un valor del mismo tipo de datos (excepto como se hizo notar antes, en la sugerencia 7).
9. Familiarícese con todas las herramientas disponibles al intentar localizar las fuentes de error: el repaso de algoritmo, el seguimiento a mano, el programa depurador del sistema, la función `assert` y las sentencias de salida depuradoras.

Resumen

C++ permite escribir programas en módulos expresados como funciones. Por tanto, la estructura de un programa puede imitar su descomposición funcional aun cuando el programa es complicado. Para hacer que su función `main` se vea exactamente como el nivel 0 de su descomposición funcional, escriba cada módulo de nivel inferior como una función. La función `main` ejecuta entonces estas otras funciones en secuencia lógica.

Las funciones se comunican por medio de dos listas: la lista de parámetros (que especifica el tipo de datos de cada identificador) en el encabezado de la función y la lista de argumentos en el código invocador. Los elementos de estas listas concuerdan en número y posición, y deben coincidir con el tipo de datos.

Parte del proceso de descomposición funcional requiere determinar qué datos debe recibir un módulo de nivel inferior y qué información se debe obtener de él. Los nombres de estos datos, junto con la precondición y la poscondición de un módulo, definen su interfaz. Los nombres de los datos se convierten en la lista de parámetros y el nombre del módulo se vuelve el nombre de la función. Con las funciones void, una llamada para la función se realiza al escribir el nombre de la función como una sentencia, encerrando los argumentos apropiados entre paréntesis.

C++ tiene dos clases de parámetros: por referencia y por valor. Los parámetros por referencia tienen tipos de datos que terminan en `&` en la lista de parámetros, mientras que los parámetros por valor no. Los parámetros que devuelven valores desde una función deben ser parámetros por referencia. Los otros deben ser parámetros por valores. Esto reduce el riesgo de errores, porque sólo se pasa una copia del valor de un argumento a un parámetro por valor y, por consiguiente, el argumento está protegido de cambios.

Además de las variables declaradas en su lista de parámetros, una función puede tener variables locales dentro de ella. El acceso a estas variables es sólo dentro del bloque en que se declaran. Las variables locales se deben inicializar cada vez que sea llamada la función en la que están contenidas porque sus valores se destruyen cuando regresa la función.

En un programa se pueden llamar funciones de más de un lugar. El mecanismo de correspondencia posicional permite el uso de distintas variables como argumentos para la misma función. Llamadas múltiples para una función, desde lugares distintos y con diferentes argumentos, pueden simplificar en gran medida la codificación de muchos programas complejos.

Comprobación rápida

1. ¿Qué elementos de una descomposición funcional corresponden a funciones en C++? (pp. 252-253)
2. ¿Qué caracteres se usan para indicar un parámetro por referencia y dónde aparece en la declaración del parámetro? (pp. 265-269)
3. ¿Dónde aparecen los argumentos y dónde los parámetros? (pp. 269-272)

4. Usted está escribiendo una función para devolver el primer nombre de una cadena que contiene un nombre completo. ¿Cuántos parámetros tiene la función y cuáles de ellos son de referencia y cuales de valor? (pp. 273-277)
5. ¿Qué clase de parámetro usaría para un valor entrante que proviene de un argumento? ¿Para un valor saliente? ¿Para un valor que entra, se cambia y regresa al argumento? (pp. 273-277)
6. ¿En qué parte del programa aparecen las declaraciones de funciones que son llamadas por `main`, con respecto a `main`? (pp. 260-262)
7. ¿Qué partes de un programa pueden tener acceso a una variable local declarada dentro del bloque de una función? (pp. 262-263)
8. Si la misma función aparece en múltiples lugares en una descomposición funcional, ¿cómo la convertiría en código en un programa? (pp. 260-263)

Respuestas

1. Módulos **2.** El carácter `&` aparece al final del nombre del tipo de parámetro. **3.** Los argumentos aparecen en llamadas de función y los parámetros en encabezados de función. **4.** Debe tener dos parámetros, uno para cada cadena. El parámetro de nombre completo debe ser un parámetro por valor, y el parámetro de nombre debe ser un parámetro por referencia. **5.** Los valores entrantes usan parámetros por valor. Los valores que regresan al argumento (out o inout) se deben pasar por parámetros por referencia. **6.** Deben ser declarados antes de usarlos, así que aparecen antes de `main`. Sin embargo, la declaración podría ser simplemente un prototipo de función y la definición real puede aparecer entonces en cualquier parte. **7.** Sólo las sentencias dentro del bloque, después de la declaración. **8.** La codifica una vez, o a su prototipo, colocándola antes de cualquier referencia a ella en el resto del programa, y la llama desde cada lugar en el programa que corresponda a su aparición en la descomposición funcional.

Ejercicios de preparación para examen

1. ¿Cuáles son las tres cosas que distinguen una función `void` de una `main`?
2. Un prototipo de función debe especificar el nombre de una función y el nombre y tipo de cada uno de sus parámetros. ¿Verdadero o falso?
3. ¿Cuándo y a dónde regresa el control una función `void`?
4. Establezca una correspondencia entre los siguientes términos y las definiciones dadas a continuación.
   - ***a***) Argumento
   - ***b***) Parámetro
   - ***c***) Llamada de función
   - ***d***) Prototipo de función
   - ***e***) Definición de función
   - ***f***) Variable local
   - ***g***) Parámetro por valor
   - ***h***) Parámetro por referencia
     - ***i***) Declaración de función sin un cuerpo.
     - ***ii***) Parámetro que recibe una copia del valor del argumento.
     - ***iii***) Variable declarada en un encabezado de función.
     - ***iv***) Declaración de función con un cuerpo.
     - ***v***) Variable o expresión listada en una llamada para una función.
     - ***vi***) Sentencia que transfiere el control a una función.
     - ***vii***) Parámetro que recibe la ubicación del argumento.
     - ***viii***) Variable declarada dentro de un bloque.
5. En el siguiente encabezado de función, ¿qué parámetros son por valor y cuáles por referencia?

```
void ExamPrep (string& name, int age, float& salary,
char level)
```

6. Si una función tiene seis parámetros, ¿cuántos argumentos deben estar en una llamada para la función?

7. ¿Qué sucede si una función asigna un nuevo valor a un parámetro por valor? ¿Qué sucede si la función asigna un nuevo valor a un parámetro por referencia?
8. ¿Qué es incorrecto con este prototipo de función?

```
void ExamPrep (phone& int, name string, age& int)
```

9. Los argumentos pueden aparecer en cualquier orden siempre y cuando tengan los tipos correctos y C++ averiguará la correspondencia. ¿Verdadero o falso?
10. Defina la encapsulación.
11. ¿Para qué dirección(es) de flujo de datos usa parámetros por referencia?
12. ¿Qué es erróneo en la siguiente función?

```
void Square (int& x)
{
    x = x * x;
    return 0;
}
```

13. ¿Qué está equivocado en la siguiente función?

```
void Power (int x, int y)
{
    int result;
    result = 1;
    while (y > 0)
    {
        result = result * x;
        y--;
    }
}
```

14. ¿Cuál es el error en la siguiente función?

```
void Power (int x, int y, int result)
{
    result = 1;
    while (y > 0)
    {
        result = result * x;
        y--;
    }
}
```

15. ¿En qué está mal la siguiente función?

```
void Power (int& x, int& y, int& result)
{
    result = 1;
    while (y > 0)
    {
        result = result * x;
        y--;
    }
}
```

16. Se puede hacer referencia a una variable local en cualquier parte dentro del bloque en que se declara. ¿Verdadero o falso?
17. Las funciones pueden ser llamadas desde otras funciones además de `main`. ¿Verdadero o falso?
18. ¿Cuál sería la precondición para una función que lee un archivo de enteros y devuelve su promedio?

Ejercicios de preparación para la programación

1. Escriba el encabezado para una función void denominada `Max` que tiene tres parámetros `int`, `num1`, `num2` y `greatest`. Los dos primeros parámetros reciben datos del invocador y `greatest` devuelve un valor. Documente el flujo de datos de los parámetros con comentarios apropiados.
2. Escriba el prototipo de función para la función del ejercicio 1.
3. Escriba la definición de función de la función del ejercicio 1 para que devuelva el mayor de los dos parámetros de entrada.
4. Escriba el encabezado para una función void nominada `GetLeast` que tome un parámetro `ifstream` denominado `infile` como un parámetro de entrada que se cambie, y que tenga un parámetro `int` nombrado `lowest` que devuelva un valor. Documente el flujo de datos de los parámetros con comentarios apropiados.
5. Escriba el prototipo de función para la función del ejercicio 4.
6. Escriba la definición de función para la función del ejercicio 4, de modo que lea `infile` como una serie de valores `int` y devuelva la entrada de entero mínimo de `infile`.
7. Añada comentarios a la definición de función precedente que expresen su precondición y poscondición.
8. Escriba el encabezado para una función llamada `Reverse` que toma dos parámetros de cadena. En el segundo parámetro, la función devuelve una cadena que es el inverso de carácter por carácter de la cadena en el primer parámetro. Los parámetros se llaman `original` y `lanigiro`. Documente el flujo de parámetros con comentarios apropiados.
9. Escriba el prototipo de función para la función del ejercicio 8.
10. Escriba la definición de función para la función del ejercicio 8.
11. Añada comentarios a la definición de función precedente que expresen su precondición y poscondición.
12. Escriba una función void, `LowerCount`, que lea una línea de `cin` y devuelva un `int` (`count`) que contenga el número de letras minúsculas en la línea. En el apéndice C encontrará una descripción de la función `islower`, que devuelve `true` si su parámetro `char` es un carácter en minúscula. Documente el flujo de parámetros con comentarios apropiados.
13. Añada comentarios a la definición de función que escribió para el ejercicio 12 que expresen su precondición y poscondición.
14. Escriba una función `void`, `GetNonemptyLine`, que tome un `ifstream` (`infile`) como un parámetro de entrada y salida, y lea las líneas del archivo hasta que encuentre una línea que contenga caracteres. Debe devolver la línea vía un parámetro de cadena llamado `line`. Documente el flujo de datos de los parámetros con comentarios apropiados.
15. Escriba una función vacía, `skipToEmpityLine`, que toma un `ifstream` (`infile`) como un parámetro de entrada y salida y que lee líneas desde el archivo hasta que encuentra una línea que no contenga caracteres. Ésta debe devolver entonces el número de líneas omitidas vía un parámetro `int` llamado `skipped`. Documente el flujo de parámetros con comentarios apropiados.
16. Escriba una función `void`, `TimeAdd`, que tome parámetros que representen dos tiempos en días, horas y minutos y que los sume para obtener un nuevo tiempo. Cada parte del tiempo es un `int`, y las horas varían de 0 a 23, mientras que los minutos van de 0 a 59. No hay límite en el intervalo de los días. Se supone que el tiempo que se añadirá es positivo. Los valores de los parámetros que representan el primer tiempo se remplazan con el resultado de sumar los dos tiempos. Enseguida se muestra un ejemplo de una llamada en que 3 días, 17 horas, 49 minutos se agregan a 12 días, 22 horas y 14 minutos.

```
días = 12;
horas = 11;
```

```
minutos = 14;
TimeAdd (días, horas, minutos, 3, 17, 49)
```

Después de la llamada, los valores de las variables son

```
días = 16
horas = 16
minutos = 3
```

Documente el flujo de parámetros con comentarios apropiados.

17. Amplíe la función `TimeAdd` del ejercicio 16 para incluir segundos.
18. Escriba una función void, `SeasonPrint`, que toma parámetros `int` que representan un mes y un día y produce en `cout` el nombre de la estación. Para los objetivos de este ejercicio, la primavera comienza el 21 de marzo, el verano el 21 de junio, el otoño el 21 de septiembre y el invierno el 21 de diciembre. Observe que el año comienza y termina durante el invierno. Se puede asumir en la función que los valores en los parámetros `month` y `day` se han validado antes de que sea llamada. Documente el flujo de parámetros con comentarios apropiados.

Problemas de programación

1. Reescriba el programa del problema 1 de programación del capítulo 6 con funciones. El programa imprimirá una gráfica de barras de las temperaturas por hora para un día, dados los datos en un archivo. Debe pedir a la función que imprima el encabezado de la gráfica y a otra función que imprima la barra de estrellas para un determinado valor de temperatura. Observe que la segunda función no imprime el valor a la izquierda de la gráfica de barras. El programa principal coordina el proceso de introducir valores y llamar estas funciones según sea necesario.

   Ahora que sus programas se tornan más complejos, es incluso más importante que use el sangrado y el estilo apropiados, identificadores significativos y comentarios pertinentes.
2. Usted está trabajando para una compañía que coloca pisos de cerámica y necesita un programa que estime el número de cajas de loseta para un trabajo. Un trabajo se estima al tomar las medidas de cada habitación en pies y pulgadas, y convertirlas en un múltiplo del tamaño de la loseta (redondeando cualquier múltiplo parcial) antes de multiplicar para obtener el número de losetas para la habitación. Una caja contiene 20 losetas, así que el número total necesario debe ser dividido entre 20 y redondeado para obtener el número de cajas. Se supone que las losetas son cuadradas.

   El programa debe solicitar inicialmente al usuario que introduzca el tamaño de la loseta en pulgadas y el número de habitaciones. Éste debe contener las dimensiones para cada habitación y dar como resultado las losetas necesarias para cada habitación. Después de introducir los datos para la última habitación, el programa debe producir también el número total de losetas necesarias, el número de cajas y cuántas losetas sobrarán.

   A continuación se da un ejemplo de la apariencia que tendría la ejecución:

```
Introduzca el número de habitaciones: 2
Introduzca el tamaño de la loseta en pulgadas: 12
Introduzca el ancho de la habitación (pies y pulgadas, separados por un
espacio): 17 4
Introduzca la longitud de la habitación (pies y pulgadas, separados por un
espacio): 9 3
La habitación requiere 180 losetas.
Introduzca el ancho de la habitación (pies y pulgadas, separados por un
espacio): 11 6
Introduzca la longitud de la habitación (pies y pulgadas, separados por un
espacio): 11 9
La habitación requiere 144 losetas.
El total de losetas requeridas es 324.
El número de cajas necesarias es 17.
Sobrarán 16 losetas.
```

Use la descomposición funcional para resolver este problema y codifique la solución con funciones siempre que tenga sentido proceder así. Su programa debe comprobar los datos no válidos como dimensiones negativas, número de habitaciones menor que uno, número de pulgadas mayor que 11, etcétera. Debe solicitar la entrada corregida cuando se detecte una entrada no válida. Ahora que sus programas se tornan más complejos, es incluso más importante que utilice sangrado y estilo apropiados, identificadores significativos y comentarios pertinentes.

3. En el problema 6 de programación del capítulo 4 se le pidió que escribiera un programa para calcular la puntuación para un turno, o cuadro, en una partida de bolos o pinos. Aquí ampliará este algoritmo a fin de calcular la puntuación de un juego completo para un jugador. Un juego consta de 10 cuadros, pero el décimo tiene algunos casos especiales que no se describieron en el capítulo 4.

   Un cuadro se juega colocando primero los diez bolos. El jugador hace rodar la bola para derribar los bolos. Si todos son derribados en el primer lanzamiento, se llama chuza, y se termina la entrada. Si son derribados menos de diez bolos en el primer lanzamiento, se registra el número de bolos derribados y el jugador realiza un segundo lanzamiento. Si los bolos restantes no son derribados en el segundo lanzamiento, se llama semipleno (spare). El cuadro termina después del segundo lanzamiento, incluso si todavía quedan bolos parados. Si en el segundo lanzamiento no se derriban todos los bolos, entonces se registra la cantidad de bolos derribados y la puntuación del cuadro es sólo el número de bolos derribados en los dos lanzamientos. Sin embargo, en el caso de una chuza o un semipleno, la puntuación para el cuadro depende de los lanzamientos en el siguiente turno y tal vez del que sigue después.

   Si el cuadro es una chuza, entonces la puntuación es igual a 10 puntos más el número de bolos derribados en los dos lanzamientos siguientes. Así, la puntuación máxima para un cuadro es treinta, lo cual ocurre cuando el cuadro es una chuza y los dos cuadros siguientes también son chuzas. Si el cuadro es un semipleno, entonces la puntuación son esos diez puntos más el número de bolos derribados en el siguiente lanzamiento.

   El último cuadro se juega un poco distinto. Si el jugador hace una chuza, entonces obtiene dos lanzamientos más de modo que se pueda calcular la puntuación para la chuza. De manera similar, si es un semipleno, entonces se da un lanzamiento extra. Si en los dos primeros lanzamientos no se derriban todos los bolos entonces la puntuación para el último cuadro es sólo la cantidad de bolos derribados y no hay lanzamiento extra. Enseguida se presenta un ejemplo de cómo podría aparecer la entrada y salida (I/O) para el inicio de una corrida:

```
Introduzca el lanzamiento para el cuadro 1: 10
¡Chuza!
Introduzca el lanzamiento para el cuadro 2: 7
Introduzca el lanzamiento para el cuadro 2: 3
¡Semipleno!
La puntuación para el cuadro 1 es 20. El total es 20.
Introduzca el lanzamiento para el cuadro 3: 5
La puntuación para el cuadro 2 es 15. El total es 35.
Introduzca el lanzamiento para el cuadro 3: 2
La puntuación para el cuadro 3 es 7. El total es 42.
Introduzca el lanzamiento para el cuadro 4: 12
Error de entrada. Por favor introduzca el número de bolos en el intervalo
de 0 a 10.
Introduzca la puntuación para el cuadro 4:
```

   Su programa debe imprimir el número de bolos derribados en cada lanzamiento y producir la puntuación para cada cuadro como se calcula. El programa debe reconocer cuando ha finalizado un cuadro (ya sea por una chuza o un segundo lanzamiento). El programa debe comprobar también la introducción de datos erróneos. Por ejemplo, un lanzamiento puede estar en el intervalo de 0 a 10 bolos y el total de los dos lanzamientos en cualquiera de los nueve primeros cuadros debe ser menor o igual a 10. Use la descomposición funcional para resolver este proble-

ma y codifique la solución con funciones según sea apropiado. Asegúrese de usar el formato y los comentarios apropiados en su código. El resultado debe tener un formato claro y nítido, y los mensajes de error deben ser informativos.

4. Escriba un programa simple de directorio telefónico en C++ que busque números en un archivo que contenga una lista de nombres y números telefónicos. Se debe indicar al usuario que introduzca el nombre y el apellido de una persona, y el programa produce entonces el número correspondiente o indica que el nombre no está en el directorio. Después de cada búsqueda, el programa debe preguntar al usuario si quiere buscar otro número y luego repetir el proceso o salir del programa. Los datos del archivo deben estar organizados de modo que cada línea contenga un nombre, un apellido y un número telefónico, separados por espacios en blanco. Puede volver al comienzo del archivo cerrándolo o abriéndolo de nuevo.

   Use la descomposición funcional para resolver el problema y codifique la solución con funciones según sea apropiado. Asegúrese de usar formato y comentarios apropiados en su código. El resultado debe tener un formato claro y nítido y los mensajes de error deben ser informativos.

5. Amplíe el programa del problema 4 para buscar direcciones también. Cambie el formato de archivo para que el nombre y el número telefónico aparezcan en una línea y la dirección aparezca en la siguiente línea de cada entrada. El programa debe preguntar al usuario si quiere buscar un número telefónico, una dirección o ambas cosas, luego efectuar la búsqueda y producir la información solicitada. El programa debe reconocer una petición válida y solicitar al usuario que introduzca de nuevo la petición. Como en el problema 4, el programa debe permitir al usuario seguir introduciendo peticiones hasta que indiquen que están hechas.

6. En el problema de programación 1 del capítulo 4 se le pidió que escribiera un programa que introduce una letra y produce la palabra correspondiente en el alfabeto fonético de la International Civil Aviation Organization. En este problema se le pide cambiar ese programa en una función y usarla para convertir una cadena de entrada introducida por el usuario en la serie de palabras que se usarían para deletrearla fonéticamente. Por ejemplo:

```
Introducir la cadena: programa
La versión fonética es: Papa Romeo Oscar Golf Romeo Alpha Mike
```

   Para facilidad de referencia, el alfabeto ICAO del capítulo 4 se repite aquí:

   A Alpha
   B Bravo
   C Charlie
   D Delta
   E Echo
   F Foxtrot
   G Golf
   H Hotel
   I India
   J Juliet
   K Kilo
   L Lima
   M Mike
   N November
   O Oscar
   P Papa
   Q Quebec
   R Romeo
   S Sierra
   T Tango
   U Uniform
   V Victor

W Whiskey
X X-ray
Y Yankee
Z Zulu

Asegúrese de usar el formato y los comentarios apropiados en su código. Proporcione los mensajes de solicitud apropiados al usuario. El resultado debe ser marcado con claridad y tener un formato nítido.

7. Se le ha pedido escribir un programa para evaluar un examen de opción múltiple. El examen tiene 20 preguntas, cada una contestada con una letra en el intervalo de `'a'` a la `'f'`. Los datos se almacenan en un archivo (`exams.dat`) donde la primera línea es la clave, que consta de 20 caracteres. Las líneas restantes en el archivo son respuestas de examen, y consisten en un número de identificación del alumno, un espacio y una cadena de 20 caracteres. El programa debe leer la clave, luego leer cada examen y producir el número de identificación y la puntuación en el archivo `scores.dat`. La introducción de datos erróneos debe producir un mensaje de error. Por ejemplo, dados los datos:

```
abcdefabcdefabcdefab
1234567 abcdefabcdefabcdefab
9876543 abddefbbbdefcbcdefac
5554446 abcdefabcdefabcdef
4445556 abcdefabcdefabcdefabcd
3332221 abcdefghijklmnopqrst
```

El programa daría los resultados en `scores.dat`

```
1234567  20
9876543  15
5554446  Muy pocas respuestas
4445556  Demasiadas respuestas
3332221  Respuestas no válidas
```

Use la descomposición funcional para resolver el problema y codifique la solución con funciones según sea apropiado. Asegúrese de usar el formato y los comentarios apropiados en su código. El resultado debe tener un formato claro y nítido, y los mensajes de error deben ser informativos.

Respuestas al seguimiento de caso práctico

1. Revise el código para el programa del Caso práctico ejecutando los módulos de entrada como funciones. Compare los dos programas. ¿Qué encuentra más fácil de leer?
2. Experimente con el programa de la hipoteca usando los siguientes valores de entrada:

Cantidad del préstamo	100000.00
Número de años	0
Tasa de interés	0
Cantidad del préstamo	50000.00
Número de meses	–1
Tasa de interés	–1

Describa lo que sucede.

3. Agregue la comprobación de errores al programa Hipoteca para considerar los problemas que afloraron en el ejercicio 2.
4. Añada una columna al resultado del programa Hipoteca que muestre cuánto interés se ha pagado.

CAPÍTULO

8

Alcance, tiempo de vida y más sobre funciones

Objetivos

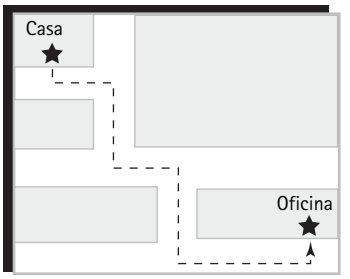
Objetivos de conocimiento

- *Conocer qué es una referencia global.*
- *Entender y poder evitar efectos secundarios indeseables.*
- *Saber cuándo usar una función de devolución de valor.*

Objetivos de habilidades

Ser capaz de:

- *Determinar qué variables en un programa son locales.*
- *Determinar qué variables son accesibles en un bloque dado.*
- *Determinar la duración de cada variable en un programa.*
- *Diseñar y codificar una función de devolución de valor para una tarea específica.*
- *Invocar de manera apropiada una función de devolución de valor.*



A medida que los programas se hacen más grandes y complicados, se incrementa en ellos el número de identificadores. Se inventan nombres de funciones, nombres de variables, identificadores constantes, etcétera. Algunos de estos identificadores se declaran dentro de bloques. Otros identificadores, nombres de funciones, por ejemplo, se declaran fuera de cualquier bloque. En este capítulo se examinan las reglas de C++ mediante las cuales una función puede tener acceso a identificadores que se declaran fuera de su propio bloque. Con estas reglas, se vuelve a la discusión del diseño de interfaz que se inició en el capítulo 7.

Por último, se examina la segunda clase de subprograma provista por C++: la *función de devolución de valor*. A diferencia de las funciones void, que devuelven resultados (si existen) a través de la lista de parámetros, una función de devolución de valor devuelve un solo resultado, el valor de la función, a la expresión de la cual fue llamada. En este capítulo se aprende cómo escribir funciones de devolución de valor definidas por el usuario.

8.1 Alcance de identificadores

Como se vio en el capítulo 7, las variables locales son las que se declaran dentro de un bloque, como el cuerpo de una función. Recuerde que no se puede tener acceso a las variables locales fuera del bloque que las contiene. La misma regla de acceso se aplica a declaraciones de constantes nombradas: el acceso a constantes locales sólo se da en el bloque en el que se declaran.

Cualquier bloque, no sólo el cuerpo de una función, puede contener declaraciones constantes y variables. Por ejemplo, esta sentencia If contiene un bloque que declara una variable local n:

```
if (alpha > 3)
{
    int n;

    cin >> n;
    beta = beta + n;
}
```

Como con cualquier variable local, ninguna sentencia fuera del bloque que contenga su declaración puede tener acceso a n.

Alcance La región del código de programa donde se permite hacer referencia (uso) a un identificador.

Si se listan todos los lugares desde los cuales se permite tener acceso a un identificador, se describiría el alcance de visibilidad o el alcance de acceso del identificador, conocido por lo común sólo como **alcance**.

C++ define varias categorías de alcance de un identificador. Se comienza con la descripción de tres de estas categorías.

1. *Alcance de clase.* Este término se refiere al tipo de datos llamado *clase*, que se introdujo de manera breve en el capítulo 4. Se pospone un análisis detallado del alcance de clase hasta el capítulo 11.

2. *Alcance local.* El alcance de un identificador declarado dentro de un bloque se extiende desde el punto de declaración hasta el final de ese bloque. También, el alcance de un parámetro de función (parámetro formal) se amplía desde el punto de declaración hasta el final del bloque que es el cuerpo de la función.
3. *Alcance global.* El alcance de un identificador declarado fuera de las funciones y clases se amplía del punto de declaración al final del archivo entero que contiene el código de programa.

Los nombres de funciones de C++ tienen alcance global. (Hay una excepción a esta regla, que se analiza en el capítulo 11 cuando se examinan las clases de C++.) Una vez declarado un nombre de función, ésta puede ser invocada por cualquier otra función en el resto del programa. En C++ no existe una función local propiamente dicha; es decir, no se puede anidar una definición de función dentro de otra definición de función.

Las variables y constantes globales son las declaradas fuera de todas las funciones. En el siguiente fragmento de código, `gamma` es una variable global a la cual se puede tener acceso directo mediante sentencias en `main` y `SomeFunc`.

```
int gamma;    // Variable global

int main()
{
    gamma = 3;
     ⋮
}

void SomeFunc()
{
    gamma = 5;
     ⋮
}
```

Cuando una función declara un identificador local con el mismo nombre que un identificador global, el identificador local tiene prioridad dentro de la función. Este principio se llama **prioridad de nombre** u **ocultación de nombre**.

Prioridad de nombre La prioridad que un identificador local de una función tiene sobre un identificador global con el mismo nombre en cualquier referencia que la función hace a ese identificador; conocida también como *ocultación de nombre*.

Aquí se presenta un ejemplo que usa tanto declaraciones locales como globales:

```
#include <iostream>

using namespace std;

void SomeFunc( float );

const int a = 17;              // Una constante global
int b;                         // Una variable global
int c;                         // Otra variable global

int main()
{
    b = 4;                     // Asignación a b global
    c = 6;                     // Asignación a c global
    SomeFunc(42.8);
    return 0;
}
```

```
void SomeFunc( float c )        // Evita el acceso a c global
{
    float b;                    // Evita el acceso a b global

    b = 2.3;                    // Asignación a b local
    cout << "a = " << a;        // Produce a global (17)
    cout << " b = " << b;       // Produce b local (2.3)
    cout << " c = " << c;       // Produce c local (42.8)
}
```

En este ejemplo, la función `SomeFunc` tiene acceso a la constante global a pero declara su propia variable local `b` y un parámetro `c`. Así, el resultado sería

```
a = 17 b = 2.3 c = 42.8
```

La variable local `b` tiene prioridad sobre la variable global `b`, ocultando a `b` global de modo eficaz de las sentencias de la función `SomeFunc`. El parámetro c bloquea también el acceso a la variable global `c` desde dentro de la función. Los parámetros de funciones actúan igual que las variables locales en este sentido; es decir, los parámetros tienen alcance local.

Reglas de alcance

Cuando usted escribe programas en C++, rara vez declara variables globales. Hay aspectos negativos en el uso de variables globales, los cuales se analizarán después. Pero cuando surge una situación en la que usted tiene una necesidad apremiante de contar con variable globales, vale la pena saber cómo C++ maneja estas declaraciones. Las reglas para tener acceso a identificadores que no se declaran localmente se llaman **reglas de alcance**.

Reglas de alcance Las reglas que determinan en qué lugar se puede tener acceso a un identificador, dado el punto donde se declaró ese identificador.

Identificador no local Con respecto a un bloque dado, cualquier identificador declarado fuera de ese bloque.

Además del acceso local y global, las reglas de alcance de C++ definen lo que sucede cuando se anidan unos bloques con otros. Cualquier cosa declarada en un bloque que contiene un bloque anidado es **no local** para el bloque interno. (Los identificadores globales son no locales con respecto a los bloques del programa.) Si un bloque tiene acceso a cualquier identificador declarado fuera de su propio bloque, es un *acceso no local*.

Aquí se presentan las reglas detalladas, sin considerar el alcance de clase y ciertas características de lenguaje que aún no se han analizado:

1. Un nombre de función tiene alcance global. Las definiciones de función no se pueden probar dentro de definiciones de función.
2. El alcance de un parámetro de función es idéntico al alcance de una variable local declarada en el bloque exterior del cuerpo de la función.
3. El alcance de una variable o constante global se extiende desde su declaración hasta el final del archivo, excepto por lo que se hace notar en la regla 5.
4. El alcance de una variable o constante local va de su declaración al final del bloque en el cual se declaró. Este alcance incluye cualquier bloque anidado, excepto por lo que se hace notar en la regla 5.
5. El alcance de un identificador no incluye ningún bloque anidado que contenga un identificador declarado localmente con el mismo nombre (los identificadores locales tienen prioridad de nombre).

A continuación se ilustra un programa que demuestra las reglas de alcance de C++. Para simplificar el ejemplo, sólo se detallan las declaraciones y encabezados. Note cómo el cuerpo del ciclo While identificado como bloque 3, localizado dentro de la función `Block2`, contiene sus propias declaraciones de variables locales.

```
// Programa Reglas de alcance

#include <iostream>

using namespace std;

void Block1( int, char& );
void Block2();

int  a1;          // Una variable global
char a2;          // Otra variable global

int main()
{
    ⋮
}

//*************************************************************************

void Block1( int   a1,          // Evita el acceso a a1 global
             char& b2 )         // Tiene el mismo alcance que c1 y d2
{
    int c1;        // Una variable local respecto a Block1
    int d2;        // Otra variable local respecto a Block1
     ⋮
}

//*************************************************************************

void Block2()
{
    int a1;        // Evita el acceso a a1 global
    int b2;        // Local respecto a Block2; sin conflicto con b2 en
                      Block1

    while ( . . . )
    {              // Block3
        int c1;    // Local respecto a Block3; sin conflicto con c1 en
                      Block1
        int b2;    // Evita el acceso no local a b2 en Block2
                   //   sin conflicto con b2 en Block1
         ⋮
    }
}
```

Se procede a examinar el programa Reglas de alcance en términos de los bloques que define, y a ver lo que significan estas reglas. En la figura 8-1 se muestran los encabezados y declaraciones del programa Reglas de alcance con los alcances de visibilidad indicados por cajas.

Cualquier cosa dentro de una caja puede referirse a cualquier otra en una caja circundante más grande, pero no se permiten referencias externas-internas. Por consiguiente, una sentencia en el bloque 3 (Block3) podría tener acceso a algún identificador declarado en el bloque 2 (Block2) o a cualquier variable global. Una sentencia en el bloque 3 no podría tener acceso a identificadores declarados en el bloque 1 (Block1) desde el exterior.

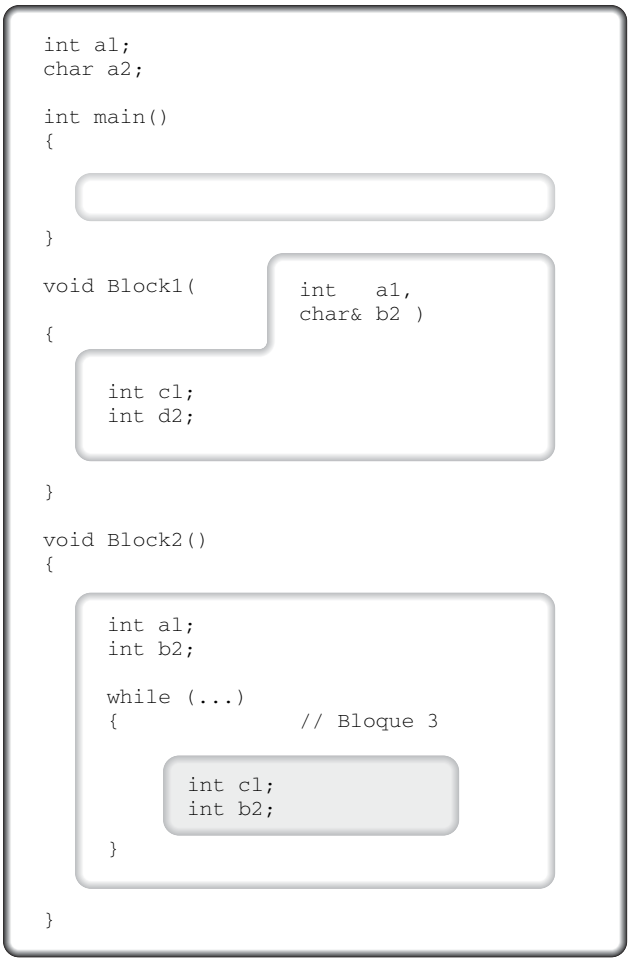


Figura 8-1 *Diagrama de alcance para el programa Reglas de alcance*

Observe que los parámetros para una función están dentro de la caja de la función, pero el nombre de la función está afuera. Si el nombre de la función estuviera dentro de la caja, ninguna función podría llamar a otra. Esto demuestra que sólo se tiene acceso globalmente a los nombres de funciones.

Imagine los nombres de las cajas de la figura 8-1 como habitaciones con paredes hechas de espejos bidireccionales, con el lado reflector hacia fuera y el lado transparente hacia dentro. Si se para en la habitación del bloque 3, podría ver hacia las habitaciones circundantes para las declaraciones de las variables globales (y cualquier cosa intermedia). Sin embargo, no podría ver hacia otras habitaciones (como el bloque 1), porque sus superficies externas reflejadas bloquearían su visión. Como resultado de esta analogía, es común usar el término *visible* al describir un alcance de acceso. Por ejemplo, la variable `a2` es visible en el programa, lo que significa que se puede tener acceso a ella desde cualquier parte del programa.

La figura 8-1 no cuenta toda la historia; representa sólo las reglas de alcance 1 a 4. Se recuerda también la regla 5. La variable `a1` se declara en tres lugares distintos en el programa Reglas de alcance. Debido a la prioridad de nombre, los bloques 2 y 3 tienen acceso a `a1` declarada en el bloque 2 en vez de a `a1` global. De manera similar, el alcance de la variable `b2` declarada en el bloque 2 *no* incluye el "orificio" creado por el bloque 3, porque éste declara su propia variable `b2`.

El compilador pone en práctica la prioridad de nombre como sigue. Cuando una expresión se refiere a un identificador, el compilador comprueba primero las declaraciones locales. Si el identificador no es local, el compilador va por cada nivel de anidación hasta que encuentra un identificador

con el mismo nombre. Allí se detiene. Si hay un identificador con el mismo nombre declarado en un nivel incluso más allá, nunca se alcanza. Si el compilador llega a las declaraciones globales (incluyendo identificadores insertados por directivas `#include`) y aún no puede hallar al identificador, surge un mensaje de error, como "UNDECLARED IDENTIFIER".

Este tipo de mensaje indica con más probabilidad un error ortográfico o uso incorrecto de mayúsculas y minúsculas, o podría indicar que el identificador no se declaró antes de la referencia a él o no se declaró en absoluto. Sin embargo, podría indicar también que los bloques están anidados de modo que el alcance del identificador no incluya la referencia.

Declaraciones y definiciones de variables

En el capítulo 7 aprendió que la terminología de C++ distingue entre una declaración de función y una definición de función. Un prototipo de función es una declaración solamente; es decir, no ocasiona que se reserve espacio de memoria para la función. En contraste, una declaración de función que incluye el cuerpo se llama definición de función. El compilador reserva memoria para las instrucciones del cuerpo de la función.

C++ aplica la misma terminología a declaraciones de variables. Una declaración de variable se convierte en una definición de variable si también reserva memoria para la variable. Todas las declaraciones de variables que se han empleado desde el comienzo han sido definiciones de variables. ¿Cómo se vería una declaración de variable si *no* fuera también una definición?

En el capítulo anterior se habló acerca del concepto de un programa multiarchivo, un programa que ocupa físicamente varios archivos que contienen piezas del programa. C++ tiene una palabra reservada, `extern`, que permite hacer referencia a una variable global localizada en otro archivo. Una declaración "normal" como

```
int someInt;
```

causa que el compilador reserve una ubicación de memoria para someInt. Por otro lado, la declaración

```
extern int someInt;
```

se conoce como *declaración externa*. Ésta expresa que `someInt` es una variable global localizada en otro archivo y que aquí no se debe reservar ningún almacenamiento para ella. Los archivos de encabezado de sistema como `iostream` contienen declaraciones externas para que los programas de usuarios puedan tener acceso a variables importantes definidas en los archivos de sistema. Por ejemplo, `iostream` incluye declaraciones como éstas:

```
extern istream cin;
extern ostream cout;
```

Estas declaraciones permiten hacer referencia a `cin` y `cout` como variables globales en su programa, pero las definiciones de variables se localizan en otro archivo suministrado por el sistema de C++.

En terminología de C++, la sentencia

```
extern int someInt;
```

es una declaración pero no una definición de `someInt`. Ésta relaciona un nombre de variable con un tipo de datos de modo que el compilador puede efectuar la comprobación de tipos. Pero la sentencia

```
int someInt;
```

es tanto una declaración como una definición de `someInt`. Es una definición porque reserva memoria para `someInt`. En C++, usted puede declarar una variable o una función muchas veces, pero puede haber sólo una definición.

Excepto en situaciones en las que es importante distinguir entre declaraciones y definiciones de variables, se continuará con el uso de la frase más general *declaración de variable* en lugar de la *definición de variable* más específica.

Espacios de nombres

Por algún tiempo se ha estado incluyendo la siguiente directiva using en los programas:

```
using namespace std;
```

¿Qué es exactamente un espacio de nombre? Como un concepto general, el espacio de nombre es otra palabra para alcance. Sin embargo, como una característica específica del lenguaje C++, un espacio de nombre es un mecanismo mediante el cual el programador puede crear un alcance designado. Por ejemplo, el archivo de encabezado estándar `cstdlib` contiene prototipos de funciones para varias funciones de biblioteca, una de las cuales es la función valor absoluto, abs. Las declaraciones están contenidas dentro de una definición de espacio de nombre como sigue:

```
// En el archivo de encabezado cstdlib:

namespace std
{
     ⋮
    int abs( int );
     ⋮
}
```

Una definición de espacio de nombre consta de la palabra `namespace`, luego un identificador de la elección del programador y después el *cuerpo del espacio de nombre* entre llaves. Se dice que los identificadores declarados dentro de un espacio de nombre tienen *alcance de espacio de nombre*. El acceso a este tipo de identificadores no puede ser desde fuera del cuerpo excepto si se emplea uno de tres métodos.

El primer método, introducido en el capítulo 2, es usar un nombre calificado: el nombre del `namespace`, seguido del operador de resolución de alcance (`::`), seguido del identificador deseado. Aquí se presenta un ejemplo:

```
#include <cstdlib>

int main()
{
    int alpha;
    int beta;
     ⋮
    alpha = std::abs(beta);    // Un nombre calificado
     ⋮
}
```

La idea general es informar al compilador que se está haciendo referencia al abs declarado en el espacio de nombre `std`, no algún otro `abs` (por ejemplo, una función global llamada `abs` que nosotros mismos podríamos haber escrito).

El segundo método es usar una sentencia llamada declaración `using` como sigue:

```
#include <cstdlib>

int main()
{
    int alpha;
```

```
    int beta;
    using std::abs;     // Una declaración using
     ⋮
    alpha = abs(beta);
     ⋮
}
```

Esta declaración `using` permite que el identificador `abs` se utilice en el cuerpo de `main` como un sinónimo para el `std::abs` más grande.

El tercer método —que ya se vio anteriormente— es para usar la directiva `using` (no se debe confundir con la declaración `using`).

```
#include <cstdlib>

int main()
{
    int alpha;
    int beta;
    using namespace std;     // Una directiva using
     ⋮
    alpha = abs(beta);
     ⋮
}
```

Con una directiva `using`, *todos* los identificadores del nombre de espacio especificado son accesibles, pero sólo en el alcance en que aparece la directiva `using`. Arriba, la directiva `using` en el alcance local (está dentro de un bloque), así que los identificadores del nombre de espacio `std` son accesibles sólo dentro de `main`. Por otro lado, si se escribe la directiva `using` fuera de las funciones (como se ha estado haciendo), como esto:

```
#include <cstdlib>

using namespace std;

int main()
{
    ⋮
}
```

entonces la directiva `using` es de alcance global; en consecuencia, los identificadores del nombre de espacio `std` son accesibles globalmente.

Colocar una directiva `using` en el alcance global puede ser conveniente. Por ejemplo, todas las funciones que se escriben pueden referirse a identificadores como `abs`, `cin` y `cout` sin tener que insertar una directiva `using` localmente en cada función. Sin embargo, las directivas globales `using` se consideran una mala idea al crear programas grandes, multiarchivo. Los programadores emplean con frecuencia varias bibliotecas, no sólo la biblioteca estándar de C++, al desarrollar software complejo. Dos o más bibliotecas pueden, sólo por conveniencia, usar el mismo identificador para propósitos completamente distintos. Si se emplean directivas globales `using`, pueden ocurrir *coincidencias de nombres* (definiciones múltiples del mismo identificador) porque todos los identificadores se han llevado al alcance global. (Los programadores de C++ hacen referencia a esto como "contaminación del nombre de espacio global".) En los siguientes capítulos se continúa con el uso de directivas globales `using` para el espacio de nombre `std` porque nuestros programas son relativamente pequeños y, por lo tanto, las coincidencias de nombres son improbables.

Dado el concepto de alcance de espacio de nombre, se refine la descripción de las categorías de alcance de C++ como sigue.

1. *Alcance de clase.* Este término se refiere al tipo de datos llamado *clase*. Se pospone una descripción detallada del alcance de clase hasta el capítulo 11.
2. *Alcance local.* El alcance de un identificador declarado dentro de un bloque se extiende del punto de declaración al final de ese bloque. También, el alcance de un parámetro de función (parámetro formal) va del punto de declaración al final del bloque que es el cuerpo de la función.
3. *Alcance de nombre de espacio.* El alcance de un identificador declarado en una definición de nombre de espacio se extiende del punto de declaración al final del cuerpo del nombre de espacio, *y* su alcance incluye el alcance de una directiva `using` que especifica el nombre de espacio.
4. *Alcance global* (o *nombre de espacio global*). El alcance de un identificador declarado fuera de los espacios de nombres, funciones y clases va del punto de declaración al final del archivo completo que contiene el código de programa.

Note que éstas son las descripciones generales de categorías de alcance y no reglas de alcance. Las descripciones no explican la ocultación de nombre (la redefinición de un identificador dentro de un bloque anidado).

8.2 Duración de una variable

Un concepto relacionado pero separado del alcance de una variable es su **duración**, el periodo durante la ejecución del programa cuando un identificador tiene asignada en realidad memoria. Se ha dicho que el almacenamiento para variables locales se crea (asigna) en el momento en que el control introduce una función. Entonces las variables están "vivas" mientras la función se está ejecutando, y finalmente el almacenamiento se destruye (se libera) cuando sale la función. En contraste, la duración de una variable global es la misma que la de todo el programa. La memoria se asigna sólo una vez, cuando el programa comienza a ejecutarse, y se libera sólo cuando termina todo el programa. Observe que el alcance es una cuestión de *tiempo de compilación*, pero la duración es un asunto de *tiempo de ejecución*.

Duración El periodo durante la ejecución de un programa cuando un identificador tiene memoria asignada.

Variable automática Una variable para la cual se asigna memoria y se libera cuando el control entra y sale del bloque en el cual está declarado.

Variable estática Una variable para la cual la memoria permanece asignada en la ejecución de todo el programa.

En C++, el almacenamiento de una **variable automática** se asigna en la entrada del bloque y se libera en la salida. Una **variable estática** es aquella cuyo almacenamiento permanece asignado el tiempo que dura el programa. Todas las variables globales son variables estáticas. Por omisión, las variables declaradas dentro de un bloque son variables automáticas. Sin embargo, se puede usar la palabra reservada `static` cuando declara una variable local. Si se procede de este modo, la variable es estática y su duración persiste de llamada de función a llamada de función:

```
void SomeFunc()
{
    float  someFloat;          // Se destruye cuando sale la función
    static int someInt;        // Retiene su valor de llamada a llamada
     ⋮
}
```

Por lo común, es mejor declarar una variable local como `estática` que usar una variable global. Al igual que una variable local, su memoria permanece asignada el tiempo que dura el programa. Pero a diferencia de una variable global, su alcance local evita que sean retocadas otras funciones del programa.

Inicializaciones en declaraciones

Una de las cosas más comunes que se realizan en los programas es declarar primero una variable y luego, en un segmento separado, asignar un valor inicial a la variable. A continuación aparece un ejemplo representativo:

```
int sum;

sum = 0;
```

C++ permite combinar estas dos sentencias en una. El resultado se conoce como una *inicialización en una declaración*. Aquí se inicializa `sum` en su declaración:

```
int sum = 0;
```

En una declaración, la expresión que especifica el valor inicial se llama *inicializador*. En la expresión anterior el inicializador es la constante 0. La coerción implícita de tipos tiene lugar si el tipo de datos del inicializador es diferente del tipo de datos de la variable.

Una variable automática se inicializa en el valor especificado cada vez que el control entra al bloque:

```
void SomeFunc( int someParam )
{
    int i = 0;                    // Se inicializa cada vez
    int n = 2 * someParam + 3;    // Se inicializa cada vez
     ⋮
}
```

En contraste, la inicialización de una variable estática (ya sea variable global o local `static` declarada explícitamente) ocurre sólo una vez, la primera vez que el control alcanza su declaración. Aquí está un ejemplo en el que se inicializan dos variables estáticas locales sólo una vez (la primera vez que se llama a la función):

```
void AnotherFunc( int param )
{
    static char ch = 'A';         // Se inicializa sólo una vez
    static int  m  = param + 1;   // Se inicializa sólo una vez
     ⋮
}
```

Aunque una inicialización da a una variable un valor inicial, es perfectamente aceptable reasignarle otro valor durante la ejecución del programa.

Hay distintas opiniones respecto a inicializar una variable en su declaración. Algunos programadores nunca lo hacen y prefieren mantener una inicialización cerca de las sentencias ejecutables que dependen de la variable. Por ejemplo,

```
int loopCount;
    ⋮
loopCount = 1;
while (loopCount <= 20)
{
    ⋮
}
```

Otros programadores sostienen que una de las causas más frecuentes de errores de programa es olvidar inicializar las variables antes de usar su contenido; inicializar cada variable en su declaración

elimina estos errores. Como con cualquier tema controvertido, la mayor parte de los programadores al parecer toman una posición intermedia entre estos dos extremos.

8.3 Diseño de interfaz

Ahora se vuelve al tema del diseño de interfaz, que se analizó primero en el capítulo 7. Recuerde que el flujo de datos por una interfaz de función puede tomar tres formas: sólo entrante, sólo saliente y entrante y saliente. Cualquier elemento que pueda ser clasificado como entrante debe ser codificado como parámetro de valor. Los elementos de las dos categorías restantes (saliente y entrante y saliente) deben ser parámetros de referencia; la única forma en que la función puede depositar resultados en los argumentos del invocador es tener las direcciones de esos argumentos. Para remarcar, se repite la siguiente tabla del capítulo 7.

Flujo de datos para un parámetro	Mecanismo de paso de argumentos
Entrante	Paso por valor
Saliente	Paso por referencia
Entrante y saliente	Paso por referencia

Como se dijo en el capítulo anterior, hay excepciones a las directrices de esta tabla. C++ requiere que los objetos de flujo I/O sean pasados por referencia debido a la manera como se ejecutan flujos y archivos. En el capítulo 12 se encuentra otra excepción.

Algunas veces es tentador omitir el paso de diseño de interfaz al escribir una función, lo que permite la comunicación con otras funciones al hacer referencia a variables globales. ¡No lo haga! Sin el paso de diseño de interfaz, estaría creando en realidad una interfaz mal estructurada y sin documentación. Excepto en circunstancias bien justificadas, el uso de variables globales es una mala práctica de programación que origina errores de programa. Estos errores son muy difíciles de localizar y, por lo común, toman la forma de efectos secundarios indeseados.

Efectos secundarios

Suponga que hizo una llamada a la función de biblioteca sqrt en su programa:

```
y = sqrt (x);
```

Efecto secundario Cualquier efecto de una función en otra que no es parte de la interfaz entre ellas definida de modo explícito.

Usted espera llamar a sqrt para hacer sólo una cosa: calcular la raíz cuadrada de la variable `x`. Se sorprendería si sqrt cambiara también el valor de su variable `x` porque por definición sqrt no hace tales cambios. Éste sería un ejemplo de un **efecto secundario** no esperado e indeseado.

Los efectos secundarios son causados a veces por una combinación de parámetros de referencia y codificación descuidada en una función. Quizá una sentencia de asignación en la función guarda un resultado temporal en uno de los parámetros de referencia, y cambia por accidente el valor de un argumento de regreso al código invocador. Como se mencionó, usar parámetros de valores evita este tipo de efecto secundario al impedir que el cambio llegue al argumento.

Los efectos secundarios ocurren también cuando una función tiene acceso a una variable global. Un error en la función podría ocasionar que el valor de una variable global sea cambiado de una manera inesperada, lo cual causaría un error en las otras funciones que tienen acceso a la variable.

Los síntomas de un efecto secundario son engañosos porque se presentan en una parte del programa cuando en realidad lo causa algo en otra parte. Para evitar este tipo de errores, el único efecto externo que debe tener una función es transferir información por la interfaz bien estructurada de la

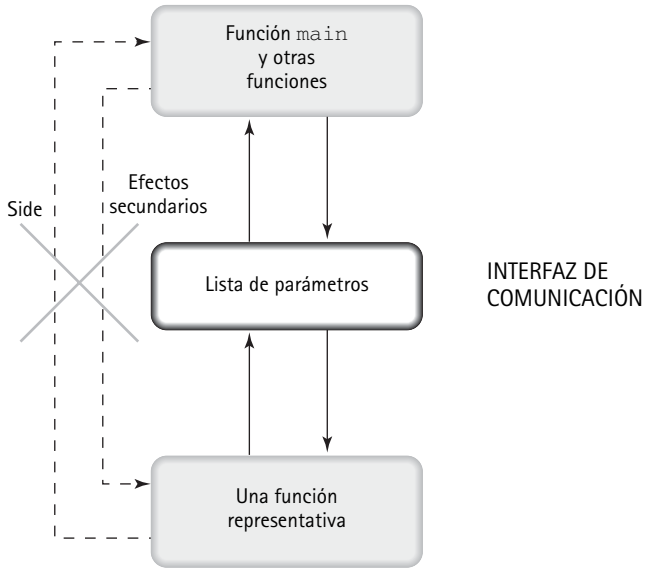


Figura 8-2 *Efectos secundarios*

lista de parámetros (véase fig. 8-2). Si la función tiene acceso a variables no locales *solamente* a través de sus listas de parámetros, y si los parámetros entrantes son parámetros de valor, entonces cada función se aísla en esencia de las otras partes del programa y no pueden ocurrir efectos secundarios.

A continuación se muestra un ejemplo corto de un programa que corre pero produce resultados incorrectos a causa de variables globales y efectos secundarios.

```
//**************************************************************************
// Programa Problema
// Éste es un ejemplo de mal diseño de programa que causa un error cuando
// se ejecuta el programa
//**************************************************************************
#include <iostream>

using namespace std;

void CountInts();

int count;           // Se supone que cuenta las líneas de entrada, pero ¿lo
                        hace en realidad?
int intVal;          // Mantiene un entero de entrada

int main()
{
    count = 0;
    cin >> intVal;
    while (cin)
    {
        count++;
        CountInts();
        cin >> intVal;
    }
```

```
    cout << count << " líneas de entrada procesadas." << endl;
    return 0;
}

//*************************************************************************

void CountInts()

// Cuenta el número de enteros en una línea de entrada (donde 99999
// es un centinela en cada línea) e imprime la cuenta
// Nota: main() ya ha leído el primer entero sobre una línea

{
    count = 0;                                  // Efecto secundario
    while (intVal != 99999)
    {
        count++;                                // Efecto secundario
        cin >> intVal;
    }
    cout << count << " enteros en esta línea." << endl;
}
```

Se supone que el programa Problema cuenta e imprime el número de enteros en cada línea de entrada. Después de que se ha procesado la última línea, debe imprimir el número de líneas. Por extraño que parezca, cada vez que se ejecuta el programa, informa que el número de líneas de entrada es el mismo que el de enteros en la última línea de entrada. Esto es porque la función `CountInts` tiene acceso a la variable global `count` y la utiliza para guardar el número de enteros en cada línea de entrada.

No hay razón para que `count` sea una variable global. Si se declara una variable local en `main` y otra variable local `count` en `CountInts`, el programa funciona correctamente. No hay conflicto entre las dos variables porque cada una es visible sólo dentro de su propio bloque.

El programa Problema demuestra también una excepción común a la regla de no tener acceso a variables globales. Desde un punto de vista técnico, `cin` y `cout` son objetos globales declarados en el archivo de encabezado `iostream`. La función `CountInts` lee y escribe directamente en estos flujos. A fin de no cometer errores en absoluto, `cin` y `cout` se deben pasar como argumentos a la función. Sin embargo, `cin` y `cout` son recursos fundamentales I/O provistos por la biblioteca estándar, y es una convención que las funciones de C++ tengan acceso directo a ellos.

Constantes globales

Contrario a lo que se podría pensar, es aceptable hacer referencia globalmente a constantes nombradas. Debido a que no es posible cambiar los valores de constantes globales mientras se ejecuta un programa, no ocurren efectos secundarios.

Hay dos ventajas para hacer referencia globalmente a constantes: facilidad de cambio y congruencia. Si necesita cambiar el valor de una constante, es más fácil cambiar sólo una declaración global que cambiar una declaración local en cada función. Al declarar una constante sólo en un lugar, se asegura también que todas las partes de un programa usen el mismo valor.

Esto no quiere decir que deba declarar *todas* las constantes globalmente. Si una constante es necesaria sólo en una función, entonces tiene sentido declararla localmente dentro de esa función.

En este punto, quizá desee dirigirse al caso práctico de resolución de problemas al final de este capítulo. El siguiente caso práctico ilustra el proceso de diseño de interfaz y el uso de parámetros de valor y referencia.

Conozca a

Ada Lovelace

El 10 de diciembre de 1815 (el mismo año en que nació George Boole). Anna Isabella (Annabella) Byron y George Gordon, Lord Byron, tuvieron una hija, Augusta Ada Byron. En Inglaterra en esa época, la fama de Byron derivó no sólo de su poesía sino también de su comportamiento escandaloso. El matrimonio tuvo dificultades desde el principio, y Anabella dejó a Byron poco después de que nació Ada. En abril de 1816, la pareja había firmado los papeles de divorcio. Byron salió de Inglaterra para nunca volver. El resto de su vida lamentó no poder ver a su hija. En algún momento le escribió:

> No te veo. No te escucho.
> Pero nadie más piensa tanto en ti.

Antes de morir en Grecia a la edad de 36 años, exclamó, "¡Oh mi pobre y bien amada hija! ¡Mi querida Ada! Dios mío, ¡la he visto!"

Mientras tanto, Annabella, quien finalmente se convertiría en baronesa por sí misma, y quien fue educada como matemática y poeta, se hizo cargo de la formación y educación de Ada. Ada recibió de Annabella sus primeros conocimientos de matemáticas, pero pronto se vio el don de Ada en el tema y fue necesaria una tutoría más extensa. Ada recibió capacitación de Augustus DeMorgan, famoso en nuestros días por uno de sus teoremas básicos del álgebra booleana, el fundamento lógico para las computadoras modernas. A la edad de 8 años, Ada había demostrado también su interés en dispositivos mecánicos y construía botes modelo detallados.

Cuando tenía 18 años, Ada visitó el Instituto de Mecánica para escuchar las conferencias de Dionysius Lardner sobre la máquina de diferencias, una máquina calculadora mecánica que construyó Charles Babbage. Ella también se interesó tanto en el dispositivo que hizo los arreglos para la presentaran con Babbage. Se dice que, al ver la máquina de Babbage, Ada fue la única persona del salón en entender de inmediato cómo funcionaba y en reconocer su importancia. Ada y Charles Babbage tuvieron una amistad duradera. Ella trabajó con él, ayudando a documentar sus diseños, traduciendo escritos sobre su trabajo y en el desarrollo de programas para sus máquinas. De hecho, hoy día Ada es reconocida como la primera programadora de la historia, y el moderno lenguaje de programación fue bautizado Ada en su honor.

Cuando Babbage diseñó su máquina analítica, Ada anticipó que ésta podría ir más allá de cálculos aritméticos y se convertiría en un manipulador general de símbolos y que, por lo tanto, tendría capacidades de largo alcance. Incluso sugirió que tal dispositivo llegaría a poder programarse con reglas de armonía y composición de modo que produciría música "científica". En efecto, Ada vislumbró el campo de la inteligencia artificial hace más de 150 años.

En 1842, Babbage dio una serie de conferencias en Turín, Italia, en relación con su máquina analítica. Uno de los asistentes fue Luigi Menabrea, quien se impresionó tanto que escribió un informe de las conferencias de Babbage. A la edad de 27 años, Ada decidió traducir el informe al inglés con la finalidad de añadir algunas de sus notas acerca de la máquina. Al final, sus notas resultaron ser el doble del material original y el documento, "The Sketch of the Analytical Engine", llegó a ser el trabajo definitivo sobre el tema.

Se hace evidente, a partir de las cartas de Ada, que sus "notas" eran por completo suyas y que Babbage hacía cambios editoriales no solicitados. En determinado momento, Ada le escribió:

> Estoy muy molesta contigo por haber alterado mi nota. Sabes que siempre estoy dispuesta a hacer cualquier modificación requerida yo misma, pero no puedo soportar que otra persona modifique mis frases.

(continúa) ▼

Ada Lovelace

Ada obtuvo el título de Condesa de Lovelace cuando se casó con Lord William Lovelace. La pareja procreó tres hijos, cuya educación familiar estuvo a cargo de la madre de Ada mientras ella continuaba con su trabajo en matemáticas. Su esposo la apoyaba en su trabajo, pero para una mujer de esa época, tal conducta era considerada casi tan escandalosa como algunas de las aventuras de su padre.

Ada Lovelace murió de cáncer en 1852, sólo un año antes de que se construyera en Suiza una máquina de diferencias práctica a partir de los diseños de Babbage. Como su padre, Ada vivió sólo hasta la edad de 36 años, y aun cuando llevaron vidas muy distintas, ella sin duda lo admiró y se inspiró en su naturaleza rebelde poco convencional. A fin de cuentas, Ada pidió ser enterrada junto a él en la propiedad de la familia.

8.4 Funciones de devolución de valor

En el capítulo 7 y en la primera parte de este capítulo, hemos estado escribiendo nuestras propias funciones void. Ahora se examina la segunda clase de subprograma en C++, la función de devolución de valor. Ya conoce varias funciones de devolución de valor provistas por la biblioteca estándar de C++: `sqrt`, `abs`, `fabs` y otras. Desde la perspectiva del invocador, la diferencia principal entre funciones void y funciones de devolución de valor es la forma en la que son llamadas. Una llamada para una función void es una sentencia completa; una llamada para una función de devolución de valor es parte de una expresión.

Desde una perspectiva de diseño, las funciones de devolución de valor se emplean cuando hay sólo un resultado devuelto por una función y ese resultado se usará directamente en una expresión. Por ejemplo, suponga que se está escribiendo un programa que calcula un reembolso prorrateado de colegiaturas para estudiantes, que ellos retiran a la mitad de un semestre. La cantidad que se reembolsará es la colegiatura total multiplicada por la fracción restante del semestre (el número de días restantes divididos entre la cantidad total de días en el semestre). Las personas que usan el programa desean poder introducir las fechas en las que comienza y termina el semestre y la fecha del retiro, y quieren que el programa calcule la fracción restante del semestre.

Debido a que cada semestre en esta escuela comienza y termina dentro de un año escolar, se puede calcular el número de días en un periodo determinando el número de día de cada fecha y restando el número de día inicial del número de día final. El número de día es el número relacionado con cada día del año si cuenta en secuencia desde el 1 de enero. Diciembre 31 tiene el número 365, excepto en años bisiestos, cuando es 366. Por ejemplo, si un semestre comienza el 3 de enero de 2001 y termina el 17 de mayo de 2001, el cálculo es como sigue.

El número de día de 03/01/2001 es 3
El número de día de 17/05/2001 es 137
La duración del semestre es 137 – 3 + 1 = 135

Se agrega 1 a la diferencia de los días porque se cuenta el primer día como parte del periodo.

El algoritmo para calcular el número de día para una fecha se complica por los años bisiestos y por los meses de distintas duraciones. Se podría codificar este algoritmo como una función void llamada `ComputeDay`. El reembolso se podría calcular mediante el siguiente segmento de código.

```
ComputeDay(startMonth, startDay, startYear, start);
ComputeDay(lastMonth, lastDay, lastYear, last);
ComputeDay(withdrawMonth, withdrawDay, withdrawYear, withdraw);
fraction = float(last - withdraw + 1) / float(last - start + 1);
refund = tuition * fraction;
```

Los primeros tres argumentos para `ComputeDay` los recibe la función, y el último se devuelve al invocador. Debido a que `ComputeDay` devuelve sólo un valor, es posible escribirlo como una función de devolución de valor en lugar de una función void. Se examinará cómo se escribiría el código invocador si se tuviera una función de devolución de valor llamada `Day` que devuelve el número de día de una fecha en un año específico.

```
start = Day(startMonth, startDay, startYear);
last = Day(lastMonth, lastDay, lastYear);
withdraw = Day(withdrawMonth, withdrawDay, withdrawYear);
fraction = float(last - withdraw + 1) / float(last - start + 1);
refund = tuition * fraction;
```

La segunda versión del segmento de código es mucho más intuitiva. Debido a que `Day` es una función de devolución de valor, se sabe de inmediato que todos sus parámetros reciben valores y que devuelve sólo un valor (el número de día para una fecha).

Considérese la definición de función para `Day`. No se preocupe acerca de cómo funciona `Day`; por ahora, debe centrar su atención en su sintaxis y estructura.

```
int Day( /* in */ int month,          // Número de mes, 1 - 12
         /* in */ int dayOfMonth,     // Día del mes, 1 - 31
         /* in */ int year       )    // Año. Por ejemplo, 2001

// Esta función calcula el número de día dentro de un año, dada la fecha.
// Da cuenta correctamente de los años bisiestos. El cálculo se basa
// en el hecho de que los meses tienen en promedio 30 días de duración.
// Así, (mes - 1) * 30 es aproximadamente el número de días en el año al
// comienzo de cualquier mes. Se usa un factor de corrección para considerar
// los casos donde el promedio es incorrecto y los años bisiestos. Se agrega
// el día del mes para producir el número de día

// Precondición:
//     1 <= month <= 12
//  && dayOfMonth está en un intervalo válido para el mes
//  && se asigna el año
// Poscondición:
//     El valor de retorno es el número de día en el intervalo 1 - 365
//                         (o 1 - 366 para un año bisiesto)

{
    int correction = 0;   // Factor de corrección para tomar en cuenta el año
                          //   bisiesto y los meses de diferente duración

    // Prueba para año bisiesto

    if (year % 4 == 0 && (year % 100 != 0 || year % 400 == 0))
        if (month >= 3)         // Si la fecha es después del 29 de febrero
            correction = 1;     //   entonces se suma 1 para el año bisiesto

    // Corregir los meses de distinta duración

    if (month == 3)
        correction = correction - 1;
    else if (month == 2 || month == 6 || month == 7)
        correction = correction + 1;
```

```
    else if (month == 8)
        correction = correction + 2;
    else if (month == 9 || month == 10)
        correction = correction + 3;
    else if (month == 11 || month == 12)
        correction = correction + 4;
    return (month - 1) * 30 + correction + dayOfMonth;
}
```

Lo primero que se observa es que la definición de función se parece a una función void, excepto por el hecho de que el encabezado comienza el tipo de datos `int` en lugar de la palabra `void`. La segunda cosa que se debe observar es la sentencia Return al final, que incluye una expresión entera entre la palabra `return` y el punto y coma.

Una función de devolución de valor devuelve un valor, no a través de un parámetro sino por medio de una sentencia Return. El tipo de datos al comienzo del encabezado declara el tipo de valor que devuelve la función. Este tipo de datos se llama *tipo de función*, aunque un término más preciso es **tipo de valor de función** (o *tipo de devolución de función* o *tipo de resultado de función*).

Tipo de valor de función (o tipo de devolución de función o tipo de resultado

La última sentencia en la función `Day` evalúa la expresión

```
(month - 1) * 30 + correction + dayOfMonth
```

y devuelve el resultado como el valor de función (véase fig. 8-3).

Ahora ya ha visto dos formas de la sentencia Return. La forma

```
return;
```

es válida *sólo* en función void. Causa que el control salga de inmediato de la función y vuelva al invocador. La segunda forma es

```
return Expresión;
```

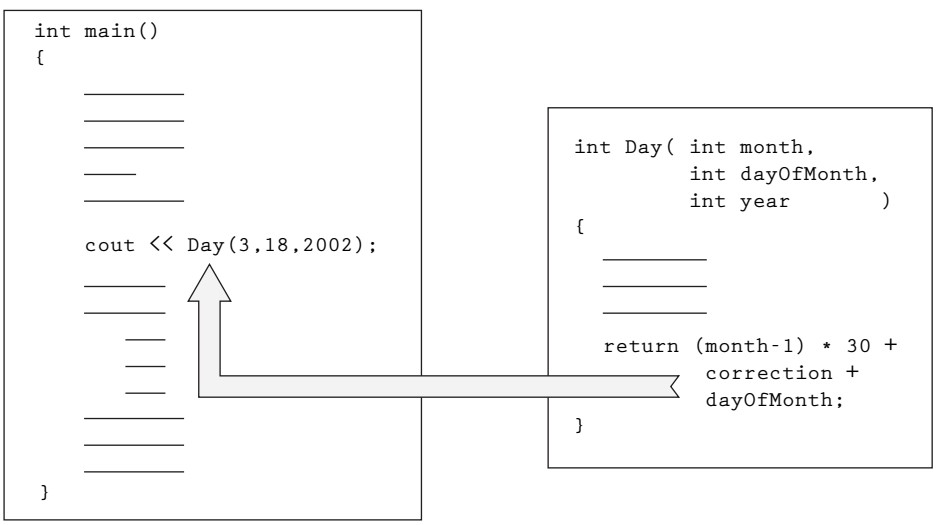


Figura 8-3 *Devolución de un valor de función para la expresión que llamó a la función*

Esta forma es válida *sólo* en una función de devolución de valor. Devuelve el control al invocador, enviando de regreso el valor de la Expresión como el valor de la función. (Si el tipo de datos de Expresión es diferente del tipo de función declarado, su valor se fuerza al tipo correcto.)

En el capítulo 7 se presentó una plantilla de sintaxis para la definición de función de una función void. Ahora se actualiza la plantilla de sintaxis para cubrir tanto las funciones void como las funciones de devolución de valor:

Definición de función

```
Tipo de datos Nombre de la función ( Lista de parámetros )
{
     Sentencia
        ⋮
}
```

Si el tipo de datos es la palabra `void`, la función está vacía; de lo contrario, es una función de devolución de valor. Observe en el área sombreada de la plantilla de sintaxis que Tipo de datos es opcional. Si omite el tipo de datos de una función, se supone `int`. Este punto se menciona sólo porque en ocasiones se encuentran programas donde falta Tipo de datos en el encabezado de la función. Muchos programadores no consideran que esta práctica sea un buen estilo de programación.

La lista de parámetros para una función de devolución de valor tiene exactamente la misma forma que para una función void: una lista de declaraciones de parámetros, separadas por comas. También, un prototipo de función para un valor de devolución de valor se ve como el prototipo para una función void excepto que comienza con un tipo de datos en lugar de `void`.

Considérense dos ejemplos más de funciones de devolución de valor. La biblioteca estándar de C++ provee una función de potencia, pow, que eleva un número de punto flotante a una potencia de punto flotante. La biblioteca no provee una función de potencia para valores `int`, así que se construirá una propia. La función recibe dos enteros, `x` y `n` (donde $n \geq 0$) y calcula x^n. Se usa un método simple, multiplicando repetidamente por `x`. Debido a que el número de iteraciones se conoce por adelantado, es apropiado un ciclo controlado por conteo. La cuenta del ciclo baja hasta 0 desde el valor inicial de `n`. Para cada iteración del ciclo, `x` se multiplica por el producto previo.

```
int Power( /* in */ int x,     // Número base
           /* in */ int n )    // Potencia para elevar la base a

// Esta función calcula x para la potencia n

// Precondición:
//     se asigna x  &&  n >= 0  &&  (x a la n) <= INT_MAX
// Poscondición:
//     El valor devuelto es x a la potencia n

{
    int result;     // Contiene las potencias intermedias de x

    result = 1;
    while (n > 0)
    {
        result = result * x;
        n--;
    }
    return result;
}
```

Observe la notación que se emplea en la poscondición de una función de devolución de valor. Debido a que la función de devolución de valor regresa un solo valor, es más conciso si se expresa simplemente a qué valor es igual. Excepto en ejemplos complicados, la poscondición se ve como esto:

```
// Poscondición
//     El valor devuelto es…
```

Otra función que se usa con frecuencia en el cálculo de probabilidades es el factorial. Por ejemplo, 5 factorial (escrito 5! en notación matemática) es $5 \times 4 \times 3 \times 2 \times 1$. Cero factorial, por definición, es igual a 1. Esta función tiene un parámetro entero. Como con la función `Power` se usa multiplicación repetida, pero se reduce el multiplicador en cada iteración.

```
int Factorial( /* in */ int n )    // Número cuyo factorial
                                   //   se calculará
// Esta función calcula n!

// Precondición:
//     n >= 0  &&  n! <= INT_MAX
// Poscondición:
//     El valor devuelto es n!

{
    int result;     // Contiene productos parciales

    result = 1;
    while (n > 0)
    {
        result = result * n;
        n--;
    }
    return result;
}
```

Una llamada para la función `Factorial` podría parecerse a esto:

```
combinations = Factorial(n) / (Factorial(m) * Factorial(n - m));
```

Funciones booleanas

Las funciones de devolución de valor no están restringidas a devolver resultados numéricos. Se pueden usar también, por ejemplo, para evaluar una condición y obtener un resultado booleano. Las funciones booleanas pueden ser útiles cuando una rama o ciclo depende de alguna condición compleja. En vez de codificar la condición directamente en la sentencia If o While, se puede llamar a una función booleana para formar la expresión de control.

Suponga que se escribe un programa que funciona con triángulos. El programa lee tres ángulos como números de punto flotante. Antes de efectuar cualquier cálculo en esos ángulos, se desea comprobar que en realidad forman un triángulo sumando los ángulos para confirmar que su suma es igual a 180 grados. Se puede escribir una función de devolución de valor que toma los tres ángulos como parámetros y devuelve un resultado booleano. Esta clase de función se vería así (recuerde del capítulo 5 que debe probar los números de punto flotante sólo para igualdad cercana):

```
#include <cmath>   // Para fabs()
  ⋮
bool IsTriangle( /* in */ float angle1,    // Primer ángulo
                 /* in */ float angle2,    // Segundo ángulo
                 /* in */ float angle3 )   // Tercer ángulo

// Esta función comprueba si sus tres valores entrantes suman 180 grados
// y forman un triángulo válido

// Precondición:
//     Se asignan angle1, angle2 y angle3
// Poscondición:
//    El valor devuelto es verdadero, si (angle1 + angle2 + angle3) está
//                            dentro de 0.00000001 de 180.0 grados
//                 de otro modo, es falso

{
    return (fabs(angle1 + angle2 + angle3 - 180.0) < 0.00000001);
}
```

El siguiente programa muestra cómo se llama a la función `IsTriangle`. (La definición de función se muestra sin su documentación para ahorrar espacio.)

```
//*********************************************************************
// Programa Triángulo
// Este programa aplica la función IsTriangle
//*********************************************************************
#include <iostream>
#include <cmath>      // Para fabs()

using namespace std;

bool IsTriangle( float, float, float );

int main()
{
    float angleA;    // Tres posibles ángulos de un triángulo
    float angleB;
    float angleC;
    cout << "Introduzca tres ángulos: ";
    cin >> angleA;
    while (cin)
    {
        cin >> angleB >> angleC;
        if (IsTriangle(angleA, angleB, angleC))
            cout << "Los tres ángulos forman un triángulo válido." << endl;
        else
            cout << "Esos ángulos no forman un triángulo." << endl;
        cout << "Introduzca tres ángulos: ";
        cin >> angleA;
    }
    return 0;
}
```

```
//******************************************************************************

bool IsTriangle( /* in */ float angle1,
                 /* in */ float angle2,
                 /* in */ float angle3 )
{
    return (fabs(angle1 + angle2 + angle3 - 180.0) < 0.00000001);
}
```

En la función `main` del programa Triángulo, la sentencia If es mucho más fácil de entender con la llamada de función de lo que sería si se codificara directamente la condición completa. Cuando una prueba condicional es del todo complicada, es pertinente una función booleana.

La biblioteca estándar de C++ provee una cantidad de funciones útiles que permiten probar el contenido de variables char. Para usarlas, se incluye (`#include`) el archivo de encabezado `cctype`. Las siguientes son algunas de las funciones disponibles; el apéndice C contiene una lista más completa.

Archivo de encabezado	Función	Tipo de función	Valor de función
<cctype>	isalpha(ch)	int	No cero si ch es una letra ('A'-'Z', 'a'-'z'); de otro modo, 0
<cctype>	isalnum(ch)	int	No cero si ch es una letra o un dígito ('A'-'Z', 'a'-'z', '0'-'9'); de otro modo, 0
<cctype>	isdigit(ch)	int	No cero si ch es un dígito ('0'-'9'); de otro modo, 0
<cctype>	islower(ch)	int	No cero si ch es una letra minúscula ('a'-'z'); de otro modo, 0
<cctype>	isspace(ch)	int	No cero si ch es un carácter de espacio en blanco (espacio, nueva línea, tab., retorno de carro, alimentación de papel); de otro modo, 0
<cctype>	isupper(ch)	int	No cero si ch es una letra mayúscula ('A'-'Z'); de otro modo, 0

Aunque devuelven valores `int`, las funciones `"is..."` se comportan como funciones booleanas. Devuelven un valor `int` que es no cero (forzado a `true` en una condición If o While) o 0 (forzado a `false` en una condición If o While). El uso de estas funciones es conveniente y constituyen programas más legibles. Por ejemplo, la prueba

```
if (isalnum(inputChar))
```

es más fácil de leer y menos propensa a errores que si codifica la prueba en la forma larga:

```
if (inputChar >= 'A' && inputChar <= 'Z' ||
    inputChar >= 'a' && inputChar <= 'z' ||
    inputChar >= '0' && inputChar <= '9'    )
```

De hecho, esta expresión lógica complicada no funciona correctamente en algunas máquinas. Se verá por qué cuando se examinen datos de caracteres en el capítulo 10.

Cuestiones de estilo

Cómo nombrar funciones de devolución de valor

En el capítulo 7 se dijo que es buen estilo usar verbos imperativos al nombrar funciones void. La razón es que una llamada para una función void es una sentencia completa y debe parecer como una instrucción para la computadora:

```
PrintResults(a, b, c);
DoThis(x);
DoThat();
```

Sin embargo, este esquema de nombres no funciona bien con las funciones de devolución de valor. Una sentencia como

```
z = 6.7 * ComputeMaximum(d, e, f);
```

se oye raro cuando se lee en voz alta: "Iguala `z` a 6.7 multiplicado por el *máximo cálculo* de `d`, `e` y `f`."

Con una función de devolución de valor, la llamada de función representa un valor dentro de una expresión. Es mejor asignar nombres que son sustantivos, o a veces adjetivos, a las cosas que representan valores, como las variables y funciones de devolución de valor. Vea cuánto mejora la pronunciación en voz alta de la siguiente sentencia:

```
z = 6.7 * Maximum(d, e, f);
```

Eso se leería como "Iguale z a 6.7 multiplicado por el *máximo* de `d`, `e` y `f`". Otros nombres que sugieren valores en vez de acciones son raíz cuadrada (`SquareRoot`), cubo (`Cube`), `Factorial`, Cuenta del alumno (`StudentCount`), suma de cuadrados (`SumOfSquares`) y número de seguro social (`SocialSecurityNum`). Como ve, son sustantivos o frases nominales.

Las funciones de devolución de valor (y variables) se nombran con frecuencia con adjetivos o frases que comienzan con *Is*. Aquí están algunos ejemplos:

```
while (Valid(m, n))
if (Odd(n))
if (IsTriangle(s1, s2, s3))
```

Al momento de elegir un nombre para una función de devolución de valor, trate de juntar sustantivos o adjetivos de modo que el nombre sugiera un valor, no una instrucción para la computadora.

Diseño de interfaz y efectos secundarios

El diseño de la interfaz para una función de devolución de valor es casi el mismo que el de la interfaz para una función void. Se escribe una lista de lo que necesita la función y lo que debe devolver. Debido a que las funciones de devolución de valor regresan únicamente un valor, hay sólo un elemento etiquetado como "saliente" en la lista: el valor devuelto por la función. Todo lo demás en la lista se marca como "entrante" y no hay parámetros entrantes y salientes.

La obtención de más de un valor de una función de devolución de valor (al modificar los argumentos del invocador) es un efecto secundario que se debe evitar. Si su diseño de interfaz solicita la devolución de múltiples valores, entonces debe usar una función void en vez de una función de devolución de valor.

Una norma general es nunca usar parámetros de referencia en la lista de parámetros de una función de devolución de valor, sino usar exclusivamente parámetros de valor. Considérese un ejemplo que demuestra la importancia de esta regla. Suponga que se define la siguiente función:

```
int SideEffect( int& n )
{
    int result = n * n;

    n++;                    // Efecto secundario
    return result;
}
```

Esta función devuelve el cuadrado de su valor entrante, pero incrementa también el argumento del invocador antes de la devolución. Ahora suponga que se llama a esta función con la siguiente sentencia:

```
y = x + SideEffect(x);
```

Si x es originalmente 2, ¿qué valor se almacena en y? La respuesta depende del orden en el cual su compilador genera un código para evaluar la expresión. Si el código compilado llama primero a la función, entonces la respuesta es 7. Si primero tiene acceso a x en la preparación para añadirla al resultado de la función, la respuesta es 6. Esta ambigüedad es precisamente la razón de por qué no se deben usar parámetros de referencia con funciones de devolución de valor. Una función que causa un resultado impredecible no tiene lugar en un programa bien escrito.

Una excepción es el caso en el que un objeto de flujo I/O se pasa a una función de devolución de valor. Recuerde que C++ permite que un objeto de flujo sea pasado sólo a un parámetro de referencia. Dentro de una función de devolución de valor, la única operación que se debe efectuar es probar el estado del flujo (para errores EOF o I/O). Una función de devolución de valor no debe efectuar operaciones de entrada o salida. Esta clase de operaciones se consideran efectos secundarios de la función. (Se debe señalar que no todos están de acuerdo con este punto de vista. Algunos programadores sienten que efectuar una operación I/O dentro de una función de devolución de valor es perfectamente aceptable. Usted encontrará opiniones divididas a este respecto.)

Hay otra ventaja en usar sólo parámetros de valor en una función de devolución de valor. Usted puede usar constantes y expresiones como argumentos. Por ejemplo, se puede llamar a la función `IsTriangle` con literales y otras expresiones:

```
if (IsTriangle(30.0, 60.0, 30.0 + 60.0))
    cout << "Una combinación de ángulos 30-60-90 forma un triángulo.";
else
    cout << "Algo está mal.";
```

Cuándo usar funciones de devolución de valor

No hay ninguna regla formal para determinar cuándo usar una función void y cuándo una función de devolución de valor, pero aquí están algunas normas:

1. Si el módulo debe devolver más de un valor o modificar algunos de los argumentos del invocador, no use una función de devolución de valor.
2. Si el módulo debe efectuar operaciones I/O, no use una función de devolución de valor. (Esta norma no tiene aceptación general.)
3. Si el módulo devuelve sólo un valor y es un valor booleano, es apropiada una función de devolución de valor.
4. Si hay sólo un valor devuelto y ese valor se va a usar de inmediato en una expresión, es apropiada una función de devolución de valor.
5. Cuando haya duda, use una función void. Se puede volver a codificar cualquier función de devolución de valor como una función void añadiendo un parámetro saliente extra para regresar el resultado calculado.
6. Si son aceptables una función void y una función de devolución de valor, use la que a su parecer sea más fácil poner en práctica.

Las funciones de devolución de valor se incluyeron en C++ para proveer una forma de simular el concepto matemático de una función. La biblioteca estándar de C++ suministra un conjunto de funciones matemáticas usadas comúnmente a través del encabezado de archivo `cmath`. En el apéndice C aparece una lista de éstas.

Información básica

Ignorar un valor de función

Una peculiaridad del lenguaje C++ es que le permite ignorar el valor devuelto por una función de devolución de valor. Por ejemplo, usted podría escribir la siguiente sentencia en su programa sin ninguna queja del compilador:

```
sqrt(x);
```

Cuando se ejecuta esta sentencia, el valor devuelto por `sqrt` se desecha pronto. Esta llamada de función no tiene efecto alguno, excepto que utiliza el tiempo de la computadora calculando un valor que nunca se usa.

Resulta claro que la llamada anterior para `sqrt` es un error. Ningún programador escribiría esa sentencia de manera intencional. Pero los programadores de C++ ocasionalmente escriben funciones de devolución de valor de un modo que permite al invocador ignorar el valor de la función. Aquí tiene un ejemplo específico de la biblioteca estándar de C++.

La biblioteca provee una función llamada remove, cuyo propósito es eliminar un archivo de disco del sistema. Lleva un solo argumento, una cadena C que especifica el nombre del archivo, y devuelve un valor de función. Este valor de función es un entero que le notifica el estado: 0 si la operación tuvo éxito y un valor distinto de cero si fracasó. A continuación se ilustra cómo podría llamar la función `remove`:

```
status = remove("junklife.dat");
if (status ! = 0)
    PrintErrorMsg();
```

Por otro lado, si supone que el sistema siempre logra borrar un archivo, puede ignorar el estado devuelto llamando a remove como si fuera una función void:

```
remove("junklife.dat");
```

La función remove es una especie de híbrido entre una función void y una función de devolución de valor. Desde el punto de vista conceptual, es una función void; su propósito principal es borrar un archivo, no calcular un valor que será devuelto. Sin embargo, literalmente es una función de devolución de valor. Devuelve un valor de función: el estado de la operación (el cual puede optar por ignorar).

En este libro no se escriben funciones híbridas. Se prefiere mantener distinto el concepto de una función void del de una función de devolución de valor. Pero hay dos razones de por qué todo programador de C++ debe saber acerca del tema de ignorar un valor de función. Primero, si de forma accidental llama a una función de devolución de valor como si fuera una función void, el compilador no evitará que usted cometa el error. Segundo, usted algunas veces encuentra este estilo de codificación en programas de otra persona y en la biblioteca estándar de C++. Varias de las funciones de biblioteca son técnicamente funciones de devolución de valor, pero el valor de función se usa solamente para devolver algo de importancia secundaria tal como un valor de estado.

Caso práctico de resolución de problemas

Perfil de salud

PROBLEMA Su abuela ha regresado del consultorio del médico y está confundida respecto a los números que él usó para evaluar su salud. La enfermera la pesó, le tomó la tensión arterial, anotó su nivel de colesterol tomado del examen de laboratorio, le pidió que se sentara y le dijo que no tardaría el médico. Después de media hora entró el médico, examinó su gráfica, sonrió y le dijo que estaba bien. Al revisar el informe que le entregaron a su abuela, usted comprende que podría haber un mercado para un programa que explique esta información a los pacientes médicos. Usted tiene un proyecto programado en su clase de ingeniería de software, de modo que decide escribir un programa que haga exactamente eso.

Antes de que pueda expresar la entrada y la salida de este programa, necesita hacer una investigación en la red. Encuentra que hay dos partes importantes para una lectura de colesterol: HDL (bueno) y LDL (malo). La relación de los dos es también importante. Encuentra varias referencias que proveen interpretaciones para un intervalo de valores de entrada. La interpretación de su peso depende de la relación entre su peso y su estatura representado por el índice de masa corporal (IMC). Grandioso, usted ya cuenta con un programa para calcular el IMC. La tensión arterial está conformada por dos valores: sistólico y diastólico. Al hablar de tensión arterial, las lecturas están dadas, por lo común, como "algo" sobre "algo", tal como 120/80. El primer valor es la tensión sistólica y el segundo es la tensión diastólica. Ahora, ya tiene información suficiente para determinar la entrada para su programa.

ENTRADA Nombre del paciente, HDL, LDL, peso en libras, estatura en pulgadas, tensión sistólica y tensión diastólica.

SALIDA El nombre del paciente y una interpretación de las lecturas de colesterol, peso y tensión arterial se escriben en el archivo "Perfil".

DISCUSIÓN La descomposición de este problema en tareas es muy directa. Los datos se solicitan, se introducen y se evalúan. La única pregunta es si se introducen todos los datos a la vez y se evalúan o se pide que cada módulo de evaluación introduzca sus propios datos. El principio de ocultación de información sugiere que se inserten los datos dentro de las funciones que evalúan los datos.

Aquí están las interpretaciones que encontró en Internet en relación con el colesterol.

HDL	*Interpretación*
< 40	Muy bajo
>= 40 y < 60	Es correcto
>= 60	Excelente
LDL	*Interpretación*
< 100	Óptimo
>= 100 y < 130	Cerca del óptimo
>= 130 y < 160	Límite alto
>= 160 y < 190	Alto
>= 190	Muy alto

Por fortuna, ya tiene una calculadora de IMC que simplemente puede convertir en una función (al eliminar las sugerencias de alimento). ¿Qué hay acerca de las lecturas de tensión arterial? De nuevo, aquí tiene lo que halló en la red al respecto.

Sistólica	*Interpretación*
< 120	Óptima
< 130	Normal
< 140	Normal alta
< 160	Hipertensión de etapa 1
< 180	Hipertensión de etapa 2
>= 180	Hipertensión de etapa 3

Diastólica	*Interpretación*
< 80	Óptima
< 85	Normal
< 90	Normal alta
< 100	Hipertensión de etapa 1
< 110	Hipertensión de etapa 2
>= 110	Hipertensión de etapa 3

Principal **Nivel 0**

```
Abrir archivo de salida
Si el archivo de salida no se abre de manera correcta
    Escribir mensaje de error
    Devolver 1
Obtener el nombre
Evaluar el colesterol
Evaluar el IMC
Evaluar la tensión arterial
Cerrar el archivo de salida
```

Obtener el nombre
salida: valor de función **Nivel 1**

```
Solicitar el nombre
Obtener el nombre
Solicitar el apellido
Obtener el apellido
Solicitar la inicial del segundo nombre
Obtener la inicial del segundo nombre
devolver nombre + " + nombre, segundo nombre + "." + apellido
```

Evaluar colesterol(entrada-salida: perfil de salud; entrada: nombre)

```
Solicitar el ingreso del nombre
Obtener datos
Evaluar los datos de acuerdo con las gráficas
Imprimir mensaje
```

El nombre se ha agregado a la lista de parámetros para que el usuario pueda solicitar datos específicos del paciente. Las dos primeras y la última líneas pueden ser traducidas directamente en C++.

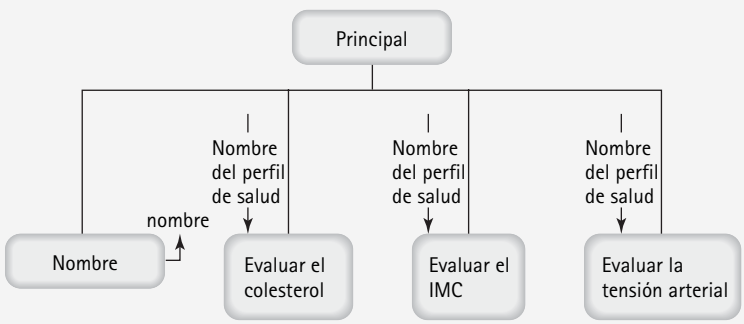
Evaluar la entrada (entrada: HDL, LDL)

```
si (HDL < 40)
    Imprimir el perfil de salud "HDL es demasiado bajo"
de otro modo si (HDL < 60)
    Imprimir el perfil de salud "HDL es correcto"
de otro modo
    Imprimir el perfil de salud "HDL es excelente"

si (LDL < 100)
    Imprimir el perfil de salud "LDL es óptimo"
de otro modo si (LDL < 130)
    Imprimir el perfil de salud "LDL es cercano al óptimo"
de otro modo si (LDL < 160)
    Imprimir el perfil de salud "LDL está en el límite alto"
de otro modo si (LDL < 190)
    Imprimir el perfil de salud "LDL es alto"
de otro modo
    Imprimir el perfil de salud "LDL es muy alto"

si (relación < 3.22)
    Imprimir el perfil de salud "La relación de LDL a HDL es buena"
de otro modo
    Imprimir el perfil de salud "La relación de LDL a HDL no es buena"
```

El algoritmo para evaluar el IMC está en el capítulo 5 y la forma de evaluar la tensión arterial es idéntica, así que se deja la escritura de su diseño como ejercicio 1 del seguimiento de caso práctico.

¿Qué módulos deben ser funciones y cuáles deben ser codificados en línea? Obtener nombre (GetName) debe ser obviamente una función de devolución de valor; evaluar colesterol (EvaluateCholesterol), evaluar índice de masa corporal (EvaluateBMI) y evaluar la tensión arterial (EvaluateBloodPressure) deben ser funciones void. La pregunta es si los módulos evaluar entrada (EvaluateInput) deben ser codificados como funciones separadas. Si este programa se fuera a codificar en un lenguaje que permitiera insertar una función dentro de una función, el buen estilo sugeriría elaborar funciones separadas de los módulos EvaluateInput. Sin embargo, C++ no permite anidar funciones. Los módulos EvaluateInput tendrían que codificarse al mismo nivel que las funciones que los llaman. Así, estos módulos se deben codificar en línea. Otra razón para proceder así es que EvaluateCholesterol, EvaluateBMI y EvaluateBloodPressure son funciones muy unidas con un propósito e interfaz claramente expresados.

GRÁFICA DE ESTRUCTURA DE MÓDULO



```
//**************************************************************************
// Programa Perfil
// Este programa introduce un nombre, peso, estatura, lecturas de tensión
// arterial y valores de colesterol. Se escriben mensajes de salud
// apropiados para cada uno de los valores de entrada en el archivo
// healthProfile. Para ahorrar espacio, se omiten de cada función los
// comentarios de precondición que documentan las suposiciones hechas
// respecto a los datos de parámetros de entrada válidos. Éstos serían
// incluidos en un programa dedicado para uso real.
//**************************************************************************

#include <fstream>
#include <iostream>
#include <string>

using namespace std;

// Prototipos de función
string Name();
void EvaluateCholesterol( ofstream& healthProfile, string name);
void EvaluateBMI( ofstream& healthProfile, string name);
void EvaluateBloodPressure( ofstream& healthProfile, string name);

int main()
{
    // Declarar y abrir el archivo de entrada
    ofstream healthProfile;
    healthProfile.open("Profile");
    if ( !healthProfile )
    { // No
        cout << "No se puede abrir el archivo." << endl;
        return 1;
    }
    string name;
    name = Name();

    // Escribir el nombre del paciente en el archivo de salida
    healthProfile << "Nombre del paciente " << name << endl;

    // Evaluar las estadísticas del paciente
    EvaluateCholesterol(healthProfile, name);
    EvaluateBMI(healthProfile, name);
    EvaluateBloodPressure(healthProfile, name);
    healthProfile << endl;

    healthProfile.close();
    return 0;
}

//**************************************************************************

string Name()
```

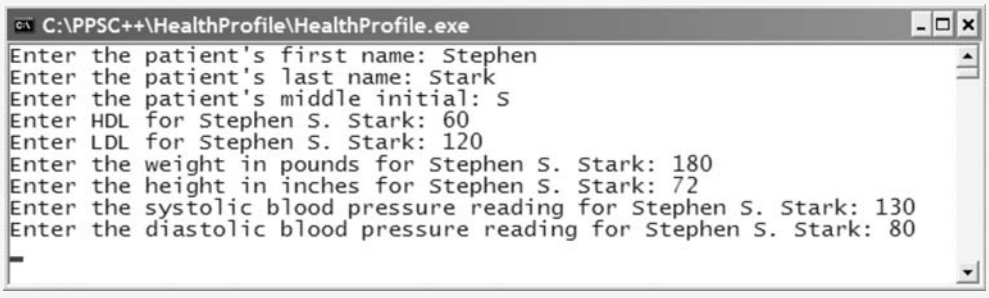
```
// Función name
// Esta función introduce un nombre y lo devuelve en el orden primer nombre,
// inicial del segundo nombre y apellido
// Poscondición:
//     El valor devuelto es la cadena compuesta por primer nombre,
//     espacio , inicial del segundo nombre, punto, espacio, apellido

{
    // Declarar el nombre del paciente
    string firstName;
    string lastName;
    char middleInitial;

    // Solicitar e introducir el nombre del paciente
    cout << "Introducir el primer nombre del paciente: ";
    cin >> firstName;
    cout << "Introducir el apellido del paciente: ";
    cin >> lastName;
    cout << "Introduzca la inicial del segundo nombre del paciente: ";
    cin >> middleInitial;
    return firstName + ' '  + middleInitial + ". " + lastName;
}

//*********************************************************************

void EvaluateCholesterol
    ( /* inout */ ofstream& healthProfile,    // Archivo de salida
      /* in */    string name )               // Nombre del paciente

// Esta función introduce HDL (colesterol bueno) y LDL (colesterol malo)
// e imprime un mensaje de salud con base en sus valores del archivo
// healthProfile.
// Precondición:
//     El archivo de salida se ha abierto con éxito
// Poscondición:
//     Los mensajes de salud apropiados para los valores de entrada
//     de HDL, LDL y su relación se han impreso en el archivo
//     healthProfile.

{
    int HDL;
    int LDL;

    // Solicitar e introducir HDL y LDL
    cout << "Introducir HDL para " << name << ": ";
    cin >> HDL;
    cout << "Introducir LDL para " << name << ": ";
    cin >> LDL;
    float ratio = LDL/HDL;                // Calcular la relación de LDL a HDL

    healthProfile << "Perfil de colesterol " << endl;
    // Imprimir un mensaje con base en el valor de HDL
    if (HDL < 40)
```

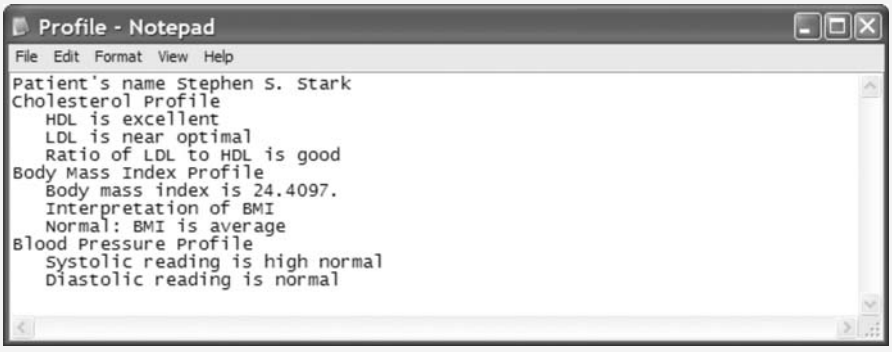
```
        healthProfile << "   HDL es demasiado bajo" << endl;
    else if (HDL < 60)
        healthProfile << "   HDL es correcto" << endl;
    else
        healthProfile << "   HDL es excelente" << endl;
    // Imprimir un mensaje con base en el valor de LDL
    if (LDL < 100)
        healthProfile << "   LDL es óptimo" << endl;
    else if (LDL < 130)
        healthProfile << "   LDL es casi óptimo" << endl;
    else if (LDL < 160)
        healthProfile << "   LDL está en el límite alto" << endl;
    else if (LDL < 190)
        healthProfile << "   LDL es alto" << endl;
    else
        healthProfile << "   LDL es muy bajo" << endl;

    if (ratio < 3.22)
        healthProfile << "   La relación de LDL a HDL es buena"
                      << endl;
    else
        healthProfile << "   La relación de LDL a HDL no es buena"
                      << endl;
}

//*****************************************************************************

void EvaluateBMI
    ( /* inout */ ofstream& healthProfile,    // Archivo de salida
      /* in */    string name )               // Nombre del paciente

// Esta función introduce el peso en libras y la estatura en pulgadas
// y calcula el índice de masa corporal (BMI imprime un mensaje
// de salud con base en el IMC). Entrada en unidades de peso inglesas
// Precondición:
//     El archivo de salida se ha abierto con éxito
// Poscondición:
//     Mensajes de salud apropiados para el IMC con base en los valores de
//     entrada de peso y estatura se han impreso en el archivo healthProfile

{
    const int BMI_CONSTANT = 703; // Constante en la fórmula de unidades
                                  // inglesas
    float pounds;
    float inches;

    // Introducir el peso y la estatura del paciente
    cout << "Introducir el peso en libras para " << name << ": ";
    cin >> pounds;
    cout << "Introducir la estatura en pulgadas para " << name << ": ";
    cin >> inches;
    float bodyMassIndex = pounds * BMI_CONSTANT / (inches * inches);
    healthProfile << "Perfil del índice de masa corporal" << endl;
```

```
    // Imprimir el índice de masa corporal
    healthProfile << "   El índice de masa corporal es " << bodyMassIndex
                  << ". " << endl;
    healthProfile << "   Interpretación del IMC " << endl;

    // Imprimir interpretación del IMC
    if (bodyMassIndex <20)
        healthProfile << "   Abajo del peso normal: el IMC es muy bajo"
                      << endl;
    else if (bodyMassIndex <=25)
        healthProfile << "   Normal: el IMC es el promedio"  << endl;
    else if (bodyMassIndex <= 30)
        healthProfile << "   Sobrepeso: el IMC es muy alto"
                      << endl;
    else
        healthProfile << "   Obeso: el IMC es peligrosamente alto"
                      << endl;
}

//*****************************************************************************

void EvaluateBloodPressure
    ( /* inout */ ofstream& healthProfile,    // Archivo de salida
      /* in */    string name )              // Nombre del paciente

// Esta función obtiene lecturas de tensión arterial (sistólica
// y diastólica) e imprime un mensaje de salud con base en sus valores
// en el archivo healthProfile
// Precondición:
//     El archivo de salida se ha abierto con éxito
// Poscondición:
//     Mensajes de salud apropiados para las lecturas de tensión arterial,
//     con base en los valores de entrada de tensión sistólica y diastólica,
//     se han impreso en el archivo healthProfile

{
    // Declarar las lecturas de tensión arterial
    int systolic;
    int diastolic;

    // Introducir las lecturas de tensión arterial del paciente
    cout << "Introducir la lectura de tensión arterial sistólica para"
         << name << ": ";
    cin  >> systolic;
    cout << "Introduzca la lectura de tensión arterial diastólica para "
         << name << ": ";
    cin >> diastolic;
    // Imprimir la interpretación de la lectura sistólica
    healthProfile << "Perfil de tensión arterial " << endl;
    if (systolic < 120)
        healthProfile << "   La lectura sistólica es óptima" << endl;
    else if (systolic < 130)
        healthProfile << "   La lectura sistólica es normal" << endl;
```

```
else if (systolic < 140)
    healthProfile << "   La lectura sistólica es normal alta"
                  << endl;
else if (systolic < 160)
    healthProfile <<
        "   La lectura sistólica indica etapa 1 de hipertensión" << endl;
else if (systolic < 180)
    healthProfile <<
        "   La lectura sistólica indica etapa 2 de hipertensión" << endl;
else
    healthProfile <<
        "   La lectura sistólica indica etapa 3 de hipertensión" << endl;

// Imprimir la interpretación de la lectura diastólica
if (diastolic < 80)
    healthProfile << "   La lectura diastólica es óptima" << endl;
else if (diastolic < 85)
    healthProfile << "   La lectura diastólica es normal" << endl;
else if (diastolic < 90)
    healthProfile << "   La lectura diastólica es normal alta"
                  << endl;
else if (diastolic < 100)
    healthProfile <<
       "   La lectura diastólica indica etapa 1 de hipertensión" << endl;
else if (diastolic < 110)
    healthProfile <<
       "   La lectura diastólica indica etapa 2 de hipertensión" << endl;
else
    healthProfile <<
       "   La lectura diastólica indica etapa 3 de hipertensión" << endl;
```

Aquí está una muestra de la entrada y la salida





La imagen mostrada corresponde a la salida producida por el programa original, escrito en idioma inglés.

PRUEBA Hay nueve entradas para este programa: dos cadenas y un carácter en Name, dos valores enteros en `EvaluateChelesterol`, dos valores de punto flotante en `EvaluateBMI` y dos valores enteros en `EvaluateBloodPressure`. El código en Name duplica el código del programa Nombre en el capítulo 4, de modo que se puede suponer que es correcto. Es necesario comprobar por separado cada una de las otras funciones, eligiendo valores que ocurren en los puntos de detención en las sentencias If de cada función.

No se ha incluido ninguna validación de datos en este caso práctico porque los valores se introducen desde una gráfica del paciente. Aunque se puede suponer que los valores de la gráfica son correctos, no se puede suponer que sean introducidos de manera correcta. En el ejercicio 3 de seguimiento de caso práctico se le pide añadir la validación de datos a este programa.

Consejo práctico de ingeniería de software

Abstracción de control, cohesión funcional y complejidad de comunicación

El programa Perfil contiene cuatro funciones, lo cual hace que la función principal se vea compleja. Sin embargo, si examina cada uno de los módulos, la estructura de control más complicada es una sentencia If con sentencias If anidadas dentro de ella.

La complejidad de un programa se oculta al reducir cada una de las estructuras de control principales a una acción abstracta que aparece como una llamada para evaluar el IMC. Las propiedades lógicas de la acción se separan de su ejecución. Este aspecto de un diseño se llama **abstracción de control**.

Abstracción de control La separación de propiedades lógicas de una acción de su ejecución.

Cohesión funcional El principio de que un módulo debe llevar a cabo exactamente una acción abstracta.

Complejidad de comunicación Una media de la cantidad de datos que pasan por la interfaz de un módulo.

La abstracción de control puede servir como una guía para decidir qué módulos codificar como funciones y cuáles codificar directamente. Si un módulo contiene una estructura de control, es un buen candidato para ser puesto en práctica como una función. Pero incluso si un módulo no contiene una estructura de control, usted aún desea considerar otros factores. ¿Es prolongado, o es llamado desde más de un lugar? Si es así, debe usar una función.

Un poco relacionado con la abstracción de control es el concepto de **cohesión funcional**, que expresa que un módulo debe llevar a cabo exactamente una abstracción.

Si puede expresar la acción que realiza un módulo en una sentencia sin ninguna conjunción (*y*), entonces es muy cohesivo. Un módulo que tiene más de un propósito principal carece de cohesión. Aparte de `main`, todas las funciones del programa Perfil tienen buena cohesión.

Un módulo que sólo satisface parcialmente un propósito carece de cohesión. Tal módulo se debe combinar con cualquier otro que tenga relación directa con él. Por ejemplo, no tendría sentido separar una función que imprime el primer dígito de una fecha porque imprimir una fecha es una acción de abstracción.

Un tercer aspecto relacionado de un diseño de módulo es su **complejidad de comunicación**, la cantidad de datos que pasan por la interfaz de un módulo; por ejemplo, el número de argumentos. La complejidad de comunicación de un módulo es con frecuencia un indicador de su cohesión. Por lo general, si un módulo requiere un gran número de argumentos, éste trata de realizar mucho o sólo satisface en forma parcial un propósito. Se debe volver y ver si hay otra manera de dividir el problema de modo que se transmita una cantidad mínima de datos entre módulos. Los módulos en Perfil tienen poca complejidad de comunicación.

Prueba y depuración

Una de las ventajas de un diseño modular es que puede probarlo mucho antes de que se haya escrito el código para todos los módulos. Si se prueba cada uno de los módulos, entonces puede ensamblar módulos en un programa completo con mucha mayor seguridad de que el programa sea correcto. En esta sección se presenta una técnica para probar un módulo por separado.

Talones y manejadores

Suponga que recibió el código para un módulo y su trabajo fue probarlo. ¿Cómo probaría un solo módulo por sí mismo? En principio, debe ser llamado por algo (a menos que sea la función `main`). Segundo, puede tener llamadas a otros módulos que no están disponibles para usted. Para probar el módulo, debe llenar estos enlaces faltantes.

Cuando un módulo contiene llamadas a otros módulos, se pueden escribir funciones ficticias llamadas **talones** para satisfacer esas llamadas.

Talón Una función ficticia que ayuda a probar parte de un programa. Un talón tiene el mismo nombre e interfaz que una función que sería llamada en realidad por la parte del programa que se está probando, pero es, por lo común, mucho más simple.

Un talón consta normalmente de una sentencia de salida que imprime un mensaje como "Función tal recién llamada". Aunque el talón es ficticio, permite determinar si la función es llamada en el momento correcto por la función `main` u otra función.

Un talón se puede usar también para imprimir un conjunto de valores que se pasan a ésta; esto dice si un módulo que está siendo probado suministra la información correcta o no. Algunas veces un talón asigna nuevos valores a sus parámetros de referencia para simular datos que se leen o resultados que se calculan para dar al módulo invocador algo en lo que siga trabajando. Debido a que se pueden elegir los valores que devuelve el talón, se tiene mejor control sobre las condiciones de la corrida de prueba.

Aquí está un talón que simula la función Name en el programa Perfil devolviendo una cadena elegida de modo arbitrario.

```
string Name()
// Talón para la función Name en el programa Perfil

{
    cout << "Aquí se llamó a Name, y se obtuvo \"John J. Smith\"."
         << endl;
    return "John J. Smith";
}
```

Este talón es más simple que la función que simula, lo cual es característico porque el objetivo de usar un talón es proveer un ambiente simple y predecible para probar un módulo.

Además de proveer un talón para cada llamada dentro del módulo, se debe proveer un programa ficticio, un **manejador**, para llamar al módulo mismo. Un programa manejador contiene el código esencial requerido para llamar al módulo que se está probando.

Manejador Una función principal simple que se emplea para llamar a una función que está siendo probada. El uso de un programa manejador permite dirigir el control del proceso de prueba.

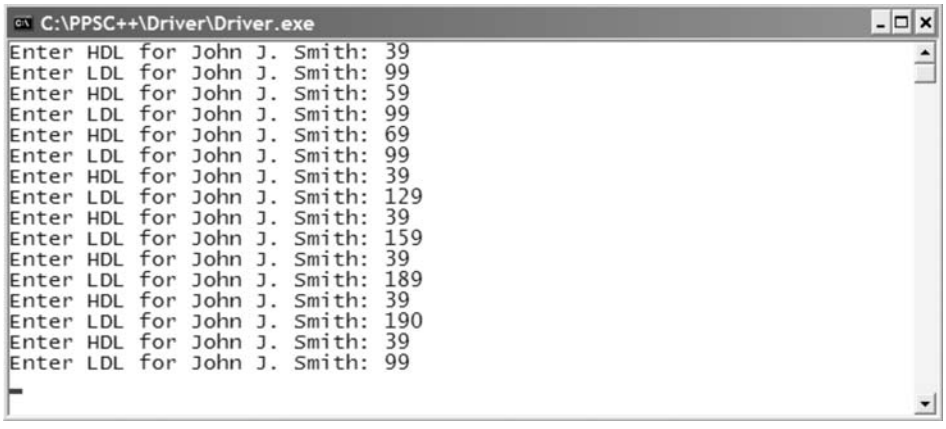
Al cercar un módulo con un programa manejador y talones, se logra el control completo de las condiciones en las que se ejecuta. Esto permite probar distintas situaciones y combinaciones que pueden revelar errores. Por ejemplo, el siguiente programa es un programa manejador para la función `EvaluateCholesterol` en el programa Perfil.

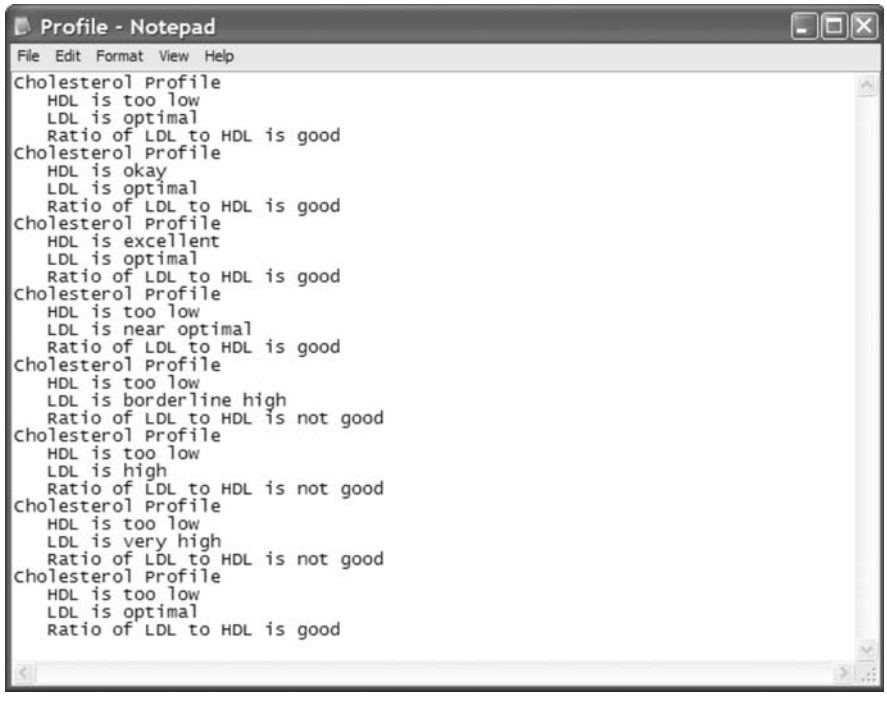
```
//**************************************************************************
// Programa manejador EvaluateCholesterol
// Este programa provee un ambiente para probar la función
// EvaluateCholesterol en ausencia del programa
// Perfil
//**************************************************************************

#include <iostream>
#include <fstream>
```

```
using namespace std;

// Prototipo de función
void EvaluateCholesterol(ofstream&, string);

int main()
{
    ofstream healthProfile;
    healthProfile.open("Profile");
    string name = "John J. Smith";
    for (int test = 1; test <= 8; test++)
        EvaluateCholesterol(healthProfile, name);
    healthProfile.close();
    return 0;
}

//*************************************************************************

void EvaluateCholesterol
    ( /* inout */ ofstream& healthProfile,    // Archivo de salida
      /* in */    string name )               // Nombre del paciente

// Esta función introduce los niveles de HDL (colesterol bueno)
// y LDL (colesterol malo) e imprime un mensaje de salud con base
// en sus valores en el archivo healthProfile.
// Precondición:
//     El archivo de salida se ha abierto con éxito
// Poscondición:
//     Se han impreso mensajes de salud apropiados para los valores
//     de entrada de HDL, LDL y su relación en el archivo
//     healthProfile.

{
    int HDL;
    int LDL;

    // Solicitar e introducir HDL y LDL
    cout << "Introducir HDL para " << name << ": ";
    cin >> HDL;
    cout << "Introducir LDL para " << name << ": ";
    cin >> LDL;

    float ratio = LDL/HDL;               // Calcular la relación de LDL a HDL
    healthProfile << "Perfil de colesterol" << endl;
    // Imprimir el mensaje con base en el valor de HDL
    if (HDL < 40)
        healthProfile << "   HDL es demasiado bajo" << endl;
    else if (HDL < 60)
        healthProfile << "   HDL es correcto" << endl;
    else
        healthProfile << "   HDL es excelente" << endl;
    // Imprimir el mensaje con base en el valor de LDL
    if (LDL < 100)
        healthProfile << "   LDL es óptimo" << endl;
    else if (LDL < 130)
        healthProfile << "   LDL es cercano al óptimo" << endl;
```

```
    else if (LDL < 160)
        healthProfile << "   LDL está en el límite alto" << endl;
    else if (LDL < 190)
        healthProfile << "   LDL es alto" << endl;
    else
        healthProfile << "   LDL es muy alto" << endl;
    if (ratio < 3.22)

        healthProfile << "   La relación de LDL a HDL es buena"
                      << endl;
    else
        healthProfile << "   La relación de LDL a HDL no es buena"
                      << endl;
}
```

El programa manejador llama a la función ocho veces, lo cual permite que se ejecuten las sentencias If. A continuación se muestra la pantalla de la entrada y la salida.





Los talones y programas manejadores son herramientas importantes en la programación en equipo. Los programadores desarrollan el diseño global y las interfaces entre los módulos. Entonces, cada programador diseña y codifica uno o más módulos y usa programas manejadores y talones para probar el código. Cuando todos los módulos se han probado y codificado, se ensamblan en lo que debe ser un programa práctico.

Para que el equipo de programación tenga éxito, es esencial que todas las interfaces de módulo sean definidas explícitamente y que los módulos codificados se adhieran de modo estricto a las especificaciones para esas interfaces. Resulta obvio que se deben evitar de manera cuidadosa las referencias a variables globales en una situación de programación en equipo porque es imposible que cada persona sepa cómo el resto del equipo está usando cada variable.

Sugerencias de prueba y depuración

1. Asegúrese de que las variables empleadas como argumentos para una función sean declaradas en el bloque donde se hace la llamada de función.
2. Defina cuidadosamente la precondición, poscondición y lista de parámetros para eliminar efectos secundarios. Las variables que se emplean sólo en una función se deben declarar como variables locales. *No* use variables locales en sus programas. (Excepción: es aceptable hacer referencia a `cin` y `cout` de modo global.)
3. Si el compilador muestra un mensaje como "UNDECLARED IDENTIFIER", compruebe que el identificador no esté mal escrito (y que de hecho, esté declarado), que el identificador está declarado antes de hacer referencia a él y que el alcance del identificador incluya la referencia a él.
4. Si pretende usar un nombre local que es igual a un nombre no local, una mala ortografía en la declaración local causará estragos. El compilador de C++ no se quejará, pero causará que toda referencia al nombre local se dirija al nombre no local.
5. Recuerde que no se puede usar el mismo identificador tanto en la lista de parámetros como en las declaraciones externas de una función.
6. Con una función de devolución de valor, asegúrese de que el encabezado y el prototipo de función comiencen con el tipo de datos correcto para la función de devolución de valor.
7. Con una función de devolución de valor, no olvide usar una sentencia

   `return` Expresión;

   para devolver el valor de función. Asegúrese de que la expresión es del tipo correcto, o de lo contrario ocurrirá coerción implícita de tipos.
8. Recuerde que una llamada para una función de devolución de valor es parte de una expresión, mientras que una llamada para una función void es una sentencia separada. (C++ suaviza esta distinción al permitir que usted llame una función de devolución de valor como si fuera una función void, ignorando el valor de retorno. Tenga cuidado aquí.)
9. En general, no use parámetros de referencia en la lista de parámetros de una función de devolución de valor. Sin embargo, se debe usar un parámetro de referencia cuando se pasa como parámetro un objeto de flujo I/O.
10. Si es necesario, use el depurador de su sistema (o use sentencias de salida depuradoras) para indicar cuándo se llama a una función y si se está ejecutando de modo correcto. Los valores de los argumentos se pueden mostrar antes de la llamada para la función (para mostrar los valores entrantes) y después (para mostrar los valores salientes). Es posible que también quiera mostrar los valores de variables locales en la función misma para indicar lo que sucede cada vez que se llama.

Resumen

El alcance de un identificador se refiere a las partes del programa en el que es visible. Los nombres de funciones en C++ tienen alcance global, así como los nombres de variables y constantes que se declaran fuera de las funciones y espacios de nombres. Las variables y constantes declaradas dentro de un bloque tienen alcance local; no son visibles fuera del bloque. Los parámetros de una función tienen el mismo alcance que las variables locales declaradas en el bloque exterior de la función.

Con raras excepciones, no se considera buena práctica declarar variables globales y referirlas directamente dentro de una función. Toda comunicación entre los módulos de un programa debe darse a través de las listas de argumentos y parámetros (y vía el valor de función enviado de regreso por una función de devolución de valor). Por otro lado, el uso de constantes globales se considera una práctica de programación aceptable porque agrega congruencia y hace más fácil cambiar un programa al tiempo que se evitan las trampas de los efectos secundarios. Las funciones bien documentas y bien diseñadas que están libres de efectos secundarios pueden volver a usarse en otros programas. Muchos programadores mantienen una biblioteca de funciones que usan de modo repetido.

La duración de una variable es el periodo durante la ejecución del programa en que se le asigna memoria. Las variable globales tienen duración estática (la memoria permanece asignada el tiempo que dura la ejecución del programa). Por defecto, las variables locales tienen duración automática (la memoria se asigna y libera en la entrada y en la salida del bloque). A una variable local se le puede dar duración estática si se emplea la palabra `static` en su declaración. Esta variable tiene la duración de una variable global pero el alcance de una variable local.

C++ permite que una variable se inicialice en su declaración. Para una variable estática, la inicialización ocurre sólo una vez, cuando el control alcanza primero su declaración. Una variable automática se inicializa cada vez que el control de tiempo llega a la declaración.

C++ provee dos clases de subprogramas, funciones void y funciones de devolución de valor, para que sean utilizadas. Una función de devolución de valor es llamada desde dentro de una expresión y devuelve un solo resultado que se emplea en la evaluación de la expresión. Para que el valor de función sea devuelto, la última sentencia ejecutada por la función debe ser una sentencia Return que contenga una expresión del tipo de datos apropiado.

Todas las reglas de alcance, así como las reglas acerca de los parámetros de referencia y valor, se aplican tanto a funciones void como a funciones de devolución de valor. Sin embargo, se considera una mala práctica de programación usar parámetros de referencia en una definición de función de devolución de valor. Proceder así incrementa el potencial de efectos secundarios accidentales. (Una excepción es cuando los objetos de flujo I/O se pasan como parámetros. En capítulos posteriores se observan otras excepciones.)

Se pueden usar talones y directores para probar funciones aisladas del resto de un programa. Son particularmente útiles en el contexto de proyectos de programación en equipo.

Comprobación rápida

1. Si una función hace referencia a una variable que no esté declarada en su bloque o su lista de parámetros, ¿la referencia es global o local? (pp. 298-303)
2. ¿De qué manera las referencias a variables globales contribuyen al potencial para efectos secundarios indeseables? (pp. 306-310)
3. Si un módulo tiene tres parámetros **`/* in */`** y un parámetro **`/* out */`**, ¿se debe poner en práctica por una función void o una función de devolución de valor? (p. 320)
4. Un programa tiene dos funciones, `Quick1` y `Quick2`. El programa en sí declara variables `check` y `qc`. La función Quick1 declara variables llamadas `quest` y `qc`. La función `Quick2` declara una variable llamada `quest` y una variable `static` llamada `forever`. ¿Cuáles de estas variables son locales? (pp. 298-303)
5. En la pregunta 4 de Comprobación rápida, ¿cuáles variables son accesibles dentro de la función `Quick1`? (pp. 298-303)
6. En la pregunta 4 de Comprobación rápida, ¿cuál es la duración de cada una de las seis variables? (pp. 306-308)

7. ¿Qué distingue a una función de devolución de valor de una función void? (pp. 312-316)
8. Dado el siguiente encabezado de función, ¿cómo escribiría una llamada para él que pase el valor 98.6 y asigne el resultado a la variable `fever`? (pp. 316-318)

```
bool TempCheck (/* in */ float temp)
```

Respuestas

1. Global. **2.** Permiten que una función afecte el estado del programa por un medio distinto a la interfaz bien definida de la lista de parámetros. **3.** Una función de devolución de valor. **4.** Las variables declaradas dentro de las funciones son locales. En `Quick1`, `quest` y `qc`. En `Quick2`, `quest` y `forever`. **5.** `check`, `quest` y `qc` declarada localmente. **6.** `check` y `qc` en el programa, junto con `forever`, son estáticas. Las variables en `Quick1`, y la variable `quest` en `Quick2` son automáticas. **7.** Usar un nombre de tipo en lugar de `void` y usar una sentencia `return` para pasar un valor de regreso al invocador. **8.** `fever = TempCheck(98.6);`

Ejercicios de preparación para examen

1. Un parámetro de función es local para todo el bloque que es el cuerpo de la función. ¿Verdadero o falso?
2. Un parámetro de referencia tiene el mismo alcance que una variable global. ¿Verdadero o falso?
3. Se puede hacer referencia a una variable global en cualquier parte dentro del programa. ¿Verdadero o falso?
4. Los nombres de funciones tienen alcance global. ¿Verdadero o falso?
5. Compare los siguientes términos con las definiciones dadas a continuación.
   - ***a***) Alcance
   - ***b***) Prioridad de nombre
   - ***c***) Reglas de alcance
   - ***d***) Identificador no local
   - ***e***) Duración
   - ***f***) Variable automática
   - ***g***) Variable estática
   - ***h***) Efecto secundario
     - ***i***) La semántica que especifica dónde se puede hacer referencia a identificadores no locales.
     - ***ii***) Una variable para la cual se asigna memoria el tiempo que dura el programa.
     - ***iii***) Cuando una función afecta a otra de una manera que no está definida por su interfaz.
     - ***iv***) La prioridad que tiene un identificador local sobre un identificador global con el mismo nombre.
     - ***v***) La región del código de programa donde se permite hacer referencia a un identificador.
     - ***vi***) Una variable que tiene memoria asignada en la entrada del bloque y liberada en la salida.
     - ***vii***) Un identificador declarado fuera del bloque actual.
     - ***viii***) El periodo en el cual un identificador tiene memoria asignada.
6. Identifique el efecto secundario en la siguiente función.

```
int ExamPrep (int param1, int& param2)
{
    if (param2 = param1)
        return param2;
    else if (param2 > param1)
        return param1;
    else
        return param1 * param2;
}
```

7. Identifique el efecto secundario en el siguiente programa (el cual usa un estilo deficiente para nombrar variables).

```
#include <iostream>

using namespace std;

string a;
int w;

bool GetYesOrNo();

int main ()
{
    cout << "Introducir el nombre:";
    cin >> a;
    w = 0;
    cout << "¿Hay datos de peso?";
    if (GetYesOrNo());
    {
        cout << "Introduzca el peso:";
        cin >> w;
    }
    cout << "El nombre es " << a << " el peso es " << w << end1;
}

bool GetYesOrNo()
{
    cout << "Introduzca sí o no: ";
    cin >> a;
    return a == "sí";
}
```

8. ¿Cuál es el alcance de espacio de nombre que se especifica con la directiva fuera de las funciones?
9. ¿Cuál es el alcance del nombre de espacio `std` en el siguiente código?

```
// Incluir directivas y prototipos de funciones aquí

int main()
{
    using namespace std;
    // Aquí está el resto del cuerpo de main
}
// Aquí están las definiciones de función
```

10. ¿Cuál es la duración de cada una de las siguientes variables?
    a) Una variable global.
    b) Una variable local en una función.
    c) Una variable local, estática en una función.
11. Rescriba la siguiente declaración e inicialización como una sola sentencia.

```
float pi;
pi = 3.14159265;
```

12. Si una variable local, estática, se inicializa en su declaración dentro de una función, ¿cuándo se inicializa la variable y con cuánta frecuencia?
13. Si una variable local, no estática, se inicializa en su declaración dentro de una función, ¿cuándo se inicializa la variable y con cuánta frecuencia?
14. Una función de devolución de valor puede tener sólo una sentencia return. ¿Verdadero o falso?
15. ¿Cuál es el error en la siguiente función?

```
bool Greater (int a, int b)
{
    if (a > b)
        return true;
}
```

16. ¿Qué está mal en la siguiente función?

```
int Average (int a, int b, int c)
{
    return (a + b + c)/3.0;
}
```

17. ¿Cuál es el error en la siguiente función?

```
void Maximum (int a, int b, int& max)
{
    if (a > b)
        max = a;
    else
        max = b;
    return max;
}
```

18. ¿Cuál es el error más comun que se comete al usar un parámetro de referencia en una función de devolución de valor?

Ejercicios de calentamiento para programación

1. El siguiente programa está escrito en un estilo muy malo que usa variables globales en lugar de parámetros y argumentos, lo cual da como resultado efectos secundarios indeseados. Rescríbalo con un buen estilo.

```
#include <iostream>
using namespace std;
int Power ();
int pow;
int x;
int result;
int main ()
{
cout << "Introducir la potencia: ";
cin >> pow;
cout << "Introducir el valor que será elevado a la potencia: ";
cin >> x;
cout << Power(x, pow);
}
int Power()
```

```
{
result = 1;
while (pow > 0)
{
result = result * x;
pow--;
}
return result;
}
```

2. Escriba el encabezado para una función `bool`, `Equals`, que tiene dos parámetros float `/* in */`, `x` y `y`.
3. Escriba un prototipo de función para la función del ejercicio 2.
4. Escriba un cuerpo para la función del ejercicio 2 que compare `x` y `y`, y devuelva `true` si su diferencia es menor que 0.00000001 y `false` en caso contrario.
5. Escriba el encabezado y el prototipo de función para una función float llamada ConeVolume que toma dos parámetros float `/* in */`, radio y altura.
6. Escriba el cuerpo para el encabezado de función del ejercicio 5. El cuerpo calcula el volumen de un cono con la fórmula siguiente.

$$\frac{1}{3}\ \pi \times radio^2 \times altura$$

7. Reescriba la función `void` descrita en los ejercicios de calentamiento para programación 4 y 6 del capítulo 7 como una función de devolución de valor. La función se llama `GetLeast` y toma un parámetro `ifstream` llamado `infile` como un parámetro de entrada que se cambia. Devuelve un valor `int` que es el valor mínimo leído del archivo.
8. Reescriba la función `void` llamada `Reverse` descrita en los ejercicios de calentamiento para programación 8 y 10 del capítulo 7 como una función de devolución de valor. Ésta debe tomar un parámetro `string` como entrada. La función devuelve una cadena que es el inverso carácter por carácter de la cadena en el parámetro. El parámetro se llama `original`.
9. Rescriba la función void `LowerCount` descrita en el ejercicio de calentamiento para programación 12 del capítulo 7 como una función de devolución de valor. La función lee una línea de `cin` y devuelve un `int` que contiene el número de letras minúsculas en la línea. En el apéndice C encontrará la descripción de la función `islower`, que devuelve `true` si su parámetro `char` es un carácter en minúscula.
10. Escriba una función `float` de devolución de valor llamada `SquareKm` que toma dos parámetros `float`, longitud y ancho, y el valor devuelto es en kilómetros cuadrados. Los parámetros son la longitud y el ancho del área en millas. El factor para convertir de millas a kilómetros es 1.6.
11. Escriba una función `bool` de devolución de valor llamada `Exhausted` que toma un parámetro `int` llamado `filmRolls`. La función sigue la pista de cuántos rollos de película se han procesado con la mezcla química en una máquina de procesamiento de fotos. Cuando el número total de rollos pasa de 1 000, devuelve `true`. Cada vez que es llamada, el valor del parámetro se agrega al total. Cuando el total pasa de 1 000, la variable que contiene el total se restablece a cero antes de que regrese la función, bajo la suposición de que la mezcla química será remplazada antes de que se procesen más rollos.
12. Escriba una función `string` de devolución de valor llamada `MonthAbrrev` que toma un valor `int` como parámetro. El parámetro, month, representa el número del mes. La función devuelve una cadena que contiene la abreviatura de tres letras para el número de mes correspondiente. Suponga que el número de mes está en el intervalo de 1 a 12.
13. Modifique la función del ejercicio 12 para manejar los números de mes que no están en el intervalo válido devolviendo la cadena `"Inv"`.
14. Escriba una función `float` de devolución de valor llamada `RunningAvg` que toma una variable `float`, `value`, como su entrada y devuelve el promedio corriente para todos los valores que se han pasado a la función desde la primera vez que se llamó al programa.

Problemas de programación

1. Usted está trabajando en un proyecto que requiere datos del clima para un lugar. El cambio diario máximo en la presión barométrica es un aspecto del clima que necesita su compañía. Usted tiene el archivo (`barometric.dat`) con lecturas barométricas de cada hora tomadas en el curso de un año. Cada línea del archivo contiene las lecturas para un solo día, separadas por espacios. Cada lectura se expresa en pulgadas de mercurio, así que es un decimal que varía de casi 28.00 a 32.00. Para cada línea de datos, es necesario determinar la lectura máxima y mínima, y producir la diferencia entre esas lecturas en el archivo `differences.dat`. Cada valor de salida debe estar en una línea separada del archivo. Una vez que se ha leído el archivo, el programa debe producir la diferencia máxima y mínima para el año en `cout`. Elabore el programa con descomposición funcional y use un estilo y documentación apropiados en su código. Su programa debe hacer uso apropiado de las funciones de devolución de valor al resolver este problema.
2. Amplíe el programa del problema 1 de modo que también produzca las lecturas barométricas máxima y mínima para cada día en el archivo `differences.dat`, y luego produzca las lecturas máxima y mínima para el año en `cout` al final de la ejecución del programa.
3. Usted está trabajando para una compañía maderera y le gustaría un programa que calcule el costo de la madera para un pedido. La compañía vende madera de pino, abeto, cedro, maple y roble. El precio de la madera es por pie tablar. Un pie tablar es igual a un pie cuadrado por una pulgada de espesor. El precio del pie tablar está dado en la siguiente tabla:

Pino	0.89
Abeto	1.09
Cedro	2.26
Maple	4.50
Roble	3.10

   La madera se vende en distintas dimensiones (especificadas en pulgadas de ancho y altura y pies de largo) que deben convertirse a pie tablar. Por ejemplo, una pieza de $2 \times 4 \times 8$ es de dos pulgadas de ancho, 4 pulgadas de alto y 8 pies de largo, y es equivalente a 5.333 pies tablares. Una entrada del usuario será en la forma de una letra y cuatro números enteros. Los enteros son: número de piezas, ancho, altura y longitud. La letra será una de P, A, C, M, R (que corresponden a las cinco clases de madera) o T, que significa total. Cuando la letra es T, no hay enteros después de ella en la línea. El programa debe imprimir el precio para cada entrada e imprimir el total después de introducir T. Aquí está un ejemplo de corrida:

   ```
   Introducir elemento: P 10 2 4 8
   10 2×4×8 Pino, costo: $47.47
   Introducir elemento: M 1 1 12 8
   1 1×12×8 Maple, costo: $36.00
   Introducir elemento: T
   Costo total: $83.47
   ```

   Elabore el programa con descomposición funcional y use el estilo y documentación apropiados en su código. Su programa debe hacer uso apropiado de funciones de devolución de valor al resolver este problema. Asegúrese de que los mensajes para el usuario sean claros y que el resultado esté marcado de manera apropiada.
4. Escriba un programa que determine el día de la semana para una fecha determinada. Usted puede inventar su propio algoritmo complejo que tome en cuenta las reglas especiales de año bisiesto y cambios en calendarios, pero éste es un caso donde tiene sentido buscar cosas que sean familiares. ¿Qué más podría requerirse para calcular valores de fechas en un intervalo amplio de tiempo? Los historiadores trabajan con fechas, pero en general no calculan a partir de ellas. Sin embargo, los astrónomos necesitan saber la deferencia en tiempo entre sucesos orbitales en el sistema solar que abarcan cientos de años. Si consulta un texto de astronomía, encontrará que hay una manera estándar de representar una fecha, llamada número de día jualiano (NDJ). Éste

es el número de días que han transcurrido desde el primero de enero del año 4713 a.C. Dado el NDJ para una fecha, hay una fórmula simple que dice el día de la semana:

```
dayOfWeek = (JDN + 1) % 7
```

El resultado está en el intervalo de 0 a 6, con 0 que representa Domingo.

El único problema restante es cómo calcular el NDJ, que no es tan simple. El algoritmo calcula varios resultados intermedios que se suman para dar el NDJ. Se examina el cálculo de cada uno de estos tres valores intermedios.

Si la fecha viene del calendario gregoriano (después del 15 de octubre de 1582), entonces calcule `intRes1` con la siguiente fórmula, de lo contrario `intRes1` será cero.

```
intRes1 = 2 - año / 100 + año / 400
```
(división de enteros)

El segundo resultado intermedio se calcula como sigue:

```
intRes2 = int(365.25 * Year)
```

Se calcula el tercer valor intermedio con esta fórmula:

```
intRes3 = int(30.6001 * (month + 1)
```

Por último, el NDJ se calcula de esta manera:

```
JDN = intRes1 + intRes2 + intRes3 + day = 1720994.5
```

Su programa debe hacer uso adecuado de las funciones de devolución de valor al resolver este problema. Las fórmulas requieren nueve cifras significativas; es posible que tenga que usar el tipo entero long y el tipo de punto flotante `double`. Su programa debe solicitar de modo apropiado la introducción de la fecha y marcar adecuadamente el resultado. Use el estilo de codificación idóneo con comentarios para documentar el algoritmo según sea necesario.

5. Con la reutilización de las funciones del problema 4 según convenga, escriba un programa en C++ que calcule el número de días entre dos fechas. Si su diseño para el problema 4 usa buena descomposición funcional, la escritura de este programa debe ser trivial. Sólo necesita introducir dos fechas, convertirlas en sus NDJ y sacar la diferencia de los NDJ.

   Su programa debe hacer uso apropiado de funciones de devolución de valor al resolver este problema. Estas fórmulas requieren nueve cifras significativas; es posible que tenga que usar el tipo entero `long` y el tipo de punto flotante `double`. Su programa debe solicitar la introducción de la fecha y marcar la salida de modo apropiado. Use un estilo de codificación adecuado con comentarios para documentar el algoritmo según sea necesario.

Seguimiento de caso práctico

1. Escriba el algoritmo para el módulo EvaluateBloodPressure.
2. Escriba un manejador y pruebe la función `EvaluateBloodPressure`, asegurándose de que se toma cada rama.
3. Sume una función booleana a los tres módulos Evaluate, lo cual es cierto si los datos son correctos y falso en caso contrario. Pruebe la entrada dentro de cada módulo para valores de entrada negativos o cero, que deben considerarse como errores. Si hay algún error, exprese en qué módulo ocurre y establezca la bandera de error.
4. Rescriba el programa `main` para probar esta bandera después de cada llamada y termine el programa si ocurre un error.

CAPÍTULO

9

Estructuras de control adicionales

Objetivos

Objetivos de conocimiento

- *Saber cómo elegir la sentencia de iteración más apropiada para un problema dado.*
- *Entender el propósito de las sentencias Break y Continue*

Objetivos de habilidades

Ser capaz de:

- *Escribir una sentencia Swith para un problema de ramificación multivía*
- *Escribir una sentencia Do-While y contrastarla con una sentencia While*
- *Escribir una sentencia For para poner en práctica un ciclo controlado por conteo*

En los capítulos precedentes se introdujeron sentencias de C++ para estructuras de secuencia, selección, ciclo y subprograma. En algunos casos se mencionó más de una forma de poner en práctica dichas estructuras. Por ejemplo, la selección se puede ejecutar mediante una estructura If-Then o una If-Then-Else. If-Then es suficiente para ejecutar cualquier estructura de selección, pero C++ proporciona If-Then-Else por conveniencia porque la rama bidireccional se emplea con frecuencia en programación.

En este capítulo se introducen cinco nuevas sentencias no esenciales, pero sí convenientes, para programar. Una, la sentencia Switch, facilita escribir estructuras de selección que tienen muchas ramas. Dos nuevas sentencias de iteración, For y Do-While, hacen más fácil programar algunos tipos de ciclos. Las otras dos, Break y Continue, son sentencias de control que se usan como parte de estructuras de iteración y selección más grandes.

9.1 La sentencia Switch

La sentencia Switch es una estructura de control de selección que permite listar cualquier número de ramas. En otras palabras, es una estructura de control para ramas multivía. Una sentencia Switch es similar a sentencias If anidadas. El valor de la **expresión switch** —que se hace corresponder con una etiqueta unida a una rama— determina cuál de las ramas se ejecuta. Por ejemplo, en la siguiente sentencia:

Expresión Switch Expresión cuyo valor determina qué etiqueta se selecciona. No puede ser una expresión de punto flotante o cadena.

```
switch (letter)
{
    case 'X' : Sentencia1;
               break;
    case 'L' :
    case 'M' : Sentencia2;
               break;
    case 'S' : Sentencia3;
               break;
    default  : Sentencia4;
}
Sentencia5;
```

En este ejemplo, `letter` es una expresión switch. La sentencia significa "Si `letter` es 'X', ejecuta Sentencia1 y sal de la sentencia Switch, y continúa con sentencia5. Si `letter` es 'L' o 'M', ejecuta Sentencia2 y continúa con Sentencia5. Si `letter` es 'S', ejecuta Sentencia3 y continúa con Sentencia5. Si `letter` es ninguno de los caracteres mencionados, ejecuta Sentencia4 y continúa con Sentencia5". La sentencia Break ocasiona una salida inmediata de la sentencia Switch. En breve se verá lo que sucede si se omiten las sentencias Break.

La plantilla de sintaxis para la sentencia Switch es

Sentencia Switch

```
switch ( Expresión integral o enum )
{
    Etiqueta de cambio . . . Sentencia
                ⋮
}
```

La expresión integral o `enum` es una expresión de tipo integral —`char`, `short`, `int`, `long`, `bool`—, o de tipo `enum` (en el siguiente capítulo se analiza `enum`). La etiqueta de Switch enfrente de una sentencia es una *etiqueta de caso* o una *etiqueta por omisión*:

Etiqueta de Switch

```
case Expresión constante :
default :
```

En una etiqueta de caso, la expresión constante es una expresión integral o `enum` cuyos operandos deben ser literales o constantes nombradas. Los siguientes son ejemplos de expresiones integrales constantes (donde `CLASS_SIZE` es una constante nombrada de tipo `int`):

```
3
CLASS_SIZE
'A'
2 * CLASS_SIZE + 1
```

El tipo de datos de Expresión constante es forzado, si es necesario, para que coincida con el tipo de la expresión switch.

En el ejemplo de apertura que prueba el valor de `letter`, éstas son las etiquetas de caso:

```
case 'X' :
case 'L' :
case 'M' :
case 'S' :
```

Como muestra ese ejemplo, una sola sentencia puede estar precedida por más de una etiqueta de caso. Cada valor de caso puede aparecer sólo una vez en una determinada sentencia Switch. Si un valor aparece más de una vez, resulta un error de sintaxis. También puede haber sólo una etiqueta por omisión en una sentencia Switch.

El flujo de control por una sentencia Switch procede así. Primero, se evalúa la expresión switch. Si uno de estos valores concuerda con uno de los valores de una etiqueta de caso, el control se bifurca a la sentencia después de esa etiqueta de caso. A partir de aquí, el control procede de modo secuencial hasta que se encuentra una sentencia Break o el final de la sentencia Switch. Si el valor de la expresión switch no coincide con ningún valor de caso, entonces sucede una de dos cosas. Si hay una etiqueta por omisión, el control se bifurca a la sentencia después de esa etiqueta. Si no hay una etiqueta por omisión, se omiten todas las sentencias dentro de Switch y el control simplemente procede a la sentencia que sigue a toda la sentencia Switch.

La siguiente sentencia Switch imprime un comentario apropiado con base en la calificación de un alumno (`grade` es de tipo `char`):

```
switch (grade)
{
    case 'A' :
    case 'B' : cout << "Buen trabajo";
               break;
    case 'C' : cout << "Trabajo promedio";
               break;
    case 'D' :
    case 'F' : cout << "Mal trabajo";
               numberInTrouble++;
               break;               // Innecesario, pero un buen hábito
}
```

Observe que la sentencia Break final es innecesaria. Sin embargo, los programadores la incluyen de cualquier manera. Una razón es que es más fácil insertar otra etiqueta de caso al final si ya está presente una sentencia Break.

Si `grade` no contiene uno de los caracteres especificados, no se ejecuta ninguna de las sentencias dentro de Switch. A no ser que una precondición de la sentencia Switch sea que `grade` es en definitiva una de 'A', 'B', 'C', 'D' o 'F', sería acertado incluir una etiqueta por omisión para una calificación no válida.

```
switch (grade)
{
    case 'A' :
    case 'B' : cout << "Buen trabajo";
               break;
    case 'C' : cout << "Trabajo promedio";
               break;
    case 'D' :
    case 'F' : cout << "Mal trabajo";
               numberInTrouble++;
               break;
    default  : cout << grade << " no es una calificación con letra válida";
               break;
}
```

Una sentencia Switch con una sentencia Break después de cada alternativa de caja se comporta exactamente como una estructura de control If-Then-Else-If. Por ejemplo, la sentencia Switch es equivalente al código siguiente:

```
if (grade == 'A' || grade == 'B')
    cout << "Buen trabajo";
else if (grade == 'C')
    cout << "Trabajo promedio";
else if (grade == 'D' || grade == 'F')
{
    cout << "Mal trabajo";
    numberInTrouble++;
}
else
    cout << grade << " no es una calificación con letra válida.";
```

¿Es cualquiera de estas dos versiones mejor que la otra? No hay respuesta absoluta a esta pregunta. Para este ejemplo particular, nuestra opinión es que la sentencia Switch es más fácil de entender debido a su forma bidimensional, como tabla. Pero algunos pueden encontrar más fácil de leer la versión If-Then-Else. Al poner en práctica una estructura de ramificación multivía, la recomendación es escribir una sentencia Switch y una If-Then-Else-If y luego compararlas en cuanto a legibilidad. Recuerde que C++ provee la sentencia Switch como un asunto de conveniencia. No se sienta obligado a usar una sentencia Switch para toda rama multivía.

Por último, se dijo que se examinaría lo que sucede si omite las sentencias Break dentro de una sentencia Switch. Se rescribirá el ejemplo de calificación con letra sin las sentencias Break:

```
switch (grade)    // Wrong version
{
    case 'A' :
    case 'B' : cout << "Buen trabajo";
    case 'C' : cout << "Trabajo promedio";
    case 'D' :
    case 'F' : cout << "Mal trabajo";
               numberInTrouble++;
    default  : cout << grade << " no es una calificación con letra válida.";
}
```

Si sucede que `grade` es 'H', el control se bifurca a la sentencia en la etiqueta por omisión y el resultado es

```
H no es una calificación con letra válida.
```

No obstante, esta opción de caso es la única que funciona correctamente. Si `grade` es 'A' el resultado es:

```
Good WorkAverage WokrPoor WorkA no es una calificación con letra válida.
```

Recuerde que después de que se toma una rama para una etiqueta de caso específica, el control procede de manera secuencial hasta que se encuentra una sentencia Break o el final de la sentencia Switch. Olvidar una sentencia Break en una opción de caso es una fuente muy común de errores en programas de C++.

Conozca a

Almirante Grace Murray Hopper

Desde 1943 hasta su muerte en el día de año nuevo de 1992, la almirante Grace Murray Hopper estuvo íntimamente relacionada con la computación. En 1991 recibió la medalla nacional de tecnología "por sus logros iniciales en el desarrollo de lenguajes de programación que simplificaron la tecnología de las computadoras y abrieron la puerta a un universo significativamente mayor de usuarios".

La almirante Hopper, cuyo nombre de soltera fue Grace Brewster Murray, nació en la ciudad de Nueva York el 9 de diciembre de 1906. Asistió a Vassar y recibió de Yale su doctorado en matemáticas. Durante los siguientes diez años dio clases de matemáticas en Vassar.

En 1943, la almirante Hopper ingresó a la Marina de los Estados Unidos y fue asignada al Bureau of Ordenance Computation Project en la universidad de Harvard como programadora en la Mark I. Después de la guerra, permaneció en Harvard como miembro de la facultad y continuó trabajando en las computadoras Mark II y Mark III de la Marina. En 1949 se unió a la Eckert-Mauchly Computer Corporation y trabajó en UNIVAC I. Fue allí donde hizo una contribución legendaria a la computación: descubrió el primer "error" de computadora, una polilla atrapada en el hardware.

La almirante Hopper tenía un compilador útil en 1952, en una época en que la idea general era que las computadoras sólo podían realizar operaciones aritméticas. Aunque no formó parte del comité que diseñó el lenguaje de computadora COBOL, tuvo una participación activa en su diseño, ejecución y uso. COBOL (que significa Common Business-Oriented Language) se desarrolló a principios de la década de 1960 y aún se usa ampliamente en el procesamiento de datos comerciales.

La almirante Hopper se retiró de la Marina en 1966, sólo para ser llamada de nuevo un año después al servicio activo de tiempo completo. Su misión era supervisar los esfuerzos de la Marina para mantener la uniformidad de los lenguajes de programación. Se dice que así como el almirante Hyman Rickover fue el padre de la Marina nuclear, la almirante Hopper fue la madre de la automatización de datos computarizados en la Marina. Prestó sus servicios al Naval Data Automation Command hasta que se retiró de nuevo en 1986 con el rango de contralmirante. Al momento de su muerte, era asesora jubilada en Digital Equipment Corporation.

Durante su vida, la almirante Hopper recibió grados honoríficos de más de 40 colegios y universidades. Fue homenajeada por sus compañeros en varias ocasiones y, entre otros premios, recibió el Computer Sciences Man of the Year otorgado por la Datta Processing Management Association, y el Contributions to Computer Science Education otorgado por el Special Interest Group for Computer Science Education de la ACM (Association for Computing Machinery).

La almirante Hopper tenía afinidad con la gente joven y le gustaba dar pláticas en colegios y universidades. Con frecuencia repartía alambres de colores, a los que llamaba nanosegundos porque tenían una longitud de casi un pie, la distancia que viaja la luz en un nanosegundo (millonésima de segundo). Su recomendación a los jóvenes era, "tú manejas las cosas, conduces a la gente. En el pasado exageramos en la dirección y olvidamos el liderazgo".

Cuando se le preguntaba de cuál de sus logros estaba más orgullosa, contestaba, "de todos los jóvenes que he capacitado durante años".

9.2 Sentencia Do-While

La sentencia Do-While es una estructura de control de iteración en la que se prueba la condición de ciclo al final (parte baja) del ciclo. Este formato garantiza que el cuerpo del ciclo se ejecute por lo menos una vez. La plantilla de sintaxis para Do-While es la siguiente:

Sentencia Do-While

```
do
    Sentencia
while ( Expresión );
```

Como es usual en C++, la sentencia consta de una sola sentencia o un bloque. Asimismo, observe que Do-While termina con punto y coma.

La sentencia

```
do
{
    Sentencia1;
    Sentencia2;
    .
    .
    .
    SentenciaN;
} while (Expresión);
```

significa "Ejecute las sentencias entre `do` y `while` siempre que la expresión aún tenga el valor `true` al final del ciclo".

Compárese un ciclo While y uno Do-While que hace la misma tarea: encuentran el primer punto en archivo de datos. Suponga que hay, por lo menos, un punto en el archivo.

Solución While

```
dataFile >> inputChar;
while (inputChar != '.')
    dataFile >> inputChar;
```

Solución Do-While

```
do
    dataFile >> inputChar;
while (inputChar != '.');
```

La solución While requiere una lectura principal para que `inputChar` tenga un valor antes de que se introduzca el ciclo. Esto no es necesario para la solución Do-While porque la sentencia de entrada dentro del ciclo se ejecuta antes de que se evalúe la condición.

Veamos otro ejemplo. Suponga que un programa necesita leer la edad de una persona de manera interactiva. El programa requiere que la edad sea positiva. Los siguientes ciclos aseguran que el valor de entrada es positivo antes de que proceda el programa.

Solución While

```
cout << "Introduzca su edad: ";
cin >> age;
```

```
while (age <= 0)
{
    cout << "Su edad debe ser positiva." << endl;
    cout << "Introduzca su edad: ";
    cin >> age;
}
```

Solución Do-While

```
do
{
    cout << "Introduzca su edad: ";
    cin >> age;
    if (age <= 0)
        cout << "Su edad debe ser positiva." << endl;
} while (age <= 0);
```

Observe que la solución Do-While no requiere que el indicador y los pasos de entrada aparezcan dos veces —una antes del ciclo y otra dentro de él—, sino que prueba el valor de entrada dos veces.

Es posible usar también Do-While para poner en práctica un ciclo controlado por conteo si se sabe por adelantado que el cuerpo del ciclo se debe ejecutar siempre por lo menos una vez. A continuación se proporcionan las dos versiones de un ciclo para sumar los enteros de 1 a `n`.

Solución While

```
sum = 0;
counter = 1;
while (counter <= n)
{
    sum = sum + counter;
    counter++;
}
```

Solución Do-While

```
sum = 0;
counter = 1;
do
{
    sum = sum + counter;
    counter++;
} while (counter <= n);
```

Si `n` es un número positivo, ambas versiones son equivalentes. Pero si `n` es 0 o negativa, los dos ciclos dan resultados distintos. En la versión While, el valor final de `sum` es 0 porque nunca se introduce el cuerpo del ciclo. En la versión Do-While, el valor final de `sum` es 1 porque el cuerpo se ejecuta una vez y *luego* se hace la prueba del ciclo.

Como la sentencia While prueba la condición antes de ejecutar el cuerpo del ciclo, se llama *ciclo de preprueba*. La sentencia Do-While hace lo opuesto y, por tanto, se conoce como un *ciclo de posprueba*. En la figura 9-1 se compara el flujo de control de los ciclos While y Do-While.

Después de examinar otras dos nuevas construcciones de iteración, se proporcionan algunas directrices para determinar cuándo usar cada tipo de ciclo.

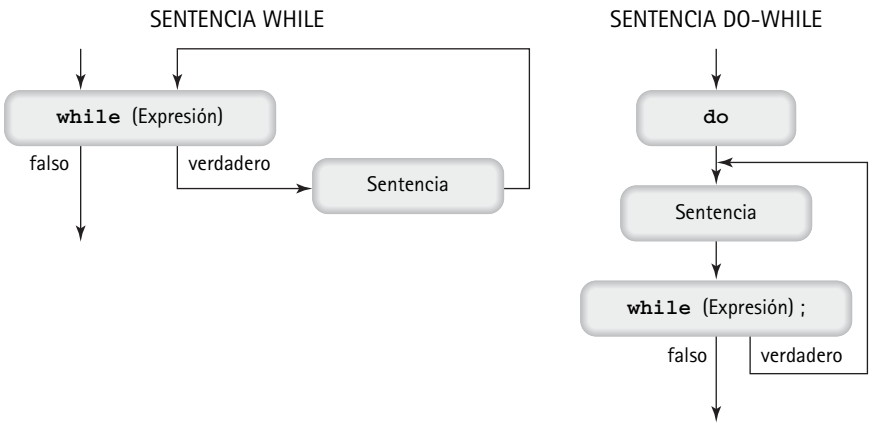


Figura 9-1 *Flujo de control: While y Do-While*

9.3 Sentencia For

La sentencia For está diseñada para simplificar la escritura de ciclos controlados por conteo. La siguiente sentencia imprime los enteros de 1 a `n`:

```
for (count = 1; count <= n; count++)
    cout << count << endl;
```

Esta sentencia For significa "Inicializar la variable de ciclo de control `count` en 1. Mientras `count` sea menor que o igual a `n`, ejecute la sentencia de salida e incrementa `count` en 1. Detenga el ciclo después de que `count` se ha incrementado a `n + 1`".

En C++, una sentencia For es sólo una notación compacta para un ciclo While. De hecho, el compilador básicamente traduce una sentencia For en un ciclo While equivalente de la siguiente manera:

```
for  ( count = 1 ; count <= n ; count++ )
     cout << count << endl;

count = 1;
while (count <= n)
{
   cout << count << endl;
   count++;
}
```

La plantilla de sintaxis para una sentencia For es

Sentencia For

```
for (Sentencia inicializadora Expresión1 ; Expresión2 )
    Sentencia
```

Expresión1 es la condición While. La sentencia inicializadora puede ser una de las siguientes: la sentencia nula (sólo un punto y coma), una sentencia de declaración (la cual siempre termina con un punto y coma). Por tanto, siempre hay un punto y coma antes de Expresión1. (Este punto y coma no se muestra en la plantilla de sintaxis porque la sentencia inicializadora siempre termina con su propio punto y coma.)

Con mucha frecuencia, una sentencia For se escribe de manera que la sentencia inicializadora comience una variable de control de ciclo y Expresión2 incremente o disminuya la variable de control de ciclo. Enseguida se muestran dos ciclos que se ejecutan el mismo número de veces (50):

```
for (loopCount = 1; loopCount <= 50; loopCount++)
    ⋮

for (loopCount = 50; loopCount >= 1; loopCount--)
    ⋮
```

Del mismo modo que los ciclos While, los ciclos Do-While y For pueden ser anidados. Por ejemplo, la estructura For anidada

```
for (lastNum = 1; lastNum <= 7; lastNum++)
{
    for (numToPrint = 1; numToPrint <= lastNum; numToPrint++)
        cout << numToPrint;
    cout << endl;
}
```

imprime el siguiente triángulo de números.

```
1
12
123
1234
12345
123456
1234567
```

Aunque las sentencias For se usan sobre todo para ciclos controlados por conteo, C++ permite escribir *cualquier* ciclo While usando una sentencia For. Para usar ciclos For inteligentemente, es necesario conocer los siguientes hechos:

1. En la plantilla de sintaxis, la sentencia inicializadora puede ser la sentencia nula, y Expresión2 es opcional. Si se omite Expresión2, no hay ninguna sentencia que el compilador pueda insertar en el final del ciclo. Como resultado, se podría escribir el ciclo While

   ```
   while (inputVal != 999)
       cin >> inputVal;
   ```

   como el ciclo For equivalente

   ```
   for ( ; inputVal != 999; )
       cin >> inputVal;
   ```

2. De acuerdo con la plantilla de sintaxis, Expresión1 —la condición While— es opcional. Si usted la omite, se asume la expresión `true`. El ciclo

   ```
   for ( ; ; )
       cout "Hi" << endl;
   ```

   es equivalente al ciclo While

   ```
   while (true)
       cout << "Hi" << endl;
   ```

   Ambos son ciclos infinitos que imprimen sin parar “Hi”.

3. La sentencia inicializadora puede ser una declaración con inicialización:

```
for (int i = 1; i <= 20; i++)
    cout << "Hi" << endl;
```

Aquí, la variable `i` tiene alcance local, aun cuando no haya llaves que establezcan un bloque. El alcance de `i` se extiende sólo hasta el final de la sentencia For. Como cualquier variable local, `i` es inaccesible fuera de este alcance (es decir, fuera de la sentencia For). Como `i` es local para la sentencia For, es posible escribir un código como el siguiente:

```
for (int i = 1; i <= 20; i++)
    cout << "Hi" << endl;
for (int i = 1; i <= 100; i++)
    cout << "Ed" << endl;
```

Este código *no* genera un error de tiempo de compilación (como "MULTIPLY DEFINED IDENTIFIER"). Se han declarado dos variables distintas denominadas `i`, cada una de las cuales es local para su propia sentencia For.*

Como ha visto, la sentencia For en C++ es una estructura muy flexible. Su uso puede variar de un simple ciclo controlado por conteo a un ciclo While "todo va" de propósito general. Algunos programadores comprimen mucho trabajo en el encabezado (la primera línea) de una sentencia For. Por ejemplo, el fragmento de programa

```
cin >> ch;
while (ch != '.')
    cin >> ch;
```

se puede comprimir en el siguiente ciclo For:

```
for (cin >> ch; ch != '.'; cin >> ch)
    ;
```

Como todo el trabajo se hace en el encabezado For, no hay nada que haga el ciclo. El cuerpo es sólo la sentencia nula.

Con las sentencias For, la recomendación es no complicar las cosas. Mientras más complicado sea el código, más se dificulta que otra persona (o usted) lo entienda y haga un seguimiento de los errores. En este libro sólo se usan ciclos For para ciclos controlados por conteo.

Enseguida se muestra un programa que contiene una sentencia For y una sentencia Switch. Éste analiza los primeros 100 caracteres leídos desde del dispositivo de entrada estándar e informa cuántos caracteres fueron letras, puntos, signos de interrogación y signos de admiración. Para la primera categoría (letras), se usa la función de biblioteca isalpha, una de las funciones "is..." que se describen en el capítulo 8. Para conservar espacio se ha omitido la documentación de interfaz para las funciones.

```
//*************************************************************************
// Programa Cuenta de caracteres
// Este programa cuenta el número de letras, puntos, signos
// de interrogación y signos de admiración encontrados en los primeros 100
// caracteres de entrada
// Suposición: la entrada consta de por lo menos 100 caracteres
//*************************************************************************
```

* En versiones de C++ previas al estándar de lenguaje ISO/ANSI, i no sería local para al cuerpo del ciclo. Su alcance se extendería hasta el final del bloque que rodea la sentencia For. En otras palabras, sería como si i se hubiera declarado fuera del ciclo. Si usted utiliza una versión antigua de C++ y su compilador le indica algo como "MULTIPLY DEFINED IDENTIFIER" en código similar al par de sentencias For anteriores, simplemente elija un nombre de variable distinto en el segundo ciclo For.

```
#include <iostream>
#include <cctype>      // Para isalpha()

using namespace std;

void IncrementCounter( char, int&, int&, int&, int& );
void PrintCounters( int, int, int, int );

int main()
{
    char inChar;            // Carácter de entrada actual
    int  loopCount;         // Variable de control de ciclo
    int  letterCount = 0;   // Número de letras
    int  periodCount = 0;   // Número de puntos
    int  questCount = 0;    // Número de signos de interrogación
    int  exclamCount = 0;   // Número de signos de admiración

    cout << "Enter your text:" << endl;
    for (loopCount = 1; loopCount <= 100; loopCount++)
    {
        cin.get(inChar);
        IncrementCounter(inChar, letterCount, periodCount,
                         questCount, exclamCount);
    }
    PrintCounters(letterCount, periodCount, questCount,
                  exclamCount);
    return 0;
}
//*****************************************************************************
void IncrementCounter( /* in */    char ch,
                       /* inout */ int& letterCount,
                       /* inout */ int& periodCount,
                       /* inout */ int& questCount,
                       /* inout */ int& exclamCount )
{
    if (isalpha(ch))
        letterCount++;
    else
        switch (ch)
        {
            case '.' : periodCount++;
                       break;
            case '?' : questCount++;
                       break;
            case '!' : exclamCount++;
                       break;
            default  : ;            // Innecesario, pero correcto
        }
}

//*****************************************************************************

void PrintCounters( /* in */ int letterCount,
                    /* in */ int periodCount,
```

```
                    /* in */ int questCount,
                    /* in */ int exclamCount )
{
    cout << endl;
    cout << "Entrada contenida" << endl
         << letterCount << " letras" << endl
         << periodCount << " puntos" << endl
         << questCount << " signos de interrogación" << endl
         << exclamCount << " signos de exclamación" << endl;
}
```

9.4 Sentencias Break y Continue

La sentencia Break, que se introdujo con la sentencia Switch, se emplea también con ciclos. Una sentencia Break produce una salida inmediata de la sentencia Switch, While, Do-While o For interna en la que aparece. Observe la palabra *interna*. Si `break` está en un ciclo que se anida dentro de otro ciclo, el control sale del ciclo interior pero no del exterior.

Una de las formas más comunes de usar `break` con ciclos es establecer un ciclo infinito, y usar pruebas If para salir del ciclo. Suponga que se desea introducir diez pares de enteros, efectuar la validación de datos y calcular la raíz cuadrada de la suma de cada par. Para la validación de datos, suponga que el primer número de cada par debe ser menor que 100 y el segundo debe ser mayor que 50. Sin embargo, se desea probar, después de cada entrada, el estado del flujo para el final de archivo. Aquí se muestra un ciclo que usa sentencias Break para realizar la tarea:

```
loopCount = 1;
while (true)
{
    cin >> num1;
    if ( !cin || num1 >= 100)
        break;
    cin >> num2;
    if ( !cin || num2 <= 50)
        break;
    cout << sqrt(float(num1 + num2)) << endl;
    loopCount++;
    if (loopCount > 10)
        break;
}
```

Observe que se podría haber usado un ciclo For para contar de 1 a 10, saliéndose de él según fuera necesario. Sin embargo, este ciclo se controla por conteo y por suceso, así que es preferible usar el ciclo While.

El ciclo anterior contiene tres puntos de salida distintos. Algunas personas se oponen terminantemente a este estilo de programación, ya que viola la idea de una sola entrada, una sola salida, analizada en relación con retornos múltiples desde una función. ¿Hay alguna ventaja en usar un ciclo infinito junto con `break`? Para contestar esta pregunta, se reescribirá el ciclo sin usar sentencias Break. El ciclo debe terminar cuando `num1` o `num2` no son válidos o `loopCount` pasa de 10. Se emplearán banderas booleanas para señalar datos inválidos en la condición While:

```
num1Valid = true;
num2Valid = true;
loopCount = 1;
```

```
while (num1Valid && num2Valid && loopCount <= 10)
{
    cin >> num1;
    if ( !cin || num1 >= 100)
        num1Valid = false;
    else
    {
        cin >> num2;
        if ( !cin || num2 <= 50)
            num2Valid = false;
        else
        {
            cout << sqrt(float(num1 + num2)) << endl;
            loopCount++;
        }
    }
}
```

Se podría argumentar que la primera versión es más fácil de seguir y entender que esta segunda. La tarea principal del cuerpo del ciclo, que calcula la raíz cuadrada de la suma de los números, es más importante en la primera versión. En la segunda, el cálculo se oscurece por estar enterrado dentro de instrucciones If anidadas. La segunda versión también tiene un flujo de control complicado.

La desventaja de usar break con los ciclos es que se puede convertir en un apoyo para quienes no acostumbran pensar con detenimiento acerca del diseño de un ciclo. Es fácil hacer uso excesivo de la técnica. A continuación se muestra un ejemplo que imprime los enteros del 1 al 5:

```
i = 1;
while (true)
{
    cout << i;
    if (i == 5)
        break;
    i++;
}
```

No hay justificación para establecer el ciclo de esta manera. Desde el punto de vista conceptual, es un ciclo puro controlado por conteo, y un simple ciclo For hace el trabajo:

```
for (i = 1; i <= 5; i++)
    cout << i;
```

El ciclo For es más fácil de entender y menos propenso a error.

Una buena norma general es usar `break` dentro de ciclos sólo como último recurso. En particular, utilícela únicamente para evitar combinaciones desconcertantes de múltiples banderas booleanas e instrucciones If anidadas.

Otra sentencia que altera el flujo de control en un programa de C++ es la sentencia Continue. Dicha sentencia, válida sólo en ciclos, termina la iteración actual del ciclo (pero no el ciclo completo). Ocasiona una bifurcación inmediata hacia el final del ciclo y omite el resto de las sentencias en el cuerpo del ciclo, en preparación para la siguiente iteración. Enseguida se proporciona un ejemplo de un ciclo de lectura en el que se desea procesar sólo números positivos en un archivo de entrada:

```
for (dataCount = 1; dataCount <= 500; dataCount++)
{
    dataFile >> inputVal;
    if (inputVal <= 0)
        continue;
    cout << inputVal;
     ⋮
}
```

Si `inputVal` es menor que 1 o igual a 0, el control se bifurca al final del ciclo. Entonces, como con cualquier ciclo, la computadora incrementa `dataCount` y efectúa la prueba del ciclo antes de pasar a la siguiente iteración.

La sentencia Continue no se emplea con frecuencia, pero se presenta para tener un panorama completo (y porque es posible encontrarla en programas de otras personas). Su propósito primordial es evitar oscurecer el proceso principal del ciclo al identificar el proceso dentro de una sentencia If. Por ejemplo, el código anterior podría escribirse con una sentencia Continue como sigue:

```
for (dataCount = 1; dataCount <= 500; dataCount++)
{
    dataFile >> inputVal;
    if (inputVal > 0)
    {
        cout << inputVal;
         :
    }
}
```

Asegúrese de ver la diferencia entre `continue` y `break`. La sentencia Continue significa "abandonar la iteración actual del ciclo y pasar a la siguiente iteración". La sentencia Break indica "salir de inmediato del ciclo completo".

9.5 Normas para elegir una sentencia de iteración

Las siguientes son normas para ayudarlo a decidir cuándo usar cada una de las tres sentencias de iteración (While, Do-While y For).

1. Si el ciclo se controla por conteo, la sentencia For es natural. Concentrar las tres acciones de control de ciclo, inicializar, probar e incrementar o disminuir un lugar (el encabezado de la sentencia For) reduce las posibilidades de olvidar incluir una de ellas.
2. Si el ciclo es controlado por suceso cuyo cuerpo se debe ejecutar por lo menos una vez, es apropiada una sentencia Do-While.
3. Si el ciclo se controla por suceso y nada se sabe acerca de la primera ejecución, use una sentencia While (o tal vez una For).
4. En caso de duda, use una sentencia While.
5. Un ciclo infinito con sentencias Break en ocasiones aclara el código pero con mucha frecuencia refleja un diseño de ciclo carente de disciplina. Utilícelo sólo después de la consideración adecuada de While, Do-While y For.

Caso práctico de resolución de problemas

El tío rico

PROBLEMA Su tío rico ha muerto, y en su escritorio usted encuentra sus dos testamentos. Uno de ellos, fechado hace varios meses, le deja a usted y sus parientes una parte sustancial de su fortuna; el otro, con fecha de la semana pasada, cede todo al vecino de su tío. Con la sospecha de que el segundo testamento es una falsificación, decide escribir un programa para analizar el estilo de escritura y comparar los testamentos. El programa lee y clasifica cada carácter. Una vez que se ha leído todo el archivo, imprime una tabla de resumen que muestra el porcentaje de letras mayúsculas, minúsculas, cifras decimales, espacios en blanco y marcas de puntuación de fin de enunciado en el archivo de datos. Los nombres de los archivos de entrada y salida se leen desde el teclado. El nombre del archivo de entrada se debe imprimir en la salida.

ENTRADA Texto en un archivo cuyo nombre se lee desde el teclado.

SALIDA Una tabla que da el nombre de cada categoría y qué porcentaje del total representa la categoría en el archivo cuyo nombre se lee desde el teclado.

ANÁLISIS Hacer esta tarea a mano sería tedioso pero muy directa. Se prepararían cinco lugares para hacer marcas, una para cada una de las categorías de símbolos que serán contados. Entonces usted leería el texto carácter por carácter, determinaría en qué categoría colocar cada uno y haría una marca en el lugar apropiado.

Es posible examinar un carácter y decir de inmediato qué categoría marcar. Puede simular "ver" con una sentencia If con ramas para las letras mayúsculas, minúsculas, dígitos, un espacio y marcas de puntuación de fin de enunciado. Una letra mayúscula se define como una entre 'A' y 'Z' inclusive; una minúscula se define como una entre 'a' y 'z' inclusive. *Pero espere:* antes de empezar a escribir algoritmos para reconocer dichas categorías, ¿no debe buscar en la biblioteca si ya existen? Sin duda, el archivo de encabezado `<cctype>` contiene funciones para reconocer letras mayúsculas, minúsculas y dígitos.

Hay una función que reconoce espacios en blanco. ¿Servirá para contar espacios? La función `isspace` devuelve verdadero si el carácter es un espacio, nueva línea, tabulador, retorno de carro o suministro de papel. El problema especifica una cuenta del número de espacios. Una nueva línea podría funcionar como el fin de una palabra como un espacio, pero el problema no pide el número de palabras; pregunta el número de espacios.

Hay una función para reconocer la puntuación, pero no la de fin de enunciado. El problema no expresa qué marcas de fin de enunciado están. ¿Qué clase de sentencias hay? Las sentencias regulares terminan en punto; las preguntas terminan en un signo de interrogación, y las exclamaciones que terminan con un signo de admiración.

SUPOSICIONES El archivo no está vacío.

Principal **Nivel 0**

```
Abrir archivos para procesarlos
Si los archivos no abren de manera correcta
    Escribir un mensaje de error
    devolver 1
Obtener un carácter
DO
    Incrementar el contador de caracteres (carácter)
    Obtener un carácter
WHILE (más datos)
Imprimir la tabla
Cerrar los archivos
```

Abrir archivos (entrada-salida: texto, tabla) **Nivel 1**

```
Solicitar el nombre de archivo de entrada
Leer el nombre de archivo
Abrir el archivo de entrada
Solicitar el nombre de archivo de salida
Leer el nombre de archivo
Abrir el archivo de salida
Imprimir el nombre del archivo de entrada en el archivo de salida
```

Incrementar contadores [entrada-salida: contador de mayúsculas (upperCaseCounter), contador de minúsculas (lowerCaseCounter), contador de espacios (blankCounter), contador de dígitos (digitCounter), contador de puntuación (punctuationCounter); entrada (In:character)]

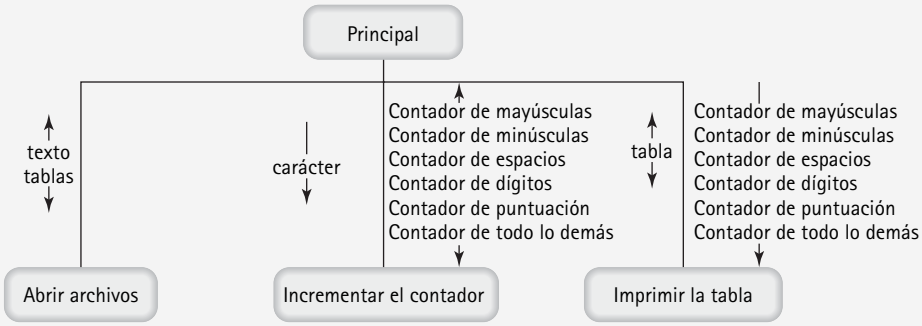
```
IF (isupper(character))
    Incrementar el contador de mayúsculas
ELSE IF (islower(character))
    Incrementar el contador de minúsculas
ELSE IF (carácter ==' ')
    Incrementar el contador de espacios
ELSE IF (isdigit(character))
    Incrementar el contador de dígitos
ELSE IF (character ==‘.’)|| (character == ‘?’) || (character == ‘!’))
    Incrementar el contador de puntuación
```

En este punto comprende que las instrucciones no indican si los porcentajes se tomarán del número total de caracteres leídos, incluso los que no corresponden a ninguna de las categorías, o del número total de caracteres que encajan en las cinco categorías. Decide suponer que todos los caracteres se deben contar, así que agrega una rama ELSE a este módulo que incrementa un contador (llamado contador de todo lo demás [allElseCounter]), para los caracteres que no caen en ninguna de las cinco categorías. Es necesario agregar la suposición al programa.

Calcular e imprimir porcentajes [entrada-salida: tablas, contador de mayúsculas (upperCaseCounter), contador de minúsculas (lowerCaseCounter), contador de espacios (blankCounter), contador de dígitos (digitCounter), contador de puntuación (punctuationCounter), contador de todo lo demás(allElseCounter)]

```
Fijar el Total para la suma de 6 contadores
Imprimir ‘Porcentaje de letras mayúsculas:’,
Contador de mayúsculas/Total * 100
Imprimir ‘Porcentaje de letras minúsculas:’,
Contador de minúsculas/Total * 100
Imprimir ‘Porcentaje de dígitos decimales:’,
Contador de dígitos/Total * 100
Imprimir ‘Porcentaje de espacios:’,
Contador de espacios/Total * 100
Imprimir ‘Porcentaje de puntuación de fin de enunciado:’,
Contador de puntuación/Total * 100
```

GRÁFICA DE ESTRUCTURA DE MÓDULO




```
//**************************************************************************
// Programa Tío rico
// Se imprime una tabla que muestra el porcentaje de caracteres en el
// archivo que pertenecen a cinco categorías: caracteres de mayúsculas,
// caracteres de minúsculas, dígitos decimales, espacios y marcas
// de puntuación de fin de enunciado
// Suposiciones: el archivo de entrada no está vacío y los porcentajes
// se basan en el número total de caracteres en el archivo
// Para ahorrar espacio, se omite de cada función los comentarios
// de precondición que documentan las suposiciones hechas acerca de los datos
// de parámetros de entrada válidos. Éstos se incluirían en un programa
// hecho para uso real
//**************************************************************************

#include <fstream>
#include <iostream>
#include <iomanip>
#include <cctype>

using namespace std;

// Prototipos de función
void OpenFiles(ifstream&, ofstream&);
void IncrementCounter(char, int&, int&, int&, int&, int&, int&);
void PrintTable(ofstream& table, int, int, int, int, int, int);

int main()
{
    // Preparar los archivos para lectura y escritura
    ifstream text;
    ofstream table;
    char character;

    // Declarar e inicializar contadores
    int uppercaseCounter = 0;        // Número de letras mayúsculas
    int lowercaseCounter = 0;        // Número de letras minúsculas
    int blankCounter = 0;            // Número de espacios
    int digitCounter = 0;            // Número de dígitos
    int punctuationCounter = 0;      // Número de caracteres de fin de
                                        enunciado '.', '?', '!'
    int allElseCounter = 0;          // Caracteres restantes
```

```
    OpenFiles(text, table);
    if ( !text || !table)
    {
        cout << "Los archivos no se abrieron de manera exitosa." << endl;
        return 1;
    }
    text.get(character);             // Introducir un carácter
    do
    { // Procesar cada carácter
            IncrementCounter(character, uppercaseCounter,
            lowercaseCounter, blankCounter, digitCounter,
            punctuationCounter, allElseCounter);
        text.get(character);
    } while (text);

    PrintTable(table, uppercaseCounter, lowercaseCounter,
        blankCounter, digitCounter, punctuationCounter,
        allElseCounter);

    text.close();
    table.close();
    return 0;
}

//*********************************************************************

void IncrementCounter(
     /* in */    char character),         // Carácter que está siendo
                                             examinado
     /* inout */ int& uppercaseCounter,   // Letras mayúsculas
     /* inout */ int& lowercaseCounter,   // Letras minúsculas
     /* inout */ int& blankCounter,       // Espacios
     /* inout */ int& digitCounter,       // Dígitos
     /* inout */ int& punctuationCounter, // '.', '?', '!'
     /* inout */ int& allElseCounter )    // Todo lo demás

// La función IncrementCounter examina el carácter e incrementa
// el contador apropiado
// Poscondición:
//     Se incrementó la categoría a la que pertenece
//     el carácter

{
    if (isupper(character))
        uppercaseCounter++;
    else if (islower(character))
        lowercaseCounter++;
    else if (character == ' ')
        blankCounter++;
    else if (isdigit(character))
        digitCounter++;
    else if ((character == '.') || (character == '?') ||
             (character == '!'))
```

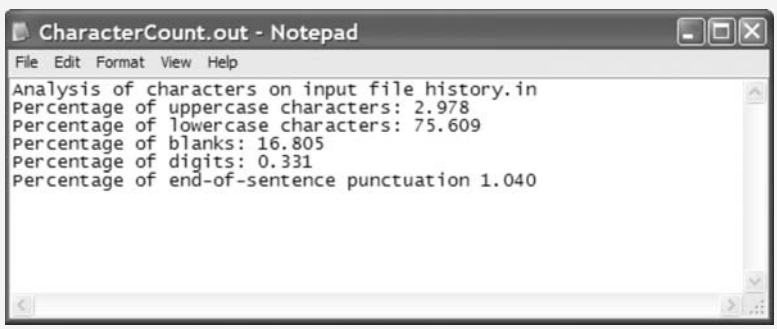
```
        punctuationCounter++;
    else
        allElseCounter++;
}

//*****************************************************************************

void PrintTable(
     /* inout */ ofstream& table,       // Archivo de salida
     /* in */ int uppercaseCounter,     // Letras mayúsculas
     /* in */ int lowercaseCounter,     // Letras minúsculas
     /* in */ int blankCounter,         // Espacios
     /* in */ int digitCounter,         // Dígitos
     /* in */ int punctuationCounter,   // '.', '?', '!'
     /* in */ int allElseCounter )      // Todo lo demás

// La función PrintTable imprime los porcentajes representados por cada
// una de las cinco categorías
// Poscondición:
//     El resultado ha sido escrito en la tabla de archivos, marcada
//     de modo apropiado

{
    // Calcular el número total de caracteres
    float total = uppercaseCounter + lowercaseCounter
        + blankCounter + digitCounter + punctuationCounter
        + allElseCounter;

    // Escribir el resultado en la tabla de flujo
    table << fixed << setprecision(3)
          << "Porcentaje de mayúsculas: "
          << uppercaseCounter / total * 100 << endl;
    table << fixed << setprecision(3)
          << "Porcentaje de minúsculas: "
          << lowercaseCounter / total * 100 << endl;
    table << fixed << setprecision(3) << "Porcentaje de espacios: "
          << blankCounter / total * 100 << endl;
    table << fixed << setprecision(3) << "Porcentaje de dígitos: "
          << digitCounter / total * 100 << endl;
    table << fixed << setprecision(3)
          << "Porcentaje de signos de fin de enunciado "
          << "Puntuación " << punctuationCounter / total * 100
          << endl;
}

//*****************************************************************************

void OpenFiles( /* inout */ ifstream& text,   // Archivo de entrada
                /* inout */ ofstream& table ) // Archivo de salida

// La función OpenFiles lee los nombres del archivo de entrada
// y el archivo de salida y los abre para procesarlos.
// Poscondición:
```

```
//     Los archivos han sido abiertos y el nombre del archivo de entrada
//     ha sido escrito en el archivo de salida
{
    string inFileName;      // Nombre de archivo de entrada especificado por
                               el usuario
    string outFileName;     // Nombre de archivo de salida especificado por el
                               usuario
    cout << "Introducir el nombre del archivo de entrada que será procesado"
    << endl;

    cin >> inFileName;
    text.open(inFileName.c_str());

    cout << "Introducir el nombre del archivo de salida" << endl;
    cin >> outFileName;
    table.open(outFileName.c_str());

    // Escribir la etiqueta en el archivo de salida
    table << "Análisis de caracteres en el archivo de salida " << inFileName
          << endl;
}
```

PRUEBA Para probarlo por completo, el programa Tío se debe ejecutar con todas las combinaciones posibles de las categorías de caracteres que se están contando. A continuación se enumera el conjunto mínimo de casos que se deben probar.

1. Están presentes todas las categorías de caracteres.
2. Están presentes cuatro de las categorías; una no está presente. (Ésta requerirá cinco corridas de prueba.)
3. Sólo están presentes caracteres que caen en una de las cinco categorías.
4. Están presentes otros caracteres.

El listado de salida que se muestra a continuación se ejecutó en un archivo de más de 4 000 caracteres. (No es un testamento, pero es suficiente para probar.)



La imagen mostrada corresponde a la salida producida por el programa original, escrito en idioma inglés.

Prueba y depuración

Las mismas técnicas de prueba empleadas con ciclos While se aplican a los ciclos Do-While y For. Sin embargo, hay algunas consideraciones adicionales con estos ciclos.

El cuerpo de un ciclo Do-While siempre se ejecuta por lo menos una vez. Así, se deben probar conjuntos de datos que muestran el resultado de ejecutar un ciclo Do-While el número mínimo de veces.

Con un ciclo For dependiente de datos, es importante probar resultados apropiados cuando el ciclo se ejecuta cero veces. Esto ocurre cuando el valor inicial es mayor que el valor final (o menor que el valor final si se reduce la variable de control de ciclo).

Cuando un programa contiene una sentencia Switch se debe probar con suficientes conjuntos de datos distintos para asegurar que cada rama se elige y ejecuta de manera correcta. Es necesario probar el programa con una expresión de cambio cuyo valor no sea ninguna de las etiquetas de caso.

Sugerencias de prueba y depuración

1. En una sentencia Switch, asegúrese de que hay una sentencia Break al final de cada opción de caso. De lo contrario, el control "cae" hasta el código en la siguiente opción de caso.
2. Las etiquetas de caso en una sentencia Switch están constituidas por valores, no variables. Sin embargo, pueden incluir constantes y expresiones nombradas relacionadas sólo con constantes.
3. Una expresión Switch no puede ser de punto flotante o cadena, y las constantes de caso no pueden ser de punto flotante o cadena.
4. Si hay una posibilidad de que el valor de la expresión switch no coincida con una de las constantes de caso, se debe proporcionar una opción por `omisión`. De hecho, es buena práctica incluir siempre una opción por `omisión`.
5. Realice una doble comprobación de las sentencias Switch largas para asegurarse de que no omitió ninguna de las ramas.
6. El ciclo Do-While es un ciclo posprueba. Si hay posibilidad de omitir todo el cuerpo del ciclo, use una sentencia While o una sentencia For.
7. El encabezado de la sentencia For (la primera línea) siempre tiene tres piezas dentro de los paréntesis. Con mucha frecuencia, la primera pieza inicializa una variable de control de ciclo; la segunda prueba la variable, y la tercera incrementa o disminuye la variable. Las tres piezas se deben separar por punto y coma. Es posible omitir cualquiera de las piezas, pero el punto y coma aún debe estar presente.
8. Con estructuras de control anidadas, la sentencia Break sólo puede salir de un nivel de anidación, el nivel más interno de Switch o ciclo en que se localiza `break`.

Resumen

La sentencia Switch es una sentencia de selección multivía. Permite que el programa elija entre un conjunto de ramas. Un Switch que contiene sentencia Break siempre se puede simular mediante una estructura If-Then-Else. Sin embargo, si es posible usar un switch, con frecuencia va a facilitar la lectura y comprensión del código. Una sentencia Switch no se puede usar con valores de punto flotante o cadena en etiquetas de caso.

Do-While es una sentencia de iteración de propósito general. Es como un ciclo While excepto que su prueba ocurre al final del ciclo, lo que garantiza por lo menos una ejecución del cuerpo del ciclo. Como con un ciclo While, uno Do-While continúa siempre y cuando la condición de ciclo sea `true`. Una sentencia Do-While es conveniente para ciclos que prueban valores de entrada y repiten si la entrada no es correcta.

La sentencia For es también una sentencia de iteración de propósito general, pero su uso más común es ejecutar ciclos controlados por conteo. La inicialización, prueba e incremento (o disminución) de la variable de control del ciclo se centra en un lugar, la primera línea de la sentencia For.

Las sentencias For, Do-While y Switch son el helado y el pastel de C++. Es posible vivir sin ellas si así se requiere, pero es muy bueno tenerlas.

Comprobación rápida

1. Si un problema requiere un ciclo puro controlado por conteo, ¿usaría una sentencia While, Do-While o For para poner en práctica el ciclo? (p. 356)
2. Cuando se ejecuta dentro de un ciclo, ¿a dónde transfiere el control de la sentencia Break? ¿Dónde prosigue el control de una sentencia Continue? (pp. 354-356)
3. Al convertir una estructura de ramificación multivía If-Else-If a una sentencia Switch, ¿qué parte del cambio se usa para poner en práctica la rama Else final de If-Else-If? (pp. 344-347)
4. ¿Qué sentencia de iteración ejecuta siempre su cuerpo por lo menos una vez? (pp. 348-350).
5. Escriba una sentencia For que cuente de -10 a 10. (pp. 350-354)

Respuestas

1. Una sentencia For. **2.** La sentencia Break sale de inmediato del ciclo. La sentencia Continue envía el control al final del ciclo. **3.** La rama `default`. **4.** Do-While. **5.** `for (int count = -10; count <= 10; count ++)`

Ejercicios de preparación para examen

1. Una expresión switch puede ser de tipo `bool`, `char`, `int` o `long`, pero no de tipo `float`. ¿Verdadero o falso?
2. Una variable declarada en la sentencia de inicialización de un ciclo For tiene alcance global. ¿Verdadero o falso?
3. Cualquier ciclo While se puede reescribir directamente como uno Do-While al cambiar sólo la sintaxis de la sentencia y mover la condición de salida al final del ciclo. ¿Verdadero o falso?
4. En un ciclo For no se permite una sentencia Break, pero sí una sentencia Continue. ¿Verdadero o falso?
5. ¿Cuáles de las sentencias de iteración en C++ son ciclos de preprueba y cuáles son ciclos posprueba?
6. ¿Qué sucede cuando olvida incluir las sentencias Break en una sentencia Switch?
7. Si omite todas las cláusulas dentro de un ciclo For (`for ( ; ; )`), ¿cuál sería la condición en un ciclo While equivalente?
8. ¿Cuántas veces se ejecuta el ciclo interno en el siguiente ciclo anidado?

```
for (int x = 1; x <= 10; x++)
  for (int y = 1; y <= 10; y++)
    for (int z = 1; z <= 10; z++)
      cout << x + y + z;
```

9. ¿Cuál sentencia de iteración elegiría para un problema en que la decisión de repetir un proceso depende de un suceso, y el proceso no puede ocurrir hasta que se ejecute por lo menos una vez?
10. ¿Qué sentencia de iteración elegiría para un problema en que la decisión de repetir el proceso depende de un contador de iteración y del estado de un archivo de entrada, y el proceso se puede omitir si el archivo está vacío?
11. ¿Qué se obtiene mediante el siguiente segmento de código si wood contiene '0'?

```
switch (wood)
{
  case 'P' : cout << "Pino";
  case 'F' : cout << "Abeto";
  case 'C' : cout << "Cedro";
  case 'O' : cout << "Roble";
```

```
  case 'M' : cout << "Arce";
  default  : cout << "Error";
}
```

12. ¿Qué se obtiene mediante el siguiente segmento de código si month contiene 8?

```
switch (month)

{
  case 1 : cout << "Enero"; break;
  case 2 : cout << "Febrero"; break;
  case 3 : cout << "Marzo"; break;
  case 4 : cout << "Abril"; break;
  case 5 : cout << "Mayo"; break;
  case 6 : cout << "Junio"; break;
  case 7 : cout << "Julio"; break;
  case 8 : cout << "Agosto"; break;
  case 9 : cout << "Septiembre"; break;
  case 10 : cout << "Octubre"; break;
  case 11 : cout << "Noviembre"; break;
  case 12 : cout << "Diciembre"; break;
  default : cout << "Error";
}
```

13. ¿Cuál es el resultado del siguiente segmento de código?

```
outCount = -1;
do
{
  inCount = 3;
  do
  {
    cout << outCount + inCount << endl;
    inCount --
  } while (inCount > 0);
  outCount++;
} while (outCount < 2);
```

14. ¿Cuál es el resultado del siguiente segmento de código?

```
for (int outCount = -1; outCount < 2; outCount++)
  for (int inCount = 3; inCount > 0; inCount--)
    cout << outCount + inCount << endl;
```

15. Reescriba el segmento de código del ejercicio 14 con ciclos While.

16. ¿Qué se imprime mediante el siguiente segmento de código?

```
for (int count = 1; count <= 4; count++)
  switch (count)
  {
    case 4 : cout << " cow?"; break;
    case 2 : cout << " now"; break;
    case 1 : cout << "How"; break;
    case 3 : cout << " brown"; break;
  }
```

17. Escriba una sola sentencia simple que tenga el mismo efecto en `cout` que el segmento de código del ejercicio 16.
18. ¿Qué produce el siguiente segmento de código (cuyo estilo de escritura es deficiente)?

```
count = 1;
for ( ; ; count++)
  if (count < 3)
    cout << count;
  else
    break;
```

Ejercicios de calentamiento para programación

1. Escriba una sentencia Switch que produzca el día de la semana para cout de acuerdo con el valor `int` en day que va de 0 a 6, donde 0 corresponde a domingo.
2. Amplíe la sentencia Switch del ejercicio 1 de modo que produzca `"Error"` si el valor en day no está en el intervalo de 0 a 6.
3. Reescriba la sentencia Switch del ejercicio 2 de modo que esté dentro de una función de devolución de valor que devuelva la cadena con el día de la semana, dado day como su parámetro.
4. Reescriba un ciclo For que use la función escrita en el ejercicio 3 para producir los días de la semana, cada uno en una línea separada. Comience con el día sábado y trabaje hacia atrás hasta domingo.
5. Cambie el ciclo For del ejercicio 4 de modo que produzca los días de la semana en orden directo. Comience con el día miércoles y pase por jueves.
6. Escriba un ciclo Do-While que solicite e introduzca una entrada del usuario de "Y" o "N". Si el usuario no introduce un valor correcto, el ciclo produce un mensaje de error y luego repite la petición de que se introduzca el valor.
7. Escriba una función de devolución de valor que tome un parámetro entero y devuelva un resultado entero. La función suma los enteros positivos, empezando en uno, hasta que la suma es igual o mayor que el valor del parámetro, y después devuelve el último entero agregado a la suma. Use un ciclo Do-While para hacer la suma.
8. Escriba un ciclo For anidado que imprima una tabla de multiplicación para los enteros de 1 a 10.
9. Escriba un ciclo For anidado que imprima un triángulo recto lleno de estrellas, una estrella en la primera línea, dos en la siguiente, etcétera, hasta la décima línea con diez estrellas.
10. Reescriba el ciclo anidado del ejercicio 9 con ciclos Do-While.
11. Reescriba el ciclo anidado del ejercicio 9 con ciclos While.
12. Escriba una función void, con ciclos For, que imprima un rectángulo de estrellas hueco cuyo ancho y altura se especifique mediante dos parámetros `int`. La parte superior y el fondo del rectángulo es una línea continua de estrellas, cada renglón entre la parte superior y el fondo consta de una estrella, luego espacios de *ancho-2* y otra estrella.
13. Escriba un ciclo For anidado que produzca las horas y minutos en el periodo de 3:15 a 7:30.
14. Amplíe el ciclo del ejercicio 13 para que también produzca segundos.
15. ¿Cuántas líneas producen los ciclos de los ejercicios 13 y 14?

Problemas de programación

1. En el problema de programación 6 del capítulo 7 se le pidió escribir un programa que introduce una cadena y produce las palabras correspondientes de la International Civil Aviation Organization que se usarían para deletrearla fonéticamente. Por ejemplo:

```
Cadena de entrada: program
La versión fonética es: Papa Romeo Oscar Golf Romeo Alpha Mike
```

Reescriba la función de este programa que devuelve la palabra que corresponde a una letra especificada. Utilice una sentencia Switch en lugar de una estructura If. Para facilidad de referencia, el alfabeto ICAO del capítulo 7 se repite aquí:

A	Alpha
B	Bravo
C	Charlie
D	Delta
E	Echo
F	Foxtrot
G	Golf
H	Hotel
I	India
J	Juliet
K	Kilo
M	Mike
N	November
O	Oscar
P	Papa
Q	Quebec
R	Romeo
S	Sierra
T	Tango
U	Uniform
V	Victor
W	Whiskey
X	X-ray
Y	Yankee
Z	Zulu

Asegúrese de usar el formato y comentarios apropiados en su código. Proporcione los mensajes apropiados al usuario. El resultado debe ser claro y tener un formato nítido.

2. En el problema 2 de programación del capítulo 4 se le pidió escribir un programa en C++ que solicita al usuario introducir su peso y el nombre de un planeta. El programa tenía como fin calcular el peso del usuario en ese planeta. Reescriba el programa de modo que la selección del factor a usar en el cálculo del peso se haga con una sentencia Switch en lugar de una estructura If. El cálculo se debe hacer en una función de devolución de valor y se debe llamar desde `main`.

Para facilidad de referencia, la información para el problema original se repite aquí. La siguiente tabla proporciona el factor mediante el cual se debe multiplicar el peso para cada planeta. El programa debe producir un mensaje de error si el usuario no escribe correctamente un nombre de planeta. El indicador y el mensaje de error deben aclarar al usuario cómo se debe introducir un nombre de planeta. Asegúrese de usar el formato y comentarios apropiados en su código. El resultado debe ser claro y tener un formato nítido.

Mercurio	0.4155
Venus	0.8975
Tierra	1.0
Luna	0.166
Marte	0.3507
Júpiter	2.5374
Saturno	1.0677
Urano	0.8947
Neptuno	1.1794
Plutón	0.0899

3. Usted trabaja para una compañía que tiene vendedores de viajes. El vendedor reclama el pago de sus ventas a un mostrador de desempeño, donde las ventas se introducen en un archivo. Cada venta se registra como una línea en el archivo `sales.dat` como un número de identificación del vendedor, un número de artículo y una cantidad, todos separados por espacios. Hay 10 vendedores, con números de identificación del 1 al 10. La compañía vende ocho productos distintos, con números de identificación del 7 al 14 (algunos productos antiguos se han discontinuado). Los precios de los productos son

Número de producto	Precio unitario
7	345.00
8	853.00
9	471.00
10	933.00
11	721.00
12	663.00
13	507.00
14	259.00

   Se le pide escribir un programa que lea en el archivo de ventas y genere un archivo separado para cada vendedor que contenga sólo sus ventas. Cada línea del archivo de ventas se copia al archivo de vendedor apropiado (`salespers1.dat` a `salespers10.dat`), sin el número de identificación del vendedor. El total para la venta (cantidad multiplicada por el precio unitario) se adjunta al registro. Al final del proceso, se debe producir el total de ventas para cada vendedor con etiquetas informativas para `cout`. Use la descomposición funcional para diseñar el programa. Asegúrese de que el programa maneja números de identificación inválidos. Si el número de identificación de un vendedor es no válido, escriba un mensaje de error para cout. Si un número de producto no es válido, escriba el mensaje de error para el archivo del vendedor y no calcule un total para esa venta. Debe haber amplia oportunidad de usar la sentencia Switch y funciones de devolución de valor en esta aplicación.
4. Elabore un programa que desarrolle un juego de adivinar un número en el que la computadora selecciona un número aleatorio en el intervalo de 0 a 100 y el usuario tiene hasta 20 intentos para adivinar. Al final de cada juego, se debe decir al usuario si gana o pierde y luego preguntarle si quiere jugar otra vez. Cuanto el usuario termina, el programa debe producir el número total de juegos ganados y perdidos. Para hacer más interesante el juego, el programa debe variar la redacción de los mensajes que produce para juegos ganados, perdidos y para pedir otro juego. Elabore 10 mensajes distintos para cada uno de estos casos y use números aleatorios para elegir entre ellos. Véa en el apéndice C.7 la información acerca de las funciones generadoras de números aleatorios. Esta aplicación debe proporcionar una buena oportunidad para que usted use una sentencia Do-While y sentencias Switch. Use la descomposición funcional para resolver el problema; escriba su código de C++ con buen estilo y comentarios ilustrativos y diviértase pensando en algunos mensajes que sorprenderán al usuario.
5. Escriba la descomposición funcional y un programa de C++ que lea un tiempo en forma numérica y lo imprima en español. El tiempo se introduce como horas y minutos, separados por un espacio. Las horas se especifican en tiempo de 24 h (15 son las 3 p.m.), pero el resultado debe ser en la forma de 12 horas a.m./p.m. Note que el mediodía y la medianoche son casos especiales. Enseguida se muestran algunos ejemplos:

   Introducir el tiempo: 12 00
   Mediodía

   Introducir el tiempo: 0 00
   Medianoche

Introducir el tiempo: 6 44
Seis cuarenta y cuatro a.m.

Introducir el tiempo: 18 11
Seis once p.m.

Escriba su código de C++ con buen estilo y comentarios ilustrativos. Esta aplicación debe darle oportunidad de usar sentencias Switch.

6. Amplíe el programa del problema 5 de modo que pregunte al usuario si quiere introducir otro tiempo, y después repita el proceso hasta que la respuesta a la pregunta sea "no". Usted debe poder codificar esto fácilmente con la sentencia Do-While.

Seguimiento de caso práctico

1. ¿Qué cambios serían necesarios en el programa Tío rico si las letras mayúsculas y minúsculas se contaran como números de la misma categoría?
2. Usted decide que está interesado en contar el número de palabras en el texto. ¿Cómo podría calcular este número?
3. Dado que puede calcular el número de palabras en el texto, ¿cómo calcularía la longitud de palabra promedio?
4. ¿Puede pensar en cualquier otra característica del estilo de escritura de una persona que pueda medir?

CAPÍTULO

10

Tipos de datos simples: integrados y definidos por el usuario

Objetivos

Objetivos de conocimiento

- *Conocer todos los tipos de datos simples que proporciona el lenguaje C++.*
- *Estar familiarizado con operadores y expresiones de C++ especializados.*
- *Entender la diferencia entre representaciones externas e internas de datos de caracteres.*
- *Comprender cómo se representan los números de punto flotante en la computadora.*
- *Entender cómo la precisión numérica limitada de la computadora puede afectar los cálculos.*
- *Conocer los conceptos de promoción y degradación de tipos.*

Objetivos de habilidades

Ser capaz de:

- *Seleccionar el tipo de datos simples más apropiado para una variable dada.*
- *Declarar y usar un tipo de enumeración.*
- *Usar sentencias For y Switch con tipos de enumeración definidos por el usuario.*
- *Crear un archivo de encabezado escrito por el usuario.*

Este capítulo representa un punto de transición en su estudio de la informática y programación en C++. Hasta el momento se ha hecho hincapié en las variables simples, estructuras de control y procesos nombrados (funciones). Después de este capítulo, la atención se centra en las formas para estructurar (organizar) datos y en los algoritmos necesarios para procesar datos en estas formas estructuradas. Para lograr esta transición, es necesario examinar con mayor detalle el concepto de tipos de datos.

Hasta ahora, se ha trabajado sobre todo con tipos de datos `int`, `char`, `bool` y `float`. Estos cuatro tipos de datos son adecuados para resolver una amplia variedad de problemas. Pero algunos programas necesitan otros tipos de datos. En este capítulo se examinan más de cerca los tipos de datos simples integrantes del lenguaje C++. Como parte de este libro, se analizan las limitaciones de la computadora para hacer cálculos. Se examina cómo dichas limitaciones causan errores numéricos y cómo evitarlos.

En ocasiones incluso los tipos de datos integrados no pueden representar de manera adecuada todos los datos de un programa. C++ tiene varios mecanismos para crear tipos de datos *definidos por el usuario*; es decir, nosotros mismos podemos diseñar nuevos tipos de datos. En este capítulo se presenta uno de dichos mecanismos: el tipo de enumeración. En capítulos posteriores se presentan más tipos de datos definidos por el usuario.

10.1 Tipos simples integrados

En el capítulo 2 se definió un tipo de datos como un conjunto específico de valores de datos (al cual se le nombra dominio) junto con un conjunto de operaciones en esos valores. Para el tipo `int`, el dominio es el conjunto de números enteros de `INT_MIN` a `INT_MAX`, y las operaciones permisibles que se han visto hasta el momento son +, –, *, /, %, ++, –– y las operaciones relacionales y lógicas. El dominio del tipo `float` es el conjunto de los números reales que una determinada computadora es capaz de representar, y las operaciones son las mismas que las del tipo `int`, excepto que se excluye el módulo (%). Para el tipo `bool`, el dominio es el conjunto de los dos valores `true` y `false`, y las operaciones permisibles son las lógicas (`!, &&, ||`) y las relacionales. El tipo `char`, aunque se usa principalmente para manipular datos de caracteres, se clasifica como un tipo integral porque utiliza enteros en la memoria para representar caracteres. Después, en este capítulo, se ve cómo funciona esto.

Los tipos `int`, `char`, `bool` y `float` tienen una propiedad en común. El dominio de cada tipo está constituido por valores de datos indivisibles o atómicos. Los tipos de datos con esta propiedad se denominan **tipos de datos simples** (o **atómicos**). Cuando se dice que un valor es atómico, se quiere decir que no se puede tener acceso a cada uno de los elementos componentes. Por ejemplo, un solo carácter de tipo `char` es atómico, pero no la cadena "Good Morning" (que se compone de 12 caracteres individuales).

Tipo de datos simples (atómicos) Tipo de datos en que cada valor es atómico (indivisible).

Otra forma de describir un tipo simple es decir que sólo se puede relacionar un valor con una variable de ese tipo. En contraste, un *tipo estructurado* es en el que una colección completa de valores se relaciona con una sola variable de ese tipo. Por ejemplo, un objeto `string` representa una colección de caracteres a los que se les asigna un solo nombre. Al comienzo del capítulo 11 se examinan tipos estructurados.

En la figura 10.1 se muestran los tipos simples que forman parte del lenguaje C++. Esta figura es parte del diagrama completo de tipos de datos de C++ que se presentó en la figura 3-1.

En esta figura, uno de los tipos, `enum`, no es un tipo de datos simples en el sentido en que lo son los tipos de datos `int` y `float`. En cambio, es un mecanismo con el que es posible definir los propios tipos de datos simples. Más adelante, en este capítulo, se examina `enum`.

Los tipos integrales `char`, `short`, `int` y `long` sólo representan enteros de diferentes tamaños. De manera similar, los tipos de punto flotante `float`, `double` y `long double`, simplemente se refieren a números de punto flotante de diferentes tamaños. ¿Qué significa *tamaños*?

En C++, los tamaños se miden en múltiplos del tamaño de un `char`. Por definición, el tamaño de un `char` es 1. En la mayoría de las computadoras, pero no en todas, el 1 significa un byte. (Recuerde, del capítulo 1, que un byte es un grupo de ocho dígitos consecutivos [unos o ceros].)

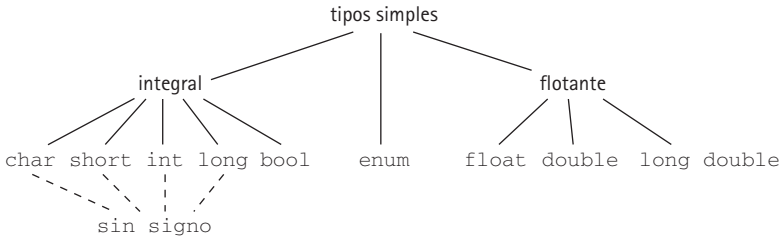


Figura 10.1 *Tipos simples C++*

Se usará la notación *sizeof*(SomeType) para denotar el tamaño de un valor de tipo SomeType. Así, por definición, *sizeof*(`char`) = 1. Aparte de `char`, los tamaños de objetos de datos en C++ son dependientes de la máquina. En determinada máquina, podría ser el caso que

sizeof (`char`) = 1
sizeof (`short`) = 2
sizeof (`int`) = 4
sizeof (`long`) = 8

En otra máquina, los tamaños podrían ser de la siguiente manera:

sizeof (`char`) = 1
sizeof (`short`) = 2
sizeof (`int`) = 2
sizeof (`long`) = 4

A pesar de estas variaciones, el lenguaje C++ garantiza que las siguientes sentencias son verdaderas:

- 1 = *sizeof* (`char`) ≤ *sizeof* (`short`) ≤ *sizeof* (`int`) ≤ *sizeof* (`long`).
- 1 ≤ *sizeof* (`bool`) ≤ *sizeof* (`long`).
- *sizeof* (`float`) ≤ *sizeof* (`double`) ≤ *sizeof* (`long double`).
- A `char` is at least 8 bits.
- A `short` is at least 16 bits.
- A `long` is at least 32 bits.

Intervalo de valores Intervalo en que deben caer los valores de un tipo numérico, especificados en términos de los valores permisibles más grandes y más pequeños.

Para datos numéricos, el tamaño de un objeto de datos determina su **intervalo de valores**. Considérense con más detalle los tamaños, intervalos de valores y constantes literales para cada uno de los tipos integrados.

Tipos integrales

Antes de examinar cómo los tamaños de tipos integrales afectan a sus posibles valores, se recuerda que la palabra reservada unsigned puede preceder el nombre de algunos tipos integrales, `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`. Los valores de dichos tipos son enteros no negativos con valores de 0 a algún valor máximo que depende de la máquina. Aunque rara vez se emplean tipos sin signo en este libro, se incluyen en esta explicación para minuciosidad.

Intervalos de valores En la tabla siguiente se presentan, a modo de ejemplo, intervalos de valores para los tipos de datos `char`, `short`, `int` y `long`, y sus variaciones sin signo (`unsigned`).

Tipo	Tamaño en bytes*	Valor mínimo*	Valor máximo*
`char`	1	–128	127
`unsigned char`	1	0	255
`short`	2	–32,768	32,767
`unsigned short`	2	0	65,535
`int`	2	–32,768	32,767
`unsigned int`	2	0	65,535
`long`	4	–2,147,483,648	2,147,483,647
`unsigned long`	4	0	4,294,967,295

* Estos valores son para una determinada máquina. Sus valores de máquina pueden ser diferentes.

Los sistemas de C++ proporcionan el archivo de encabezado `climits`, del cual es posible determinar los valores máximo y mínimo para su máquina. Dicho archivo de encabezado define las constantes `CHAR_MAX` y `CHAR_MIN`, `SHRT_MAX` y `SHRT_MIN`, `INT_MAX` e `INT_MIN`. y `LONG_MAX` y `LONG_MIN`. Los tipos sin signo tienen un valor mínimo de 0 y valores máximos definidos por `UCHAR_MAX`, `USHRT_MAX` y `ULONG_MAX`. Para hallar los valores específicos para su computadora podría imprimirlos de la manera en que se muestra a continuación:

```
#include <climits>
using namespace std;
    ⋮
cout << "Largo máximo = " << LONG_MAX << endl;
cout << "Largo mínimo = " << LONG_MIN << endl;
    ⋮
```

Constantes literales En C++, las constantes `bool` válidas son `true` y `false`. Las constantes enteras se pueden identificar en tres bases de números distintas: decimal (base 10), octal (base 8) y hexadecimal (base 16). Así como el sistema numérico decimal tiene 10 dígitos, 0 al 9, el sistema octal tiene ocho dígitos, 0 al 7. El sistema hexadecimal tiene los dígitos 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E y F, que corresponden a los valores decimales 0 al 15. Los valores octal y hexadecimal se usan en el software del sistema (compiladores, enlazadores y sistemas operativos, por ejemplo) para hacer referencia directa a bits individuales en una celda de memoria y controla dispositivos de hardware. Dichas manipulaciones de objetos de nivel bajo en una computadora son tema de estudio más avanzado y están fuera del alcance de este libro.

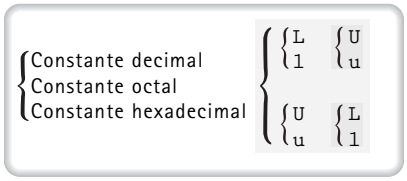
En la tabla siguiente se muestran ejemplos de constantes enteras en C++. Observe que una `L` o bien una `U` (ya sean mayúsculas o minúsculas) se pueden añadir al final de una constante para indicar `long` o `unsigned`, respectivamente.

Constante	Tipo	Observaciones
`1658`	`int`	Entero decimal (base 10).
`03172`	`int`	Entero octal (base 8). Comienza con 0 (cero). El decimal equivalente es 1658.
`0x67A`	`int`	Entero hexadecimal (base 16). Comienza con 0 (cero), después `x` o `X`. El decimal equivalente es 1658.
`65535U`	`unsigned int`	Las constantes sin signo terminan en `U` o `u`.
`421L`	`long`	Constante `long` explícita. Termina en `L` o `l`.
`53100`	`long`	Constante `long` implícita, suponiendo que el `int` máximo de la máquina sea, por ejemplo, `32767`.
`389123487UL`	`unsigned long`	Las constantes `long` sin signo terminan en `UL` o `LU` en ninguna combinación de letras mayúsculas y minúsculas.

Observe que esta tabla presenta sólo constantes numéricas para tipos integrales. Más adelante, en una sección aparte, se analizan constantes `char`.

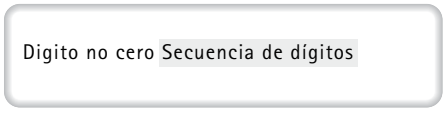
Enseguida se muestra la plantilla de sintaxis para una constante entera:

Constante entera



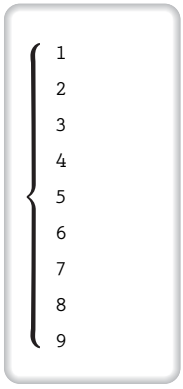
La constante decimal es un dígito no cero seguido, de manera opcional, por una secuencia de dígitos decimales:

Constante decimal



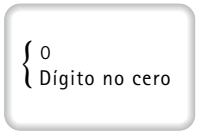
El dígito no cero, la secuencia de dígitos y el dígito, se definen como sigue:

Dígito no cero



Secuencia de dígitos

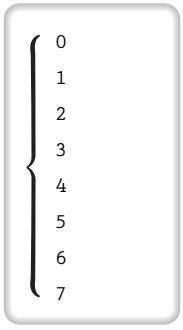
Dígito



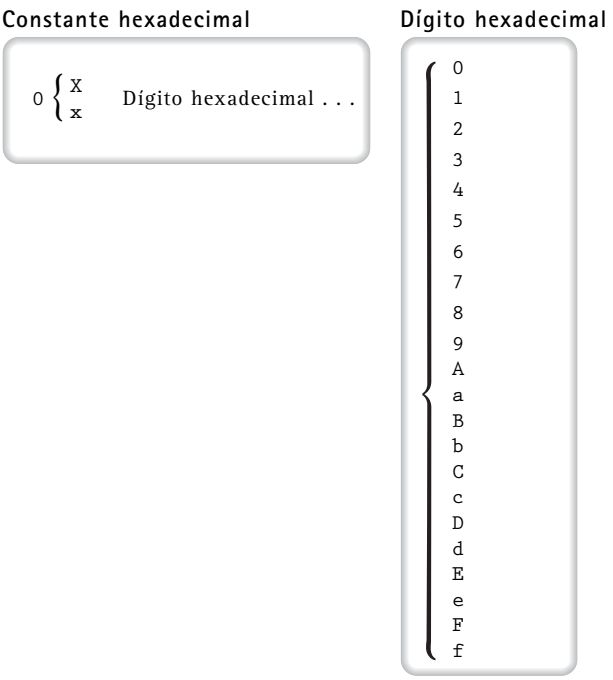
La segunda forma de la constante entera, constante octal, tiene la siguiente sintaxis:

Constante octal

Dígito octal



Por último, la constante hexadecimal se define como



Tipos de punto flotante

Intervalos de valores Enseguida se muestra una tabla que proporciona información, a modo de ejemplo, de valores para los tres tipos de punto flotante (`float`, `double` y `long double`). En dicha tabla se presenta, para cada tipo, el valor positivo máximo y el valor positivo mínimo (una pequeña fracción que es muy cercana a cero). Los números negativos tienen el mismo intervalo, pero el signo opuesto. Los intervalos de valores se expresan en notación exponencial (científica), donde 3.4E+38 significa 3.4×10^{38}.

Tipo	Tamaño en bytes*	Valor positivo mínimo*	Valor positivo máximo*
float	4	3.4E–38	3.4E+38
double	8	1.7E–308	1.7E+308
long double	10	3.4E–4932	1.1E+4932

* Estos valores son para una máquina específica. Los valores de su máquina pueden ser diferentes.

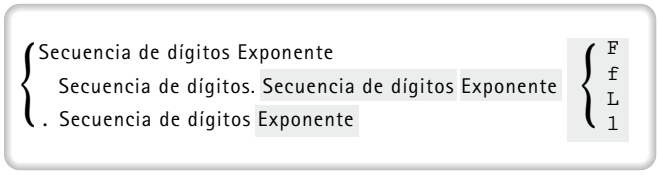
El archivo de encabezado estándar `cfloat` define las constantes `FLT_MAX` y `FLT_MIN`, `DBL_MAX` y `DBL_MIN`, y `LDBL_MAN` y `LDBL_MIN`. Para determinar los intervalos de valores de su máquina, usted podría escribir un programa corto que imprima dichas constantes.

Constantes literales Cuando usa una constante de punto flotante como 5.8 en un programa de C++, se supone que su tipo es `double` (precisión doble). Si almacena el valor en una variable `float`, la computadora coerciona su tipo de `double` a `float` (precisión simple). Si insiste en que una constante sea de tipo `float` en vez de `double`, puede anexar una `F` o una `f` al final de la constante. De modo similar, un sufijo `L` o `l` significa una constante `long double`. Enseguida se muestran algunos ejemplos de constantes de punto flotante en C++.

Constante	Tipo	Observaciones
`6.83`	`double`	Por omisión, las constantes de punto son de tipo `double`.
`6.83F`	`float`	Las constantes `float` explícitas terminan en `F` o `f`.
`6.83L`	`long double`	Las constantes `long double` explícitas terminan en `L` o `l`.
`4.35E-9`	`double`	Notación exponencial, que significa 4.35×10^{-9}.

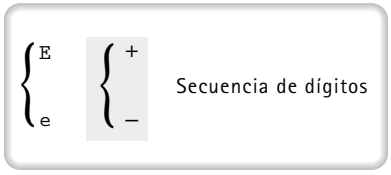
A continuación se muestra la plantilla para una constante de punto flotante en C++:

Constante de punto flotante



La secuencia de dígitos es la misma que se definió en la sección de constantes enteras, una secuencia de dígitos decimales (base 10). La forma del Exponente es:

Exponente



10.2 Más operadores de C++

C++ tiene una rica variedad, en ocasiones desconcertante, de operadores que le permiten manipular valores de los tipos de datos simples. Los operadores que ha conocido hasta el momento incluyen el operador de asignación (`=`), los operadores aritméticos (`+`, `-`, `*`, `/`, `%`), los operadores de incremento y decremento (`++`, `--`), los operadores relacionales (`==`, `!=`, `<`, `<=`, `>`, `>=`), y los operadores lógicos (`!`, `&&`, `||`). En algunos casos, un par de paréntesis se considera un operador; a saber, el operador de llamada de función,

```
ComputeSum(x, y);
```

y el operador de conversión de tipo,

```
y = float(someInt);
```

C++ tiene también muchos operadores especializados que pocas veces se encuentran en otros lenguajes de programación. A continuación se muestra una tabla de dichos operadores adicionales. Conforme se inspeccione la tabla, no se atemorice, una rápida revisión será suficiente.

Operador		Observaciones
Operadores de asignación combinados		
`+=`	Sumar y asignar	
`-=`	Restar y asignar	
`*=`	Multiplicar y asignar	
`/=`	Dividir y asignar	
Operadores de incremento y decremento		
`++`	Preincremento	Ejemplo: `++someVar`
`++`	Posincremento	Ejemplo: `someVar++`
`- -`	Predecremento	Ejemplo: `- -someVar`
`- -`	Postdecremento	Ejemplo: `someVar- -`
Operadores por bits		Sólo operandos enteros
`<<`	Corrimiento a la izquierda	
`>>`	Corrimiento a la derecha	
`&`	AND por bits	
`|`	OR por bits	
`^`	EXCLUSIVE OR por bits	
`~`	Complemento (invertir todos los bits)	
Más operadores de asignación combinados		Sólo operandos enteros
`%=`	Módulo y asignación	
`<<=`	Corrimiento a la izquierda y asignación	
`>>=`	Corrimiento a la derecha y asignación	
`&=`	AND por bits y asignación	
`|=`	OR por bits y asignación	
`^=`	EXCLUSIVE OR por bits y asignación	
Otros operadores		
`()`	Moldeo	
`sizeof`	Tamaño del operando en bytes	Forma: `sizeof` Expr `or sizeof`(Type)
`?:`	Operador condicional	Forma: Expr1 `?` Expr2 : Expr3

Los operadores de esta tabla, junto con los que ya conoce, comprenden la mayoría de los operadores de C++, pero no todos. En capítulos posteriores se introducen algunos operadores adicionales conforme vaya siendo necesario.

Operadores de asignación y expresiones de asignación

C++ tiene varios operadores de asignación. El signo igual (=) es el operador de asignación básico. Cuando se combina con sus dos operandos, forma una **expresión de asignación** (*no* una sentencia de asignación). Toda expresión de asignación tiene un *valor* y un *efecto secundario*, a saber, que el valor se almacena en el objeto denotado por el lado izquierdo. Por ejemplo, la expresión

```
delta = 2 * 12
```

tiene el valor 24 y el efecto secundario de guardar este valor en `delta`.

Expresión de asignación Expresión de C++ con (1) un valor y (2) el efecto secundario de guardar el valor de la expresión en un lugar de la memoria.

Sentencia de asignación Sentencia formada al anexar un punto y coma a una expresión.

En C++, cualquier expresión se convierte en una **sentencia de asignación** cuando se termina mediante un punto y coma. Las tres siguientes son sentencias de C++ válidas, aunque las dos primeras no tienen efecto en absoluto al momento de la ejecución:

```
23;
2 * (alpha + beta);
delta = 2 * 12;
```

La tercera sentencia de expresión *es* útil debido a su efecto secundario de almacenar 24 en `delta`.

Como una asignación es una expresión, no una sentencia, se permite usarla en cualquier parte de una expresión. Aquí tiene una sentencia que almacena el valor 20 en `firstInt`, el valor 30 en `secondInt` y el valor 35 en `thirdInt`:

```
thirdInt = (secondInt = (firstInt = 20) + 10) + 5;
```

Algunos programadores de C++ usan este estilo de codificación, pero otros lo consideran difícil de leer y propenso a errores.

En el capítulo 5 se recomendó tener precaución contra el error de usar el operador `=` en lugar de `==`:

```
if (alpha = 12)  // Erróneo
    :
else
    :
```

La condición en la sentencia If es una expresión de asignación, no una expresión relacional. El valor de la expresión es 12 (interpretada en la condición If como `true`), así que la cláusula else nuca se ejecuta. Peor aún, el efecto secundario de la expresión de asignación es guardar 12 en `alpha` y, por tanto, destruye su contenido previo.

Además del operador `=`, C++ tiene varios operadores de asignación combinados (`+=`, `*=` y los otros listados en la tabla de operadores). Dichos operadores tienen la semántica siguiente:

Sentencia	Sentencia equivalente
`i += 5;`	`i = i + 5;`
`pivotPoint *= n + 3;`	`pivotPoint = pivotPoint * (n + 3);`

Los operadores de asignación combinados son otro ejemplo de "helado y pastel". En ocasiones son convenientes para escribir una línea de código de manera más compacta, pero puede prescindir de ellos.

Operadores de incremento y decremento

Los operadores de incremento y decremento (`++` y `--`) operan sólo en variables, no en constantes o expresiones arbitrarias. Suponga que una variable `someInt` contiene el valor 3. La expresión `++someInt` denota preincremento. El efecto secundario de incrementar `someInt` ocurre primero, así que el valor resultante de la expresión es 4. En contraste, la expresión `someInt++` denota posincremento. El valor de la expresión es 3, y *después* tiene lugar el efecto de incrementar `someInt`. En el siguiente código se ilustra la diferencia entre pre y posincremento.

```
int1 = 14;
int2 = ++int1;
// Assert: int1 == 15  &&  int2 == 15

int1 = 14;
int2 = int1++;
// Assert: int1 == 15  &&  int2 == 14
```

Usar efectos secundarios a la mitad de expresiones más grandes es siempre un poco arriesgado. Es fácil cometer errores de semántica, y la lectura del código puede ser confusa. Examínese el siguiente ejemplo:

```
a = (b = c++) * - -d / (e += f++);
```

Algunas personas se entretienen buscando cuánto pueden hacer con el mínimo de teclazos posibles. Pero es necesario recordar que el desarrollo serio de software requiere escribir códigos que otros programadores puedan leer y entender. El uso excesivo de efectos secundarios impide este objetivo. Por mucho, el uso más común de `++` y `--` es hacer el incremento o decremento como una sentencia de expresión separada:

```
count++;
```

Aquí no se emplea el valor de la expresión, pero se obtiene el efecto secundario deseado de incrementar `count`. En este ejemplo, no importa si usa preincremento o posincremento.

Operadores por bits (a nivel de bits)

Los operadores por bits listados en la tabla de operadores (`<<`, `>>`, `&`, `|`, etcétera) se utilizan para manipular bits individuales dentro de una celda de memoria. En este libro no se analiza el uso de dichos operadores; el tema de las operaciones a nivel de bit se estudia con más frecuencia en un curso acerca de organización de computadoras y programación con lenguaje ensamblador. Sin embargo, se señalan dos características acerca de los operadores por bits.

Primero, los operadores integrados `<<` y `>>` son los operadores de desplazamiento a la izquierda y desplazamiento a la derecha, respectivamente. Su propósito es llevar los bits dentro de una celda de memoria y desplazarlos a la izquierda o a la derecha. Por supuesto, estos operadores se han estado usando todo el tiempo, pero en un contexto completamente distinto, entrada y salida de programa. El archivo de encabezado `iostream` usa una técnica avanzada de C++ denominada *sobrecarga de operador* para dar significados adicionales a estos dos operadores. Un operador de sobrecarga es el que tiene significados múltiples, lo cual depende de los tipos de datos de sus operandos. Al examinar el operador `<<`, el compilador determina por el contexto si se desea una operación de desplazamiento a la izquierda o una operación de salida. En particular, si el primer operando (lado izquierdo) denota un flujo de salida, entonces es una operación de salida. Si el primer operando es una variable entera, es una operación de desplazamiento a la izquierda.

Segundo, se repite la advertencia del capítulo 5: no confundir los operadores `&&` y `||` con los operadores `&` y `|`. La sentencia

```
if (i == X & j == 4)    // Erróneo
    k = 20;
```

es correcta desde el punto de vista sintáctico porque `&` es un operador válido (el operador AND por bits). El programa que contiene esta sentencia se compila de manera correcta, pero se ejecuta de modo incorrecto. Aunque no se examina lo que hacen los operadores AND y OR por bits, tenga cuidado al usar los operadores relacionales `&&` y `||` en sus expresiones lógicas.

Operación de moldeo (cast)

Se ha visto que C++ permite con mucha libertad que el programador combine tipos de datos en expresiones, en operaciones de asignación, en el paso de argumentos y en la devolución de un valor de función. Sin embargo, la coerción implícita de tipos tiene lugar cuando se combinan tipos de datos distintos. En lugar de depender de la coerción implícita de tipos en una sentencia como

```
intVar = floatVar;
```

se recomienda usar un moldeo explícito de tipos para mostrar que la conversión de tipos es intencional:

```
intVar = int(floatVar);
```

En C++, la operación de moldeo viene en dos formas:

```
intVar = int(floatVar);      // Notación funcional
intVar = (int) floatVar;     // Notación de prefijo. Se requieren paréntesis
```

La primera forma se denomina notación funcional porque se parece a una llamada de función. En realidad no es una llamada de función (no hay un subprograma de nombre `int` definido por el usuario o predefinido), pero tiene la sintaxis y aspecto visual de una llamada de función. La segunda forma, notación de prefijo, no tiene parecido con ninguna característica de lenguaje familiar en C++. En esta notación, los paréntesis rodean el nombre del tipo de datos, no la expresión que se convierte. La notación de prefijo es la única forma disponible en el lenguaje C; C++ agregado a la notación funcional.

Aunque la mayoría de los programadores de C++ usan la notación funcional para la operación de moldeo, hay una restricción en su uso. El nombre del tipo de datos debe ser un identificador simple. Si el nombre del tipo consta de más de un identificador, se *debe* usar la notación de prefijo. Por ejemplo,

```
myVar = unsigned int(someFloat);      // No
myVar = (unsigned int) someFloat;     // Sí
```

Operador `sizeof`

El operador `sizeof` es un operador unario que produce el tamaño, en bytes, de su operando. El operando puede ser un nombre de variable, como en

```
sizeof someInt
```

o el operando puede ser el nombre de un tipo de datos, ubicado entre paréntesis:

```
sizeof(float)
```

Usted podría hallar los tamaños de varios tipos de datos de su máquina si usa un código como el siguiente:

```
cout << "El tamaño de un short es " << sizeof(short) << endl;
cout << "El tamaño de un int es " << sizeof(int) << endl;
cout << "El tamaño de un long es " << sizeof(long) << endl;
```

Operador `?:`

El último operador de la tabla es `?:`, denominado en ocasiones operador condicional. Es un operador ternario (tres operandos) con la sintaxis siguiente:

Expresión condicional

```
Expresión1 ? Expresión2 : Expresión3
```

Enseguida se muestra cómo funciona. Primero, la computadora evalúa la expresión 1. Si el valor es `true`, entonces el valor de la expresión completa es la Expresión 2; de lo contrario, el valor de toda la expresión es la Expresión 3. (Sólo se evalúa una de las Expresiones, ya sea la 1 o la 2.) Un ejemplo

clásico de su uso es igualar una variable max con el valor más grande de las dos variables a y b. Con una sentencia If, se procedería de esta manera:

```
if (a > b)
    max = a;
else
    max = b;
```

Con el operador `?:`, se puede usar la sentencia de asignación siguiente:

```
max = (a > b) ? a : b;
```

Enseguida se presenta otro ejemplo. El valor absoluto de un número *x* se define como

$$|x| = \begin{cases} x, & \text{if } x \geq 0 \\ -x, & \text{if } x < 0 \end{cases}$$

Para calcular el valor absoluto de una variable `x` y guardarlo en `y`, se podría usar el operador `?:` de la siguiente manera:

```
y = (x >= 0) ? x : -x;
```

En los ejemplos de `max` y de valor absoluto se usaron paréntesis alrededor de la expresión que se prueba. Dichos paréntesis son innecesarios porque, como se verá en breve, el operador condicional tiene muy poca precedencia. Pero es conveniente incluir los paréntesis por claridad.

Precedencia de operadores

En la tabla siguiente se resume la precedencia de los operadores en C++, incluyendo todos los vistos anteriormente excepto los operadores a nivel de bits. (En el apéndice B se encuentra la lista completa.) En la tabla, los operadores se agrupan por nivel de precedencia, y una línea horizontal separa cada nivel de precedencia del siguiente nivel inferior.

Precedencia (máxima a mínima)		
Operador	Asociatividad	Observaciones
`()`	Izquierda a derecha	Llamada de función y moldeo de estilo de función
`++ - -`	Derecha a izquierda	++ y -- como operadores de sufijo
`++ - - ! Unario + Unario - (moldeo) sizeof`	Derecha a izquierda Derecha a izquierda	++ y -- como operadores de prefijo
`* / %`	Izquierda a derecha	
`+ -`	Izquierda a derecha	
`< <= > >=`	Izquierda a derecha	
`== !=`	Izquierda a derecha	
`&&`	Izquierda a derecha	
`||`	Izquierda a derecha	
`?:`	Derecha a izquierda	
`= += -= *= /=`	Derecha a izquierda	

La columna identificada como *Asociatividad* describe el orden de agrupamiento. Dentro de un nivel de precedencia, la mayoría de los operadores se agrupan de izquierda a derecha. Por ejemplo,

```
a - b + c
```

significa

```
(a - b) + c
```

y no

```
a - (b + c)
```

No obstante, algunos operadores se agrupan de derecha a izquierda; en particular los operadores unarios, los de asignación y el `?:`. Por ejemplo, examine los operadores de asignación. La expresión

```
sum = count = 0
```

significa

```
sum = (count = 0)
```

Esta asociatividad tiene sentido porque la operación de asignación es, por naturaleza, una operación de derecha a izquierda.

Advertencia: aunque la precedencia y la asociatividad del operador dictan el *agrupamiento* de operadores con sus operandos, C++ no define el *orden* en que se evalúan las subexpresiones. Por tanto, usar efectos secundarios en las expresiones requiere cuidado extra. Por ejemplo, si `i` contiene actualmente 5, la sentencia

```
j = ++i + i;
```

almacena ya sea 11 o 12 en `j`, dependiendo del compilador que se use. Veamos por qué. Hay tres operadores en la sentencia de expresión anterior: `=`, `++` y `+`. El operador `++` tiene la precedencia más alta, así que opera sólo en `i`, no en la expresión `i + i`. El operador de suma tiene mayor precedencia que el operador de asignación, lo que da paréntesis implícitos de la manera siguiente:

```
j = (++i + i);
```

Hasta el momento, todo bien. Pero ahora se plantea esta pregunta: en el operador de adición, ¿se evalúa primero el operando izquierdo o el derecho? El lenguaje C++ no dicta el orden. Si un compilador genera código para evaluar primero el operando izquierdo, el resultado es 6 + 6 o 12. Otro compilador podría generar código para evaluar primero el operando derecho, y se obtendría 6 + 5 u 11. Para asegurar la evaluación de izquierda a derecha en este ejemplo, es necesario forzar el orden con dos sentencias separadas:

```
++i;
j = i + i;
```

La moraleja aquí es que si usa efectos secundarios múltiples en las expresiones, incrementa el riesgo de resultados inesperados o incongruentes. Para los principiantes en C++, es mejor evitar efectos secundarios innecesarios.

10.3 Trabajar con datos de caracteres

Se han estado usando variables `char` para guardar datos de caracteres, como el carácter 'A' o 'e' o '+':

```
char someChar;
    ⋮
someChar = 'A';
```

Sin embargo, como `char` se define como un tipo integral y `sizeof(char)` es igual a 1, es posible usar también una variable `char` para guardar una constante entera pequeña (por lo común, un byte). Por ejemplo,

```
char counter;
    ⋮
counter = 3;
```

En computadoras con una cantidad muy limitada de espacio de memoria, los programadores en ocasiones usan el tipo `char` para ahorrar memoria cuando trabajan con enteros pequeños.

Una pregunta natural es: ¿cómo establece la computadora la diferencia entre datos enteros y datos de caracteres localizados en una celda de memoria? La respuesta es: ¡la computadora *no puede* determinar la diferencia! Para explicar esta situación sorprendente, es necesario examinar con detalle cómo se almacenan los datos en una computadora.

Conjuntos de caracteres

Cada computadora utiliza un conjunto particular de caracteres, el conjunto de todos los caracteres posibles con los que es capaz de trabajar. Dos conjuntos de caracteres muy usados en la actualidad son el de caracteres ASCII y el de caracteres EBCDIC. El conjunto ASCII es el que usa la gran mayoría de las computadoras, mientras que el conjunto EBCDIC se encuentra sobre todo en computadoras centrales IBM. El conjunto ASCII consta de 128 caracteres distintos y el conjunto EBCDIC tiene 256. En el apéndice E se muestran los caracteres disponibles en ambos conjuntos.

Un conjunto de caracteres desarrollado recientemente, denominado *Unicode,* permite muchos más caracteres distintos que los conjuntos ASCII o EBCDIC. El conjunto Unicode se inventó sobre todo para acomodar alfabetos más grandes y símbolos de varios idiomas. En C++, el tipo de datos `wchar_t`, en vez de `char`, se usa para caracteres Unicode. De hecho, `wchart_t` se puede usar para otros, tal vez poco usados, conjuntos de "caracteres amplios" además de Unicode. En este libro no se examina Unicode o el tipo `wchar_t`. Se continúa con el estudio del tipo `char` y los conjuntos de caracteres ASCII y EBCDIC.

Representación externa Forma (carácter) imprimible de un valor de datos.

Representación interna Forma en que se guardan valores de datos dentro de la unidad de memoria.

Sin importar el conjunto de caracteres que se use, cada carácter tiene una **representación externa**, la forma como aparece en un dispositivo I/O como una impresora, y una **representación interna**, la forma como se guarda dentro de la unidad de memoria de la computadora. Si usa la constante `char` 'A' en un programa de C++, su representación externa es la letra *A*. Es decir, si la imprime verá una *A*, como esperaba. No obstante, su representación interna es un valor entero. Los 128 caracteres ASCII tienen representaciones internas de 0 a 127; los caracteres EBCDIC, de 0 a 255. Por ejemplo, la tabla ASCII del apéndice E muestra que el carácter 'A' tiene representación interna 65, y el carácter b tiene representación interna 98.

Considérese de nuevo la sentencia

```
someChar = 'A';
```

Suponiendo que nuestra máquina usa el conjunto de caracteres ASCII, el compilador traduce la constante 'A' en el entero 65. La sentencia se podría haber escrito como

```
someChar = 65;
```

Ambas sentencias tienen exactamente el mismo efecto, el de almacenar 65 en `someChar`. Sin embargo, *no* se recomienda la segunda versión. No es tan comprensible como la primera, y no es portable (el programa no funcionará de modo correcto en una máquina que usa el conjunto EBCDIC, el cual emplea representación interna distinta; 193 para 'A').

Ya se mencionó que la computadora no puede establecer la diferencia entre datos de caracteres y enteros en la memoria. Ambos se almacenan internamente como enteros. Sin embargo, cuando se efectúan operaciones I/O, la computadora hace lo correcto, utiliza la representación interna que corresponde al tipo de datos de la expresión que se imprime. Considere, por ejemplo, el segmento de código siguiente:

```
// Se supone que en este ejemplo se usa el conjunto de caracteres ASCII
int  someInt = 97;
char someChar = 97;

cout << someInt << endl;
cout << someChar << endl;
```

Cuando se ejecutan estas sentencias, el resultado es

```
97
a
```

Cuando el operador `<<` produce `someInt`, imprime la secuencia de caracteres 9 y 7. Para producir `someChar`, imprime el carácter simple `a`. Aunque ambas variables contienen el valor 97 internamente, el tipo de datos de cada una determina cómo se imprime.

¿Qué cree que produzca la siguiente secuencia de sentencias?

```
char ch = 'D';

ch++;
cout << ch
```

Si su respuesta fue `E`, es correcto. La primera sentencia declara ch y la inicializa en el valor entero 68 (en ASCII). La sentencia siguiente incrementa `ch` a 69 y luego se imprime su representación externa (la letra *E*). Ampliando esta idea de incrementar una variable `char` se podrían imprimir las letras *A* a *G*, como sigue:

```
char ch;

for (ch = 'A'; ch <= 'G'; ch++)
    cout << ch;
```

Este código inicializa ch en 'A' (65 en ASCII). Cada vez mediante el ciclo, se imprime la representación externa de `ch`. En la iteración final, se imprime *G* y `ch` se incrementa a 'H' (72 en ASCII). Entonces la prueba de ciclo es falsa, así que termina el ciclo.

Constantes `char` de C++

En C++, las constantes `char` se presentan de dos modos. El primero, el cual se ha estado usando de manera regular, es un carácter imprimible simple entre comillas sencillas:

```
'A'   '8'   ')'   '+'
```

Observe que se dice carácter *imprimible*. El conjunto de caracteres incluye ambos caracteres imprimibles y *caracteres de control* (o *caracteres no imprimibles*). No se da a entender que se imprimirán los caracteres de control, sino que se usan para controlar la pantalla, impresora y otros dispositivos de hardware. Si examina la tabla de caracteres ASCII, es evidente que los caracteres imprimibles son los que tienen valores enteros 32-126. Los caracteres restantes (con valores 0-31 y 127) son caracteres de control no imprimibles. En el conjunto de caracteres EBCDIC, los caracteres de control son los que tienen valores de 0 a 63 y de 250 a 255 (y algunos que están entremezclados con los caracteres imprimibles). Un carácter de control que ya conoce es el de nueva línea, que hace que el cursor de la pantalla avance a la línea siguiente.

Para acomodar los caracteres de control, C++ proporciona una segunda forma de la constante `char`: la *secuencia de escape*. Una secuencia de escape es uno o más caracteres precedidos por una diagonal invertida (\). Ya se ha familiarizado con la secuencia de escape \n, que representa el carácter de nueva línea. A continuación se presenta la descripción completa de las dos formas de la constante `char` en C++:

1. Un solo carácter imprimible, excepto una comilla sencilla (`'`) o una diagonal invertida (`\`), entre comillas sencillas.
2. Una de las siguientes sentencias de escape entre comillas sencillas:

`\n`	Nueva línea (alimentación de línea en ASCII)
`\t`	Tabulador horizontal
`\v`	Tabulador vertical
`\b`	Tecla de retroceso
`\r`	Retorno de carro
`\f`	Suministro de papel
`\a`	Alerta (una campanada o pip)
`\\`	Diagonal invertida
`\'`	Comilla simple (apóstrofo)
`\"`	Comilla doble
`\0`	Carácter nulo (todos los bits 0)
`\ddd`	Equivalente octal (uno, dos o tres dígitos octales que especifican el valor entero del carácter deseado)
`\xddd`	Equivalente hexadecimal (uno o más dígitos hexadecimales que especifican el valor entero del carácter deseado)

Aunque una secuencia de escape se escribe como dos o más caracteres, cada una representa un solo carácter en el conjunto de caracteres. El carácter de alerta (`\a`) es lo mismo que lo que se denomina carácter BEL en ASCII y EBCDIC. Para hacer sonar la campana en su computadora, se puede producir un carácter de alerta como:

```
cout << '\a';
```

En la lista de secuencias de escape anterior, las entradas marcadas como *equivalente octal* y *equivalente hexadecimal* permiten hacer referencia a cualquier carácter del conjunto de caracteres de su máquina al especificar su valor entero ya sea de manera octal o hexadecimal.

Observe que es posible usar una secuencia de escape dentro de una cadena, del mismo modo que se usa cualquier carácter imprimible dentro de una cadena. La sentencia

```
cout << "\aWhoops!\n";
```

hace sonar la campana, muestra Whoops! y termina la línea de salida. La sentencia

```
cout << "She said \"Hi\"";
```

produce `She said "Hi"`, y no termina la línea de salida.

Técnicas de programación

¿Qué se puede hacer con datos de caracteres de un programa? En el capítulo anterior se incluyeron diferentes categorías de caracteres. Las posibilidades son interminables y dependen, por supuesto, del problema particular a resolver. Sin embargo, varias técnicas tienen un uso tan extenso que es importante revisarlas.

Comparación de caracteres En capítulos anteriores se vieron ejemplos de comparar la igualdad de caracteres. Se han usado pruebas como

```
if (ch == 'a')
```

y

```
while (inputChar != '\n')
```

Los caracteres se pueden comparar también por medio de `<`, `<=`, `>` y `>=`. Por ejemplo, si la variable `firstLetter` contiene la primera letra del apellido de una persona, es posible probar si comienza con *A* a *H* mediante esta prueba:

```
if (firstLetter >= 'A' && firstLetter <= 'H')
```

En un nivel de pensamiento, una prueba como ésta es razonable si considera que < significa "viene antes" en el conjunto de caracteres y que > significa "viene después". En otro nivel, la prueba tiene incluso más sentido cuando considera que la representación subyacente de un carácter es un número entero. La máquina literalmente compara ambos valores enteros con el significado matemático de menor o mayor que.

Cuando escribe una expresión lógica para comprobar si un carácter está dentro de cierto intervalo de valores, en ocasiones tiene que recordar el conjunto de caracteres que usa su máquina. En el capítulo 8 se sugirió que una prueba como

```
if (ch >= 'a' && ch <= 'z')
```

funciona de manera correcta en algunas máquinas, pero no en otras. En ASCII, esta prueba If se comporta correctamente porque las letras minúsculas ocupan 26 posiciones consecutivas en el conjunto de caracteres. Sin embargo, en EBCDIC, hay un espacio entre las letras minúsculas `i` y `j` que incluye caracteres no imprimibles, y hay otro entre *r* y *s*. (Hay espacios similares entre las letras mayúsculas *I* y *J* y entre *R* y *S*.) Si su máquina usa EBCDIC, debe expresar de otro modo la prueba If para asegurarse de que *sólo* incluya los caracteres deseados. Un mejor método es aprovechar las funciones "`is...`" suministradas por la biblioteca estándar a través del encabezado de archivo `cctype`. Si remplaza la prueba If anterior con la siguiente:

```
if (islower(ch))
```

entonces su programa es más portable; la prueba funciona de manera correcta en cualquier máquina, sin importar su conjunto de caracteres. Es buena idea familiarizarse bien con estas funciones de biblioteca de prueba de caracteres (apéndice C). Le pueden ahorrar tiempo y facilitarle la escritura de programas portables.

Conversión de caracteres de dígitos a enteros Suponga que se desea convertir un dígito, que se lee en forma de carácter, en su equivalente numérico. Debido a que los caracteres de dígitos '0' al '9' son consecutivos en los caracteres ASCII y EBCDIC, restar '0' de cualquier dígito en forma de carácter proporciona el dígito en forma numérica:

```
'0' - '0' = 0
'1' - '0' = 1
'2' - '0' = 2
 ⋮
```

Por ejemplo, en ASCII, '0' tiene representación interna 48, y '2' tiene representación interna 50. Por tanto, la expresión

```
'2' - '0'
```

es igual a 50 – 48 y evalúa 2.

¿Por qué habría que hacer esto? Recuerde que cuando el operador de extracción (>>) lee datos en una variable `int`, el flujo de entrada falla si se encuentra un carácter inválido. (Una vez que ha fallado el flujo, ninguna entrada más tendrá éxito.) Suponga que está escribiendo un programa que solicita a un usuario experimentado introducir un número del 1 al 5. Si la variable de entrada es de tipo `int` y el usuario escribe de modo accidental una letra del alfabeto, el programa está en problemas. Para prevenir esta posibilidad, usted podría leer la respuesta del usuario como un carácter y convertirlo en un número, y efectuar la comprobación de errores en el camino. Enseguida se presenta un segmento de código que demuestra la técnica:

```
#include <cctype>   // Para isdigit()
using namespace std;
    :
void GetResponse( /* out */ int& response )

// Poscondición:
//     Se ha solicitado al usuario que introduzca un dígito del 1
//     al 5 (de manera repetida, y con mensajes de error,
//     si el dato es inválido)
//  && 1 <= response <= 5

{
    char inChar;
    bool badData = false;

    do
    {
        cout << "Introduzca un número del 1 al 5: ";
        cin >> inChar;
        if ( !isdigit(inChar) )
            badData = true;                  // No es un dígito
        else
        {
            response = int(inChar - '0');
            if (response < 1 || response > 5)
                badData = true;              // Es un dígito, pero
        }                                    // está fuera del intervalo
        if (badData)
            cout << "Por favor intente de nuevo" << endl;
    } while (badData);
}
```

Conversión a minúsculas y mayúsculas Al trabajar con datos de caracteres, en ocasiones se encuentra que es necesario convertir una letra minúscula a mayúscula, o viceversa. Por fortuna, la técnica de programación requerida para hacer dichas conversiones es fácil; todo lo que se necesita es una simple llamada a una función de biblioteca. Por medio del encabezado de archivo `cctype`, la biblioteca estándar proporciona no sólo las funciones "`is...` " que se han explicado, sino también dos funciones de devolución de valor denominadas `toupper` y `tolower`. A continuación se presentan sus descripciones:

Archivo de encabezado	Función	Tipo de función	Valor de función
<cctype>	toupper(ch)	char*	Equivalente en mayúsculas de ch, si ch es una letra minúscula; ch, en caso contrario
<cctype>	tolower(ch)	char	Equivalente en minúsculas de ch, si ch es una letra mayúscula; ch, en caso contrario

*Técnicamente, tanto el argumento como el valor de retorno son de tipo int. Pero desde el punto de vista conceptual, las funciones operan en datos de caracteres.

Observe que el valor devuelto por cada función es el carácter original si no se cumple la condición. Por ejemplo, tolower ('M') devuelve el carácter 'm', mientras tolower ('+') devuelve '+'.

Una característica de estas dos funciones es que permite al usuario responder a determinados avisos de introducción de información mediante el uso de letras mayúsculas o minúsculas. Por ejemplo, si desea permitir *S* o *s* para una respuesta "Sí" del usuario, y *N* o *n* para "No", se podría hacer lo siguiente:

```
#include <cctype>     Para toupper()
using namespace std;
    :
cout << "Introducir S o N: ";
cin >> inputChar;
if (toupper(inputChar) == 'Y')
{
    ⋮
}
else if (toupper(inputChar) == 'N')
{
    ⋮
}
else
    PrintErrorMsg();
```

A continuación se proporciona una función denominada Lower, que es la Implementación de la función tolower. (Usted no desperdiciaría tiempo escribiendo esta función porque tolower ya está disponible.) Dicha función devuelve el equivalente en minúscula de una letra mayúscula. En ASCII, cada letra minúscula tiene exactamente 32 posiciones más allá de la letra mayúscula correspondiente. Y en EBCDIC, las minúsculas están 64 posiciones *antes* de sus letras mayúsculas correspondientes. Para hacer que trabaje la función Lower tanto en máquinas basadas en ASCII como en las basadas en EBCDIC, se define una constante DISTANCE que tenga el valor

```
'a' - 'A'
```

En ASCII, el valor de esta expresión es 32. En EBCDIC, el valor es –64.

```
#include <cctype>     // Para isupper()
using namespace std;
    ⋮
char Lower( /* in */ char ch )

// Poscondición:
//     Valor de función == equivalente en minúsculas de ch, si ch es
//                         una letra mayúscula
//                      == ch en caso contrario
```

```
{
    const int DISTANCE = 'a' - 'A';  // Distancia fija entre
                                     // letras mayúsculas
                                     // y minúsculas
    if (isupper(ch))
        return ch + DISTANCE;
    else
        return ch;
}
```

Acceso a los caracteres en una cadena En la última sección se proporcionó una descripción de un segmento de código que solicita al usuario una respuesta "Sí" o "No". El código aceptó una respuesta 'S' o 'N' en letras mayúsculas o minúsculas. Si un problema requiere que el usuario escriba la palabra completa *Sí* o *No* en cualquier combinación de letras mayúsculas o minúsculas, el código se torna más complicado. Leer la respuesta del usuario como una cadena en un objeto `string` denominado `inputStr`, requeriría una estructura larga If-Then-Else-If para comparar `inputStr` con "`sí`", "`Sí`", "`sÍ`", etcétera.

Como opción, se inspeccionará sólo el primer carácter de la cadena de entrada, comparándola con 'S', 's', 'N' o 'n', y luego ignorar el resto de la cadena. La clase `string` permite tener acceso a un carácter de una cadena al dar su número de posición en corchetes:

```
Objeto string [ Posición ]
```

Dentro de una cadena, el primer carácter está en la posición 0, el segundo en la posición 1, etcétera. Por tanto, el valor de Posición debe ser mayor que o igual a 0 y menor que o igual a la longitud de cadena menos 1. Por ejemplo, si `inputStr` es un objeto `string` y `ch` es una variable `char`, la sentencia

```
ch = inputStr[2];
```

tiene acceso al carácter en la posición 2 de la cadena (el tercer carácter) y lo copia en `ch`.

Ahora es posible bosquejar el código para leer una respuesta "Sí" o "No", al comprobar sólo la primera letra de esa respuesta.

```
string inputStr;
    ⋮
cout << "Introduzca Sí o No: ";
cin >> inputStr;
if (toupper(inputStr[0]) == 'Y')
{
    ⋮
}
else if (toupper(inputStr[0]) == 'N')
{
    ⋮
}
else
    PrintErrorMsg();
```

Enseguida se presenta otro ejemplo de acceso a caracteres dentro de una cadena. El programa siguiente pide al usuario escribir el nombre de un mes y luego proporciona como resultado cuántos días hay en ese mes. La entrada puede ser en mayúsculas o minúsculas, y el programa permite una entrada aproximada. Por ejemplo, las entradas `Febrero`, `FEBRERO`, `fEbRe`, `feb`, `fe`, `f` y `fyz34x` son

interpretadas como Febrero porque es el único mes que empieza con *f*. Sin embargo, la entrada Ma es rechazada porque podría representar Marzo o Mayo. Para ahorrar espacio, se ha omitido la documentación de interfaz para la función `DaysInMonth` en este programa.

```
//**************************************************************************
// Programa Número de días
// Este programa solicita de modo repetido un mes y produce el
// número de días de ese mes. Se permite una entrada aproximada: sólo
// se examinan los caracteres necesarios para determinar el mes
//**************************************************************************
#include <iostream>
#include <cctype>     // Para toupper()
#include <string>     // Para tipo string

using namespace std;

string DaysInMonth( string );

int main()
{
    string month;    // Valor de entrada del usuario
    do
    {
        cout << "Nombre del mes (o q para salir) : ";
        cin >> month;
        if (month != "q")
            cout << "El número de días en " << month << " es "
                 << DaysInMonth(month) << endl;
    } while (month != "q");
    return 0;
}

//**************************************************************************

string DaysInMonth( /* in */ string month )
{
    string::size_type i;                    // Variable de control de ciclo
    string badData = "** Mes inválido **";  // Mensaje de datos

    // Convertir a mayúsculas

    for (i = 0; i < month.length(); i++)
        month[i] = toupper(month[i]);

    // Asegúrese que la longitud es por lo menos × para próximas pruebas

    month = month + "  ";

    // Examinar el primer carácter, después otros si es necesario

    switch (month[0])
    {
        case 'J' : if (month[1] == 'A' ||     // Enero
                       month[2] == 'L'   )    // Julio
```

```
                    return "31";
                else if (month[2] == 'N')  // Junio
                    return "30";
                else
                    return badData;
        case 'F' : return "28 o bien 29";;    // Febrero
        case 'M' : if (month[2] == 'R' ||     // Marzo
                    month[2] == 'Y'   )       // Mayo
                    return "31";
                else
                    return badData;
        case 'A' : if (month[1] == 'P')       // Abril
                    return "30";
                else if (month[1] == 'U')  // Agosto
                    return "31";
                else
                    return badData;
        case 'S' :                            // Septiembre
        case 'N' : return "30";               // Noviembre
        case 'O' :                            // Octubre
        case 'D' : return "31";               // Diciembre
        default  : return badData;
    }
}
```

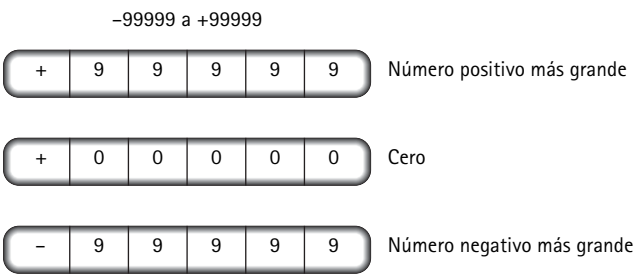
10.4 Más acerca de números de punto flotante

Se han usado números de punto flotante desde que se introdujeron en el capítulo 2, pero no se han examinado con detenimiento. Los números de punto flotante tienen propiedades especiales cuando se usan en la computadora. Hasta ahora, en cierta medida se han ignorado dichas propiedades, pero es el momento de considerarlas minuciosamente.

Representación de números de punto flotante

Suponga que se tiene una computadora en la que cada ubicación de memoria es del mismo tamaño y se divide en un signo más cinco dígitos decimales. Cuando se define una constante o variable, la ubicación asignada a ésta consta de cinco dígitos y un signo. Cuando se define una variable o constante `int`, la interpretación del número almacenado en ese lugar es directa. Cuando se define una variable o constante `float`, el número almacenado ahí tiene una parte entera y una parte fraccionaria, por lo que es necesario codificar para representar ambas partes.

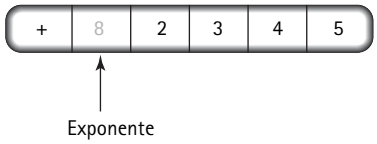
Veamos cuál podría ser la apariencia de dichos números codificados. Es posible representar el intervalo de números enteros con cinco dígitos; de –99 999 a +99 999:



La **precisión** (el número de dígitos que se puede representar) es cinco dígitos, y cada número dentro de ese intervalo se puede representar de modo exacto.

Precisión Número máximo de dígitos significativos.

¿Qué sucede si se permite que uno de esos dígitos (el de la izquierda, por ejemplo) represente un exponente?



Entonces +82 345 representa al número $+2\ 345 \times 10^8$. El intervalo de números que ahora se pueden representar es mucho más grande:

-9999×10^9 a 9999×109

o bien,

−9,999,000,000,000 a +9,999,000,000,000

Sin embargo, la precisión ahora es sólo de cuatro dígitos; esto es, sólo es posible representar exactamente números de cuatro dígitos en el sistema. ¿Qué sucede con los números con más dígitos? Los cuatro dígitos de la izquierda se representan de manera correcta, y los de la derecha —o dígitos menos significativos— se pierden (se supone que son 0). En la figura 10-2 se muestra lo que sucede. Observe que 1 000 000 se puede representar exactamente, pero -4 932 426 no, porque el esquema de codificación limita a cuatro **dígitos significativos.**

Dígitos significativos Dígitos desde el primer no cero a la izquierda hasta el último no cero a la derecha (más cualquier dígito 0 que sea exacto).

Para ampliar el esquema de codificación a fin de representar números de punto flotante, se debe tener la posibilidad de representar exponentes negativos. Aquí se muestran dos ejemplos:

$$7394 \times 10^{-2} = 73.94$$

y

$$22 \times 10^{24} = .0022$$

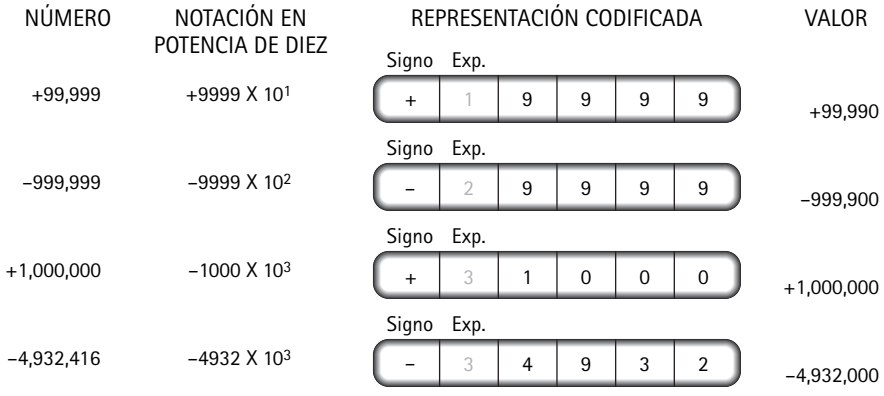


Figura 10-2 *Codificación por medio de exponentes positivos*

Figura 10-3 *Codificación con exponentes positivos y negativos*

Como el esquema no incluye un signo para el exponente, se cambiará un poco. El signo existente se convierte en el signo del exponente, y se añade un signo a la izquierda para representar el signo del número (véase la figura 10-3).

Todos los números entre $-9\ 999 \times 10^9$ y $9\ 999 \times 10^9$ se pueden representar ahora con exactitud hasta cuatro dígitos. Añadir exponentes negativos al esquema permite representar números fraccionarios tan pequeños como 1×10^{-9}.

En la figura 10-4 se muestra cómo se codificarían algunos números de punto flotante. Observe que la precisión aún es de sólo cuatro dígitos. Los números 0.1032, –5.406 y 1 000 000 se pueden representar de modo exacto. Sin embargo, el número 476.0321 con siete dígitos significativos, se representa como 476.0; no es posible representar el *321*. (Es necesario señalar que algunas computadoras realizan el *redondeo* en lugar del truncamiento simple al desechar dígitos sobrantes. Con la suposición de cuatro dígitos significativos, dicha máquina guardaría 476.0321 como 476.0, pero guardaría 476.0823 como 476.1. Se continúa el análisis suponiendo truncamiento simple en lugar de redondeo.)

Aritmética con números de punto flotante

Cuando se usa aritmética de enteros, los resultados son exactos. Sin embargo, la aritmética de punto flotante pocas veces es exacta. Para entender por qué, se sumarán los tres números de punto flotante *x*, *y* y *z* por medio del esquema de codificación.

NÚMERO	NOTACIÓN EN POTENCIA DE DIEZ	REPRESENTACIÓN CODIFICADA (Signo, Exp.)	VALOR
0.1032	$+1032 \times 10^{-4}$	+ – 4 1 0 3 2	0.1032
–5.4060	-5406×10^{-3}	– – 3 5 4 0 6	–5.406
–0.003	-3000×10^{-6}	– – 6 3 0 0 0	–0.0030
476.0321	$+4760 \times 10^{-1}$	+ – 1 4 7 6 0	476.0
1,000,000	$+1000 \times 10^{3}$	+ + 3 1 0 0 0	1,000,000

Figura 10-4 *Codificación de algunos números de punto flotante*

Primero se suma x a y y luego se suma z al resultado. Enseguida se efectúan las operaciones en orden distinto, se suma y a z y luego se suma x al resultado. La ley asociativa de la aritmética establece que las dos respuestas deben ser iguales, pero ¿en realidad lo son? Se usarán los siguientes valores para x, y y z:

$x = -1324 \times 10^3 \quad y = 1325 \times 10^3 \quad z = 5424 \times 10^0$

Enseguida se presenta el resultado de agregar z a la suma de x y y:

$$\begin{array}{lrl} \text{(x)} & -1324 \times 10^3 & \\ \text{(y)} & \underline{1325 \times 10^3} & \\ & 1 \times 10^3 & = 1000 \times 10^0 \\ & & \\ \text{(x+y)} & 1000 \times 10^0 & \\ \text{(z)} & \underline{5424 \times 10^0} & \\ & 6424 \times 10^0 & \leftarrow \text{(x+y)+z} \end{array}$$

Ahora se presenta el resultado de añadir x a la suma de y y z:

$$\begin{array}{lrl} \text{(y)} & 1325000 \times 10^0 & \\ \text{(z)} & \underline{5424 \times 10^0} & \\ & 1330424 \times 10^0 & = 1330 \times 10^3 \text{ (truncado a cuatro dígitos)} \\ \text{(y+z)} & 1330 \times 10^3 & \\ \text{(x)} & \underline{-1324 \times 10^3} & \\ & 6 \times 10^3 & = 6000 \times 10^0 \leftarrow \text{x + (y + z)} \end{array}$$

Estas dos respuestas son lo mismo en el lugar de los miles, pero después de eso son distintas. El error de esta discrepancia se denomina **error representativo.**

Error representativo Error aritmético que ocurre cuando la precisión del resultado verdadero de una operación aritmética es mayor que la precisión de la máquina.

Debido a los errores representativos, no es aconsejable usar una variable de punto flotante como la variable de control de ciclo. Como la precisión podría perderse en los cálculos relacionados con números de punto flotante, es difícil predecir cuándo (o incluso *si*) una variable de control de ciclo de tipo `float` (o `double` o `long double`) será igual al valor de terminación. Un ciclo controlado por conteo con una variable de control de punto flotante puede comportarse de manera impredecible.

Asimismo, debido a los errores representativos, nunca debe comparar números de punto flotante en cuanto a igualdad exacta. Rara vez hay dos números de punto flotante exactamente iguales y, por tanto, debe compararlos sólo para igualdad cercana. Si la diferencia entre los dos números es menor que algún valor pequeño aceptable, puede considerarlos iguales para los fines del problema dado.

Implementación de números de punto flotante en la computadora

Las computadoras limitan la precisión de los números de punto flotante, aunque las máquinas modernas usan aritmética binaria en vez de decimal. En nuestra representación, se usan sólo 5 dígitos para simplificar los ejemplos, y algunas computadoras están limitadas a sólo 4 o 5 dígitos de precisión. Un sistema más representativo podría proporcionar 6 dígitos significativos para valores `float`, 15 dígitos para valores `double` y 19 para el tipo `long double`. Se ha mostrado sólo el exponente de un solo dígito, pero la mayoría de los sistemas permiten 2 dígitos para el tipo `float` y exponentes de hasta 4 dígitos para el tipo `long double`.

Cuando declara una variable de punto flotante, se supone que parte de la ubicación de memoria contiene el exponente y que el número (de nombre *mantisa*) está en el resto de la ubicación. El sistema se denomina representación de punto flotante porque el número de dígitos significativos está fijo y se permite conceptualmente que el punto decimal flote (se mueva a distintas posiciones según sea necesario). En nuestro esquema de codificación, todo número se almacena como cuatro dígitos,

donde el dígito a la izquierda es cero y el exponente se ajusta según el caso. Se dice que, en este caso, los números están *normalizados*. El número 1 000 000 se guarda como

+	+	3	1	0	0	0

y 0.1032 se almacena como

+	–	4	1	0	3	2

La normalización proporciona la máxima precisión posible.

Números modelo Cualquier número real que se pueda representar exactamente como un número de punto flotante en la computadora se denomina *número modelo*. Un número real cuyo valor no es posible representar de modo exacto se aproxima mediante el número modelo más cercano a él. En nuestro sistema con cuatro dígitos de precisión, 0.3021 es un número modelo. Los valores 0.3021409, 0.3021222 y 0.30209999999 son ejemplos de números reales que están representados en la computadora por el mismo número modelo. En la tabla siguiente se muestran los números modelo para un sistema de punto flotante incluso más simple que tiene un dígito en la mantisa y un exponente que puede ser –1, 0 o 1. El cero es un caso especial, porque tiene el mismo valor sin importar el exponente.

0.1×10^{-1}	0.1×10^{0}	$0.1 \times 10^{+1}$
0.2×10^{-1}	0.2×10^{0}	$0.2 \times 10^{+1}$
0.3×10^{-1}	0.3×10^{0}	$0.3 \times 10^{+1}$
0.4×10^{-1}	0.4×10^{0}	$0.4 \times 10^{+1}$
0.5×10^{-1}	0.5×10^{0}	$0.5 \times 10^{+1}$
0.6×10^{-1}	0.6×10^{0}	$0.6 \times 10^{+1}$
0.7×10^{-1}	0.7×10^{0}	$0.7 \times 10^{+1}$
0.8×10^{-1}	0.8×10^{0}	$0.8 \times 10^{+1}$
0.9×10^{-1}	0.9×10^{0}	$0.9 \times 10^{+1}$

La diferencia entre un número real y el número modelo que lo representa es una forma de error de representación llamado *error de redondeo*. El error de redondeo se puede medir de dos modos. El *error absoluto* es la diferencia entre el número real y el número modelo. Por ejemplo, el error absoluto de representar 0.3021409 mediante el número modelo 0.3021 es 0.0000409. El *error relativo* es el error absoluto dividido entre el número real y a veces se expresa como porcentaje. Por ejemplo, 0.0000409 dividido entre 0.3021409 es 0.000135 o 0.0135%.

El error absoluto máximo depende del *intervalo modelo*, la diferencia entre dos números adyacentes modelo. En nuestro ejemplo, el intervalo entre 0.3021 y 0.3022 es 0.0001. El error absoluto máximo del sistema, para este intervalo, es menor que 0.0001. Añadir dígitos de precisión ocasiona que el intervalo modelo (y, por consiguiente, el error absoluto máximo) sea más pequeño.

El intervalo modelo no es un número fijo; varía con el exponente. Para ver por qué varía el intervalo, considere que el intervalo entre 3021.0 y 3022.0 es 1.0, que es 10^4 veces más grande que el intervalo entre 0.3021 y 0.3022. Esto tiene sentido porque 3021.0 es simplemente 0.3021 multiplicado por 10^4. Así, un cambio en el exponente de los números modelo adyacentes para el intervalo tiene un efecto equivalente en el tamaño del intervalo. En términos prácticos, esto significa que se ceden dígitos significativos en la parte fraccionaria para representar números con partes enteras grandes. Esto se ilustra en la figura 10-5, donde se grafican los números modelo listados en la tabla anterior.

También se puede usar el error relativo y absoluto para medir el error de redondeo que resulta de los cálculos. Por ejemplo, suponga que se multiplica 1.0005 por 1 000. El resultado correcto es 1 000.5, pero debido al error de redondeo, la computadora de cuatro dígitos produce 1 000.0 como su resultado. El error absoluto del resultado calculado es 0.5 y el error relativo es 0.05%. Ahora suponga que se multiplica 100 050.0 por 1 000. El resultado correcto es 100 050 000, pero la compu-

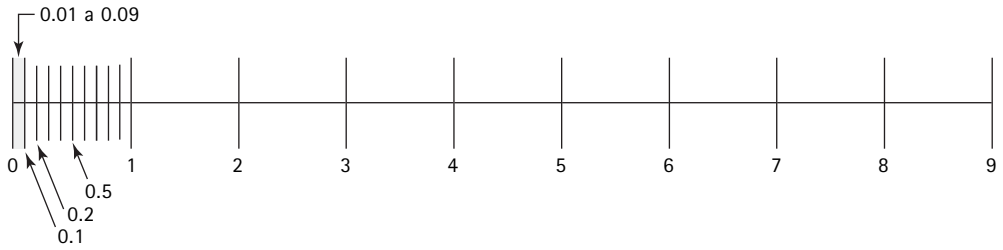


Figura 10–5 *Representación gráfica de números modelo*

tadora produce 100 000 000 como resultado. Si se examina el error relativo, aún es un modesto 0.05%, pero el error absoluto ha crecido a 50 000. Observe que este ejemplo es otro caso de cambio de tamaño del intervalo modelo.

Si es más importante considerar el error absoluto o el error relativo depende de la situación. Es inaceptable que en la auditoría de una compañía se descubra un error de contabilidad de $50 000; el hecho de que el error relativo sea sólo 0.05% no es importante. Por otro lado, un error relativo de 0.05% es inaceptable para representar fechas prehistóricas porque el error en las técnicas de medición se incrementa con la época. Es decir, si se habla de una fecha de hace casi 10 000 años, un error absoluto de 5 años es aceptable; si la fecha es de hace 100 000 000, un error absoluto de 50 000 años es igualmente aceptable.

Comparación de números de punto flotante Se se recomendó cautela respecto a comparar la igualdad exacta de los números de punto flotante. La exploración de errores representativos en este capítulo revela por qué los cálculos no pueden producir los resultados esperados aunque parezca que así debe ser. En el capítulo 5 se escribió una expresión que compara la igualdad de dos variables de punto flotante r y s mediante la función valor absoluto de punto flotante fabs:

```
fabs(r - s) < 0.00001
```

Del análisis del modelo de números modelo, ahora es posible reconocer que la constante 0.00001 en esta expresión representa un error absoluto máximo. Se puede generalizar esta expresión como

```
fabs(r - s) < ERROR_TERM
```

donde `ERROR_TERM` es un valor que se debe determinar para cada problema de programación.

¿Qué sucede si se desea comparar números de punto flotante con una medida de error relativo? Se debe multiplicar el término de error por el valor en el problema en el que el error sea relativo. Por ejemplo, si se desea probar si `r` y `s` son "iguales" dentro de 0.05% de `s`, se escribe la siguiente expresión:

```
fabs(r - s) < 0.0005 * s
```

Recuerde que la elección del error aceptable, y si debe ser absoluto o relativo, depende del problema en cuestión. Los términos de error que se han mostrado en las expresiones de ejemplo son por completo arbitrarias y podrían no ser apropiadas para la mayoría de los problemas. Al resolver un problema relacionado con la comparación de números de punto flotante, por lo común se desea un término de error lo más pequeño posible. En ocasiones la elección se especifica en la descripción de problema o es razonablemente obvia. Algunos casos requieren análisis cuidadoso de las matemáticas del problema y los límites representativos de la computadora en particular. Esta clase de análisis son el dominio de una rama de las matemáticas de nombre *análisis numérico* y están fuera del alcance de este libro.

Subdesbordamiento y desbordamiento Además de los errores representacionales o de representación, hay otros dos problemas a observar en la aritmética de punto flotante. Considérese un cálculo relacionado con números pequeños:

$$\begin{array}{r} 4210 \times 10^{-8} \\ \times\ 2000 \times 10^{-8} \\ \hline 8420000 \times 10^{-16} \end{array} = 8420 \times 10^{-13}$$

Este valor no se puede representar en nuestro esquema debido a que el exponente –13 es muy pequeño. El mínimo es –9. Una manera de resolver el problema es fijar el resultado del cálculo en 0.0. Es obvio que cualquier respuesta que dependa de este cálculo no será exacta.

El desbordamiento es un problema más grave debido a que no existe recurso lógico cuando ocurre. Por ejemplo, el resultado del cálculo

$$\begin{array}{r} 9999 \times 10^{9} \\ \times\ 1000 \times 10^{9} \\ \hline 9999000 \times 10^{18} \end{array} = 9999 \times 10^{21}$$

no se puede guardar; entonces, ¿qué se debe hacer? Para ser congruentes con la respuesta para el subdesbordamiento, se podría fijar el resultado en 9999×10^{9} (el valor representable máximo en este caso). No obstante, esto parece intuitivamente equivocado. La opción es detener con un mensaje de error.

C++ no define lo que debe suceder en caso de desbordamiento o subdesbordamiento. Las diferentes ejecuciones de C++ resuelven el problema de distintas maneras. Usted podría causar intencionalmente un desbordamiento con su sistema y ver qué sucede. Algunos sistemas imprimen un mensaje de error en tiempo de ejecución, como "FLOATING POINT OVERFLOW". En otros sistemas, se podría obtener el número más grande que es posible representar.

Aunque se están analizando problemas con números de punto flotante, los números enteros también pueden causar desbordamiento negativo o positivo. La mayoría de las ejecuciones de C++ ignoran el desbordamiento de enteros. Para ver cómo controla la situación su sistema, debe intentar sumar 1 a `INT_MAX` y –1 a `INT_MIN`. En la mayoría de los sistemas, sumar 1 a `INT_MAX` fija el resultado para `INT_MIN`, un número negativo.

En ocasiones se puede evitar el desbordamiento si los cálculos se ordenan cuidadosamente. Suponga que desea saber cuántas manos distintas de póquer de cinco cartas se pueden repartir con una baraja. Lo que se busca es el número de *combinaciones* de 52 cartas tomadas 5 a la vez. La fórmula matemática estándar para el número de combinaciones de n cosas tomadas r a la vez es

$$\frac{n!}{r!(n-r)!}$$

Se podría usar la función `Factorial` que se mostró en el capítulo 8 y escribir esta fórmula en una sentencia de asignación:

```
hands = Factorial(52) / (Factorial(5) * Factorial(47));
```

El único problema es que 52! es un número muy grande (alrededor de 8.0658×10^{67}). Y 47! también es grande (casi 2.5862×10^{59}). Ambos números están más allá de la capacidad de la mayoría de los sistemas para representarlos como enteros (52! requiere 68 dígitos de precisión). Aunque sea posible representar en muchas máquinas como números de punto flotante, aún se pierde la mayor parte de la precisión. Sin embargo, si se vuelven a ordenar los cálculos, se puede lograr un resultado exacto en cualquier sistema con 9 o más dígitos de precisión. ¿Cómo? Considere que la mayoría de las multiplicaciones para calcular 52! se cancelan cuando el producto se divide entre 47!

$$\frac{52!}{5!\times 47!} = \frac{52\times 51\times 50\times 49\times 48\times 47\times 46\times 45\times 44\times \ldots}{(5\times 4\times 3\times 2\times 1)\times (47\times 46\times 45\times 44\times \ldots)}$$

Así, sólo se tiene que calcular

```
hands = 52 * 51 * 50 * 49 * 48 / Factorial(5);
```

lo que significa que el numerador es 311 857 200 y el denominador es 120. En un sistema con nueve o más dígitos de precisión, se obtiene una respuesta exacta: 2 598 960 manos de póquer.

Error de cancelación Otro tipo de error que puede suceder con números de punto flotante se denomina *error de cancelación*, una forma de error de representación que ocurre cuando se suman o restan números de magnitudes ampliamente distintas. Examínese un ejemplo:

$$(1 + 0.00001234 - 1) = 0.00001234$$

Las leyes de la aritmética establecen que esta ecuación debe ser verdadera. Pero, ¿es verdadera si la computadora hace la aritmética?

$$\begin{array}{r} 100000000 \times 10^{-8} \\ +\quad 1234 \times 10^{-8} \\ \hline 100001234 \times 10^{-8} \end{array}$$

Para cuatro dígitos, la suma es 1000×10^{-3}. Ahora la computadora resta 1:

$$\begin{array}{r} 1000 \times 10^{-3} \\ -1000 \times 10^{-3} \\ \hline 0 \end{array}$$

El resultado es 0, no .00001234.

En ocasiones se puede evitar la suma de dos números de punto flotante que sean muy diferentes en tamaño si los cálculos se ordenan de manera correcta. Suponga que un problema requiere que se sumen muchos números de punto flotante pequeños a un número grande de punto flotante. El resultado es más exacto si el programa suma primero los números más pequeños para obtener un número más grande, y luego agrega la suma al número grande.

Información básica

Implicaciones prácticas de precisión limitada

Un análisis de los errores de representación, de desbordamiento, subdesbordamiento y cancelación puede parecer meramente académico. De hecho, estos errores tienen implicaciones prácticas importantes en muchos problemas. Esta sección finaliza con tres ejemplos que ilustran cómo la precisión limitada puede tener efectos costosos o incluso desastrosos.

Durante el programa espacial Mercury, varias de las naves espaciales cayeron a una distancia considerable de sus puntos de aterrizaje calculados. Esto retardaba la recuperación de la nave espacial y del astronauta, de modo que ambos corrían cierto peligro. Con el tiempo, se encontró que el problema se debió a una representación imprecisa del periodo de rotación de la Tierra en el programa que calculó el punto de aterrizaje.

(continúa) ▼

Implicaciones prácticas de precisión limitada

Como parte de la construcción de una presa hidroeléctrica, era necesario tender un largo conjunto de cables de alta tensión para enlazar la presa con el punto de distribución de potencia más cercano. Dichos cables tenían que ser de varias millas de longitud y cada uno debía ser una unidad continua. (Debido a la salida de alta potencia desde la presa, no era posible empalmar cables más cortos.) El costo de los cables resultó elevado y tenían que colgar entre los dos puntos. Resultó que eran demasiado cortos, así que fue necesario fabricar otro conjunto. El problema se debió a errores de precisión al calcular la longitud de la curva catenaria (la curva que forma un cable cuando cuelga entre dos puntos).

Una auditoría a un banco encontró una cuenta misteriosa con una gran cantidad de dinero en ella. El seguimiento de la cuenta llevó a un programador sin escrúpulos que había usado precisión limitada en su favor. El banco calculaba el interés en sus cuentas hasta una precisión de un décimo de centavo. Los décimos de centavo no se agregaban a las cuentas de los clientes, de modo que el programador sumaba y depositaba en una cuenta a su nombre los décimos extra de todas las cuentas. Debido a que el banco tenía miles de cuentas, estas pequeñas cantidades sumaban una gran cantidad de dinero. Y debido a que el resto de los programas del banco no usaban tanta precisión en sus cálculos, el esquema pasó inadvertido durante muchos meses.

La moraleja de esta explicación es doble: (1) Los resultados de los cálculos de punto flotante son por lo común imprecisos, y dichos errores pueden tener consecuencias graves, y (2) si usted trabaja con números muy grandes o muy pequeños, necesita más información de la que proporciona este libro y debe consultar uno de análisis numérico.

Consejo práctico de ingeniería de software

Cómo elegir un tipo de datos numéricos

Un primer encuentro con todos los tipos de datos numéricos de C++ podría abrumarlo. Para ayudar a elegir una opción, quizá se sienta tentado a lanzar una moneda al aire. Debe resistir esta tentación, porque cada tipo de datos existe por una razón. Aquí tiene algunas directrices:

1. En general, `int` es preferible.

   Como regla, debe usar tipos de punto flotante sólo cuando sea absolutamente necesario; es decir, cuando en definitiva necesite valores fraccionarios. La aritmética de punto flotante no sólo está sujeta a errores de representación; también es significativamente menor que la aritmética de enteros en la mayoría de las computadoras.

   Para datos enteros ordinarios, use `int` en lugar de `char` o `short`. Es fácil cometer errores de desbordamiento con estos tipos de datos tan pequeños. (Sin embargo, el tipo `char` es apropiado para datos de caracteres.)
2. Use `long` sólo si el intervalo de valores `int` en su máquina es muy restrictivo.

   Comparado con `int`, el tipo `long` requiere más espacio de memoria y tiempo de ejecución.
3. Use `double` y `long double` sólo si necesita números muy grandes o pequeños, o si los valores `float` de su máquina no llevan suficientes dígitos de precisión.

   El costo de usar `double` y `long double` es el mayor espacio de memoria y tiempo de ejecución.

(continúa) ▼

Cómo elegir un tipo de datos numéricos

4. Evite las formas `unsigned` de tipos integrales.

   Dichos tipos son sobre todo para manejar bits dentro de una celda de memoria, un tema que no se aborda en este libro. Usted podría pensar que declarar una variable `unsigned` evita guardar por accidente un número negativo en la variable. Sin embargo, el compilador de C++ no evita que eso suceda. Más adelante, en este capítulo, se explica por qué.

Si sigue estas directrices, encontrará que los tipos simples que usa con más frecuencia son `int` y `float`, junto con `char` para datos de caracteres y `bool` para datos booleanos. Sólo en raras ocasiones necesita las variaciones más largas y más cortas de estos tipos fundamentales.

10.5 Datos definidos por el usuario

El concepto de tipo de datos es fundamental para todos los lenguajes de programación de uso extenso. Una de las ventajas del lenguaje C++ es que permite a los programadores crear nuevos tipos de datos, hechos a la medida para satisfacer las necesidades de un determinado programa. Gran parte del resto de este libro trata de los tipos de datos definidos por el usuario. En esta sección se examina cómo crear los propios tipos de datos.

Sentencia Typedef

La *sentencia Typedef* permite introducir un nuevo nombre para un tipo existente. Su plantilla de sintaxis es

Sentencia Typedef

```
typedef Nombre de tipo existente Nombre de tipo nuevo;
```

Antes de que el tipo de datos `bool` fuera parte del lenguaje de C++, muchos programadores usaban un código como el siguiente para simular un tipo booleano:

```
typedef int Boolean;
const int TRUE = 1;
const int FALSE = 0;
    ⋮
Boolean dataOK;
    ⋮
dataOK = TRUE;
```

En este código, la sentencia Typedef ocasiona que el compilador sustituya la palabra `int` cada vez que aparezca la palabra `Boolean` en el resto del programa.

La sentencia Typedef proporciona una forma muy limitada de definir tipos de datos propios. De hecho, Typedef no crea un nuevo tipo de datos en absoluto: sólo crea un nombre adicional para un tipo de datos existente. Respecto al compilador, el dominio y operaciones del tipo booleano anterior son idénticos al dominio y operaciones del tipo `int`.

A pesar de que Typedef no puede crear un nuevo tipo de datos, es una herramienta valiosa para escribir programas documentados por sí mismos. Antes de que `bool` fuera un tipo integrado, el código de programa que usaban los identificadores `Boolean`, `TRUE` y `FALSE` era más descriptivo que el que usaba `int`, 1 y 0 para operaciones booleanas.

Los nombres de tipos definidos por el usuario obedecen las mismas reglas de alcance que se aplican a los identificadores en general. La mayoría de los tipos, como `Boolean`, se definen de manera global, aunque es razonable definir un nuevo tipo dentro de un subprograma si es el único lugar que usa. Las directrices que determinan dónde se debe definir una constante nombrada se aplican también a los tipos de datos.

Tipos de enumeración

Tipo de enumeración Tipo de datos definido por el usuario cuyo dominio es un conjunto ordenado de valores literales expresados como identificadores.

C++ permite al usuario identificar un nuevo tipo simple al listar (enumerar) los valores literales que constituyen el dominio del tipo. Dichos valores literales deben ser *identificadores*, no números. Los identificadores están separados por comas, y la lista se encuentra entre llaves. Los tipos de datos definidos de este modo se denominan **tipos de enumeración**. Enseguida se presenta un ejemplo:

```
enum Days {SUN, MON, TUE, WED, THU, FRI, SAT};
```

Esta declaración crea un nuevo tipo de datos denominado Days. Mientras que Typedef sólo crea un sinónimo para un tipo existente, un tipo de enumeración como Days es un nuevo tipo y es distinto de cualquier tipo existente.

Enumerador Uno de los valores del dominio de un tipo de enumeración.

Los valores en el tipo `Days` —`SUN`, `MON`, `TUE`, etcétera— se denominan **enumeradores**. Los enumeradores son *ordenados*, en el sentido de que `SUN < MON <TUE... < FRI < SAT`. Aplicar operadores relacionales a enumeradores es como aplicarlos a caracteres. La relación que se prueba es si un enumerador "viene antes" o "viene después" en el orden del tipo de datos.

Ya se vio que la representación interna de una constante `char` es un entero no negativo. Los 128 caracteres ASCII se representan en la memoria como los enteros 0 al 127. Los valores en un tipo de enumeración se representan también internamente como enteros. Por omisión, el primer enumerador tiene el valor entero 0; el segundo tiene el valor 1, etcétera. La declaración del tipo de enumeración `Days` es similar al siguiente conjunto de declaraciones:

```
typedef int Days;
const int SUN = 0;
const int MON = 1;
const int TUE = 2;
  ⋮
const int SAT = 6;
```

Si hay alguna razón para que usted desee representaciones diferentes para enumeradores, puede especificarlas de manera explícita como:

```
enum Days {SUN = 4, MON = 18, TUE = 9, . . .   };
```

Rara vez hay alguna razón para asignar valores específicos a enumeradores. Con el tipo Days, se está interesado en los días de la semana, no en la forma en que la máquina los guarda internamente. Ya no se analiza más esta característica, aunque en ocasiones podrá verla en programas de C++.

Observe el estilo utilizado para escribir con mayúsculas los enumeradores. Como en esencia los enumeradores son constantes nombradas, se escribe en mayúsculas todo el identificador. Esto es una elección de estilo. Muchos programadores de C++ usan letras mayúsculas y minúsculas cuando inventan nombres para los enumeradores.

A continuación se muestra la sintaxis para la declaración de un tipo de enumeración. Es una versión simplificada que se amplía después en el capítulo.

Declaración de enumeración

```
enum Nombre { Enumerador, Enumerador…};
```

Cada enumerador tiene la forma siguiente:

Enumerador

```
Identificador = ConstIntExpression
```

donde la ConstIntExpression opcional es una expresión entera compuesta sólo de constantes literales o nombradas.

Los identificadores usados como enumeradores deben seguir las reglas para cualquier identificador de C++. Por ejemplo,

```
enum Vowel {'A', 'E', 'I', 'O', 'U'};     // Error
```

es indebida porque los elementos no son identificadores. La declaración

```
enum Places {1st, 2nd, 3rd};     // Error
```

es indebida porque los identificadores no pueden comenzar con dígitos. En las declaraciones

```
num Starch {CORN, RICE, POTATO, BEAN};
enum Grain {WHEAT, CORN, RYE, BARLEY, SORGHUM};     // Error
```

el tipo `Starch` y el tipo `Grain` se permiten individualmente, pero no juntos. Los identificadores con el mismo alcance deben ser únicos. CORN no se puede definir dos veces.

Suponga que está escribiendo un programa para una clínica veterinaria. El programa debe seguir la pista de diferentes clases de animales. El siguiente tipo de enumeración se podría usar para dicho propósito.

Identificador de tipo ↓ — Valores literales en el dominio

```
enum Animals {RODENT, CAT, DOG, BIRD, REPTILE, HORSE, BOVINE, SHEEP};

Animals inPatient;
Animals outPatient;
```

Creación de dos variables de tipo `Animals`

`RODENT` es una literal, uno de los valores en el tipo de datos `Animals`. Debe quedar claro que `RODENT` no es un nombre de variable; es uno de los valores que se pueden guardar en las variables `inPatient` y `outPatient`. Considérense las clases de operaciones que tal vez habría que efectuar en variables de tipos de enumeración.

Asignación La sentencia de asignación

```
inPatient = DOG;
```

no asigna a `inPatient` la cadena de caracteres “DOG”, tampoco el contenido de una variable de nombre DOG. Asigna el *valor* `DOG`, que es uno de los valores en el dominio del tipo de datos `Animals`.

La asignación es una operación válida, siempre y cuando el valor que se guarde sea de tipo `Animals`. Las dos sentencias siguientes

```
inPatient = DOG;
outPatient = inPatient;
```

son aceptables. Cada expresión del lado derecho es de tipo `Animals`; `DOG` es una literal de tipo `Animals` e `inPatient` es una variable de tipo `Animals`. Aunque se sabe que la representación subyacente de `DOG` es el entero 2, el compilador advierte del uso de esta asignación:

```
inPatient = 2;   // Not allowed
```

Aquí tiene la regla precisa:

> *La coerción implícita de tipos se define a partir de un tipo de enumeración a un tipo integral, pero no de un tipo integral a un tipo de enumeración.*

Al aplicar esta regla a las sentencias

```
someInt = DOG;   // Válido
inPatient = 2;   // Error
```

se ve que la primera sentencia almacena el número 2 en `someInt` (como resultado de la coerción implícita de tipos), pero la segunda produce un error en tiempo de compilación. La restricción de almacenar un valor entero en una variable de tipo `Animals` es evitar que de modo accidental guarde un valor que esté fuera del intervalo:

```
inPatient = 65;   // Error
```

Incremento Suponga que desea "incrementar" el valor en `inPatient` de modo que se convierta en el siguiente valor en el dominio:

```
inPatient = inPatient + 1;   // Error
```

Esta sentencia es indebida por la siguiente razón. El lado derecho es correcto porque la coerción implícita de tipos le permite agregar `inPatient` a 1; el resultado es un valor `int`. Pero la operación de asignación no es válida porque no se puede almacenar un valor `int` en `inPatient`. La sentencia

```
inPatient++;   // Error
```

también es inválida porque el compilador considera que tiene la misma semántica que la sentencia de asignación anterior. Sin embargo, se puede evitar la regla de coerción de tipos si usa una conversión de tipo *explícito*, un moldeo de tipos, como sigue:

```
inPatient = Animals(inPatient + 1);   // Correct
```

Cuando usa el moldeo de tipos, el compilador supone que usted sabe lo que hace y lo permite.

Incrementar una variable de un tipo de enumeración es muy útil en ciclos. En ocasiones es necesario un ciclo que procese todos los valores en el dominio del tipo. Se podría intentar el siguiente ciclo For:

```
Animals patient;

for (patient=RODENT; patient <= SHEEP; patient++)  // Error
    ⋮
```

Sin embargo, como se explicó antes, el compilador se quejará respecto a la expresión patient++. Para incrementar patient, se debe usar una expresión de asignación y un moldeo de tipos:

```
for (patient=RODENT; patient <= SHEEP; patient=Animals(patient + 1))
    ⋮
```

La única precaución aquí es que cuando el control termina el ciclo, el valor de `patient` es 1 *mayor que* el valor más grande en el dominio (SHEEP). Si quiere usar `patient` fuera del ciclo, debe reasignarle un valor que esté dentro del intervalo apropiado para el tipo `Animals`.

Comparación La operación más común efectuada en valores de tipos de enumeración es la comparación. Cuando usted compara dos valores, su orden se determina mediante el orden en el cual lista los enumeradores en la declaración de tipo. Por ejemplo, la expresión

```
inPatient <= BIRD
```

tiene el valor `true` si `inPatient` contiene el valor `RODENT`, `CAT`, `DOG` o `BIRD`.

También es posible usar valores de un tipo de enumeración en una sentencia Switch. Debido a que `RODENT`, `CAT`, etcétera, son literales, pueden aparecer en etiquetas de caso:

```
switch (inPatient)
{
    case RODENT  :
    case CAT     :
    case DOG     :
    case BIRD    : cout << "Jaula";
                   break;
    case REPTILE : cout << "Terrario";
                   break;
    case HORSE   :
    case BOVINE  :
    case SHEEP   : cout << "Establo";
}
```

Entrada y salida El flujo I/O se define sólo para los tipos integrados básicos (`int`, `float`, etcétera), no para tipos de enumeración definidos por el usuario. Los valores de tipos de enumeración se deben introducir o sacar de manera indirecta.

Para introducir valores, una estrategia es leer una cadena que deletree una de las constantes en el tipo de enumeración. La idea es introducir la cadena y traducirla en una de las literales en el tipo de enumeración al examinar sólo las letras que se requieran para determinar lo que es.

Por ejemplo, el programa de clínica veterinaria podría leer el tipo de animal como una cadena; luego, asignar uno de los valores de tipo `Animals` a ese paciente. *Cat, dog, horse* y *sheep* se pueden determinar por su primera letra. *Bovine, bird, rodent* y *reptile* no se pueden determinar hasta que se examina la segunda letra. El siguiente fragmento de programa lee en una cadena que representa un nombre de animal y lo convierte en uno de los valores en el tipo `Animals`.

```
#include <cctype>    // Para toupper()
#include <string>    // Para tipo string
    :
string animalName;
    :
cin >> animalName;
switch (toupper(animalName[0]))
{
    case 'R' : if (toupper(animalName[1]) == 'O')
                   inPatient = RODENT;
               else
                   inPatient = REPTILE;
               break;
    case 'C' : inPatient = CAT;
               break;
```

```
    case 'D' : inPatient = DOG;
               break;
    case 'B' : if (toupper(animalName[1]) == 'I')
                   inPatient = BIRD;
               else
                   inPatient = BOVINE;
               break;
    case 'H' : inPatient = HORSE;
               break;
    default :  inPatient = SHEEP;
}
```

Los valores de tipo de enumeración no se pueden imprimir de manera directa. La impresión se hace por medio de una sentencia Switch que imprime una cadena de caracteres que corresponde al valor.

```
switch (inPatient)
{
    case RODENT  : cout << "Roedor";
                   break;
    case CAT     : cout << "Gato";
                   break;
    case DOG     : cout << "Perro";
                   break;
    case BIRD    : cout << "Pájaro";
                   break;
    case REPTILE : cout << "Reptil";
                   break;
    case HORSE   : cout << "Caballo";
                   break;
    case BOVINE  : cout << "Bovino";
                   break;
    case SHEEP   : cout << "Oveja";
}
```

Se podría preguntar, ¿por qué no usar sólo un par de letras o un número entero como código para representar a cada animal en un programa? La respuesta es que se usan tipos de enumeración para hacer más legibles los programas; son otra forma de hacer que el código sea más autodocumentado.

Devolución de un valor de función Se han estado usando funciones de devolución de valor para calcular y devolver valores de tipos integrados como `int`, `float` y `char`:

```
int Factorial( int );
float CargoMoment( int );
```

C++ permite que un valor de devolución de función sea de *cualquier* tipo de datos, integrado o definido por el usuario, excepto un arreglo (un tipo de datos que se examinará en capítulos posteriores).

En la última sección se escribe una sentencia Switch para convertir una cadena de entrada en un valor de tipo `Animals`. Se escribirá una función de devolución de valor que efectúa esta tarea. Observe cómo el encabezado de función declara el tipo de datos del valor de retorno como `Animals`.

```
Animals StrToAnimal( /* in */ string str )
{
    switch (toupper(str[0]))
```

```
    {
        case 'R' : if (toupper(str[1]) == 'O')
                       return RODENT;
                   else
                       return REPTILE;
        case 'C' : return CAT;
        case 'D' : return DOG;
        case 'B' : if (toupper(str[1]) == 'I')
                       return BIRD;
                   else
                       return BOVINE;
        case 'H' : return HORSE;
        default :  return SHEEP;
    }
}
```

En esta función, ¿por qué no se incluyó una sentencia Break después de cada alternativa de caso? Porque cuando una de las alternativas ejecuta una sentencia Return, el control termina de inmediato la función. No es posible que el control "pase" a la siguiente alternativa.

A continuación se muestra un código que llama a la función `StrToAnimal`:

```
enum Animals {RODENT, CAT, DOG, BIRD, REPTILE, HORSE, BOVINE, SHEEP};

Animals StrToAnimal( string );
  ⋮
int main()
{
    Animals inPatient;
    Animals outPatient;
    string  inputStr;
     ⋮
    cin >> inputStr;
    inPatient = StrToAnimal(inputStr);
     ⋮
    cin >> inputStr;
    outPatient = StrToAnimal(inputStr);
     ⋮
}
```

Tipos de datos nombrados y anónimos

Los tipos de enumeración que se han examinado, `Animals` y `Days`, se denominan **tipos nombrados** porque sus declaraciones incluyen nombres para los tipos. Las variables de estos nuevos tipos de datos se declaran por separado usando los identificadores de tipo `Animals` y `Days`.

Tipo nombrado Tipo definido por el usuario, cuya declaración incluye un identificador de tipo que da un nombre al tipo.

C++ también permite introducir un nuevo tipo directamente en una declaración de variable. En lugar de las declaraciones

```
enum CoinType {NICKEL, DIME, QUARTER, HALF_DOLLAR};
enum StatusType {OK, OUT_OF_STOCK, BACK_ORDERED};


CoinType   change;
StatusType status;
```

se podría escribir

```
enum {NICKEL, DIME, QUARTER, HALF_DOLLAR} change;
enum {OK, OUT_OF_STOCK, BACK_ORDERED} status;
```

Tipo anónimo Tipo que no tiene un identificador de tipo relacionado.

Un nuevo tipo declarado en una declaración de variable se denomina **tipo anónimo** porque no tiene un nombre, es decir, no tiene un identificador de tipo relacionado con él.

Si es posible crear un tipo de datos en una declaración de variable, ¿por qué preocuparse de una declaración de tipo separada que crea un tipo nombrado? Los tipos nombrados, igual que las constantes nombradas, hacen que un programa sea más legible, entendible y fácil de modificar. Asimismo, declarar un tipo y declarar una variable de ese tipo son dos conceptos distintos; es mejor mantenerlos separados.

Ahora se presenta una plantilla de sintaxis más completa para una declaración de tipo de enumeración. Dicha plantilla muestra que el nombre de tipo es opcional (produce un tipo anónimo) y que se puede incluir opcionalmente una lista de variables en la declaración.

Declaración enum

```
enum Nombre {Enumerador, Enumerador,...} Nombre de variable , Nombre de variable… ;
```

Encabezados de archivo escritos por el usuario

Cuando crea sus propios tipos de datos es común hallar que uno de ellos puede ser útil en más de un programa. Por ejemplo, usted podría estar trabajando en varios programas que requieran un tipo de enumeración que consta de los 12 meses del año. En lugar de escribir la sentencia

```
enum Months
{
    JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE,
    JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER
};
```

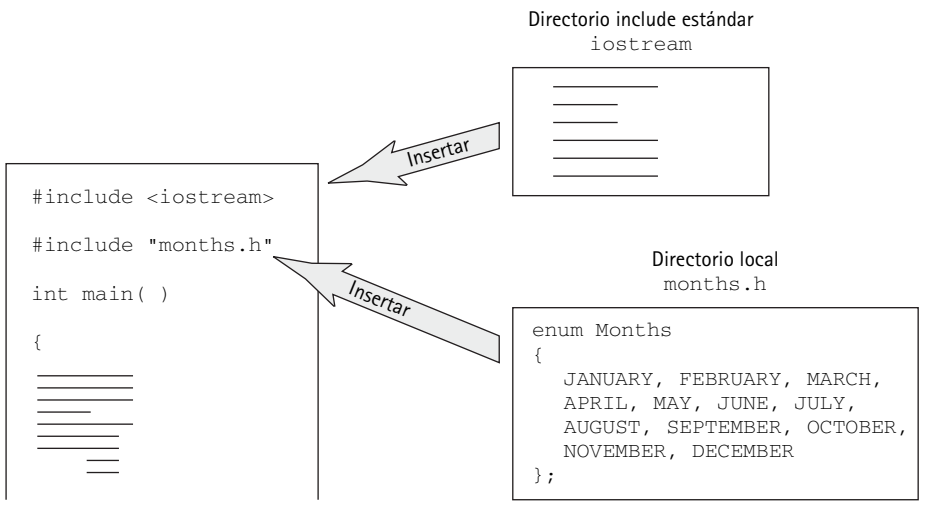


Figura 10-6 *Incluir encabezados de archivo*

al comienzo de cada programa que usa el tipo `Months`, puede colocar esta sentencia en un archivo separado de nombre `months.h`. Luego, usar `months`.h del mismo modo que usa los encabezados de archivo proporcionados por el sistema, como `iostream` y `cmath`. Al usar una directiva `#include` se pide al preprocesador de C++ que inserte físicamente el contenido del archivo en su programa. (Aunque muchos sistemas C++ usan la extensión de nombre de archivo `.h` [o ninguna extensión en absoluto] para denotar archivos de encabezado, otros sistemas emplean extensiones como `.hpp` o `.hxx`.)

Cuando coloca entre paréntesis angulares el nombre de un encabezado de archivo, como en

```
#include <iostream>
```

el preprocesador busca el archivo en el *directorio include* estándar, un directorio que contiene todos los archivos de encabezado proporcionados por el sistema de C++. Por otro lado, puede colocar el nombre de un archivo de encabezado entre comillas, como:

```
#include "months.h"
```

En este caso, el preprocesador busca el archivo en el directorio actual del programador. Este mecanismo permite escribir los propios encabezados de archivo que contienen declaraciones de tipo y declaraciones constantes. Se puede usar una directiva `#include` simple en lugar de volver a escribir las declaraciones en cada programa que las necesite (véase la figura 10-6).

10.6 Más acerca de la coerción de tipos

Según lo aprendido en el transcurso de varios capítulos, C++ realiza la coerción implícita de tipos siempre que se utilicen valores de tipos de datos distintos en lo siguiente:

1. Expresiones aritméticas y relacionales
2. Operaciones de asignación
3. Paso de argumentos
4. Retorno del valor de función desde una función de devolución de valor

Para el elemento 1 —expresiones de tipo mixto—, el compilador de C++ sigue un conjunto de reglas para la coerción de tipos. Para los elementos 2, 3 y 4, el compilador sigue un segundo conjunto de reglas. Se examinará cada una de estas dos reglas.

Coerción de tipos en expresiones aritméticas y relacionales

Promoción (ensanchamiento) Conversión de un valor de un tipo "menor" a uno "mayor" de acuerdo con la precedencia de tipos de datos de un lenguaje de programación.

Suponga que una expresión aritmética consta de un operador y dos operandos; por ejemplo, `3.4*sum` o `var1/var2`. Si los dos operandos son de tipos de datos distintos, entonces uno de ellos es **promovido** (o **ensanchado**) temporalmente para que coincida con el tipo de datos del otro. Para entender de manera exacta qué significa promoción, se examinará la regla para coerción de tipos en una expresión aritmética.*

Paso 1: Cada valor `char`, `short`, `bool` o de enumeración es promovido (ensanchado) a `int`. Si ambos operandos son ahora `int`, el resultado es una expresión `int`.

Paso 2: Si el paso 1 aún deja una expresión de tipo mixto, se usa la siguiente precedencia de tipos:

mínimo → máximo

* La regla que se da para la coerción de tipos es una versión simplificada de la regla hallada en la definición del lenguaje C++. La regla completa tiene más qué decir acerca de tipos sin signo, que rara vez se usan en este libro.

```
int, unsigned int, long, unsigned long, float, double, long double
```

El valor del operando del tipo "menor" es promovido al del tipo "mayor", y el resultado es una expresión de ese tipo.

Un ejemplo simple es la expresión `someFloat+2`. Esta expresión no tiene valores `char`, `short`, `bool` o de enumeración, así que el paso 1 deja una expresión de tipo mixto. En el paso 2, `int` es un tipo "menor" que `float`, así que el valor 2 es coercionado temporalmente al valor `float`, es decir, 2.0. Entonces tiene lugar la adición, y el tipo de la expresión completa es `float`.

Esta descripción de coerción de tipo se cumple también para expresiones relacionales como

```
someInt <= someFloat
```

El valor de `someInt` se coerciona temporalmente a representación de punto flotante antes de que tenga lugar la comparación. La única diferencia entre expresiones aritméticas y relacionales es que el tipo resultante de una expresión relacional es siempre `bool`; el valor `true` o `false`.

Enseguida se proporciona una tabla que describe el resultado de promover un valor de un tipo simple a otro en C++:

De	A	Resultado de la promoción
`double`	`long double`	Mismo valor, que ocupa más espacio de memoria
`float`	`double`	Mismo valor, que ocupa más espacio de memoria
Tipo integral	Tipo de punto flotante	Equivalente de punto flotante del valor entero; la parte fraccionaria es cero
Tipo integral	Su contraparte `unsigned`	Mismo valor, si el número original es no negativo; un número positivo radicalmente distinto, si el número original es negativo
Tipo integral con signo tipo integral `unsigned`	Tipo integral con signo más grande Tipo integral más grande (ya sea con signo o sin signo)	Mismo valor, que ocupa más espacio de memoria Mismo valor no negativo, que ocupa más espacio de memoria

NOTA: el resultado de promover un `char` a un `int` depende del compilador. Algunos compiladores tratan a `char` como `unsigned char`, así que la promoción produce siempre un entero no negativo. Con otros compiladores, `char` significa signed `char`, así que la promoción de un valor negativo produce un entero negativo.

La nota al pie de la tabla sugiere un problema potencial si pretende escribir un programa de C++ portable. Si usa el tipo `char` sólo para almacenar datos de caracteres, no hay problema. C++ garantiza que cada carácter en el conjunto de caracteres de la máquina (como ASCII) está representado como un valor no negativo. Al usar datos de caracteres, la promoción de `char` a `int` proporciona el mismo resultado en cualquier máquina con cualquier compilador.

Pero si se pretende ahorrar memoria usando el tipo `char` para manipular enteros con signos pequeños, entonces la promoción de dichos valores al tipo `int` puede producir diferentes resultados en máquinas distintas. Es decir, una máquina puede promover valores `char` negativos a valores `int` negativos, mientras que el mismo programa en otra máquina podría promover valores `char` negativos a valores `int` *positivos*. La moraleja es: a menos que necesite hasta el último espacio de memoria, no use `char` para manejar números pequeños con signo. Use `char` sólo para almacenar datos de caracteres.

Coerción de tipos en asignaciones, paso de argumentos y retorno de una función de valor

En general, la promoción de un valor de un tipo a otro no ocasiona pérdida de información. Piense en la promoción como mover sus tarjetas de beisbol de una caja de zapatos a una caja de zapatos más

grande. Todas las tarjetas aún caben en la nueva caja y hay espacio de sobra. Por otro lado, la **degradación** (o **estrechamiento**) de valores de datos puede causar pérdida de información. La degradación es como mover una caja de zapatos llena de tarjetas de beisbol a una caja más pequeña; algo se tiene que desechar.

Degradación (estrechamiento) Conversión de un valor de un tipo "superior" a un tipo "menor" de acuerdo con la precedencia del tipo de datos de un lenguaje de programación. La degradación puede causar pérdida de información.

Considere una operación de asignación

$$v = e$$

donde v es una variable y e es una expresión. En relación con los tipos de datos de v y e, aquí tiene las tres posibilidades

1. Si los tipos v y e son lo mismo, no se requiere ningún tipo de coerción.
2. Si el tipo de v es "mayor" que el de e (usando la precedencia de tipos explicada con la promoción), entonces el valor de e se promueve al tipo de v antes de que se almacene en v.
3. Si el tipo de v es "menor" que el de e, el valor de e se degrada al tipo de v antes que se almacene en v.

La degradación, que se puede considerar como la disminución de un valor, puede causar pérdida de información:

- La degradación de un tipo integral más grande a un tipo integral más corto (como `long` a `int`) da como resultado la eliminación de los bits de la izquierda (más significativos) en la representación de números binarios. El resultado puede ser un número drásticamente distinto.
- La degradación de un tipo de punto flotante a un tipo integral causa el truncamiento de la parte fraccionaria (y un resultado indefinido si la parte de números enteros no encajara en la variable de destino). El resultado de truncar un número negativo depende de la máquina.
- La degradación de un tipo de punto flotante más grande a un tipo de punto flotante más pequeño (como `double` a `float`) podría dar como resultado pérdida de dígitos de precisión.

La descripción de coerción de tipos en una operación de asignación se cumple también para el paso de argumentos (la localización de argumentos en parámetros) y para la devolución de un valor de función con una sentencia Return. Por ejemplo, suponga que `INT_MAX` en su máquina es 32767 y que tiene la siguiente función:

```
void DoSomething( int n )
{
    ⋮
}
```

Si se llama a la función con una sentencia

```
DoSomething(50000);
```

entonces el valor 50000 (que es implícitamente de tipo `long` porque es más grande que `INT_MAX`) se degrada a un valor más pequeño distinto que cabe en una ubicación `int`. De manera similar, la Implementación de la función

```
int SomeFunc( float x )
{
     ⋮
    return 70000;
}
```

causa la degradación del valor 70000 a un valor `int` más pequeño porque `int` es el tipo declarado del valor de devolución de función.

Una consecuencia interesante de la coerción implícita de tipos es la inutilidad de declarar una variable como `unsigned`, con la esperanza de que el compilador evite que usted cometa un error como:

```
unsignedVar = -5;
```

El compilador no se queja en absoluto. Genera un código para coercionar el tipo `int` a un valor `unsigned int`. Si usted ahora imprime el valor de `unsignedVar`, verá un entero positivo de aspecto extraño. Como se indicó, los tipos `unsigned` son más apropiados para técnicas avanzadas que manipulan bits individuales dentro de celdas de memoria. Es mejor evitar usar `unsigned` para cálculos numéricos ordinarios.

Caso práctico de resolución de problemas

Análisis estadístico de texto

PROBLEMA El problema del tío rico del capítulo 9 y los ejercicios de Seguimiento de caso práctico lo intrigaron, así que decide cambiar y mejorar el programa. En lugar de calcular porcentajes de grupos de caracteres, sólo mostrará las cuentas. Asimismo, determinará la longitud de palabra promedio y la longitud de sentencia promedio. Como acaba de aprender acerca de los tipos enumerados y las sentencias Switch, decide volver a hacer el diseño con estas construcciones.

ANÁLISIS Las respuestas del ejercicio de Seguimiento de caso práctico del capítulo 9 hacen pensar que el número de líneas nuevas, signos de puntuación y espacios dan una buena aproximación del número de palabras. Sin embargo, si alguno de estos caracteres aparece consecutivamente, sólo el primero se debe considerar como símbolo de fin de palabra. Es posible usar una variable booleana de fin de palabra (endOfWord) que se fija en verdadero cuando se encuentra un símbolo de fin de palabra. El contador de palabras se debe incrementar cuando endOfWord es falso, después de que endOfWord se fijó en verdadero. Cuando se lee un carácter alfanumérico, endOfWord se fija en falso.

ENTRADA Texto en el archivo cuyo nombre se lee desde el teclado.

SALIDA Una tabla proporcionada por el archivo cuyo nombre se lee desde el teclado y muestra los siguientes valores.

Número total de caracteres alfanuméricos
Número de letras mayúsculas
Número de letras minúsculas
Número de dígitos
Número de caracteres ignorados
Número de palabras
Número de oraciones
Longitud de palabra promedio
Longitud de oración promedio

Principal **Nivel 0**

```
Abrir archivos para proceso
    Si los archivos no se abren de manera correcta
    Escribir un mensaje de error
    devolver 1
Obtener un carácter
DO
    Procesar carácter (carácter)
    Obtener un carácter
WHILE (más datos)
Imprimir la tabla
```

El módulo Abrir archivos en el programa Tío rico se puede usar de modo directo. De hecho, sólo es necesario descomponer el módulo Procesar Carácter, aunque el módulo Imprimir tabla se modificará un poco.

PROCESAR CARÁCTER En el programa Tío rico se usó una sentencia If para determinar a qué categoría pertenece un carácter. En este programa se usará una sentencia Switch con etiquetas de caso de un tipo enumerado. Las categorías son mayúsculas, minúsculas, dígitos, fin de palabra, fin de sentencia e ignorar.

```
enum Features {UPPER, LOWER, DIGIT, EOW, EOS, IGNORE};
```

Este módulo es donde se debe establecer el cambio de fin de palabra (endOfWord). Se debe establecer en falso cuando se declara y en verdadero cuando se encuentra un símbolo de fin de palabra, y restablecer en falso cuando se encuentra un carácter alfanumérico. Para que este proceso funcione de manera adecuada, endOfWord se debe marcar como una variable estática. Una variable estática es una variable local que mantiene su valor de una invocación a otra.

(Entrada: carácter; entrada-salida: contador de mayúsculas, contador de minúsculas, contador de dígitos, contador de palabras, contador de sentencias, contador de ignorar) **Nivel 1**

```
Establecer endOfWord (estática) en falso

SWITCH (Decodificar (carácter))
   UPPER:    Incrementar el contador de mayúsculas (uppercaseCounter)
             establecer el fin de palabra (endOfWord) en falso
   LOWER:    Incrementar el contador de minúsculas (lowercaseCounter)
             Fijar endOfWord en falso
   DIGIT:    Incrementar el contador de dígitos (digitCounter)
             Fijar endOfWord en falso
   EOW:      SI NO es el fin de palabra
                incrementar el contador de palabras
                Fijar endOfWord en verdadero;
   EOS:      incrementar el contador de sentencia
   IGNORE:   incrementar el contador de ignorar (ignoreCounter)
```

Decodificar (entrada: carácter) **Nivel 2**
Salida: valor de función; característica

```
IF isupper(carácter))
    devolver UPPER
ELSE IF islower(carácter)
    devolver LOWER
ELSE IF isdigit(carácter)
    devolver DIGIT
ELSE
    SWITCH(carácter)
        '.':
        '?':
        '!': devolver EOS
        ' ':
        ',':
        ';':
        ':':
        '\n': devolver EOW;
    devolver IGNORE;
```

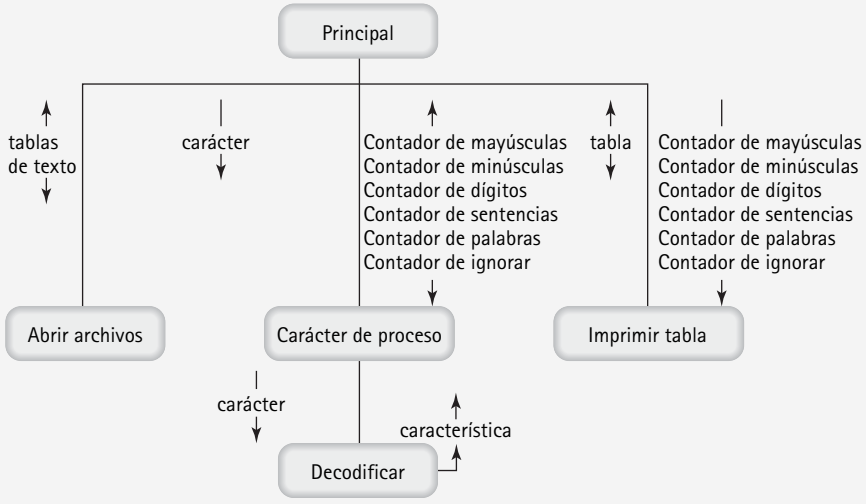
Observe que es posible usar una sentencia de cambio (Switch) en la última rama else porque los caracteres se pueden usar como etiquetas de caso. Si se hace coincidir cualquiera de las dos primeras, el control fluye hacia el tercer caso, que tiene un `return` junto a ella que ocasiona que la ejecución salte hasta el final de la sentencia Switch. Lo mismo aplica para las etiquetas de caso cuarta a séptima, que fluyen hacia la última etiqueta de caso, la cual tiene junto a ella un `return`.

A medida que examine este algoritmo comprenderá que los marcadores de fin de sentencia son también marcadores de fin de palabra. Sin embargo, usted también quiere mantener las cuentas separadas, así que decide atender este problema en el módulo Imprimir resultados sumando el número de sentencias al número de palabras.

Imprimir tabla (entrada-salida: tabla, contador de mayúsculas, contador de minúsculas, contador de dígitos, contador de sentencias, contador de palabras, contador de ignorar) **Nivel 1**

```
Fijar el número total de caracteres alfanuméricos (totalAlphaNum) en contador de
    mayúsculas + contador de minúsculas + contador de dígitos

Imprimir en la tabla "Número total de caracteres alfanuméricos:"
    totalAlphaNum
Imprimir en la tabla: "Número de letras mayúsculas:" uppercaseCounter
Imprimir en la tabla "Número de letras minúsculas:" lowercaseCounter
Imprimir en la tabla "Número de dígitos:" digitCounter
Imprimir en la tabla "Número de caracteres ignorados:" ignoreCounter


Fijar el contador de palabras en contador de palabras + contador de sentencias

Imprimir en la tabla "Número de palabras:" wordCounter
Imprimir en la tabla "Número de sentencias:" sentenceCounter
Imprimir en la tabla "Longitud de palabra promedio:" float(totalAlphaNum)/
    wordCounter
Imprimir en la tabla "Longitud de sentencia promedio:"
        float(wordCounter)/sentenceCounter
```

GRÁFICA DE ESTRUCTURA DE MÓDULOS




```
//**************************************************************************
// Programa Estilo
// Calcular un análisis estilístico de las siguientes características
// de texto:
//   número de palabras
//   longitud de palabra promedio
//   número de sentencias
//   longitud de sentencia promedio
//   número de letras mayúsculas
//   número de letras minúsculas
//   número de dígitos
// Para ahorrar espacio, se omiten de cada función los comentarios
// de precondición que documentan las suposiciones hechas acerca
// de los datos de parámetros de entrada válidos. Éstos se incluirían
// en un programa dedicado a uso real
//**************************************************************************

#include <fstream>
#include <iostream>
#include <iomanip>
#include <cctype>

using namespace std;

enum Features {UPPER, LOWER, DIGIT, IGNORE, EOW, EOS};

// Prototipos de función
void OpenFiles(ifstream&, ofstream&);
Features Decode(char character);
void ProcessCharacter(char, int&, int&, int&, int&, int&, int&);
void PrintTable(ofstream& table, int, int, int, int, int, int);

int main()
{
```

```
    // Preparar archivos para lectura y escritura
    ifstream text;
    ofstream table;
    OpenFiles(text, table);
    if (!text || !table )
    {
        cout << "Los archivos no se abrieron con éxito." << endl;
        return 1;
    }

    char character;                 // Carácter de entrada

    // Declarar e inicializar contadores
    int uppercaseCounter = 0;
    int lowercaseCounter = 0;
    int digitCounter = 0;
    int wordCounter = 0;
    int sentenceCounter = 0;
    int ignoreCounter = 0;


    text.get(character);            // Introducir un carácter
    do
    { // Procesar cada carácter
        ProcessCharacter(character, uppercaseCounter,
            lowercaseCounter, digitCounter, sentenceCounter,
            wordCounter, ignoreCounter);
        text.get(character);        // Introducir un carácter
    } while (text);

    PrintTable(table, uppercaseCounter, lowercaseCounter,
        digitCounter, sentenceCounter, wordCounter, ignoreCounter);
    text.close();
    table.close();
    return 0;
}

//*****************************************************************************

Features Decode( /* in */ char character ) // Carácter decodificado

// La función Decode examina el carácter y devuelve su tipo
// Poscondición:
//     El valor de retorno es el tipo enumerado al que pertenece
//     el carácter

{
    if (isupper(character))
        return UPPER;
    else if (islower(character))
        return LOWER;
    else if (isdigit(character))
        return DIGIT;
```

```
    else
        switch  (character)
        {
            case '.'  :
            case '?'  :
            case '!'  :  return EOS;

            case ' '  :
            case ','  :
            case ';'  :
            case ':'  :
            case '\n' :  return EOW;

        }
    return IGNORE;
}

//*********************************************************************

void OpenFiles( /* inout */ ifstream& text,   // Archivo de entrada
                /* inout */ ofstream& table ) // Archivo de salida

// La función OpenFiles lee los nombres del archivo de entrada
// y el archivo de salida y los abre para procesamiento; el nombre
// del archivo de entrada se escribe en el archivo de salida
// Poscondición:
//     Se han abierto los archivos y se ha escrito el nombre del archivo
//     de entrada en el archivo de salida

{
    string inFileName;
    string outFileName;
    cout << "Introduzca el nombre del archivo que será procesado" << endl;
    cin >> inFileName;
    text.open(inFileName.c_str());
    cout << "Introduzca el nombre del archivo de salida" << endl;
    cin >> outFileName;
    table.open(outFileName.c_str());
    table << "Análisis de caracteres en el archivo de entrada" << inFileName
          << endl << endl;
}

//*********************************************************************

void PrintTable
    ( /* inout */ ofstream& table,      // Archivo de salida
      /* in */ int uppercaseCounter,    // Letras mayúsculas
      /* in */ int lowercaseCounter,    // Letras minúsculas
      /* in */ int digitCounter,        // Dígitos
      /* in */ int sentenceCounter,     // '.', '?', '!'
      /* in */ int wordCounter,         // Palabras
      /* in */ int ignoreCounter )      // Todo lo demás
```

```
// La función PrintTable imprime los porcentajes representados por
// cada una de las cinco categorías
// Poscondición:
//     El resultado se ha escrito en la tabla de archivos, marcados
//     de manera apropiada

{
    int totalAlphaNum;
    totalAlphaNum = uppercaseCounter + lowercaseCounter
        + digitCounter;

    // Imprimir los resultados en la tabla de archivos
    table << "Número total de caracteres alfanuméricos: "
          << totalAlphaNum << endl;
    table << "Número de letras mayúsculas: " << uppercaseCounter
          << endl;
    table << "Número de letras minúsculas: " << lowercaseCounter
          << endl;
    table << "Número de dígitos: " << digitCounter << endl;
    table << "Número de caracteres ignorados: " << ignoreCounter
          << endl;

   // Agregar el número de marcadores de fin de sentencia a la cuenta de palabras
    wordCounter = wordCounter + sentenceCounter;

    // Escribir el resto de los resultados en la tabla de archivos
    table << "Número de palabras: " <<  wordCounter << endl;
    table << "Número de sentencias: " << sentenceCounter << endl;
    table << "Longitud de palabra promedio: " << fixed << setprecision(2)
          << float(totalAlphaNum)/ wordCounter  << endl;
    table << "Longitud de sentencia promedio: " << fixed << setprecision(2)
          << float(wordCounter) / sentenceCounter << endl;

}

//*****************************************************************************

void ProcessCharacter
    ( /* in  */   char character,        // Carácter que
                                          // se procesará
      /* inout */ int& uppercaseCounter, // Letras mayúsculas
      /* inout */ int& lowercaseCounter, // Letras minúsculas
      /* inout */ int& digitCounter,     // Dígitos
      /* inout */ int& sentenceCounter,  // '.', '?', '!'
      /* inout */ int& wordCounter,      // Palabras
      /* inout */ int& ignoreCounter )   // Todo lo demás

// La función ProcessCharacter examina el carácter e incrementa
// el contador apropiado.
// Poscondición:
//     Se incrementó la categoría a la que pertenece
//     el carácter
```

```
{
    static bool endOfWord = false;

    switch (Decode(character))
    {
        case UPPER :  uppercaseCounter++;
                      endOfWord = false;
                      break;
        case LOWER :  lowercaseCounter++;
                      endOfWord = false;
                      break;
        case DIGIT :  digitCounter++;
                      endOfWord = false;
                      break;
        case EOW   :  if (!endOfWord)
                      {
                          wordCounter++;
                          endOfWord = true;
                      }
                      break;
        case EOS   :  sentenceCounter++;
                      break;
        case IGNORE:  ignoreCounter++;
                      break;
    }
}
```

PRUEBA Se tomará una muestra de texto, se calcularán a mano las estadísticas y se compararán los resultados con el del programa.

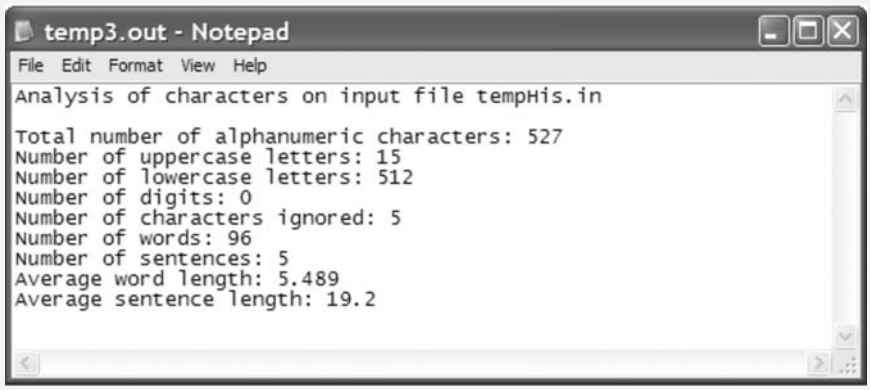
Entrada

```
The Abacus (which appeared in the sixteenth century) was the first
calculator. In the middle of the seventeenth century Blaise Pascal, a French
mathematician, built and sold gear-driven mechanical machines which performed
whole number addition and subtraction.  (Yes, the language Pascal is named
for him.)

Later in the seventeenth century a German mathematician Gottfried Wilhelm von
Leibniz built the first mechanical device designed to do all four whole number
operations: addition, subtraction, multiplication and division.  The state of
mechanical gears and levers at that time was such that the Leibniz machine
was not very reliable.
```

Resultados

Número total de caracteres alfanuméricos:	527
Número de letras mayúsculas:	15
Número de letras minúsculas:	512
Número de dígitos:	0
Número de caracteres ignorados:	5 (dos pares de paréntesis y un guión)
Número de palabras:	96
Número de enunciados:	5
Longitud de palabra promedio:	5.489
Longitud de enunciados promedio:	19.2

Resultado del programa:

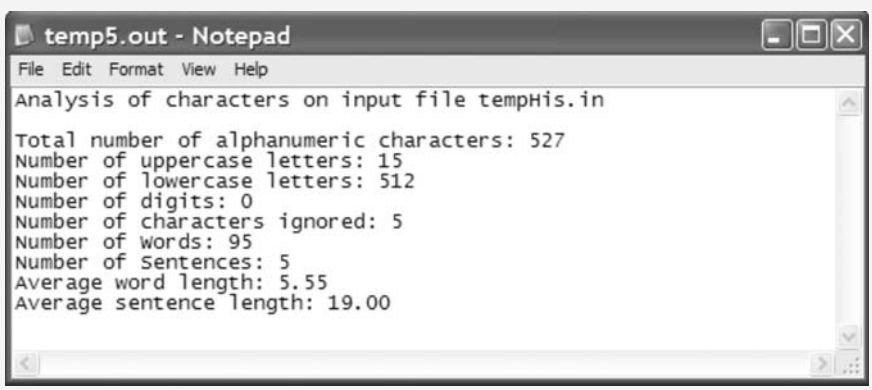


La imagen mostrada corresponde a la salida producida por el programa original, escrito en idioma inglés.

El número de palabras, la longitud de palabra promedio y la longitud de enunciado promedio son erróneos. Vuelva a contar el número de palabras y de nuevo resultan 96. Usted prestó atención al caso donde los marcadores de fin de enunciado terminan las palabras añadiendo un número de enunciados al número de palabras. Pero `endOfWord` no identificó cuando se hallaron los marcadores de fin de sentencia. El problema en la función `ProcessCharacter` se corrige de manera siguiente:

```
case EOS   :  sentenceCounter++;
              endOfWord = true;
              break;
```

Al ejecutar el programa se obtiene el resultado siguiente.



La imagen mostrada corresponde a la salida producida por el programa original, escrito en idioma inglés.

El número de palabras difiere en uno. Ahora lo ve. Contó "gear-driven" como dos palabras; el programa la cuenta como una. En los ejercicios de Seguimiento de caso práctico se pide examinar una solución para este problema.

Prueba y depuración

Datos de punto flotante

Cuando un problema requiere el uso de números de punto flotante muy grandes, pequeños o precisos, es importante recordar las limitaciones del sistema particular en uso. Al probar un programa que efectúa cálculos de punto flotante, determine con antelación el margen de error aceptable y después diseñe sus datos de prueba para intentar llevar el programa más allá de esos límites. Compruebe de manera correcta la exactitud de los resultados calculados. (Recuerde que cuando calcula a mano los resultados correctos, una calculadora de bolsillo puede tener *menos* precisión que su sistema de computadora.) Si el programa produce resultados aceptables cuando se dan datos del peor de los casos, tal vez funcione correctamente en datos representativos.

Cómo hacer frente a los errores de entrada

Varias veces en este libro se ha requerido probar los programas para datos inválidos y escribir un mensaje de error. Escribir un mensaje de error es necesario, pero es sólo el primer paso. Se debe decidir también qué debe hacer después el programa. El problema en sí y la gravedad del error deben determinar qué acción tomar en cualquier condición de error. El método adoptado depende también de si el programa se ejecuta o no de manera interactiva.

En un programa que lee sus datos sólo de un archivo de entrada, no hay interacción con la persona que introdujo los datos. El programa, por tanto, debe intentar ajustar los elementos de datos malos, si es posible.

Si el elemento de datos inválidos no es esencial, el programa puede omitirlo y continuar; por ejemplo, si un programa que promedia calificaciones de exámenes encuentra una puntuación negativa, puede simplemente omitirla. Si es posible hacer una conjetura bien informada respecto al valor probable de los datos malos se puede establecer en ese valor antes de ser procesado. En cualquier caso, es necesario escribir un mensaje que exprese que se encontró un elemento de datos inválido y esbozar los pasos emprendidos. Esta clase de mensajes componen un *informe de excepción*.

Si el elemento de datos es esencial y no es posible ninguna conjetura se debe terminar el proceso. Es necesario escribir un mensaje al usuario con toda la información posible sobre el elemento de datos inválido.

En un ambiente interactivo, el programa puede solicitar al usuario que proporcione otro valor. El programa debe indicar al usuario qué es incorrecto en los datos originales. Otra posibilidad es escribir una lista de acciones y pedir al usuario que elija entre ellas.

Estas sugerencias acerca de cómo manejar datos malos suponen que el programa reconoce valores de datos malos. Hay dos métodos para la detección de errores: pasivo y activo. La detección pasiva de errores deja que el sistema los detecte. Esto podría parecer lo más fácil, pero el programador renuncia a controlar el proceso cuando ocurre un error. Un ejemplo de detección de error pasivo es el error de división entre cero del sistema.

La detección activa de errores significa pedir al programa que compruebe errores posibles y determinar una acción apropiada si ocurre un error. Un ejemplo de detección activa de errores sería leer un valor y usar una sentencia If para ver si el valor es 0 antes realizar la división.

Sugerencias de prueba y depuración

1. Evite generar efectos secundarios innecesarios en las expresiones. La prueba

```
if ((x = y) < z)
    ⋮
```

es menos clara y más propensa a error que la secuencia equivalente de sentencias

```
x = y;
if (y < z)
    ⋮
```

Asimismo, si omite accidentalmente los paréntesis alrededor de la operación de asignación, como esto:

```
if (x = y < z)
```

entonces, de acuerdo con la precedencia del operador de C++, *no* se asigna a `x` el valor de `y`. Se le asigna el valor 1 o 0 (el valor coercionado del resultado booleano de la expresión relacional `y < z`).

2. Es posible que los programas que dependen de un determinado conjunto de caracteres de la máquina no se ejecuten de manera correcta en otra máquina. Compruebe que la biblioteca estándar proporciona las funciones de manejo de caracteres. Funciones como `tolower`, `toupper`, `isalpha` e `iscntrl` informan de manera automática del conjunto de caracteres empleado.
3. No compare directamente la igualdad de valores de punto flotante. En vez de eso, compruebe la igualdad aproximada. La tolerancia para igualdad aproximada depende del problema particular que esté resolviendo.
4. Use enteros sólo si trabaja con números enteros. La computadora puede representar de manera exacta cualquier entero, siempre que esté dentro del intervalo permisible de valores de la máquina. También, la aritmética de enteros es más rápida que la aritmética de punto flotante en la mayoría de las máquinas.
5. Tome en cuenta los errores de representación, de cancelación, de desbordamiento positivo y negativo. Si es posible, intente ordenar los cálculos de su programa para evitar que los números de punto flotante se vuelvan demasiado grandes o pequeños.
6. Si su programa incrementa el valor de un entero positivo y de repente el resultado se vuelve un número negativo, debe sospechar de desbordamiento de enteros. En la mayoría de las computadoras, sumar 1 a `INT_MAX` produce `INT_MIN`, un número negativo.
7. Excepto cuando lo necesite, evite combinar tipos de datos en expresiones, operaciones de asignación, paso de argumentos y la devolución de un valor de función. Si debe combinar tipos, el moldeo explícito de tipos puede evitar sorpresas causadas por la coerción implícita de tipos.
8. Considere usar tipos de enumeración para que sus programas sean más legibles, entendibles y modificables.
9. Evite escribir datos anónimos. Asigne un nombre a cada tipo definido por el usuario.
10. Los valores de tipos de enumeración no se pueden introducir o producir de manera directa.
11. La degradación de tipos puede ocasionar menor precisión o corrupción de datos.

Resumen

Un tipo de datos es un conjunto de valores (el dominio) junto con las operaciones que pueden aplicarse a dichos valores. Los tipos de datos simples son aquellos cuyos valores son atómicos (indivisibles).

Los tipos integrales en C++ son `char`, `short`, `int`, `long` y `bool`. Los tipos integrales de uso común son `int` y `char`. El tipo `char` se puede usar para guardar enteros numéricos pequeños (por lo común, un byte) o, con mayor frecuencia, para almacenar datos de caracteres. Los datos de caracteres incluyen tanto caracteres imprimibles como no imprimibles. Los caracteres no imprimibles, los que controlan el comportamiento de dispositivos de hardware, se representan en C++ como secuencias de escape, como \n. Cada carácter se representa internamente como un entero no negativo de acuerdo con el conjunto de caracteres particular (como ASCII o EBCDIC) que usa la computadora.

Los tipos de punto flotante integrados en el lenguaje C++ son `float`, `double` y `long double`. Los números de punto flotante se representan en la computadora con una mantisa y un exponente. Dicha representación permite números mucho más grandes o mucho más pequeños que los que se puede representar con los tipos integrales. La representación de punto flotante también permite efectuar cálculos en números con partes fraccionarias.

Sin embargo, hay desventajas al usar números de punto flotante en cálculos aritméticos. Los errores representativos, por ejemplo, pueden afectar la exactitud de los cálculos de un programa. Al usar números de punto flotante, recuerde que si dos números son muy distintos entre sí en tamaño, sumarlos o

restarlos puede producir una respuesta errónea. Recuerde también que la computadora tiene un intervalo limitado de números que puede representar. Si un programa intenta calcular un valor demasiado grande o muy pequeño, podría enviar un mensaje de error cuando se ejecuta el programa.

C++ permite al programador definir tipos de datos adicionales. La sentencia Typedef es un simple mecanismo para renombrar un tipo existente, aunque el resultado no es un nuevo tipo de datos. Un tipo de enumeración, creado al listar los identificadores que constituyen el dominio, es un nuevo tipo de datos distinto de cualquier tipo existente. Los valores de un tipo de enumeración se pueden asignar, comparados en expresiones relacionales, usados como etiquetas de caso en una sentencia Switch, pasados como argumentos y devueltos como valores de función. Los tipos de enumeración son muy útiles en la escritura de programas claros, autodocumentados. En capítulos posteriores se examinarán características del lenguaje que permiten crear tipos incluso más eficaces definidos por el usuario.

Comprobación rápida

1. ¿Se considera `bool` un tipo integral o un tipo de enumeración? (pp. 373-376)
2. ¿Cuál es la diferencia entre el operador `&&` y el operador `&`? (pp. 377-380)
3. ¿De qué manera se representan los caracteres no imprimibles como literales en un programa de C++? (pp. 384-386)
4. Si se fuera a agrandar la porción del exponente de una representación de punto flotante, ¿qué parte de la representación disminuiría y cuál sería el efecto de este cambio? (pp. 392-397)
5. ¿Cómo se nombra al error que resulta cuando la precisión de un cálculo es mayor que la que admite la computadora? (pp. 394-395)
6. Si se asigna un `int` a un `long`, ¿el tipo de conversión es una promoción o degradación? (pp. 409-412)
7. ¿Usaría un `int` o `long` para representar un número de seguro social? (pp. 372-373)
8. Escriba una definición de tipo de enumeración para las cuatro estaciones del año. (pp. 402-407)
9. Escriba un encabezado de ciclo For que itera el tipo Estaciones definido en la pregunta anterior. (pp. 404-405)
10. ¿Un tipo anónimo definido por el usuario puede ser un parámetro en una función? (pp. 407-408)
11. En una directiva include, ¿cómo difieren en escritura el nombre de un archivo de encabezado en el directorio actual y el de un archivo de encabezado en el directorio include estándar? (pp. 408-409)

Respuestas

1. Es un tipo integral. **2.** `&&` es un AND lógico de un par de valores `bool`, mientras que `&` es un AND a nivel de bits de un par de valores integrales. **3.** Por medio de secuencias de escape o con sus valores enteros. **4.** Disminuiría la mantisa. El cambio incrementaría el intervalo de la representación pero se reduciría su precisión. **5.** Error de representación. **6.** Una promoción. **7.** Un `long`, a menos que el sistema admita valores `int` de 64 bits. **8.**
```
enum Seasons {SPRING, SUMMER, WINTER, AUTUMN};
```
9.
```
for (quarter = SPRING;
     quarter <= AUTUMN;
     quarter = Seasons(quarter + 1))
```
10. No. **11.** Va entre comillas y no entre paréntesis angulares.

Ejercicios de preparación para examen

1. Todos los tipos integrales en C++ pueden ser con signo o sin signo. ¿Verdadero o falso?
2. La diferencia entre una expresión de asignación y una sentencia de asignación es un punto y coma. ¿Verdadero o falso?
3. El operador sizeof se puede usar para determinar si el tipo `int` de una máquina es de 32 o 64 bits de largo. ¿Verdadero o falso?
4. Los números de punto flotante pocas veces son exactamente iguales. ¿Verdadero o falso?
5. Los valores de tipos de enumerador se deben escribir en mayúsculas. ¿Verdadero o falso?

6. ¿Cuáles son los cinco tipos integrales en C++?
7. ¿Qué sucede si el dígito principal de una literal entera es un cero? ¿Qué pasa si el cero va seguido de una X?
8. ¿Qué grupo de operadores de C++ tiene la precedencia menor de todos?
9. ¿Cuál es la diferencia en efecto de escribir `count++` en lugar de `++count`?
10. ¿En qué situación es necesario usar la notación de prefijo para la operación de moldeo en lugar de la notación funcional?
11. ¿Qué funciones podría usar para simplificar la siguiente expresión?

```
(reply == 'N' || reply == 'n' || reply == 'Y' || reply == 'y')
```

12. Si la variable name tipo `string`, contiene "Abigail", ¿a qué es igual la expresión `name[3]`?
13. Si una computadora tiene un tipo de punto flotante con cinco dígitos de precisión decimal, y un exponente de un dígito, ¿cuál sería el resultado de sumar 3.8281×10^4 y 2.4531×10^0?
14. Explique la diferencia entre error absoluto y relativo al comparar si dos números de punto flotante son iguales.
15. ¿Qué es erróneo en el siguiente par de declaraciones de tipo enumeración?

```
enum Colors {RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET};
enum Flowers{ROSE, DAFFODIL, LILY, VIOLET, COSMOS, ORCHID};
```

16. Dada la declaración de Colors en el ejercicio 15, ¿cuál es el valor de la expresión `(YELLOW + 1)`?
17. Dado el segmento de código:

```
enum Flowers{ROSE, DAFFODIL, LILY, VIOLET, COSMOS, ORCHID};
Flowers choice;
choice = LILY;
choice++;
```

¿Por qué el compilador envía un mensaje de error de tipo inválido para la última línea?

18. ¿Por qué es imposible usar un tipo anónimo con un parámetro de función?

Ejercicios de calentamiento para programación

1. Escriba el valor entero 5
   a) como una literal de tipo `int`
   b) como una literal de tipo `long`
   c) como una literal de tipo `unsigned long`
   d) en base 8 (octal)
   e) en base 16 (hexadecimal)
2. Escriba el valor de punto flotante 3.14159265
   a) como una literal de tipo `float`
   b) como una literal de tipo `double`
   c) como una literal de tipo `long double`
   d) como una literal de tipo `double` con un exponente 0
3. Escriba las sentencias de expresión de asignación que hacen lo siguiente:
   a) sume 7 a la variable `days`
   b) multiplique el valor de la variable `radius` por 6.2831853
   c) reste 40 de la variable `workHours`
   d) divida la variable average entre la variable `count`
4. Escriba una expresión cuyo resultado sea el número de bits en un valor de tipo `long`.
5. Use las reglas de precedencia de C++ para eliminar cualquier paréntesis innecesario de las expresiones siguientes:
   a) `((a * b) + (c * d))`
   b) `((a * b) / (c * d))`
   c) `((a + b) + ((c / (d + e)) * f))`

d) `(((a + b) / (c + d)) * (e + f))`
e) `((-a + b) <= (c * d)) && ((a + b) >= (c - d))`

6. Escriba un segmento de código que introduzca un carácter, y si es un carácter numérico, lo convierta en un valor `int`.
7. Escriba un ciclo For que examine una cadena de nombre inLine, y cuente las veces que aparece el carácter `'e'`.
8. Escriba una función de devolución de valor con una sentencia Switch anidada que devuelva el número correspondiente al nombre de mes que se suministra vía un parámetro de cadena. El nombre del mes puede estar en mayúsculas, minúsculas o una combinación de ambas.
9. Escriba una sentencia If que sume 12 a `hour` si la primera letra en la cadena ampm es una `'P'` o una `'p'`.
10. Escriba una expresión que devuelva `true` si la variable float `balance`, es igual a `audit` con un error absoluto de 0.001 o menos.
11. Escriba una expresión que devuelva `true` si la variable float, `year`, es igual a `epoch` dentro de 0.00001% del valor en `epoch`. (Analice detenidamente este punto.)
12. Declare un tipo de enumeración que conste de los nueve planetas en su orden por distancia hacia el Sol (Mercurio primero, Plutón al último).
13. Escriba una función de devolución de valor que convierta un planeta del tipo de enumeración declarado en el ejercicio 12. Si la cadena no es un nombre de planeta apropiado, devolver TIERRA.
14. Escriba una función de devolución de valor que convierta un planeta del tipo de enumeración declarada en el ejercicio 12 en la cadena correspondiente. El planeta es un parámetro de entrada y la cadena es devuelta por la función. Si la entrada no es un planeta válido, devolver `"Error"`.
15. Escriba una sentencia For que imprima los nombres de los planetas en orden, con el tipo de enumeración declarado en el ejercicio 12 y la función declarada en el ejercicio 14.

Problemas de programación

1. En el problema de programación 2 del capítulo 4 se pidió escribir un programa en C++ que pide al usuario introducir su peso y el nombre de un planeta. En el capítulo 9, en el problema de programación 2 se pidió escribir un programa con una sentencia Switch. Ahora, reescriba el programa de modo que utilice un tipo enumerado para representar el planeta. Si hizo los ejercicios de calentamiento para programación 12 al 15, reescribir esto es muy fácil.

   Para facilidad de referencia, se repite aquí la información para el problema original. En la tabla siguiente se proporciona el factor por el que se debe multiplicar el peso para cada planeta. El programa debe producir un mensaje de error si el usuario no escribe un nombre de planeta correcto. La leyenda de orientación (prompt) y el mensaje de error deben aclarar al usuario cómo se debe introducir el nombre de un planeta. Asegúrese de usar el formato apropiado y los comentarios pertinentes en su código. El resultado debe ser marcado con claridad y tener un formato nítido.

Mercurio	0.4155
Venus	0.8975
Tierra	1.0
Luna	0.166
Marte	0.3507
Júpiter	2.5374
Saturno	1.0677
Urano	0.8947
Neptuno	1.1794
Plutón	0.0899

2. En el problema 3 de programación del capítulo 9 se pidió escribir un programa que generara archivos de informes de ventas para un conjunto de vendedores de viajes. Ahí se usó un entero en el intervalo de 1 a 10 para representar números de identificación para los vendedores. Reescriba

el programa de modo que utilice un tipo de enumeración cuyos valores sean los nombres de las personas (los puede inventar). El formato del archivo de ventas debe remplazar el número de identificación del vendedor con una cadena que es el apellido de la persona, de modo que una línea del archivo contenga un nombre, un número de elemento y una cantidad. Por conveniencia, se repite aquí la otra información relacionada con el problema.

La compañía vende ocho productos distintos, con números de identificación de 7 a 14 (algunos productos antiguos se han discontinuado). Los precios unitarios de los productos son:

Número de producto	Precio unitario
7	345.00
8	853.00
9	471.00
10	933.00
11	721.00
12	663.00
13	507.00
14	259.00

El programa lee en el archivo ventas y genera un archivo separado para cada vendedor que contiene sólo sus ventas. Cada línea del archivo ventas se copia al archivo de vendedor apropiado, sin el nombre del vendedor. Los nombres de archivo deben ser el nombre del vendedor con `.dat` anexado (tal vez sea necesario ajustar los nombres que no funcionan como nombres de archivo en su computadora, por ejemplo nombres unidos mediante un guión o nombres con apóstrofos). El total para la venta (cantidad multiplicada por precio unitario) se anexa al registro. Al final del proceso, se deben producir las ventas totales para cada vendedor con etiquetas informativas para `cout`. Use la descomposición funcional para diseñar el programa. Asegúrese de que el programa maneje correctamente nombres inválidos de vendedores. Si el nombre de un vendedor es inválido, escriba un mensaje de error para `cout`. Si es inválido el número de producto, escriba el mensaje de error para el archivo del vendedor y no calcule un total para la venta.

3. Usted toma una clase de geología y el profesor desea escribir un programa para ayudar a los alumnos a aprender periodos del tiempo geológico. El programa debe permitir que el usuario introduzca un intervalo de fechas prehistóricas (en millones de años) y luego producir los periodos que están incluidos en ese intervalo. Cada vez que se hace esto, se pregunta al usuario si quiere continuar. El objetivo del ejercicio es que el alumno intente averiguar cuándo comenzó cada periodo, de modo que pueda elaborar una gráfica de tiempo geológico. Dentro del programa, represente los periodos con un tipo de enumeración constituido por sus nombres. Tal vez quiera crear una función que determine el periodo correspondiente a una fecha, y otra que devuelva la cadena correspondiente a cada identificador en la enumeración. Después puede usar un ciclo For para producir la serie de periodos en el intervalo. Los periodos de tiempo geológico son:

Nombre del periodo	Fecha de inicio (millones de años)
Cuaternario	2.5
Terciario	65
Cretáceo	136
Jurásico	192
Triásico	225
Pérmico	280
Carbonífero	345
Devónico	395
Silúrico	435
Ordovícico	500
Cámbrico	570
Precámbrico	4500 o antes

Use la descomposición funcional para resolver el problema. Asegúrese de usar un buen estilo de codificación y comentarios ilustrativos. Las leyendas de orientación y los mensajes de error producidos deben ser claros e informativos.

4. El programa educacional que escribió para el problema 4 fue un gran éxito. El profesor de geología quiere que usted escriba otro programa como ayuda para enseñar el tiempo geológico. En este programa, la computadora selecciona una fecha en el tiempo geológico y lo presenta al alumno. Éste conjetura qué periodo corresponde a la fecha. Se permite al alumno continuar haciendo suposiciones hasta que obtenga la respuesta correcta. Luego, el programa pregunta al usuario si el alumno quiere probar de nuevo, y repite el proceso si la respuesta es "sí". De nuevo debe usar un tipo de enumeración que conste de los nombres de los periodos. En este caso, es probable que quiera determinar una función que devuelva el periodo correspondiente a una cadena que contenga el nombre de un periodo (el programa debe funcionar con cualquier estilo de uso de mayúsculas en los nombres). Es posible que desee también una función que devuelva el periodo para una fecha dada.

   Use el diseño funcional para resolver el problema. Asegúrese de usar un buen estilo de documentación y comentarios ilustrativos. Las leyendas de orientación y los mensajes de error producidos deben ser claros e informativos. Tal vez quiera agregar un poco de interés al programa y mantener un registro del número de suposiciones que hace el usuario, y ofrecer distintos niveles de elogio y motivación dependiendo de los logros de usuario.

5. Escriba un programa de C++ que determina el número más grande para el cual su computadora puede representar su factorial con un tipo `long double`. El factorial es el producto de todos los números desde el número uno hasta el número dado. Por ejemplo, diez factorial (escrito 10!) es

   ```
   1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10 = 3628800
   ```

   Como puede ver, el factorial crece para ser un número grande muy rápido. Su programa debe seguir multiplicando el factorial previo por el siguiente entero, luego restar uno y comprobar si la diferencia entre el factorial y el factorial menos uno es menor que uno —una tolerancia de error. Cuando se alcanza la precisión máxima del tipo, y se trunca la menor cantidad de dígitos significativos para permitir que se almacene la mayor cantidad de dígitos significativos del producto, entonces restar uno no debe tener efecto en el valor. Sin embargo, debido a que las representaciones de punto flotante pueden ser inexactas, la expresión

   ```
   abs((number - 1) - number)
   ```

   podría no ser igual a 1. Por eso necesita incluir una tolerancia de error pequeña en la comparación.

   Use la descomposición funcional para resolver el problema. Codifique el programa con un buen estilo y comentarios ilustrativos. Para mantener informado al usuario del progreso, tal vez quiera producir todos los valores factoriales intermedios. El número máximo y su factorial se deben marcar con claridad.

Seguimiento de caso práctico

1. En el programa Estilo, los módulos ProcessCharacter y PrintTable toman seis contadores como parámetros, lo que constituye una lista de parámetros muy larga. ¿Es posible manejar esto de otra manera?
2. ¿Cómo podría determinar si un guión se debe contar como un símbolo de fin de palabra o una interrupción en la palabra debido a espaciamiento?
3. Ponga en práctica el cambio descrito en su respuesta al ejercicio 2.
4. endOfWord se restablece en `false` cada vez que se lee un carácter alfanumérico. Así, se establece en sí mismo una y otra vez. ¿Puede pensar en un esquema que permitiría establecerlo sólo una vez?
5. ¿Se debe agregar la detección de error al programa Estilo? Explique.

CAPÍTULO

11

Tipos estructurados, abstracción de datos y clases

Objetivos

Objetivos de conocimiento

- *Entender el concepto general del tipo unión de C++.*
- *Comprender la diferencia entre especificación e implementación de un tipo de datos abstractos.*
- *Entender cómo el compilador impone la encapsulación y la ocultación de información.*

Objetivos de habilidades

Ser capaz de:

- *Declarar un tipo de datos* `struct` *(registro), una estructura de datos cuyos componentes pueden ser heterogéneos.*
- *Tener acceso a una variable* `struct`.
- *Definir una estructura de registro jerárquica.*
- *Tener valores guardados en un registro jerárquico.*
- *Declarar un tipo* `class` *de C++.*
- *Declarar objetos de clase, dada la declaración de un tipo* `class`.
- *Escribir código cliente que invoca funciones miembros de clases.*
- *Poner en práctica las funciones miembros de clases.*
- *Organizar el código para una clase de C++ en dos archivos: el archivo de especificación (*`.h`*) y el archivo de implementación.*
- *Escribir un constructor de clase de C++.*

En el capítulo anterior se examinó el concepto de un tipo de datos y se consideró cómo definir tipos de datos simples. En el presente capítulo se amplía la definición de un tipo de datos para incluir tipos estructurados. Se comienza con un análisis de tipos estructurados en general y luego se examinan dos tipos estructurados proporcionados por el lenguaje C++: struct y unión.

A continuación se introduce el concepto de *abstracción de datos*, la separación de las propiedades lógicas de un tipo de datos de su implementación. La abstracción de datos es importante porque permite crear tipos de datos que, de otro modo, no estarían disponibles en un lenguaje de programación. Otro beneficio de la abstracción de datos es la capacidad para producir *software comercial*, piezas de software que se pueden usar una y otra vez en programas distintos, ya sea por el creador del software o por cualquier otro programador que desee usarlas.

El concepto principal para practicar la abstracción de datos es el *tipo de datos abstractos*. En este capítulo se examinan en detalle tipos de datos abstractos y se introduce la característica del lenguaje C++ diseñada de manera expresa para crearlos: la *clase*. Se concluye con dos casos prácticos que demuestran la abstracción de datos, tipos de datos abstractos y clases de C++.

11.1 Tipos de datos simples contra estructurados

Tipo de datos estructurados Tipo de datos en el que cada valor es una colección de componentes y cuya organización se caracteriza por el método empleado para tener acceso a componentes individuales. Las operaciones permisibles en un tipo de datos estructurados incluyen el almacenamiento y la recuperación de componentes individuales.

En el capítulo 10 se examinan tipos de datos simples o atómicos. Un valor en un tipo simple es un solo elemento de datos; no se puede descomponer en partes componentes. Por ejemplo, cada valor int es un número entero único y no se puede descomponer más. En contraste, un **tipo de datos estructurados** es uno en el que cada valor es una *colección* de elementos componentes. La colección completa recibe un solo nombre; sin embargo, aún es posible tener acceso a cada componente. Un ejemplo de un tipo de datos estructurados en C++ es la clase string, empleada para crear y controlar cadenas. Cuando declara una variable `myString` como de tipo `string`, `myString` no representa sólo un valor de datos atómico; representa una colección completa de caracteres. Pero se puede tener acceso a cada uno de los componentes de la cadena (al usar una expresión como `myString[3]`, la cual tiene acceso al valor char en la posición 3).

Los tipos de datos simples, tanto integrados como definidos por el usuario, son los bloques de construcción para tipos estructurados. Un tipo estructurado reúne un conjunto de valores componentes y, en general, impone una disposición específica (véase la figura 11-1). El método empleado para tener acceso a componentes individuales de un tipo estructurado depende de cómo estén dispuestos éstos. Conforme se analizan varias formas de estructurar datos, se consideran los mecanismos de acceso correspondientes.

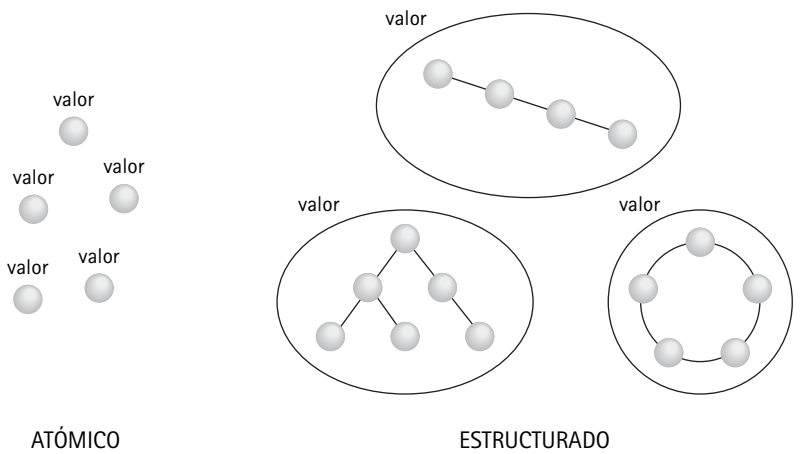


Figura 11-1 *Tipos de datos atómicos (simples) y estructurados*

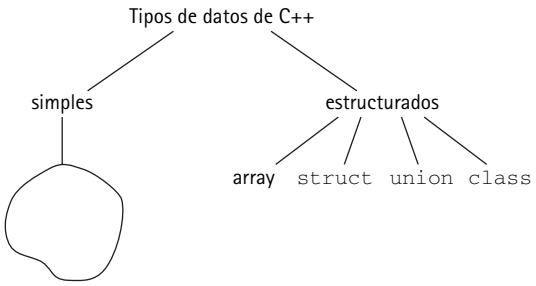


Figura 11-2 *Tipos estructurados de C++*

En la figura 11-2 se muestran los tipos estructurados disponibles en C++. Esta figura es una parte del diagrama completo presentado en la figura 3-1.

En este capítulo se examinan los tipos `struct`, `union` y `class`. Los tipos de datos array son el tema del capítulo 12.

11.2 Registros (structs)

Registro (estructura en C++) Tipo de dados estructurado con un número fijo de componentes a los que se tiene acceso por medio de un nombre. Los componentes pueden ser heterogéneos (de tipos distintos).

Campo (miembro en C++) Componente de un registro.

En informática, un **registro** es un tipo de datos estructurado, *heterogéneo*. Heterogéneo significa que cada uno de los componentes de un registro puede ser un tipo de datos diferente. Cada componente de un registro se denomina **campo** del registro, y cada campo recibe un nombre conocido como *nombre de campo*. C++ usa su propia terminología con registros. Un registro se denomina **estructura**; los campos de un registro se nombran **miembros** de la estructura, y cada miembro tiene un *nombre*.*

En C++, los tipos de datos de registro se declaran, en general, de acuerdo con la siguiente sintaxis:

Declaración struct

```
struct Nombre del tipo
{
      Lista de miembros
};
```

donde el nombre del tipo es un identificador que da un nombre al tipo de datos y la lista de miembros se define como

Lista de miembros

```
Tipo de datos Nombre del miembro ;
Tipo de datos Nombre del miembro ;
                  ⋮
```

*Técnicamente, una `struct` de C++ es casi idéntica al tipo `class` que se introduce después en este capítulo. Sin embargo, en C una `struct` tiene las propiedades de un registro, y la mayoría de los programadores de C++ continúan con el uso de `struct` en su papel tradicional de representar directamente un registro. En este libro se mantiene esta práctica estándar.

La palabra reservada `struct` es una abreviatura para *estructura*. Debido a que la palabra *estructura* tiene muchos otros significados en informática, se usará *struct* o *registro* para evitar confusiones respecto a lo que se hace referencia.

Tal vez reconozca que la sintaxis de una lista de miembros es casi idéntica a una serie de declaraciones de variable. Tenga cuidado: una declaración `struct` es una declaración de tipo, y aun se deben declarar variables de este tipo para cualquier ubicación de la memoria que se relacionará con los nombres de miembros. Por ejemplo, se usará una struct para describir un alumno en una clase. Se desea guardar el nombre y el apellido, el promedio global previo a esta clase, la calificación en tareas de programación, la calificación en preguntas de cultura general, la calificación del examen final y la calificación final del curso.

```
// Declaraciones de tipos

enum GradeType {A, B, C, D, F};

struct StudentRec
{
    string    firstName;
    string    lastName;
    float     gpa;            // Promedio de puntos
    int       programGrade;   // Suponga 0..400
    int       quizGrade;      // Suponga 0..300
    int       finalExam;      // Suponga 0..300
    GradeType courseGrade;
};

// Declaraciones de variables

StudentRec firstStudent;
StudentRec student;
int        grade;
```

Observe que, tanto en este ejemplo como en la plantilla de sintaxis, una declaración `struct` termina en un punto y coma. Hasta ahora ha aprendido a no escribir un punto y coma después de la llave derecha de una sentencia compuesta (bloque). Sin embargo, la lista de miembros en una declaración `struct` no se considera una sentencia compuesta; las llaves son sólo sintaxis requerida en la declaración. Una declaración `struct`, como todas las sentencias de declaración de C++, deben terminar con un punto y coma.

`firstName` (nombre), `lastName` (apellido), `gpa` (promedio de puntos), `programGrade` (calificación en programación), `quizGrade` (calificación en cultura general), `finalExam` (examen final) y

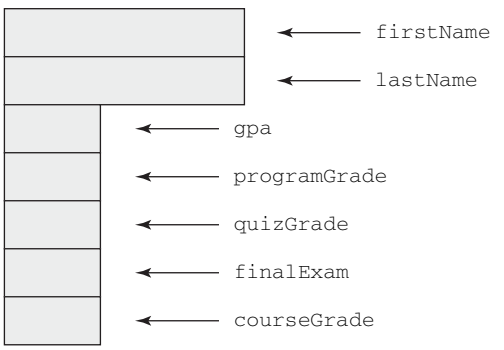


Figura 11-3 *Patrón para una struct*

`courseGrade` (calificación del curso) son nombres de miembros dentro del tipo `struct`, `StudentRec`. Estos nombres constituyen la lista de miembros. Observe que a cada nombre de miembro se le asigna un tipo. Asimismo, los nombres de miembros deben ser únicos dentro de un tipo `struct`, del mismo modo que los nombres de variables deben ser únicos dentro de un bloque.

`firstName` y `lastName` son de tipo `string`. `gpa` es un miembro `float`. `programGrade`, `quizGrade` y `finalExam` son miembros `int`. `courseGrade` es un tipo de datos de enumeración conformado por las calificaciones A a D y F.

Ninguno de estos miembros struct se relaciona con áreas de la memoria hasta que se declara una variable del tipo `StudentRec`. `StudentRec` es sólo un patrón para una struct (véase la figura 11-3). Las variables `firstStudent` y `student` son variables de tipo `StudentRec`.

Acceso a componentes individuales

Para tener acceso a un miembro de una variable struct, usted proporciona el nombre de la variable, seguido de un punto, y después el nombre del miembro. Esta expresión se denomina **selector de miembro**. La plantilla de sintaxis es

Selector de miembro Expresión usada para tener acceso a componentes de una variable struct. Se forma al usar el nombre de la variable struct y el nombre del miembro, separados por un punto.

Selector de miembro

Variable struct . Nombre de miembro

Esta sintaxis para seleccionar los componentes individuales de una estructura tiene el nombre de *notación de punto*. Para tener acceso al promedio de puntos de `firstStudent`, se escribiría

```
firstStudent.gpa
```

Para tener acceso a la calificación de examen final de `student`, se escribiría

```
student.finalExam
```

El componente de una `struct` al cual tiene acceso el selector de miembro, se trata como cualquier otra variable del mismo tipo. Se podría usar en una sentencia de asignación, pasado como un argumento, etcétera. En la figura 11-4 se muestra la variable `struct`, `student`, con el selector de miembro para cada miembro. En este ejemplo se supone que cierto proceso ya ha tenido lugar, así que los valores se guardan en algunos de los componentes.

Se demostrará el uso de dichos selectores de miembro. Al usar la variable `student`, el siguiente segmento de código introduce una calificación de examen final; añade la calificación de programación, la calificación de cultura general y la calificación del examen final, y luego asigna una calificación con letra al resultado.

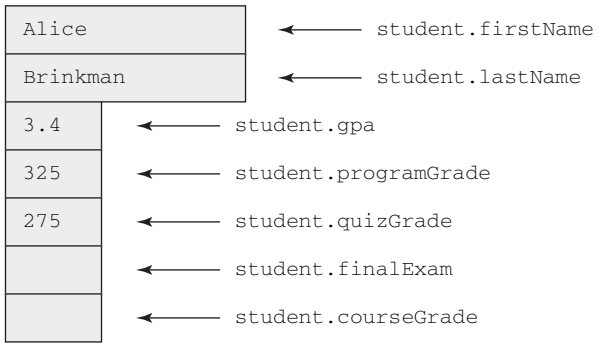


Figura 11-4 *Variable struct,* `student`*, con selectores de miembro*

```
cin >> student.finalExam;
grade = student.finalExam + student.programGrade +
        student.quizGrade;
if (grade >= 900)
    student.courseGrade = A;
else if (grade >= 800)
    student.courseGrade = B;
else  .
      .
      .
```

Operaciones de agregación en structs

Operación de agregación Operación en una estructura de datos como un todo, a diferencia de una operación en un componente individual de la estructura de datos.

Además de tener acceso a componentes individuales de una variable struct, en algunos casos es posible usar **operaciones de agregación**. Una operación de agregación es la que maneja la estructura como una unidad completa.

En la tabla siguiente se resumen las operaciones de agregación permitidas en variables struct:

Operación de agregación	¿Permitida en structs?
I/O	No
Asignación	Sí
Aritmética	No
Comparación	No
Paso de argumentos	Sí, por valor o por referencia
Retorno como un valor de devolución de una función	Sí

De acuerdo con la tabla, una variable struct se puede asignar a otra. Sin embargo, ambas variables deben ser declaradas como del mismo tipo. Por ejemplo, dadas las declaraciones

```
StudentRec student;
StudentRec anotherStudent;
```

la sentencia

```
anotherStudent = student;
```

copia el contenido de la variable struct, `student`, en la variable `anotherStudent`, miembro por miembro.

Por otro lado, las operaciones aritméticas de agregación y comparaciones no están permitidas (sobre todo porque no tendrían sentido):

```
student = student * anotherStudent;     // No permitida
if (student < anotherStudent)           // No permitida
```

Además, no se permite la agregación I/O:

```
cin >> student;                         // No permitida
```

Se debe introducir o producir una variable struct, un miembro a la vez:

```
cin >> student.firstName;
cin >> student.lastName;
⋮
```

De acuerdo con la tabla, un registro (struct) completo se puede pasar como un argumento, ya sea por valor o por referencia, y se puede devolver una struct como el valor de una función de devolución de valor. Se definirá una función que toma una variable `StudentRec` como parámetro.

La tarea de esta función es determinar si la calificación de un alumno en un curso es congruente con su promedio de puntos (*grade point average*, *GPA*). Se define *congruente* para indicar que la calificación del curso corresponde al GPA redondeado. El GPA se calcula en una escala de cuatro puntos, donde A es 4, B es 3, C es 2, D es 1 y F es 0. Si el GPA redondeado es 4 y la calificación del curso es A, entonces la función devuelve `true`. Si el GPA redondeado es 4 y la calificación del curso no es A, entonces la función devuelve `false`. Cada una de las otras calificaciones se prueba de la misma manera.

La función `Consistent` se codifica a continuación. El parámetro `aStudent`, una variable struct de tipo `StudentRec`, se pasa por valor.

```
bool Consistent( /* in */ StudentRec aStudent )

// Precondición:
//     0.0 <= aStudent.gpa <= 4.0
// Poscondición:
//     Valor de función == true, si la calificación del curso es congruente
//                              con el GPA global
//                    == false, en caso contrario

{
    int roundedGPA = int(aStudent.gpa + 0.5);

    switch (roundedGPA)
    {
        case 0: return (aStudent.courseGrade == F);
        case 1: return (aStudent.courseGrade == D);
        case 2: return (aStudent.courseGrade == C);
        case 3: return (aStudent.courseGrade == B);
        case 4: return (aStudent.courseGrade == A);
    }
}
```

Más acerca de declaraciones struct

Para completar el análisis inicial de estructuras de C++ se proporciona una plantilla de sintaxis más completa para una declaración de tipo `struct`:

Declaración struct

```
struct Nombre de tipo
{
    Lista de miembros
} Lista de variables ;
```

Como puede ver en la plantilla de sintaxis, dos elementos son opcionales: nombre de tipo (el nombre del tipo `struct` que se declara) y lista de variables (lista de nombres de variables entre la llave derecha y el punto y coma). Hasta ahora, en los ejemplos se ha declarado un nombre de tipo,

pero no se ha incluido una lista de variables. La lista de variables le permite no sólo declarar un tipo `struct` sino también declarar variables de ese tipo, todo en una sentencia. Por ejemplo, es posible escribir las declaraciones

```
struct StudentRec
{
    string firstName;
    string lastName;
      ⋮
};

StudentRec firstStudent;
StudentRec student;
```

de modo más compacto en la forma

```
struct StudentRec
{
    string firstName;
    string lastName;
      ⋮
} firstStudent, student;
```

En este libro se evita combinar declaraciones de variable con declaraciones de tipo, de preferencia para mantener ambos conceptos separados.

Si se omite el nombre de tipo, pero se incluye la lista de variables, se crea un tipo anónimo:

```
struct
{
    int   firstMember;
    float secondMember;
} someVar;
```

Aquí, `someVar` es una variable de tipo anónimo. Ninguna otra variable de este tipo se puede declarar porque el tipo no tiene nombre. Por tanto, `someVar` no participa en operaciones de agregación como la asignación o paso de argumentos. Las advertencias dadas en el capítulo 10 contra la escritura anónima de tipos de enumeración se aplican también a tipos `struct`.

Enlace de elementos similares

En los capítulos 9 y 10 había un número de contadores que tenían que ser pasados a dos funciones separadas. Por ejemplo, aquí tiene la llamada para la función `ProcessCharacter` del capítulo 10:

```
ProcessCharacter(character, uppercaseCounter,
    lowercaseCounter, digitCounter, sentenceCounter,
    wordCounter, ignoreCounter);
```

La interfaz se puede hacer mucho más simple si se reúnen dichos contadores en un registro como sigue:

```
struct Counters
{
    int uppercaseCounter;
    int lowercaseCounter;
```

```
    int digitCounter;
    int sentenceCounter;
    int wordCounter;
    int ignoreCounter;
};
```

Los prototipos para `ProcessCharacter` y `PrintTable` se convierten en

```
Counters counters;
void ProcessCharacter
    ( /* in */    char character,      // Carácter que será procesado
      /* inout */ Counters& counters );// Contadores de categorías
void PrintTable
    ( /* inout */ ofstream& table,     // Archivo de salida
      /* inout */ Counters counters);  // Contadores de categorías
```

De hecho, es posible hacer más simple el código al añadir una función para inicializar los contadores.

```
void InitializeCounters
    ( /* inout */ Counters& counters); // Contadores de categorías
```

Por supuesto, se tendrán que reescribir los cuerpos de `ProcessCharacter` y `PrintTable`, y se tendrá que escribir el cuerpo de `InitializeCounters`. No se tomará el espacio para mostrar todo este código, pero de la implementación de `InitializeCounters` mostrada aquí, se ven las clases de cambios que son necesarios en las otras funciones.

```
void InitializeCounters
    ( /* inout */ Counters& counters) // Contadores de categorías
// Los contadores para distintas categorías se fijan en cero
// Poscondición:
//    Se ha establecido en cero cada campo de los contadores de variables struct
{
   counters.uppercaseCounter = 0;
   counters.lowercaseCounter = 0;
   counters.digitCounter = 0;
   counters.wordCounter = 0;
   counters.sentenceCounter = 0;
   counters.ignoreCounter = 0;
}
```

¿Ve qué tan simple se vuelve el código para la función principal?

```
int main()
{
    // Preparar los archivos para lectura y escritura
    ifstream text;
    ofstream table;
    OpenFiles(text, table);
    if ( !text || !table)
    {
        cout << "Los archivos no se abrieron con éxito." << endl;
        return 1;
    }
```

```
    char character;                 // Carácter de entrada
    Counters counters;              // Contiene los contadores de categorías
    InitializeCounters(counters);   // Establece en cero a los contadores
    text.get(character);            // Introducir un carácter
    do
    { // Procesar cada carácter
        ProcessCharacter(character, counters);
        text.get(character);        // Introducir un carácter
    } while (text);
    PrintTable(table, counters);
    text.close();
    table.close();
    return 0;
}
```

Registros jerárquicos

Se han visto ejemplos en los que los componentes de un registro son variables y cadenas simples. Un componente de un registro puede ser también otro registro. Los registros cuyos componentes son por sí mismos registros, se denominan **registros jerárquicos**.

Registro jerárquico Registro en el que por lo menos uno de los componentes es por sí mismo un registro.

Considérese un ejemplo en el que es apropiada una estructura jerárquica. Un pequeño almacén de máquinas guarda información acerca de cada una de sus máquinas. Hay información descriptiva (como el número de identificación, una descripción de la máquina, la fecha de compra y el costo); también se conserva información estadística (como el número de días caídos, el índice de fallas y la fecha del último servicio). ¿Cuál es una forma razonable de presentar toda esta información? Primero, considérese una estructura de registro horizontal (no jerárquico) que contiene esta información.

```
struct MachineRec
{
    int    idNumber;
    string description;
    float  failRate;
    int    lastServicedMonth;  // Suponga 1..12
    int    lastServicedDay;    // Suponga 1..31
    int    lastServicedYear;   // Suponga 1900..2050
    int    downDays;
    int    purchaseDateMonth;  // Suponga 1..12
    int    purchaseDateDay;    // Suponga 1..31
    int    purchaseDateYear;   // Suponga 1900..2050
    float  cost;
};
```

El tipo `MachineRec` tiene 11 miembros. Hay aquí tanta información detallada que es difícil percatarse de lo que representa el registro. Veamos si es posible reorganizarla en una estructura jerárquica que tenga más sentido. Es posible organizar la información en dos grupos: información que cambia e información que no cambia. Hay también dos fechas que es necesario conservar: la de compra y la del último servicio. Estas observaciones hacen pensar en usar un registro que describa una fecha, uno que describa los datos estadísticos y un registro global que contenga los otros dos como componentes. Las siguientes declaraciones de tipo reflejan dicha estructura.

```
struct Date
{
    int month;  // Suponga 1..12
    int day;    // Suponga 1..31
    int year;   // Suponga 1900..2050
};
struct Statistics
{
    float    failRate;
    Date     lastServiced;
    int      downDays;
};
struct MachineRec
{
    int             idNumber;
    string          description;
    Statistics      history;
    Date            purchaseDate;
    float           cost;
};

MachineRec machine;
```

El contenido de un registro de máquina es mucho más que obvio. Dos de los componentes del tipo `struct`, `MachineRec`, son por sí mismos structs: `purchaseDate` es de tipo `struct`, `Date`; e history es de tipo `struct`, `Statistics`. Uno de los componentes de tipo `struct`, `Statistics`, es una `struct` de tipo `Date`.

¿Cómo se tiene acceso a los componentes de una estructura jerárquica como ésta? Se elaboran las expresiones de acceso (selectores de miembros) para los miembros de los structs insertados de izquierda a derecha, comenzando con el nombre de la variable `struct`. Aquí tiene algunas expresiones y los componentes a los que tienen acceso:

Expresión	Componente al que se tiene acceso
`machine.purchaseDate`	Variable struct `Date`
`machine.purchaseDate.month`	`month` miembro de una variable struct `Date`
`machine.purchaseDate.year`	`year` miembro de una variable struct `Date`
`machine.history.lastServiced.year`	`year` miembro de una variabale struct `Date` contenida en una struct de tipo `Statistics`

La figura 11-5 es una representación diagramática de machine con valores. Observe con cuidado cómo se tiene acceso a cada componente.

11.3 Uniones

En la figura 11-2 se presenta un diagrama que muestra los cuatro tipos estructurados disponibles en C++. Se han analizado los tipos `struct` y ahora se consideran los tipos *union*.

En C++, una unión se define como una estructura que contiene sólo uno de sus miembros a la vez durante la ejecución del programa. Enseguida se presenta una declaración de un tipo `union` y una variable de unión:

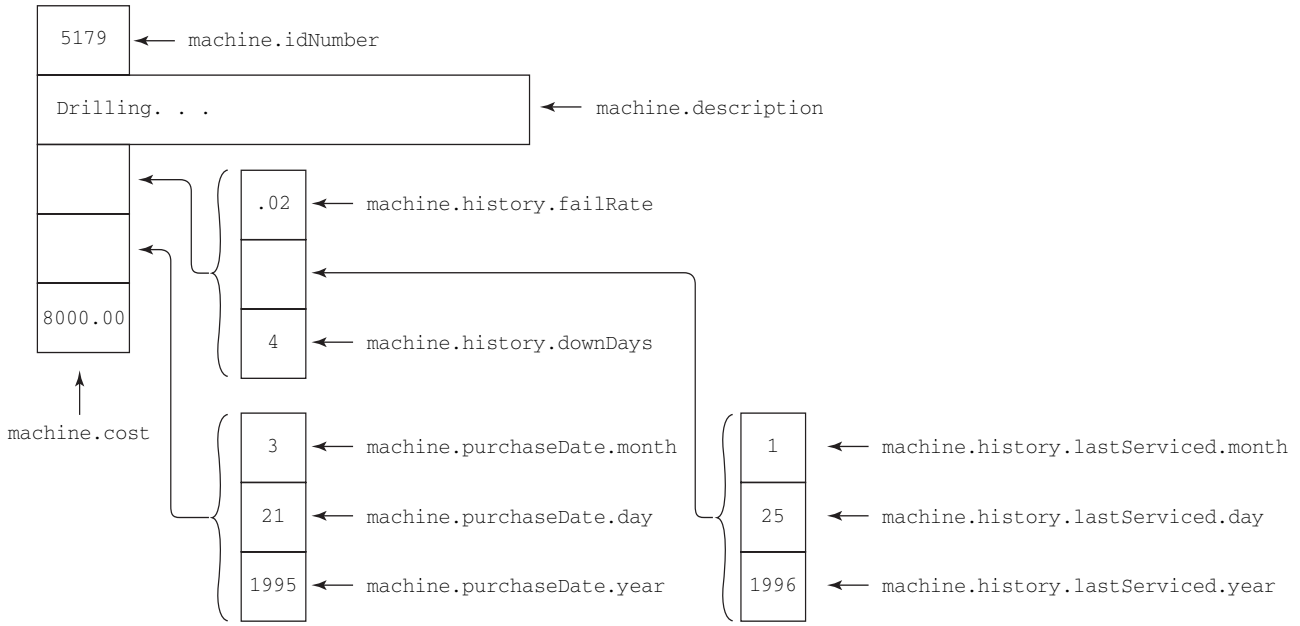


Figura 11.5 *Registros jerárquicos en la variable machine*

```
union WeightType
{
    long  wtInOunces;
    int   wtInPounds;
    float  wtInTons;
};

WeightType weight;
```

La sintaxis para declarar un tipo `union` es idéntica a la que se mostró para el tipo `struct`, excepto que la palabra `union` sustituye a `struct`.

En tiempo de ejecución, el espacio de memoria asignado a la variable `weigth` *no* incluye espacio para tres componentes distintos. En cambio, `weight` puede contener sólo uno de los siguientes valores: *ya sea* un valor `long` *o* un valor `int` *o* un valor `float`. Se supone que el programa nunca necesitará un peso en onzas, en libras y en toneladas al mismo tiempo mientras se ejecuta. El propósito de una unión es conservar memoria al forzar varios valores a usar el mismo espacio de memoria, uno a la vez. El siguiente código muestra cómo podría ser usada la variable `weight`.

```
weight.wtInTons = 4.83;
 ⋮
// Ya no se necesita el peso en toneladas. Vuelva a usar el espacio de memoria.

weight.wtInPounds = 35;
 ⋮
```

Después de la última sentencia de asignación se ignora el valor `float` previo 4.83 y lo sustituye el valor `int` 35.

Es muy razonable argumentar que la unión no es una estructura de datos en absoluto. No representa una colección de valores; representa sólo un valor simple de entre varios valores *posibles*. Por otro lado, las uniones se agrupan con los tipos estructurados como resultado de su similitud con los structs.

Hay mucho más que decir acerca de las uniones, incluso temas sutiles relacionados con su declaración y uso. Sin embargo, dichos temas son más apropiados en un estudio avanzado de estructuras de datos y programación de sistemas. Se han introducido las uniones sólo para presentar un cuadro completo de los tipos estructurados proporcionados por C++ y para familiarizarlo a usted con la idea general en caso de que encuentre uniones en otros programas de C++.

11.4 Abstracción de datos

A medida que el software que se desarrolla se hace más complejo, se diseñan al mismo tiempo algoritmos y estructuras de datos. Se avanza de la estructura de datos lógica o abstracta contemplada en el nivel superior a través del proceso de refinamiento hasta que se alcanza la codificación concreta en C++. Se han ilustrado dos formas de representar la estructura lógica de un registro de máquina en un inventario de almacén. La primera utiliza un registro en el cual se definieron (concretaron) todos los componentes al mismo tiempo. La segunda usa un registro jerárquico en el cual las fechas y estadísticas que describen la historia de una máquina se definieron en un registro de menor nivel.

Considérense de nuevo las dos formas en que se representa la estructura de datos lógicos.

```
// ***************** Versión 1 *****************

struct MachineRec
{
    int    idNumber;
    string description;
    float  failRate;
    int    lastServicedMonth;  // Suponga 1..12
    int    lastServicedDay;    // Suponga 1..31
    int    lastServicedYear;   // Suponga 1900..2050
    int    downDays;
    int    purchaseDateMonth;  // Suponga 1..12
    int    purchaseDateDay;    // Suponga 1..31
    int    purchaseDateYear;   // Suponga 1900..2050
    float  cost;
};

// ***************** Versión 2 *****************

struct Date
{
    int month;  // Suponga 1..12
    int day;    // Suponga 1..31
    int year;   // Suponga 1900..2050
};
struct Statistics
{
    float    failRate;
    Date     lastServiced;
    int      downDays;
};
struct MachineRec
{
    int            idNumber;
    string         description;
```

```
    Statistics     history;
    DateType       purchaseDate;
    float          cost;
};
```

¿Cuál de estas dos representaciones es mejor? La segunda es mejor por dos razones.

Primera, agrupa los elementos de modo lógico. Las estadísticas y las fechas son entidades dentro de sí mismas. Se podría desear una fecha o una historia de máquina en otra estructura de registro. Si se definen las fechas y estadísticas sólo dentro de `MachineRec` (como en la primera estructura), sería necesario definirlas de nuevo para cada estructura de datos que las necesite, de modo que se obtengan definiciones múltiples de la misma identidad lógica.

Segunda, los detalles de las entidades (estadísticas y fechas) son llevados a un nivel inferior en la segunda estructura. El principio de diferir detalles al nivel más bajo posible se debe aplicar para diseñar estructuras de datos, así como para diseñar algoritmos. Cómo se representa una historia de máquina o una fecha es irrelevante para nuestro concepto de registro de máquina, así que no es necesario especificar los detalles hasta que sea oportuno escribir algoritmos para manejar esos miembros.

Llevar los detalles de la implementación de un tipo de datos a un nivel inferior separa la descripción lógica de la implementación. Este concepto es análogo a la abstracción de control que se analizó en el capítulo 8. La separación de las propiedades lógicas de un tipo de datos de sus detalles de implementación se denomina **abstracción de datos**, que es un objetivo de programación efectiva y la base sobre la que se determinan tipos de datos abstractos. (En la siguiente sección se analiza el concepto de tipos de datos abstractos.)

Abstracción de datos Separación de las propiedades lógicas de un tipo de datos de su implementación.

Por último, todas las propiedades lógicas se deben definir en términos de los tipos de datos concretos y las rutinas escritas para manejarlos. Si la implementación está diseñada de modo apropiado, es posible usar las mismas rutinas para manejar la estructura en una amplia variedad de aplicaciones. Por ejemplo, si se tiene una rutina para comparar fechas, es posible usarla para comparar las fechas que representan los días en que se compró o dio mantenimiento al equipo, o las fechas que representan los cumpleaños de las personas. El concepto de diseñar una estructura de bajo nivel y escribir rutinas para manipularla es la base para los tipos `class` de C++, que se examinan más adelante en este capítulo.

11.5 Tipos de datos abstractos

Vivimos en un mundo complejo. Durante el transcurso de cada día, se bombardea de manera constante con información, hechos y detalles. Para arreglárselas con la complejidad, la mente humana se engarza en la *abstracción*; el acto de separar las cualidades esenciales de una idea u objeto de los detalles de cómo funciona o se compone.

Con la abstracción, se centra la atención en el *qué,* no el *cómo*. Por ejemplo, la comprensión acerca de los automóviles se basa en gran medida en la abstracción. La mayoría sabe *qué* hace el motor (impulsa el automóvil), pero pocos saben, o desean saber, con precisión *cómo* funciona internamente la máquina. La abstracción permite argumentar, pensar en y usar los automóviles sin tener que saber todo acerca de cómo funcionan.

En el mundo del diseño de software, se reconoce que la abstracción es una necesidad absoluta para manejar proyectos de software inmensos y complejos. En cursos introductorios de informática, los programas son, en general, pequeños (quizá 50 a 200 líneas de código) y comprensibles en su totalidad por una persona. Sin embargo, los grandes productos de software comercial compuestos de cientos de miles, incluso millones, de líneas de código no pueden ser diseñados, comprendidos o probados por completo sin usar la abstracción de varias formas. Para manejar la complejidad, los desarrolladores de software usan por lo regular dos técnicas de abstracción importantes: abstracción de control y abstracción de datos.

En el capítulo 8 se definió la abstracción de control como la separación de las propiedades lógicas de una acción de su implementación. Se participa en la abstracción de control siempre que se escribe una función que reduce un algoritmo complicado a una acción abstracta efectuada por una llamada de función. Al invocar una función de biblioteca, como en la expresión

```
4.6 + sqrt(x)
```

se depende sólo de la *especificación* de la función, una descripción escrita de lo que hace. Se puede usar la función sin necesidad de conocer su *implementación* (los algoritmos que consiguen el resultado). Al invocar la función sqrt, el programa es menos complejo porque están ausentes todos los detalles relacionados con el cálculo de raíces cuadradas.

Las técnicas de abstracción se aplican también a datos. Todo tipo de dato consta de un conjunto de valores (el dominio) junto con una colección de operaciones permisibles en éstos. En la sección precedente se describe la abstracción de datos como la separación de las propiedades lógicas de un tipo de datos de sus detalles de implementación. La abstracción de datos participa cuando se requiere un tipo de datos que no forma parte del lenguaje de programación. Se puede definir el nuevo tipo de datos como un **tipo de datos abstractos (TDA)** si se centra la atención sólo en sus propiedades lógicas y se posponen los detalles de su implementación.

Tipo de datos abstractos Tipo de datos cuyas propiedades (dominio y operaciones) son especificadas independientemente de cualquier implementación particular.

Como en el caso de la abstracción de control, un tipo de datos abstractos tiene tanto una especificación (el *qué*) como una implementación (el *cómo*). La especificación de un TDA describe las características de los valores de datos, así como el comportamiento de cada una de las operaciones en esos valores. El usuario del TDA necesita entender sólo la especificación, no la implementación, para usarlo. Enseguida se presenta una especificación muy informal de una lista de TDA:

TIPO
    IntList
DOMINIO
    Cada valor de IntList es una colección de hasta 100 números enteros separados.
OPERACIONES
    Insertar un elemento en la lista.
    Borrar un elemento de la lista.
    Buscar un elemento en la lista.
    Devolver la longitud actual de la lista.
    Clasificar la lista en orden ascendente.
    Imprimir la lista.

Observe la ausencia total de detalles de implementación. No se ha mencionado cómo se podrían guardar en realidad los datos en un programa o cómo se podrían realizar las operaciones. Ocultar los detalles de implementación reduce la complejidad para el usuario y también lo protege de cambios en la implementación.

A continuación se muestra la especificación para otro TDA, uno que tal vez sea útil para representar el tiempo en un programa.

TIPO
    Time
DOMINIO
    Cada valor de tiempo es un tiempo del día en forma de horas, minutos y segundos.
OPERACIONES
    Establecer el tiempo.
    Imprimir el tiempo.

Incrementar el tiempo en un segundo.
Comparar la igualdad de dos tiempos.
Determinar si un tiempo es "menor que" (viene antes) otro.

Representación de datos Forma concreta de datos usada para representar valores abstractos de un tipo de datos abstractos.

La especificación de un TDA define valores de datos abstractos y operaciones abstractas para el usuario. En última instancia, el TDA debe ser ejecutado en código de programa. Para poner en práctica un TDA, el programador debe hacer dos cosas:

1. Elegir una **representación de datos** concreta de los datos abstractos con tipos de datos que ya existen.
2. Poner en práctica cada una de las operaciones permisibles en términos de instrucciones de programa.

Para poner en práctica el TDA IntList, se podría usar una representación de datos concreta que consiste en dos elementos: una estructura de datos de 100 elementos (como un *arreglo*, tema del siguiente capítulo) y una variable `int` que sigue la pista de la longitud actual de la lista. Para ejecutar las operaciones de IntList, es necesario crear algoritmos con base en la representación de datos elegida. En los dos capítulos siguientes se analiza en detalle la estructura de datos del arreglo y su uso al poner en práctica TDA de listas.

Para implantar el TDA `Time`, se podrían usar tres variables `int` para la representación de datos (una para las horas, una para los minutos y una para los segundos). O se podrían usar tres cadenas como la representación de datos. La especificación del TDA no confina ninguna representación de datos particular. Siempre y cuando se satisfaga la especificación, se tiene la libertad de elegir entre representaciones alternativas de datos y sus algoritmos relacionados. La elección se podría basar en la eficiencia del tiempo (velocidad de implementación de los algoritmos); la eficiencia de espacio (uso económico del espacio de memoria), o la simplicidad y legibilidad de los algoritmos. Con el tiempo, adquirirá conocimiento y experiencia que lo ayude a decidir qué implementación es mejor para un contexto particular.

Fundamentos teóricos

Categorías de operaciones con tipos de datos abstractos

En general, las operaciones básicas relacionadas con un tipo de datos abstractos son de tres categorías: **constructors (constructores)**, **transformers (transformadores)** y **observers (observadores)**.

Una operación que crea un nuevo caso de un TDA (como una lista) es un constructor. Las operaciones que insertan un elemento en una lista y borran un elemento de una lista son transformadores. Una operación que toma una lista y la anexa al final de una segunda lista es también un transformador.

Constructor Operación que crea un nuevo caso (variable) de un TDA.

Transformador Operación que determina un nuevo valor del TDA, dados uno o más valores previos del tipo.

Observador Operación que permite observar el estado de un caso de un TDA sin cambiarlo.

Iterador Operación que permite procesar, uno a la vez, los componentes en un caso de un TDA.

Una función booleana que devuelve `true` si está vacía una lista y `false` si contiene algún componente, es un ejemplo de un observador. Una función booleana que prueba si determinado valor está en la lista es otro observador.

Algunas operaciones son combinaciones de observadores y constructores. Una operación que toma dos listas y las combina en una tercera lista (nueva) es tanto un observador (de las dos listas existentes) como un constructor (de la tercera lista).

Además de las tres categorías básicas de operaciones de TDA, en ocasiones se define una cuarta categoría: **iterators (iteradores)**.

Un ejemplo de un iterador es una operación que devuelve el primer elemento de una lista que es llamada inicialmente y devuelve el siguiente con cada llamada sucesiva.

11.6 Clases en C++

En capítulos anteriores se ha trabajado con valores de datos como cantidades pasivas en las que influirán funciones. En el capítulo 10 se vieron contadores de categorías como datos pasivos y se efectuaron operaciones como funciones que tomaron sus valores como parámetros. De manera similar, antes, en este capítulo, se estudió el registro de un alumno como una cantidad pasiva, con una struct como la representación de datos, y la implementación de la operación `Consistent` como una función que recibe una struct como un parámetro (véase la figura 11-6).

Esta separación de operaciones y datos no corresponde muy bien con la noción de un tipo de datos abstractos. Después de todo, un TDA consta de valores de datos como de operaciones en esos valores. Es preferible ver un TDA como la definición de una estructura de datos *activa*, una que combina datos y operaciones en una sola unidad cohesiva (véase la figura 11-7). C++ apoya este punto de vista debido a que proporciona un tipo estructurado, integrado, conocido como **clase**.

En la figura 11-2 se listan los cuatro tipos estructurados disponibles en el lenguaje C++: el arreglo, el registro (`struct`), la unión y la clase. Una clase es un tipo estructurado de modo específico para representar tipos de datos abstractos. Es similar a un registro, pero casi siempre se diseña de modo que sus componentes (**miembros de clase**) incluyan no sólo datos, sino también funciones que manejen esos datos.* Enseguida se muestra una declaración de clase en C++ que corresponde al TDA `Time` que se definió en la sección anterior:

Clase Tipo estructurado en un lenguaje de programación que se usa para representar un tipo de datos abstractos.

Miembro de clase Componente de una clase. Los miembros de clase pueden ser datos o funciones.

```
class Time
{
public:
    void Set( int, int, int );
    void Increment();
    void Write() const;
    bool Equal( Time ) const;
    bool LessThan( Time ) const;
```

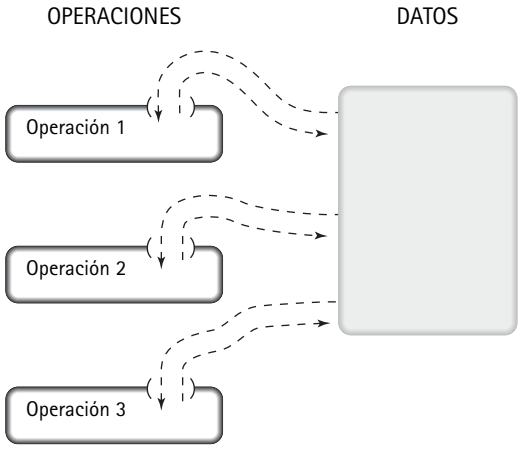


Figura 11-6 *Datos y operaciones como entidades separadas*

* Como se observó antes, en C++ una `struct` es casi idéntica a una `class`. Pero debido a su herencia de la construcción `struct` de C, la mayoría de los programadores usan `class` para poner en práctica un TDA, y limitan el uso de `struct` a aplicaciones donde se necesita un registro que no tiene operaciones relacionadas.

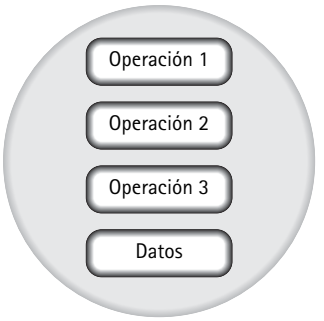


Figura 11–7 *Datos y operaciones enlazados en una sola unidad*

```
private:
    int hrs;
    int mins;
    int secs;
};
```

(Por ahora, debe ignorar la palabra `const` que aparece en algunos de los prototipos de función. Más adelante, en este capítulo, se explica este uso de `const`.)

La clase `Time` tiene ocho miembros: cinco funciones miembro (`Set`, `Increment`, `Write`, `Equal`, `LessThan`) y tres variables miembro (`hrs`, `mins`, `secs`). Como es de suponer, las tres variables miembro forman la representación de datos concreta para el TDA `Time`. Las cinco funciones miembro corresponden a las operaciones listadas para el TDA `Time`: establezca el tiempo (en horas, minutos y segundos pasados como argumentos a la función `Set`), incremente el tiempo en un segundo, imprima el tiempo, compare la igualdad de dos tiempos y determine si un tiempo es menor que otro. Si bien la función `Equal` compara la igualdad de dos variables `Time`, su lista de parámetros tiene sólo un parámetro (una variable `Time`). De manera similar, la función `LessThan` tiene sólo un parámetro, aun cuando se compara dos veces. Después se verá la razón.

Al igual que una declaración `struct`, la declaración de `Time` define un tipo de datos, pero no crea variables del tipo. Las variables de clase (en general denominadas **objetos de clase** o **casos de clase**) se determinan al usar declaraciones ordinarias de variables:

Objeto de clase (caso de clase) Variable de tipo `class`.

```
Time startTime;
Time endTime;
```

Cualquier software que declara y maneja objetos `Time` se denomina **cliente** de la clase.

Cliente Software que declara y maneja objetos de una clase particular.

Según se observa en la declaración precedente de la clase `Time`, se ven las palabras reservadas `public` y `private`, cada una seguida de dos puntos. Los datos o funciones, o ambos, declarados entre las palabras `public` y `private` constituyen la interfaz pública; los clientes pueden tener acceso directamente a estos miembros de clase. Los miembros de clase declarados después de la palabra `private` son considerados información privada y son inaccesibles para los clientes. Si el código cliente intenta entrar a un elemento privado, el compilador señala un error.

Sólo las funciones miembro de clase pueden tener acceso a miembros de clase privados. En la clase `Time`, sólo las funciones miembro `Set`, `Increment`, `Write`, `Equal` y `LessThan`, no el código cliente, pueden tener acceso a las variables privadas `hrs`, `mins` y `secs`. Dicha separación de miembros de clase en las partes pública y privada es un sello del diseño de TDA. Para conservar correctamente las propiedades de un TDA, un caso del TDA se debe manejar *sólo* a través de operaciones que forman la interfaz pública. Más adelante, en este capítulo, será necesario ampliar este tema.

Cuestiones de estilo

Declaración de miembros de clase públicos y privados

C++ no requiere que usted declare los miembros de clase públicos y privados en un orden fijo. Distintas variaciones son posibles.

Por omisión, los miembros de clase son privados; la palabra `public` se debe usar para "abrir" cualquier miembro para acceso público. Por tanto, las declaraciones de la clase Time se podrían escribir de la siguiente manera:

```
class Time
{
    int hrs;
    int mins;
    int secs;
public:
    void Set( int, int, int );
    void Increment();
    void Write() const;
    bool Equal( Time ) const;
    bool LessThan( Time ) const;
};
```

Por omisión, las variables `hrs`, `mins` y `secs` son privadas. La parte pública se extiende de la palabra `public` hasta el final de la declaración de clase.

Incluso con la parte privada ubicada primero, algunos programadores usan la palabra reservada `private` para ser lo más explícito posible:

```
class Time
{
private:
    int hrs;
    int mins;
    int secs;
public:
    void Set( int, int, int );
    void Increment();
    void Write() const;
    bool Equal( Time ) const;
    bool LessThan( Time ) const;
};
```

Se prefiere localizar la parte pública primero para centrar la atención en la interfaz pública y restar importancia a la representación de datos privados:

```
class Time
{
public:
    void Set( int, int, int );
    void Increment();
```

(continúa) ▼

Declaración de miembros de clase públicos y privados

```
    void Write() const;
    bool Equal( Time ) const;
    bool LessThan( Time ) const;
private:
    int hrs;
    int mins;
    int secs;
};
```

Este estilo se usa en el resto del libro.

Respecto a la accesibilidad publica contra la privada, ahora es posible describir de modo más completo la diferencia entre structs y clases de C++. C++ define una struct como una clase cuyos miembros son, por omisión, públicos. En contraste, los miembros de una clase son, en consecuencia, privados. Además, es más común usar sólo datos, no funciones, como miembros de una struct. Observe que *puede* declarar miembros de struct como privados y *puede* incluir funciones miembros en una struct, pero después podría usar también una clase. Por consiguiente, la mayoría de los programadores usan la struct de la manera tradicional en C, como una forma de representar una estructura de registro, y ponen en práctica los TDA sólo con clases.

Clases, objetos de clase y miembros de clase

Es importante hacer hincapié en que una clase es un tipo, no un objeto de datos. Como cualquier tipo, una clase es un modelo a partir del cual se crean (o *ejemplifican*) muchos objetos de ese tipo. Considere un tipo como un cortador de galletas y los objetos de ese tipo como las galletas.

Las declaraciones

```
Time time1;
Time time2;
```

crean dos objetos de la clase `Time`: `time1` y `time2`. Cada función tiene sus propias copias de `hrs`, `mins` y `secs`, los miembros de datos privados de la clase. En un determinado momento durante la implementación del programa, las copias de `time1` de `hrs`, `mins` y `secs` podrían contener los valores 5, 30 y 10, y las copias de `time2` podrían contener los valores 17, 58 y 2. La figura 11-8 representa los objetos de clase `time1` y `time2`.

(En realidad, el compilador de C++ no consume memoria al colocar copias de una función miembro, por ejemplo, `Increment`, en `time1` y `time2`. El compilador genera sólo una copia física de `Increment`, y cualquier objeto de clase ejecuta esta única copia de la función. No obstante, el diagrama de la figura 11-8 es una buena imagen mental de dos objetos de clase distintos.)

Asegúrese de tener clara la diferencia entre los términos *objeto de clase* y miembro de clase. En la figura 11-8 se ilustran dos objetos de clase `Time`, y cada objeto tiene ocho miembros.

Operaciones integradas en objetos de clase

Por muchas razones, las clases definidas por el programador son como tipos integrados. Usted puede declarar tantos objetos de una clase como quiera. Puede pasar objetos de clase como argumentos a funciones y devolverlos como valores de función. Como cualquier variable, un objeto de clase puede ser automático (creado cada vez que el control alcanza su declaración y destruido cuando termina el programa).

Por otras razones, C++ trata los structs y las clases de modo distinto de los tipos integrados. La mayoría de las operaciones integradas no se aplican a structs o clases. No es posible usar el operador `+` para sumar dos objetos `Time`, ni el operador `==` para comparar la igualdad de dos objetos `Time`.

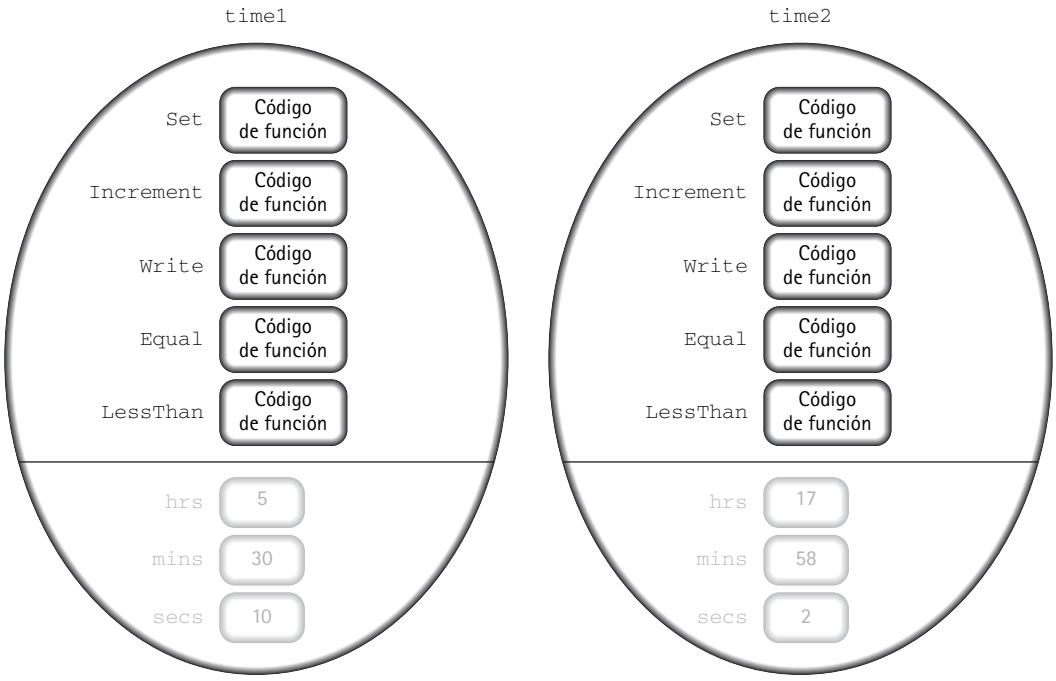


Figura 11-8 *Vista conceptual de dos objetos de clase*

Dos operaciones integradas que son válidas para objetos `struct` y de clase son la selección de miembro (.) y la asignación (=). Al igual que con los structs, es posible seleccionar un miembro de una clase por medio de la notación de punto. Es decir, se escribe el nombre del objeto de clase, luego un punto, después el nombre del miembro. La sentencia

```
time1.Increment();
```

invoca a la función `Increment` para el objeto `time1`, con la intención de añadir un segundo al tiempo guardado en `time1`. La otra operación integrada, asignación, realiza la asignación de agregación de un objeto de clase a otro con la siguiente semántica: si `x` y `y` son objetos de la misma clase, entonces la asignación `x = y` copia los miembros de datos de `y` a `x`. A continuación se presenta un fragmento de código cliente que demuestra la selección y asignación de miembros.

```
Time time1;
Time time2;
int  inputHrs;
int  inputMins;
int  inputSecs;

time1.Set(5, 20, 0);
// Afirmar: time1 corresponde a 5:20:0

cout << "Introduzca horas, minutos, segundos: ";
cin >> inputHrs >> inputMins >> inputSecs;
time2.Set(inputHrs, inputMins, inputSecs);

if (time1.LessThan(time2))
    DoSomething();
time2 = time1;                      // Asignación miembro a miembro
time2.Write();
// Afirmar: se ha producido 5:20:0
```

Ya se destacó que las funciones `Equal` y `LeesThan` tienen sólo un parámetro cada una, aun cuando comparan dos objetos `Time`. En la sentencia If del segmento de código anterior, se está comparando `time1` y `time2`. Debido a que `LeesThan` es una función de miembro de clase, se invoca al dar el nombre de un objeto de clase (`time1`), luego un punto, después el nombre de la función (`LeesThan`). Sólo queda un elemento sin especificar: el objeto de clase con el que `time1` se debe comparar (`time2`). Por tanto, la función `LeesThan` requiere un solo parámetro, no dos. Aquí tiene otra forma de explicarlo: si una función miembro de clase representa una operación binaria (dos operandos), el primer operando aparece a la izquierda del operador punto, y el segundo está en la lista de parámetros. (Para generalizar, una operación *n*-aria tiene *n* – 1 operandos en la lista de parámetros. Así, una operación unaria, como `Write` o `Increment` en la clase `Time`, tiene una lista de parámetros vacía.)

Además de la selección y asignación de miembros, algunos otros operadores integrados son válidos para los objetos de clase y structs. Dichos operadores se usan para manejar direcciones de memoria, por lo que se pospone su análisis para más adelante, en el libro. Por ahora, considere `.` y `=` como los únicos operadores integrados válidos.

Desde el comienzo se ha trabajado con las clases de C++ en un contexto particular: entrada y salida. El archivo de encabezado estándar `iostream` contiene las declaraciones de dos clases, `istream` y `ostream`, que manejan la I/O de un programa. La biblioteca estándar de C++ declara a cin y cout como objetos de estas clases:

```
istream cin;
ostream cout;
```

La clase `istream` tiene muchas funciones miembro, dos de las cuales, `get` e `ignore`, ya se han visto en sentencias como:

```
cin.get(someChar);
cin.ignore(200, '\n');
```

Como con cualquier objeto de clase de C++, la notación de punto se usa para seleccionar una función miembro particular a invocar.

Las clases de C++ se han usado también al ejecutar el archivo I/O. El encabezado de archivo `fstream` contiene declaraciones para las clases `ifstream` y `ofstream`. El código cliente

```
ifstream dataFile;

dataFile.open("input.dat");
```

declara un objeto de clase `ifstream` denominado `dataFile`, luego invoca la función miembro de clase `open` para intentar abrir un archivo `input.dat` para la entrada.

No se examinan en detalle las clases `istream`, `ostream`, `ifstream` y `ofstream` y todas sus funciones miembro. Estudiarlas rebasaría los objetivos del libro. Lo que importa reconocer es que las clases y objetos son fundamentales para toda actividad I/O en un programa de C++.

Alcance de clase

Ya se mencionó que los nombres de miembros deben ser únicos dentro de una struct. Además, en el capítulo 8 se mencionaron cuatro clases de alcance en C++: alcance local, alcance global, alcance de espacio de nombre y *alcance de clase*. El alcance de clase se aplica a los nombres de miembro dentro de structs, uniones y clases. Decir que un nombre de parámetro tiene alcance de clase significa que el nombre está enlazado a esa clase (o struct o unión). Si resulta que el mismo identificador se declara fuera de la clase, los dos identificadores no están relacionados. Considérese un ejemplo.

La clase `Time` tiene una función miembro de nombre `Write`. En el mismo programa, otra clase (como `SomeClass`) podría tener también una función miembro denominada `Write`. Además, el programa podría tener una función `Write` global que no se relaciona en absoluto con ninguna clase. Si el programa tiene sentencias como

```
Time      checkInTime;
SomeClass someObject;
int       n;
 ⋮
checkInTime.Write();
someObject.Write();
Write(n);
```

entonces el compilador de C++ no tiene problema para distinguir entre las tres funciones `Write`. En las dos primeras llamadas de función, la notación de punto denota la selección de miembro de clase. La primera sentencia invoca la función `Write` de la clase `Time`, y la segunda invoca la función `Write` de la clase `SomeClass`. La sentencia final no usa la notación de punto, así que el compilador sabe que la función llamada es la función global `Write`.

Ocultación de información

Desde el punto de vista conceptual, un objeto de clase tiene una pared invisible alrededor de él. Dicha pared, denominada **barrera de abstracción**, evita que el código cliente entre a datos privados y funciones. La barrera impide también que el objeto de clase tenga acceso a datos y funciones fuera del objeto. Esta barrera es una característica crítica de clases y tipos de datos abstractos.

Para que un objeto de clase comparta información con el mundo exterior (es decir, con clientes), debe haber un espacio en la barrera de abstracción. Dicho espacio es la interfaz pública, los miembros de clase declarados como `públicos`. La única forma en que un cliente puede modificar la información interna del objeto de clase es indirectamente, a través de las operaciones en la interfaz pública. Los ingenieros tienen un concepto similar denominado **caja negra**. Una caja negra es un módulo o dispositivo cuyos trabajos internos no están a la vista. El usuario de la caja negra depende sólo de la especificación escrita de *qué* hace ésta, no de *cómo* lo hace. El usuario conecta cables a la interfaz y supone que el módulo funciona de modo correcto al satisfacer la especificación (véase la figura 11-9).

Barrera de abstracción Pared invisible alrededor de un objeto de clase que encapsula los detalles de implementación. La pared puede ser traspasada sólo a través de la interfaz pública.

Caja negra Dispositivo eléctrico o mecánico cuyos trabajos internos no están a la vista.

Ocultación de información Encapsulación y ocultación de detalles de implementación para evitar que el usuario de una abstracción dependa de los detalles o los maneje de modo incorrecto.

En el diseño de software, el concepto de caja negra se denomina **ocultación de información**. La ocultación de información evita que el usuario de una clase tenga que conocer todos los detalles de la implementación. Asimismo, la ocultación de información asegura al encargado de una clase que el usuario no pueda entrar directamente a ningún código privado o datos y comprometa la exactitud de la implementación.

Usted ya está familiarizado con la encapsulación y ocultación de información. En el capítulo 7 se analizó la posibilidad de ocultar la implementación de una función en un archivo separado. En este capítulo se verá cómo ocultar las implementaciones de funciones miembros de clase colocándolas en archivos separados del código cliente.

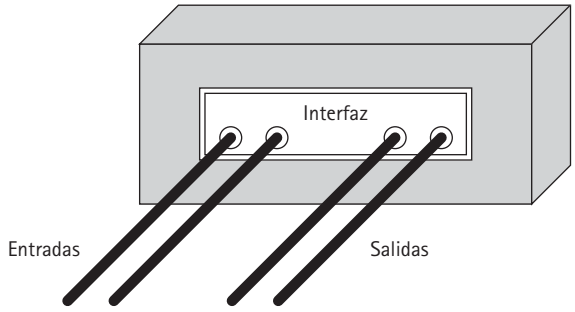


Figura 11–9 *Una caja negra*

El creador de una clase C++ es libre de elegir cuáles miembros son privados y cuáles son públicos. Sin embargo, hacer públicos los miembros de datos (como en una `struct`) permite al cliente inspeccionar y modificar los datos de modo directo. Debido a que la ocultación de información es fundamental para la abstracción de datos, la mayoría de las clases exhibe un patrón característico: la parte privada contiene datos y la parte pública las funciones que manejan los datos.

La clase `Time` ejemplifica esta organización. Los miembros de datos `hrs`, `mins` y `secs` son privados, así que el compilador prohíbe al cliente el acceso directo a estos miembros. La siguiente sentencia de cliente establece como resultado un error en tiempo de compilación:

```
Time checkInTime;

checkInTime.hrs = 9;     // Prohibido en código cliente
```

Como las funciones miembro de clase pueden tener acceso a datos privados, el creador de la clase puede ofrecer un producto confiable, sabiendo que el acceso externo a los datos privados es imposible. Si es aceptable permitir que el cliente *inspeccione* (pero no modifique) miembros de datos privados, una clase podría proporcionar funciones de observación. La clase `Time` tiene tres funciones de esa clase: `Write`, `Equal` y `LessThan`. Debido a que esas funciones de observador no están dedicadas a modificar los datos previos, se declaran con la palabra `const` seguida de la lista de parámetros:

```
void Write() const;
bool Equal( Time ) const;
bool LessThan( Time ) const;
```

C++ hace referencia a estas funciones como *funciones miembro* `const`. Dentro del cuerpo de una función miembro `const`, ocurre un error en tiempo de compilación si alguna sentencia intenta modificar un miembro de datos privado. Aunque aun no lo requiere el lenguaje, es buena práctica declarar como `const` las funciones miembro que no modifican datos privados.

11.7 Archivos de especificación e implementación

Un tipo de datos abstractos consta de dos partes: una especificación y una implementación. La especificación describe el comportamiento del tipo de datos sin hacer referencia a su implementación. La implementación crea una barrera de abstracción al ocultar la representación concreta de datos, así como el código para las operaciones.

La declaración de clase `Time` sirve como la especificación de `Time`. Esta declaración presenta la interfaz pública al usuario en la forma de prototipos de función. Para poner en práctica la clase `Time`, se deben proporcionar definiciones de funciones (declaraciones con cuerpos) para todas las funciones miembro.

En C++, es habitual (aunque no requerido) empaquetar la declaración y la implementación de clase en archivos separados. Un archivo —el *archivo de especificación*— que es un archivo de encabezado (`.h`) que contiene sólo la declaración de clase. El segundo archivo —*archivo de implementación*— contiene las definiciones de funciones para todas las funciones miembro de clase. Considérese primero el archivo de especificación.

Archivo de especificación

A continuación se muestra el archivo de especificación para la clase `Time`. En nuestro sistema de computadora, se ha nombrado el archivo `Time.h`. La declaración de clase es la misma que se presentó antes, con una excepción importante: se incluyen precondiciones y poscondiciones de función para especificar la semántica de las funciones miembro con la mayor claridad posible para el usuario de la clase.

```
//****************************************************************
// ARCHIVO DE ESPECIFICACIÓN (Time.h)
// Este archivo proporciona la especificación
// de un tipo de datos abstractos Time
//****************************************************************

class Time
{
public:
    void Set( /* in */ int hours,
              /* in */ int minutes,
              /* in */ int seconds );
        // Precondición:
        //     0 <= hours <= 23  &&  0 <= minutes <= 59
        //  && 0 <= seconds <= 59
        // Poscondición:
        //     El tiempo se establece de acuerdo con los parámetros entrantes
        // NOTA:
        //     Esta función deber ser llamada antes que
        //     cualquiera de las otras funciones miembro

    void Increment();
        // Precondición:
        //     La función Set ha sido invocada por lo menos una vez
        // Poscondición:
        //     El tiempo ha sido adelantado en un segundo, con
        //     23:59:59 que comienza de nuevo en 0:0:0

    void Write() const;
        // Precondición:
        //     The Set function has been invoked at least once
        // Poscondición:
        //     El tiempo ha sido generado en la forma HH:MM:SS

    bool Equal( /* in */ Time otherTime ) const;
        // Precondición:
        //     The Set function has been invoked at least once
        //     para este tiempo y otro tiempo (otherTime)
        // Poscondición:
        //     Valor de función == true, si este tiempo es igual a otherTime
        //                      == false, en caso contrario

    bool LessThan( /* in */ Time otherTime ) const;
        // Precondición:
        //     The Set function has been invoked at least once
        //     for both this time and otherTime
        //  && Este tiempo y otherTime representan tiempos
        //     en el mismo día
        // Poscondición:
        //     Valor de función == true, si este tiempo está antes
        //                              en el día que otherTime
        //                      == false, en caso contrario
private:
```

```
    int hrs;
    int mins;
    int secs;
};
```

Observe las precondiciones para las funciones `Increment`, `Write` y `LessThan`. Es responsabilidad del cliente fijar el tiempo antes de incrementarlo, imprimirlo o probarlo. Si el cliente no fija el tiempo, el efecto de cada una de estas funciones es indefinido.

En principio, un archivo de especificación no debe revelar ningún detalle de implementación al usuario de la clase. El archivo debe especificar *qué* hace cada función miembro sin revelar cómo lo hace. Sin embargo, como puede ver en la declaración de clase, hay un detalle de implementación que es visible al usuario: la representación de datos concreta del TDA que se lista en la parte privada. Sin embargo, la representación de datos aún es considerada información oculta en el sentido de que el compilador prohíbe al código cliente entrar de modo directo a los datos.

Archivo de implementación

El archivo de especificación (`.h`) para la clase `Time` contiene sólo la declaración de clase. El archivo de implementación debe proporcionar las definiciones de función para todas las funciones miembro de clase. En los comentarios de inicio del siguiente archivo de implementación, se documenta el nombre de archivo como `Time.cpp`. Es posible que su sistema use un sufijo de nombre de archivo distinto para los archivos de código fuente, tal vez `.c`, `.C` o bien `.cxx`.

Se recomienda que primero examine brevemente el código de C++ siguiente, sin preocuparse mucho de las nuevas características de lenguaje, como poner un prefijo al nombre de cada función con los símbolos

```
Time::
```

Inmediatamente después del código de programa se explican las nuevas características.

```
//*****************************************************************
// ARCHIVO DE IMPLEMENTACIÓN (Time.cpp)
// Este archivo pone en práctica las funciones miembro de Time
//*****************************************************************
#include "Time.h"
#include <iostream>

using namespace std;
// Miembros de clase privados:
//     int hrs;
//     int mins;
//     int secs;

//*****************************************************************

void Time::Set( /* in */ int hours,
                /* in */ int minutes,
                /* in */ int seconds )

// Precondición:
//     0 <= hours <= 23  &&  0 <= minutes <= 59
//  && 0 <= seconds <= 59
// Poscondición:
//     hrs == hours  &&  mins == minutes  &&  secs == seconds
// NOTA:
```

```
//     Esta función DEBE ser llamada antes que
//     cualquier otra función miembro

{
    hrs = hours;
    mins = minutes;
    secs = seconds;
}

//****************************************************************

void Time::Increment()

// Precondición:
//     Se ha apelado a la función Set por lo menos una vez
// Poscondición:
//     El tiempo ha sido adelantado un segundo, con
//     23:59:59 que vuelve a comenzar en 0:0:0

{
    secs++;
    if (secs > 59)
    {
        secs = 0;
        mins++;
        if (mins > 59)
        {
            mins = 0;
            hrs++;
            if (hrs > 23)
                hrs = 0;
        }
    }
}

//****************************************************************

void Time::Write() const

// Precondición:
//     Se ha invocado la funcion Set por lo menos una vez
// Poscondición:
//     El tiempo ha sido generado en la forma HH:MM:SS

{
    if (hrs < 10)
        cout << '0';
    cout << hrs << ':';
    if (mins < 10)
        cout << '0';
    cout << mins << ':';
    if (secs < 10)
        cout << '0';
    cout << secs;
}
```

```
//*****************************************************************

bool Time::Equal( /* in */ Time otherTime ) const

// Precondición:
//     The Set function has been invoked at least once
//     para este tiempo y otherTime
// Poscondición:
//     Valor de función == true, si este tiempo es igual a otherTime
//                      == false, en caso contrario

{
    return (hrs == otherTime.hrs && mins == otherTime.mins &&
            secs == otherTime.secs);
}

//*****************************************************************

bool Time::LessThan( /* in */ Time otherTime ) const
// Precondición:
//     Se ha invocado la función Set por lo menos una vez
//     para este tiempo y otherTime
//  && Este tiempo y otherTime representan tiempos en el
//     mismo día
// Poscondición:
//     Valor de función == true, si este tiempo está antes
//                              que otherTime en el día
//                      == false, en caso contrario
{
    return (hrs < otherTime.hrs ||
            hrs == otherTime.hrs && mins < otherTime.mins ||
            hrs == otherTime.hrs && mins == otherTime.mins
                                 && secs < otherTime.secs);
}
```

Este archivo de implementación demuestra varios puntos importantes.

1. El archivo comienza con la directiva de preprocesador

   ```
   #include "Time.h"
   ```

   Tanto el archivo de implementación como el código cliente deben incluir (`#include`) el archivo de especificación. En la figura 11-10 se ilustra este acceso compartido en el archivo de especificación. Esta coparticipación garantiza que todas las declaraciones relacionadas con una abstracción son congruentes. Es decir, tanto `client.cpp` como `Time.cpp` deben hacer referencia a la misma declaración de la clase `Time` localizada en `Time.h`.
2. Cerca de la parte superior del archivo de implementación se ha incluido un comentario que expresa de nuevo los miembros privados de la clase `Time`.

   ```
   // Miembros de clase privados:
   //     int hrs;
   //     int mins;
   //     int secs;
   ```

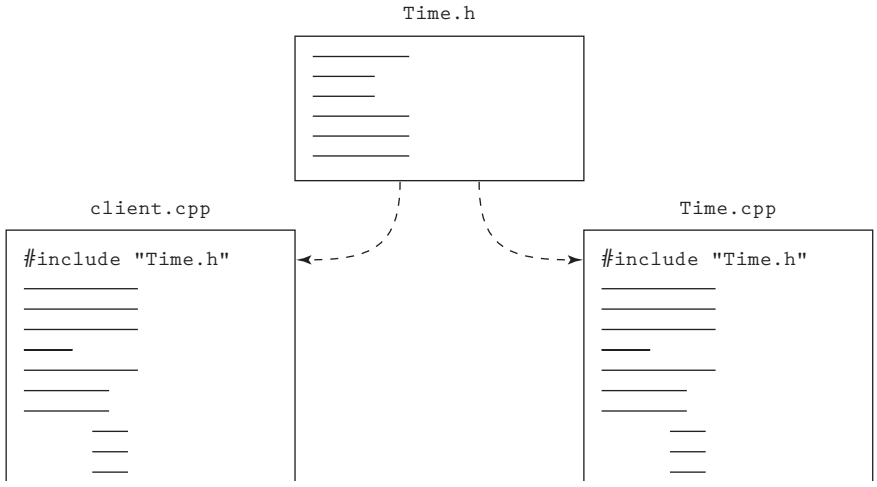


Figura 11-10 *Acceso compartido a un archivo de especificación*

Este comentario recuerda al lector que cualesquiera referencias a estos identificadores son referencias a miembros de clase privados.

3. En el encabezado de cada definición de función, el nombre de clase (`Time`) y el operador de resolución de alcance (`::`) prefijan al nombre de la función miembro. Como se explicó, es posible que varias clases tengan funciones miembro con el mismo nombre, por ejemplo, `Write`. Además, podría haber una función global `Write` que no sea miembro de ninguna clase. El operador de resolución de alcance elimina cualquier incertidumbre respecto a qué función particular se define.
4. Aunque los clientes de una clase deben usar el operador punto para referirse a los miembros de clase [como `startTime.Write()`], los miembros de una clase se refieren entre sí directamente sin usar la notación punto. Al examinar los cuerpos de las funciones `Set` e `Increment`, es posible observar que las sentencias hacen referencia directa a las variables miembro hrs, mins y secs sin usar el operador punto.

   Una excepción a esta regla ocurre cuando una función miembro maneja dos o más objetos de clase. Considere la función `Equal`. Suponga que el código cliente tiene dos objetos de clase, `startTime` y `endTime`, y que usa la sentencia

```
if (startTime.Equal(endTime))
    ⋮
```

   En tiempo de ejecución, el objeto `startTime` es el objeto para el cual se invoca la función `Equal`. En el cuerpo de la función `Equal`, la expresión relacional

```
hrs == otherTime.hrs
```

   se refiere a los miembros de clase de dos objetos de clase distintos. El identificador simple hrs se refiere al miembro hrs del objeto de clase para el cual se invoca la función (es decir, `startTime`). La expresión `otherTime.hrs` se refiere al miembro hrs del objeto de clase que es pasado al parámetro, `otherTime`, como un argumento de función: `endTime`.
5. `Write`, `Equal` y `LessThan` son funciones observadoras; no modifican los datos privados de la clase. Debido a que éstas han sido declaradas como funciones miembro `const`, el compilador evita que asignen nuevos valores a los datos privados. El uso de `const` es una ayuda para el usuario de la clase (como una señal visual de que esta función no modifica ningún dato privado) y una ayuda para el encargado de implementar la clase (como una forma de evitar la modificación accidental de los datos). Observe que la palabra `const` debe aparecer tanto en la función prototipo (en la declaración de clase) como en el encabezado de la definición de función.

Compilación y enlace de un programa multiarchivo

Ahora que se ha creado un archivo de especificación y un archivo de implementación para la clase `Time`, ¿cómo se usan estos archivos en los programas? Primero examinemos el concepto de *compilación separada* de archivos de código fuente.

En capítulos anteriores se ha hecho referencia al concepto de programa multiarchivo, un programa dividido en varios archivos que contiene código fuente. En C++ es posible compilar cada uno de dichos archivos por separado y en diferentes tiempos. El compilador traduce cada archivo de código fuente en un archivo de código objeto. En la figura 11-11 se muestra un programa multiarchivo que consta de los archivos de código fuente `myprog.cpp`, `file2.cpp` y `file3.cpp`. Cada uno de estos archivos se puede compilar de modo independiente y obtener archivos de código objeto `myprog.obj`, `file2.obj` y `file3.obj`. Aunque cada archivo .obj contiene código de lenguaje de máquina, aún no está en forma ejecutable. El programa ligador del sistema reúne el código objeto para formar un archivo de programa ejecutable. (En la figura 11-11 se usan los sufijos de nombre de archivo `.cpp`, `.obj` y `.exe`. Es posible que su sistema de C++ use convenciones distintas de nombre de archivo.)

Los archivos como `file2.cpp` y `file3.cpp` contienen por lo común definiciones de función para funciones que son llamadas por el código en `myprog.cpp`. Un beneficio importante de la compilación separada es que modificar el código en sólo un archivo requiere volver a compilar únicamente ese archivo. El nuevo archivo `.obj` se liga de nuevo con los otros archivos `.obj` existentes. Por supuesto, si una modificación de un archivo afecta el código de otro (por ejemplo, cambiar la interfaz de una función al alterar el número de tipos de datos de los parámetros de función), entonces los archivos afectados también necesitan ser modificados y recompilados.

De regreso a la clase `Time`, supongamos que se ha usado el editor del sistema para crear los archivos `Time.h` y `Time.cpp`. Ahora se puede compilar `Time.cpp` en código objeto. Si se está trabajando en la línea de comandos del sistema operativo, se usa un comando similar para lo siguiente:

```
cc -c Time.cpp
```

En este ejemplo, se supone que `cc` es el nombre del comando que invoca al compilador de C++ o al ligador, o a ambos, lo que depende de varias opciones dadas en la línea de comandos. La opción de línea de comandos `-c` significa, en muchos sistemas, "compilar pero no ligar". En otras palabras, este comando produce un archivo de código objeto, por ejemplo, `Time.obj`, pero no intenta ligar este archivo con ningún otro.

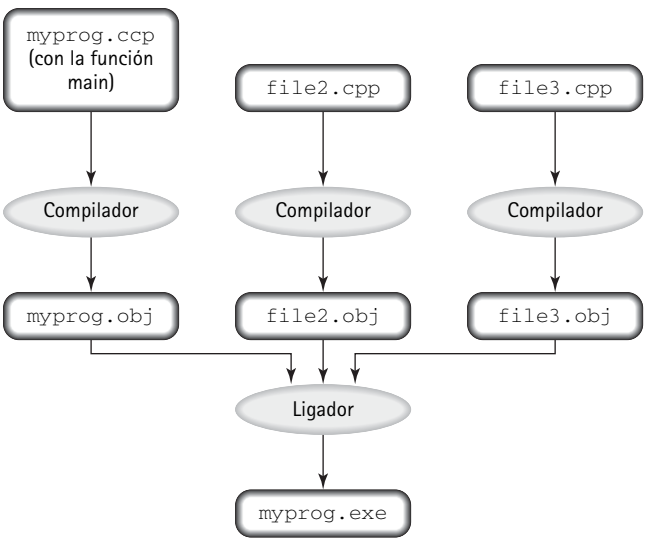


Figura 11-11 *Compilación separada y ligamiento*

Un programador que desea usar la clase `Time` escribe un código que incluye (`#include`) el archivo `Time.h`, después declara y usa objetos `Time`:

```
#include "Time.h"
 ⋮
Time appointment;

appointment.Set(15, 30, 0);
appointment.Write();
 ⋮
```

Si este código cliente está en un archivo llamado `diary.cpp`, un comando del sistema operativo como

```
cc diary.cpp Time.obj
```

compila el programa cliente en código objeto, liga éste con `Time.obj` y produce un programa ejecutable (véase la figura 11-12).

La mecánica de compilar, ligar y ejecutar varía de una computadora a otra. En los ejemplos que usan el comando cc suponga que está trabajando en la línea de comandos del sistema operativo. Muchos sistemas de C++ proporcionan un *ambiente integrado*, un programa que reúne al editor, compilador y ligador en un solo paquete. Los ambientes integrados lo regresan al editor cuando ocurre un error en tiempo de compilación o en tiempo de ligado, identificando el lugar del error. Algunos ambientes integrados manejan también *archivos de proyecto*. Éstos contienen información acerca de los archivos que constituyen un programa multiarchivo. Con los archivos de proyecto, el sistema recompila o religa de manera automática cualquier archivo que haya caducado como resultado de cambios en otros archivos del programa.

Cualquiera que sea el ambiente que use, el de la línea de comandos o uno integrado, el proceso global es el mismo: se compila cada uno de los archivos de código fuente en código objeto, se ligan los archivos de objeto en un programa ejecutable y luego se ejecuta el programa.

Antes de dejar el tema de los programas multiarchivo, se hace hincapié en un punto importante. En relación con la figura 11-12, los archivos `Time.h` y `Time.obj` deben estar disponibles para otros usuarios de la clase `Time`. El usuario necesita examinar `Time.h` para ver qué hacen los objetos `Time` y cómo usarlos. El usuario también debe ser capaz de ligar su programa con `Time.obj` para producir

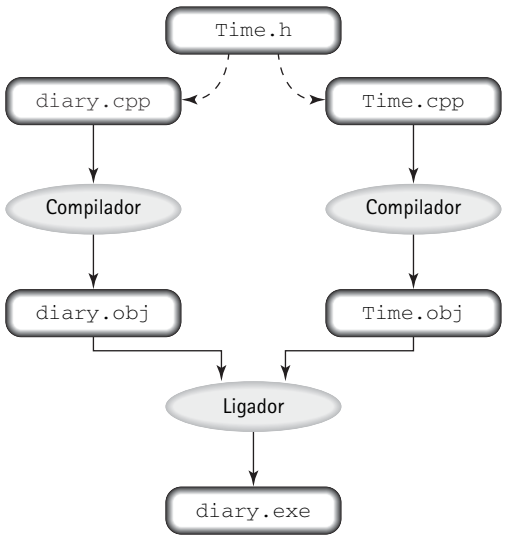


Figura 11-12 *Ligado con el archivo de implementación* `Time`

un programa ejecutable. Pero el usuario *no* necesita ver `Time.cpp`. La implementación de `Time` debe considerarse como una caja negra. El propósito principal de la abstracción es simplificar el trabajo del programador reduciendo la complejidad. Los usuarios de una abstracción no deben tener que examinar su implementación para aprender cómo usarla, ni deben escribir programas que dependan de los detalles de implementación. En el último caso, ningún cambio en la implementación podría "descomponer" los programas del usuario. En el capítulo 7, en el recuadro de Consejo práctico de ingeniería de software titulado "Ocultación conceptual contra física de una implementación de función" se analizaron los riesgos de escribir un código que depende de los detalles de implementación.

11.8 Inicialización garantizada con constructores de clases

La clase `Time` que se ha estado analizando tiene una desventaja. Depende del cliente para invocar la función `Set` antes de llamar a cualquier otra función miembro. Por ejemplo, la precondición de la función `Increment` es

```
// Precondición:
//     Se ha invocado la función Set por lo menos una vez
```

Si el cliente no apela primero a la función Set, esta precondición es falsa y se rompe el contrato entre el cliente y la implementación de la función. Como las clases casi siempre encapsulan datos, el creador de una clase no debe depender del usuario para inicializar los datos. Si el usuario olvida hacerlo podrían ocurrir resultados desagradables.

C++ proporciona un mecanismo, denominado *constructor de clase*, para garantizar la inicialización de un objeto de clase. Un constructor es una función miembro que se invoca de modo implícito siempre que se determina un objeto de clase.

Una función constructora tiene un nombre inusual: el de la clase misma. Se cambiará la clase `Time` añadiendo dos constructores de clase:

```
class Time
{
public:
    void Set( int, int, int );
    void Increment();
    void Write() const;
    bool Equal( Time ) const;
    bool LessThan( Time ) const;
    Time( int, int, int );          // Constructor
    Time();                         // Constructor
private:
    int hrs;
    int mins;
    int secs;
};
```

Esta declaración de clase incluye dos constructores, diferenciados por sus listas de parámetros. El primero tiene tres parámetros `int` que, como se verá, se emplean para inicializar los datos privados cuando se crea un objeto de clase. El segundo constructor no tiene parámetros e inicializa el tiempo en algún valor por defecto, como 00:00:00. Un constructor sin parámetros se conoce en C++ como *constructor por omisión*.

Las declaraciones de constructor son únicas en dos sentidos. Primero, como se mencionó, el nombre de la función es el mismo que el de la clase. Segundo, se omite el tipo de datos de la función. La razón es que un constructor no puede devolver un valor de función. Su propósito es sólo inicializar los datos privados de un objeto de clase.

En el archivo de implementación, las definiciones de función para dos constructores `Time` podrían parecerse a lo siguiente:

```
//*****************************************************************

Time::Time( /* in */ int initHrs,
            /* in */ int initMins,
            /* in */ int initSecs )
// Constructor

// Precondición:
//     0 <= initHrs <= 23  &&  0 <= initMins <= 59
//  && 0 <= initSecs <= 59
// Poscondición:
//     hrs == initHrs  &&  mins == initMins  &&  secs == initSecs
{
    hrs = initHrs;
    mins = initMins;
    secs = initSecs;
}

//*****************************************************************

Time::Time()

// Constructor por omisión

// Poscondición:
//     hrs == 0  &&  mins == 0  &&  secs == 0

{
    hrs = 0;
    mins = 0;
    secs = 0;
}
```

Invocación de un constructor

Aunque un constructor es un miembro de una clase, nunca se invoca con la notación de punto. Un constructor se invoca de manera automática siempre que se crea un objeto de clase. La declaración cliente

```
Time lectureTime(10, 30, 0);
```

incluye una lista de argumentos a la derecha del nombre del objeto de clase que es declarado. Cuando dicha declaración se encuentra en tiempo de ejecución, el primer constructor (parametrizado) se invoca de modo automático e inicializa los datos privados de `lectureTime` en el tiempo 10:30:00. La declaración cliente

```
Time startTime;
```

no tiene lista de argumentos después del identificador `startTime`. El constructor por omisión (sin parámetros) se invoca de modo implícito, e inicializa los datos privados de `startTime` en el tiempo 00:00:00.

Recuerde que una declaración en C++ es una sentencia genuina y puede aparecer en cualquier parte entre las sentencias ejecutables. Colocar las declaraciones entre sentencias ejecutables es muy útil al crear objetos de clase cuyos valores iniciales no se conocen hasta el momento de la ejecución. Enseguida se muestra un ejemplo:

```
cout << "Introduzca la hora de la cita en horas, minutos y segundos: ";
cin >> hours >> minutes >> seconds;

Time appointmentTime(hours, minutes, seconds);
cout << "La hora de la cita es ";
appointmentTime.Write();
 ⋮
```

Especificación revisada y archivos de implementación para `Time`

Al incluir constructores para la clase `Time`, se asegura que cada objeto de clase se inicializa antes de cualquier llamada posterior a las funciones miembro de clase. Uno de los constructores permite al código cliente especificar un tiempo inicial; el otro crea un tiempo inicial de 00:00:00 si el cliente no especifica un tiempo. Como resultado de estos constructores, es *imposible* que un objeto `Time` esté en un estado no inicializado después de que se crea. Como resultado, se puede borrar del archivo de especificación `Time` la advertencia de llamar a `Set` antes que a cualquier otra de las funciones miembro. También es posible eliminar todas las precondiciones de función que requieren que Set sea llamado previamente. Enseguida se muestra el archivo de especificación `Time` revisado:

```
//*****************************************************************
// ARCHIVO DE ESPECIFICACIÓN (Time.h)
// Este archivo proporciona la especificación
// de un tipo de datos abstractos Time
//*****************************************************************

class Time
{
public:
    void Set( /* in */ int hours,
              /* in */ int minutes,
              /* in */ int seconds );
        // Precondición:
        //     0 <= hours <= 23  &&  0 <= minutes <= 59
        //  && 0 <= seconds <= 59
        // Poscondición:
        //     El tiempo se fija de acuerdo con los parámetros entrantes

    void Increment();
        // Postcondition:
        //     El tiempo ha sido adelantado un segundo, como
        //     23:59:59 que vuelve a empezar en 00:00:00

    void Write() const;
        // Poscondición:
        //     El tiempo ha sido generado en la forma HH:MM:SS

    bool Equal( /* in */ Time otherTime ) const;
        // Poscondición:
```

```
        //     Valor de función == true, si este tiempo es igual a
        //                      == false, en caso contrario

    bool LessThan( /* in */ Time otherTime ) const;
        // Precondición:
        //     Este tiempo y otherTime representan tiempos en el
        //     mismo día
        // Poscondición:
        //     Valor de función == true, si este tiempo está antes
        //                              en el día que
        //                      == false, en caso contrario

    Time( /* in */ int initHrs,
          /* in */ int initMins,
          /* in */ int initSecs );
        // Precondición:
        //     0 <= initHrs <= 23  &&  0 <= initMins <= 59
        //  && 0 <= initSecs <= 59
        // Poscondición:
        //     Se construye el objeto de clase
        //  && El tiempo se fija de acuerdo con los parámetros entrantes

    Time();
        // Poscondición:
        //     Se construye el objeto de clase  &&  Time es 00:00:00
private:
    int hrs;
    int mins;
    int secs;
};
```

Para ahorrar espacio, no se incluye el archivo de implementación revisado. Los únicos cambios son:

1. La inclusión de las definiciones de función para los dos constructores de clase que se presentaron antes.
2. La eliminación de las precondiciones de función que expresan que la función `Set` debe ser invocada antes.

En este punto se podría preguntar si es necesaria la función `Set`. Después de todo, tanto la función `Set` como el constructor parametrizado al parecer hacen lo mismo, fijan el tiempo de acuerdo con los valores pasados como argumentos, y las ejecuciones de las dos funciones son, en esencia, idénticas. La diferencia es que `Set` puede ser invocada para un objeto de clase existente siempre y con tanta frecuencia como se desee, mientras que el constructor parametrizado es invocado sólo una vez, al momento en que se determina el objeto de clase. Por tanto, se retiene la función `Set` para proporcionar la máxima flexibilidad a los clientes de la clase.

El siguiente es un programa cliente completo que invoca todas las funciones miembro `Time`. Observe que la función main comienza con la creación de dos objetos `Time`, uno usando el constructor parametrizado y el otro usando el constructor por omisión. El resultado se muestra después del programa.

```
//*************************************************************
// Programa TimeClient
// Éste es un cliente muy simple de la clase Time
```

```
//*********************************************************************
#include <iostream>
#include "Time.h"   // Para la clase Time

using namespace std;

int main()
{
    Time time1(5, 30, 0);
    Time time2;
    int  loopCount;

    cout << "time1: ";
    time1.Write();
    cout << "  time2: ";
    time2.Write();
    cout << endl;
    if (time1.Equal(time2))
        cout << "Los tiempos son iguales" << endl;
    else
        cout << "Los tiempos NO son iguales" << endl;

    time2 = time1;
    cout << "time1: ";
    time1.Write();
    cout << "  time2: ";
    time2.Write();
    cout << endl;

    if (time1.Equal(time2))
        cout << "Los tiempos son iguales" << endl;
    else
        cout << "Los tiempos NO son iguales" << endl;

    time2.Increment();
    cout << "Nuevo time2: ";
    time2.Write();
    cout << endl;
    if (time1.LessThan(time2))
        cout << "time1 es menor que time2" << endl;
    else
        cout << "time1 NO es menor que time2" << endl;

    if (time2.LessThan(time1))
        cout << "time2 es menor que time1" << endl;
    else
        cout << "time2 NO es menor que time1" << endl;

    time1.Set(23, 59, 55);
    cout << "Incrementar time1 desde 23:59:55:" << endl;
    for (loopCount = 1; loopCount <= 10; loopCount++)
    {
        time1.Write();
        cout << ' ';
```

```
        time1.Increment();
    }
    cout << endl;
    return 0;
}
```

El resultado de ejecutar el programa TimeClient es como sigue.

```
time1: 05:30:00  time2: 00:00:00
Los tiempos NO son iguales
time1: 05:30:00  time2: 05:30:00
Los tiempos son iguales
New time2: 05:30:01
time1 es menor que time2
time2 NO es menor que time1
Incrementar time1 desde 23:59:55:
23:59:55 23:59:56 23:59:57 23:59:58 23:59:59 00:00:00 00:00:01
00:00:02 00:00:03 00:00:04
```

Directrices para usar constructores de clase

La clase es una característica esencial del lenguaje para crear tipos de datos abstractos en C++. El mecanismo de clase es una herramienta de diseño poderosa, pero junto con este poder vienen las reglas para usar las clases de manera correcta.

C++ tiene algunas reglas intrincadas respecto al uso de constructores, muchas de las cuales se relacionan con características del lenguaje que aún no se han explicado. A continuación se muestran algunas normas pertinentes en este punto.

1. Un constructor no puede devolver un valor de función, por tanto la función se declara sin un tipo de valor de retorno.
2. Una clase puede proporcionar varios constructores. Cuando se declara un objeto de clase, el compilador elige el constructor apropiado de acuerdo con el número y tipos de datos de los argumentos al constructor.
3. Los argumentos para un constructor se pasan colocando la lista de argumentos inmediatamente después del nombre del objeto de clase que es declarado:

   ```
   SomeClass anObject(arg1, arg2);
   ```

4. Si un objeto de clase se declara sin una lista de argumentos, como en la sentencia

   ```
   SomeClass anObject;
   ```

   entonces el efecto depende de qué constructores proporciona la clase (si existe).

   Si la clase no tiene constructores en absoluto, la memoria se asigna para anObject, pero sus miembros de datos privados están en un estado inicializado.

   Si la clase tiene constructores, entonces el constructor por omisión (sin parámetros) se invoca si hay uno. Si la clase tiene constructores pero ningún constructor por omisión, ocurre un error de sintaxis.

Antes de dejar el tema de los constructores, se da un repaso breve de otra función miembro especial apoyada por C++: el *destructor de clase*. Así como un constructor se invoca de modo explícito cuando se crea un objeto de clase, un destructor se invoca implícitamente cuando se destruye un objeto de clase; por ejemplo, cuando el control sale del bloque en que se declara un objeto local. Un destructor de clase se nombra del mismo modo que un constructor, excepto que el primer carácter es una tilde (~):

```
class SomeClass
{
public:
    ⋮
    SomeClass();     // Constructor
    ~SomeClass();    // Destructor
private:
    ⋮
};
```

En los siguientes capítulos no se usarán destructores; los tipos de clases que se escribirán no tienen necesidad de efectuar acciones especiales al momento en que se destruye un objeto de clase. En el capítulo 15 se analizan en detalle los destructores y se describen las situaciones en que es necesario usarlos.

Caso práctico de resolución de problemas

Nombre de tipo de datos abstractos

En el capítulo 4 se mostró un nombre en distintos formatos. En el capítulo 8, el programa Perfil de salud usó un nombre. Añadir un nombre a los programas IMC e Hipoteca tendría sentido. Con frecuencia los nombres son piezas de información necesarias. Se dejará esta duplicación de esfuerzo y se hará el trabajo definitivamente; se escribirá el código para apoyar un nombre como un tipo de datos abstractos.

El formato para este caso práctico es un poco distinto. Debido a que se está desarrollando sólo un componente de software, un TDA, y no un programa completo, se omiten las secciones de entrada y salida. En cambio, se incluyen dos secciones tituladas Especificación del TDA e Implementación del TDA.

PROBLEMA Diseñe y ejecute un tipo de datos abstractos para representar un nombre. Haga el dominio y las operaciones suficientemente generales para ser empleados en cualquier programa que necesite seguir la pista de un nombre. La especificación informal del TDA se muestra a continuación.

TIPO
    Name
DOMINIO
    Cada Name es un nombre en la forma de primer nombre, segundo nombre y apellido.
OPERACIONES
    Construir un caso Name
    Establecer un nombre
    Leer un nombre
    Inspeccionar el primer nombre
    Inspeccionar el segundo nombre
    Inspeccionar el apellido
    Inspeccionar la inicial del segundo nombre
    Comparar dos nombres para “before”, “same” y “after”

¿De dónde vienen estas operaciones? En el capítulo 14 se examina el diseño orientado a objetos y cómo elegir las operaciones, pero ahora suponga que éstas son una buena representación. Observe que no hay una operación para producir el nombre en algún formato particular. Como se permite que el usuario inspeccione las partes del nombre, éste puede combinar las partes para satisfacer su propósito.

ANÁLISIS Se crea el TDA Name en dos etapas: especificación, seguida de implementación. El resultado de la primera etapa es un archivo de especificación de C++ (.h) que contiene la declaración de una clase `Name`. El archivo debe describir al usuario la semántica precisa de cada operación de TDA. La especificación informal dada antes

sería inaceptable para el usuario del TDA. Las descripciones de las operaciones son muy imprecisas y ambiguas para que le sirvan al usuario.

La segunda etapa (implementación) requiere que *a*) se elija una representación de datos concreta para un nombre, y *b*) implementar cada una de las operaciones como una definición de función de C++. El resultado es un archivo de implementación de C++ que contiene dichas definiciones de función.

ESPEFICACIÓN DEL TDA El dominio de nuestro TDA es el conjunto de los nombres constituidos por el primer nombre, segundo nombre y apellido. Para representar el TDA como código de programa, se usa una clase de C++ llamada Name. Las operaciones de TDA se convierten en funciones miembro públicas de la clase. Ahora se especificarán las operaciones con más cuidado.

Construir un nuevo caso Name Para esta operación se usa un constructor por omisión de C++. El constructor para `Time` determina las horas, segundos y minutos en ceros. Los espacios serían el equivalente lógico para la clase `Name`.

Establecer el nombre El cliente debe proporcionar tres argumentos para esta operación: primer nombre, segundo nombre y apellido. Aunque todavía no se ha determinado una representación de datos concreta para un nombre, se debe decidir qué tipos de datos debe usar el cliente para estos argumentos. La elección lógica es para que cada valor sea representado por una cadena. Después de pensar al respecto, se comprende que esta operación podría ser codificada como un constructor parametrizado. Sin embargo, ¿qué sucede si se quiere cambiar un nombre (fijarlo) después que ya ha sido construido? Se proporcionarán dos operaciones que toman las tres partes del nombre como parámetros: una como constructor y otra como función vacía.

Leer un nombre Esta operación solicita y lee las tres partes del nombre desde el dispositivo de entrada estándar.

Inspeccionar los primeros nombres, segundos nombres y apellidos Las tres operaciones son operaciones de observador. Permiten el acceso al cliente, de modo indirecto, a los datos privados. En la clase Name se representan estas operaciones como funciones miembro de devolución de valor con los siguientes prototipos:

```
string first();
string middle();
string last();
```

¿Por qué son necesarias estas operaciones de observador? ¿Por qué no se permite simplemente que la representación de datos del nombre sea pública, en lugar de privada, para que el cliente tenga acceso directo a los valores? La respuesta es que se debe permitir al cliente inspeccionar *pero no modificar* estos valores. Si los datos son públicos, un cliente podría cambiar los datos y, por consiguiente, comprometer la integridad de un caso.

Inspeccionar la inicial del segundo nombre Esta operación requiere que la primera letra del segundo nombre sea extraída y devuelta. En el capítulo 4 se tuvo acceso a la inicial del segundo nombre como una cadena de un carácter. ¿Se debe devolverla como una cadena aquí o como una variable `char`? En realidad no importa, siempre que haya congruencia en el código y en la documentación. Se hará de esta función una función `char`, sólo para dar un ejemplo distinto.

Comparar dos nombres Esta observación compara dos nombres y determina si el primero va antes del segundo, si son lo mismo o el primero va después del segundo. Para indicar el resultado de la comparación, se define un tipo de enumeración con tres valores:

enum RelationType {BEFORE, SAME, AFTER};

Después se puede codificar la operación de comparación como una función miembro de clase que devuelve un valor del tipo:

```
RelationType ComparedTo( /* in */ Name otherName ) const;
```

Debido a que ésta es una función miembro de clase, el nombre que se compara con otherName es el objeto de clase para el cual se invoca la función miembro. Por ejemplo, el siguiente código cliente prueba si name1 va antes de name2.

```
Name name1;
Name name2;
 ⋮
if (name1.ComparedTo(name2) == BEFORE)
  DoSomething();
```

Ya casi estamos listos para escribir el archivo de especificación para la clase Name. Sin embargo, la declaración `class` requiere que se incluya la parte privada, las variables privadas que son la representación de datos concreta del TDA. Elegir una representación de datos concreta de modo apropiado pertenece a la fase de implementación del TDA, no a la fase de especificación. Pero, para satisfacer el requerimiento de declaración de clase de C++, ahora se elige una representación de datos. La representación más simple para un nombre son tres valores `string`: uno para el primer nombre, otro para el segundo nombre y otro para el apellido. Entonces, aquí tiene el archivo de especificación que contiene la declaración de la clase Name (junto con la declaración del tipo de enumeración `RelationType`).

```
//****************************************************************
// ARCHIVO DE ESPECIFICACIÓN (Name.h)
// Este archivo da la especificación del tipo de datos
// abstractos Name. Hay dos constructores: uno toma el primer nombre,
// segundo nombre y apellido como parámetros, y el segundo fija el primer
// nombre,segundo nombre y apellido en espacios en blanco
//****************************************************************

#include <iostream>
#include <string>

using namespace std;

enum RelationType{BEFORE, SAME, AFTER};

class Name
{
public:
    Name();
        // Constructor por omisión
        // Poscondición:
        //     el primer nombre, el segundo nombre y el apellido han sido
                fijados en espacios en blanco
    Name( /* in */ string firstName,
          /* in */ string middleName,
          /* in */ string lastName );
        // Constructor parametrizado
        // Poscondición:
        //     primero va el primer nombre, luego el segundo nombre Y
        //     por último el apellido

    void SetName( /* in */ string firstName,
                  /* in */ string middleName,
                  /* in */ string lastName );
    // Poscondición:
    //     primero va el primer nombre, luego el segundo nombre Y
    //     por último el apellido
```

```
    void ReadName();
    // Poscondición:
    //     Se solicita el primer nombre y se lee del dispositivo de entrada
    //     estándar

    string FirstName() const;
    // Poscondición:
    //     El valor de retorno es el primer nombre de esta persona

    string LastName() const;
    // Poscondición:
    //     El valor de retorno es el apellido de esta persona

    string MiddleName() const;
    // Poscondición:
    //     El valor de retorno es el segundo nombre de esta persona

    char MiddleInitial() const;
    // Poscondición:
    //     El valor de retorno es la inicial del segundo nombre de esta persona

    RelationType ComparedTo( /* in */ Name otherName ) const;
    // Poscondición:
    //   El valor de retorno es
    //          BEFORE, si este nombre va antes de
    //          SAME, si este nombre y otherName son lo mismo
    //          AFTER, si este nombre va después de

private:
    string first;             // Primer nombre de la persona
    string last;              // Apellido de la persona
    string middle;            // Segundo nombre de la persona

};
```

IMPLEMENTACIÓN DEL TDA Ya se ha elegido una representación de datos concretos para un nombre, mostrada en la especificación como las variables de cadena `first`, `middle` y `last`. Ahora se debe poner en práctica cada función miembro de clase, colocando las definiciones de función en un archivo de implementación de C++ denominado Name.cpp. Conforme se ponen en práctica las funciones miembro, se analizan también estrategias de prueba que permitan convencerse de que las implementaciones son correctas.

Constructores de clase FirstName, MiddleName y LastName. Las implementaciones de estas funciones son tan directas que no es necesaria explicación alguna.

Nombre (Entrada: firstName, middleName, lastName)

Establecer first en firstName
Establecer middle en middleName
Establecer last en lastName

Nombre()

Establecer first en “”
Establecer middle en “”
Establecer last en “”

Fijar el nombre (Entrada: firstName, middleName, lastName)

Fijar first en firstName
Fijar middle en middleName
Fijar last en lastName

FirstName()
Salida: valor de función

Devolver first

MiddleName()
Salida: valor de función

Devolver middle

LastName()
Salida: valor de función

Devolver last

ReadName()

Solicitar el nombre
Leer el primer nombre
Solicitar el segundo nombre
Leer el segundo nombre
Solicitar el apellido
Leer el apellido

PRUEBA Las funciones observadoras `FirstName`, `MiddleName` y `LastName` se pueden usar para comprobar que los constructores de clase, `ReadName` y `SetName` funcionan correctamente. El código

```
Name someName("Susan", "Margaret", "Smith");
cout << someName.FirstName() << ' ' << someName.MiddleName() << ' '
     << someName.LastName() << endl;
Name someName2();
cout << someName2.FirstName() << ' ' << someName2.MiddleName() << ' '
     << someName2.LastName() << endl;
someName2.SetName("Margaret", "Susan", "Smith");
cout << someName2.FirstName() << ' ' << someName2.MiddleName() << ' '
     << someName2.LastName() << endl;
```

debe imprimir tres líneas: `Susan Margaret Smith`, una línea con cinco espacios en blanco y una línea que contiene `Margaret Susan Smith`.

Función inicial Middle Es necesario extraer el primer carácter del segundo nombre. Eso es fácil. Es posible extraerlo de manera directa usando corchetes: `middle[0]`. Recuerde que el primer carácter en una cadena está en la posición cero.

Función ComparedTo Si se fuera a comparar dos nombres en nuestras cabezas, se observaría primero el segundo nombre. Si fueran distintos los segundos nombres, se sabría de inmediato cuál nombre va primero, si los segundos nombres fueran iguales, se considerarían los primeros nombres. Si éstos coinciden, se tendría que ver el apellido. Como suele suceder, es posible usar este algoritmo directamente en la función

ComparedTo(Entrada: otherName)
Salida: Function value

```
Si last < otherName.last
    Devolver BEFORE
Si last > otherName.last
    Devolver AFTER

// Los apellidos son iguales. Comparar los primeros nombres
Si first < otherName.first
    Devolver BEFORE
Si first > otherName.first
    Devolver AFTER

// Los apellidos y los primeros nombres son iguales. Comparar los segundos nombres
Si middle < otherName.middle
    Devolver BEFORE
Si middle < otherName.middle
    Devolver AFTER

// Los nombres son iguales
Devolver SAME
```

PRUEBA Al probar esta función, se debe asegurar que cada trayectoria se tome por lo menos una vez. En los ejercicios 2 y 3 de Seguimiento de caso práctico, se le pide diseñar datos de prueba para esta función y escribir un manejador que haga la prueba.

Enseguida se muestra el archivo de implementación que contiene definiciones de función para todas las operaciones de TDA:

```
//****************************************************************
// ARCHIVO DE IMPLEMENTACIÓN (Name.cpp)
// El archivo ejecuta las funciones miembro Name
//****************************************************************

#include "Name.h"
#include <iostream>

Name::Name()

// Constructor por omisión
// Poscondición:
//   first, middle, se han establecido en espacios en blanco
{
    first = " ";
    middle = " ";
    last = " ";
}
```

```
//*****************************************************************

Name::Name( /* in */ string firstName,     // Primer nombre
            /* in */ string middleName,   // Segundo nombre
            /* in */ string lastName )    // Apellido

// Constructor parametrizado
// Poscondición:
//     El primer nombre ha sido guardado en first; el segundo nombre se almacenó
//     en middle; el apellido se guardó en last

{
    first = firstName;                    // Parámetros de asignación
    last = lastName;
    middle = middleName;
}

//*****************************************************************

void Name::SetName( /* in */ string firstName,    // Primer nombre
                    /* in */ string middleName,  // Segundo nombre
                    /* in */ string lastName )   // Apellido

// Poscondición:
//     El primer nombre ha sido guardado en first; el segundo nombre se almacenó
//     en middle; el apellido se guardó en last

{
  first = firstName;                      // Parámetros de asignación
  last = lastName;
  middle = middleName;
}

//*****************************************************************

void Name::ReadName()

// Poscondición:
//     Nombre  avisar y leer del dispositivo de salida

{
    cout << "Enter first name: ";     // Prompt for first name
    cin >> first;                     // Obtener primer nombre
    cout << "Enter middle name: ";    // Aviso para segundo nombre
    cin >> middle;                    // Obtener segundo nombre
    cout << "Enter last name: ";      // Aviso de apellido
    cin >> last;                      // Obtener apellido
}

//*****************************************************************

string Name::FirstName() const
```

```
// Poscondición:
//   Valor es primero

{
    return first;
}


//****************************************************************

string Name::LastName() const

// Poscondición:
//   Valor es apellido

{
    return last;
}


//****************************************************************

string Name::MiddleName() const

// Poscondición:
//   Valor es segundo

{
    return middle;
}


//****************************************************************

char Name::MiddleInitial() const

// Poscondición:
//   Valor es segunda inicial

{
    return middle[0];
}

//****************************************************************

RelationType Name::ComparedTo( /* in */ Name otherName ) const

// Precondición:
//     Input parameter contains a valid name
// Poscondición:
//     Return value is
//          BEFORE, if this name comes before otherName
```

```
//          SAME, if this name and otherName are the same
//          AFTER, if this name is after otherName
{
    if (last < otherName.last)
        return BEFORE;
    else if (otherName.last < last)
        return AFTER;
    else if (first < otherName.first)
        return BEFORE;
    else if (otherName.first < first)
        return AFTER;
    else if (middle < otherName.middle)
        return BEFORE;
    else if (otherName.middle < middle)
        return AFTER;
    else
        return SAME;
 }
```

Un nombre es una entidad lógica para la cual ahora se ha desarrollado una implementación. Se ha diseñado, implementado y probado un TDA de nombre que podemos usar (o cualquier otro programador) siempre que se tenga un nombre como parte de los datos del programa. Si en el futuro se descubre que serían útiles más operaciones en un nombre, se pueden poner en práctica, probar o añadir al conjunto de operaciones de nombre.

Se ha dicho que la abstracción de datos es un principio importante de diseño de software. Lo que se ha hecho aquí es un ejemplo de abstracción de datos. De ahora en adelante, cuando un problema necesite un nombre, se puede detener la descomposición en el nivel lógico. No es necesario preocuparse respecto a implementar un nombre cada vez.

Prueba y depuración

Probar y depurar una clase de C++ equivale a probar y depurar cada función miembro de la clase. Todas las técnicas que ha aprendido, repasos de algoritmos, repasos de código, seguimientos a mano, manejadores de prueba, verificación de precondiciones y poscondiciones, depurador de sistema, función `assert` y salidas depuradas, pueden participar.

Considere cómo se podría probar la clase `Time` de este capítulo. Aquí tiene la declaración de clase, abreviada, excluyendo las precondiciones y poscondiciones de función:

```
class Time
{
public:
    void Set( /* in */ int hours,
              /* in */ int minutes,
              /* in */ int seconds );
        // Precondición: . . .
        // Poscondición: . . .

    void Increment();
        // Poscondición: . . .

    void Write() const;
```

```
        // Poscondición: . . .

    bool Equal( /* in */ Time otherTime ) const;
        // Poscondición: . . .

    bool LessThan( /* in */ Time otherTime ) const;
        // Precondición: . . .
        // Poscondición: . . .

    Time( /* in */ int initHrs,
          /* in */ int initMins,
          /* in */ int initSecs );
        // Precondición: . . .
        // Poscondición: . . .

    Time();
        // Poscondición: . . .
private:
    int hrs;
    int mins;
    int secs;
};
```

Para probar esta clase por completo, se debe probar cada una de las funciones miembro. Se recorrerá el proceso de probar sólo una de ellas: la función `Increment`.

Cuando se puso en práctica la función `Increment` se supone que se inició con un algoritmo de seudocódigo, se ejecutó un repaso de algoritmo y se tradujo el código en la siguiente función de C++.

```
void Time::Increment()

// Poscondición:
//     El tiempo ha avanzado un segundo, con
//     23:59:59 que se reinicia en 00:00:00

{
    secs++;
    if (secs > 59)
    {
        secs = 0;
        mins++;
        if (mins > 59)
        {
            mins = 0;
            hrs++;
            if (hrs > 23)
                hrs = 0;
        }
    }
}
```

Ahora se efectúa un repaso de código, comprobando que el código de C++ concuerda fielmente con el algoritmo de seudocódigo. En este punto (o antes, durante el repaso del algoritmo) se hace un seguimiento a mano para comprobar que la lógica es correcta.

Para el seguimiento a mano, se deben seleccionar valores de `hrs`, `mins` y `secs` que aseguren la cobertura del código. Para ejecutar cada trayectoria a través del flujo de control, se necesitan casos en los que ocurran las siguientes condiciones:

1. La primera condición If es falsa.
2. La primera condición if es verdadera y la segunda es falsa.
3. La primera condición If es verdadera, la segunda es verdadera y la tercera es falsa.
4. La primera condición If es verdadera, la segunda es verdadera y la tercera es verdadera.

A continuación se presenta una tabla que muestra valores de `hrs`, `mins` y `secs` que corresponden a estos cuatro casos. Para cada caso se escribe también lo que se espera serán los valores de las variables después de ejecutar el algoritmo.

	Valores iniciales			Resultados esperados		
Caso	`hrs`	`mins`	`secs`	`hrs`	`mins`	`secs`
1	10	5	30	10	5	31
2	4	6	59	4	7	0
3	13	59	59	14	0	0
4	23	59	59	0	0	0

Al usar los valores iniciales para cada caso, un seguimiento a mano del código confirma que el algoritmo produce los resultados deseados.

Por último, se escribe un manejador de prueba para la función `Increment`, sólo para asegurarse que la comprensión de la lógica del algoritmo es la misma que la de la computadora. Enseguida se muestra un posible manejador de prueba:

```
#include <iostream>
#include "Time.h"

using namespace std;

int main()
{
    Time time;
    int      hours;
    int      minutes;
    int      seconds;
    cout << "Introducir un tiempo (use horas < 0 para terminar): ";
    cin >> hours >> minutes >> seconds;
    while (hours >= 0)
    {
        time.Set(hours, minutes, seconds);
        time.Increment();
        cout << "El tiempo incrementado es ";
        time.Write();
        cout << endl;
        cout << "Introducir un tiempo (use horas < 0 para terminar): ";
        cin >> hours >> minutes >> seconds;
    }
    return 0;
}
```

Ahora se compila el manejador de prueba y `Time.cpp`, se liga con los dos archivos de objeto y se ejecuta el programa. Para datos de entrada, se suministran por lo menos los cuatro casos de prueba analizados anteriormente. El resultado del programa debe coincidir con los resultados deseados.

Ahora que se ha probado la función `Increment`, se pueden aplicar los mismos pasos a las funciones miembro de clase restantes. Se puede crear un manejador de prueba separado para cada función, o se puede escribir sólo un manejador que pruebe todas las funciones. La desventaja de escribir sólo un manejador es que diseñar distintas combinaciones de valores de entrada para probar con rapidez varias funciones a la vez puede ser complicado.

Antes de dejar el tema de probar una clase, se debe destacar un punto importante. Aun cuando se ha probado por completo una clase, todavía es posible que surjan errores. Considérense dos ejemplos con la clase `Time`. El primero es la sentencia cliente

```
time.Set(24, 0, 0);
```

El segundo ejemplo es la comparación

```
if (time1.LessThan(time2))
    ⋮
```

donde el programador pretende que `time1` sean las 11:00:00 en un miércoles y `time2` sea 1:20:00 en un jueves. (El resultado de la prueba es `false`, no `true`, como espera el programador.) ¿Ve el problema? En cada ejemplo, el cliente ha violado la precondición de función. La precondición de `Set` requiere que el primer argumento tenga un valor de 0 a 23. La precondición de `LessThan` requiere que los dos tiempos sean del mismo día, no de días distintos.

Si ha sido bien probada una clase y hay errores cuando el código cliente usa la clase, compruebe siempre las precondiciones de la función miembro. Se pueden perder muchas horas intentando depurar una función miembro de clase cuando, de hecho, la función es correcta. El error puede estar en el código cliente.

Sugerencias de prueba y depuración

1. Las declaraciones de los tipos `struct` y `class` deben terminar con punto y coma.
2. Asegúrese de especificar el selector de miembro total al hacer referencia a un componente de una variable struct u objeto de clase.
3. Evite usar tipos `struct` anónimos.
4. Respecto a los punto y coma, las declaraciones y definiciones de funciones miembros de clase se usan de igual manera que cualquier función de C++. El prototipo de función miembro, localizado en la declaración de clase, termina con un punto y coma. El encabezado de función, la parte de la definición de función que precede al cuerpo, no termina con un punto y coma.
5. Al implementar una función miembro de clase, no olvide poner un prefijo al nombre de función con el nombre de la clase y el operador de resolución de alcance (::)

   ```
   void Time::Increment()
   {
       ⋮
   }
   ```

6. Por ahora, las únicas operaciones integradas que se aplican a variables `struct` y objetos de clase son la selección (.) y asignación de miembro (=). Para efectuar otras operaciones, como comparar dos variables struct u objetos de clase, se debe tener acceso a cada uno de los componentes (en el caso de variables struct) o escribir funciones miembro de clase (en el caso de objetos de clase).
7. Si una función miembro de clase inspecciona pero no modifica los datos privados, es una buena idea convertirla en una función miembro `const`.
8. Una función miembro de clase no usa la notación de punto para tener acceso a miembros privados del objeto de clase para el cual se invocó la función. En contraste, una función miembro *debe* usar notación de punto para tener acceso a miembros privados de un objeto de clase que se pasa a éste como un argumento.

9. Para evitar errores causados por datos no inicializados, es buena práctica incluir siempre un constructor de clase al diseñar una clase.
10. Un constructor de clase se declara sin un tipo de valor de retorno y no puede devolver un valor de función.
11. Si un cliente de una clase tiene errores que parezcan estar relacionados con la clase, comience por comprobar las precondiciones de las funciones miembro de clase. Los errores pueden estar en el cliente, no en la clase.

Resumen

Además de poder crear tipos de datos automáticos definidos por el usuario, es posible crear tipos de datos estructurados. En un tipo de datos estructurados, se da un nombre a un grupo entero de componentes. Con muchos tipos estructurados, es posible tener acceso al grupo como un todo, o se puede entrar por separado a cada componente.

El registro es una estructura de datos para agrupar datos heterogéneos —elementos de datos que son de tipos distintos. El acceso a cada uno de los componentes de un registro es por nombre. En C++ se hace referencia a los registros como *estructuras* o structs. Se puede usar una variable struct para referirse al `struct` como un todo, o bien es posible usar un selector de miembro para tener acceso a cualquier miembro individual (componente) del `struct`. Los structs completos del mismo tipo pueden ser asignados directamente entre sí, pasados como argumentos o devueltos como valores de devolución de función. Sin embargo, la comparación de structs, se debe hacer miembro a miembro. Su lectura y escritura se debe hacer también miembro a miembro.

La abstracción de datos es una técnica poderosa para reducir la complejidad e incrementar la confiabilidad de programas. Separar las propiedades de un tipo de datos de los detalles de su implementación libera al usuario del tipo de tener que escribir un código que dependa de una implementación particular del tipo. Dicha separación asegura también al encargado de implementar el tipo que el código cliente no puede comprometer de manera accidental una implementación correcta.

Un tipo de datos abstractos (TDA) es un tipo cuya especificación va separada de su implementación. La especificación anuncia las propiedades abstractas del tipo. La implementación consta de (a) una representación de datos concreta y (b) las implementaciones de las operaciones de TDA. En C++, un TDA puede comprenderse mediante el mecanismo de clase. Los miembros de clase pueden ser designados como públicos o privados. Por lo común, los miembros privados son la representación de datos concretos del TDA y los miembros públicos son las funciones correspondientes a las operaciones de TDA.

Entre las funciones miembro públicas de una clase, con frecuencia el programador incluye uno o más constructores de clase —funciones invocadas de modo automático siempre que se determina un objeto de clase.

Separar la compilación de unidades de programa es primordial para la separación de la especificación de la implementación. La declaración de una clase de C++ se coloca, por lo regular, en un archivo de especificación (`.h`), y las implementaciones de las funciones miembro de clase residen en otro archivo: el archivo de implementación. El código cliente se compila por separado del archivo de implementación de clase, y los dos archivos de código objeto resultantes se ligan para formar un archivo ejecutable. A través de la compilación separada, el usuario de un TDA puede tratar al TDA como un componente listo para usarse sin saber siquiera cómo se pone en práctica.

Comprobación rápida

1. ¿De qué manera una `union` de C++ conserva espacio de memoria? (pp. 439-441.)
2. ¿Cómo se crea una implementación a partir de una especificación para un tipo de datos abstractos? (pp. 441-444.)
3. ¿En qué modo ayuda el uso de un archivo de especificación para proporcionar encapsulación y ocultación de información para una clase? (pp. 452-457.)
4. ¿Qué está contenido entre las llaves de una definición de `struct`? (pp. 431-433.)
5. ¿Qué operador se usa como selector de miembro de una `struct`? (pp. 433-434.)

6. ¿Cómo se denomina una estructura de datos que se implementa mediante una `struct` que contiene otros tipos de `struct`? (pp. 438-439.)
7. ¿De qué manera escribiría una expresión para tener acceso al miembro hour de una `struct` que, por sí misma, es un miembro de nombre time, de una variable `struct` denominada `date`? (pp. 438-439.)
8. Los miembros de una `struct`, al igual que `class`, pueden incluir variables, constantes y funciones. ¿Qué atributo por omisión de un miembro es diferente entre una `class` y una `struct`? (pp. 447-448.)
9. Escriba la declaración de un objeto de clase, denominado `today`, de clase `Date`. (p. 448.)
10. ¿Cómo nombraría a la función miembro `GetDay` (que no toma parámetros) de un objeto denominado `Today` de clase `Date`? (pp. 448-450.)
11. ¿Dónde aparece el operador de resolución de alcance `::` en una definición de función miembro dentro de un archivo de implementación? (pp. 454-457.)
12. ¿Qué archivo, especificación o implementación, omite cuerpos de funciones miembro? (pp. 452-454.)
13. ¿Qué distingue a un constructor de clase de otras funciones miembro? (pp. 460-461.)

Respuestas

1. Permitiendo que un tipo tenga valores de tipos distintos en diferentes tiempos, evitando la necesidad de asignar almacenaje para todos los tipos distintos a la vez. **2.** Al elegir una representación de datos y escribir después las operaciones para el ADT con instrucciones de programa. **3.** Permite al código cliente ver la interfaz formal para la clase, pero ninguno de los detalles de implementación que el creador de la clase desea ocultar. **4.** Una lista de miembros del `struct`. **5.** El punto. **6.** Un registro jerárquico. **7.** `date.time.hour`. **8.** Las clases tienen, por omisión, miembros privados; los structs tienen, por omisión, miembros públicos. **9.** `Date today`. **10.** `today.GetDay()`. **11.** En el encabezado entre el nombre de la clase y el nombre de la función. **12.** El archivo de especificación. **13.** Su nombre es idéntico al de la clase y nunca es vacío ni tiene un tipo de valor de retorno.

Ejercicios de preparación para examen

1. Una `struct` no puede tener otra `struct` como miembro. ¿Verdadero o falso?
2. Una `union` es una `struct` que puede contener sólo uno de sus miembros a la vez. ¿Verdadero o falso?
3. El alcance de clase se aplica a clases, structs y uniones. ¿Verdadero o falso?
4. La notación de punto la emplean sólo los clientes de clase para hacer referencia a miembros. Otros miembros de clase no necesitan usar la notación de punto. ¿Verdadero o falso?

```
struct Name
{
  string first;
  string middle;
  string last;
};

Name yourName;
Name myName;
```

¿Cuál es el contenido de las dos variables Name después de cada una de las siguientes sentencias, suponiendo que se ejecutan en el orden listado?

a) `yourName.first = "George";`
b) `yourName.last = "Smith";`
c) `myName = yourName;`
d) `myName.middle = "Nathaniel";`
e) `yourName.middle = myName.middle[0] + ".";`

6. ¿Cuáles son las tres operaciones de agregación permitidas en structs?
7. ¿Cuál es la diferencia entre una unión y un tipo de enumeración?
8. Dada la declaración del tipo Name en el ejercicio 5 y las siguientes declaraciones:

```
struct studentRecord
{
  Name studentName;
  Name teacherName;
  int gradeNumber;
  string grades;
}

studentRecord sally;
```

   a) ¿Cómo asignaría el nombre Sally Ellen Strong al campo `studentName` de variable `sally`?
   b) ¿Cómo asignaría el número de grado 7 al campo de `sally`?
   c) ¿Cómo asignaría la cuarta letra del campo `grades` a la variable `char` de nombre `spring`?
9. ¿Qué sucede cuando una struct se pasa como argumento a un parámetro de valor de una función? ¿Cómo difiere esto de pasarlo a un parámetro de referencia?
10. Dada la siguiente declaración de unión:

```
union GradeUnion
{
  char gradeLetter;
  int gradeNumber;
}

GradeUnion grade;
```

    ¿Qué hace cada una de las siguientes sentencias, suponiendo que se ejecutan en el orden mostrado?
    a) `cin >> grade.gradeLetter;`
    b) `if (grade.gradeLetter >= 'A' && grade.gradeLetter <= 'D')`
    c) `grade.gradeNumber =  4 - int(grade.gradeLetter - 'A');`
11. ¿Cuáles son las dos propiedades principales de un tipo de datos abstractos que se definen en su especificación?
12. ¿En qué difiere la representación de datos del dominio de un tipo de datos abstractos?
13. Por omisión, ¿los miembros de clase son públicos o privados?
14. Dado un objeto, de nombre `current`, de clase `Time`: ¿cómo escribiría la llamada para su función miembro vacía, llamada `plus`, que añade un número entero de minutos al tiempo actual? El número de minutos a añadir está en la variable `period`.
15. ¿Qué clase de operación, constructor, observador, transformador o iterador realiza una función que se declara como `const`?
16. Una clase puede tener constructores múltiples, todos con el mismo nombre. ¿Cómo decide el compilador a cuál llamar?
17. Una clase llamada `Calendar` tiene un constructor por omisión y uno que acepta un valor entero que especifica el año. ¿Cómo invocaría a cada uno de éstos para un objeto de nombre `primary`? Use el año actual como el argumento para el segundo constructor.
18. ¿Cómo se distingue el nombre de una clase del de un constructor?

Ejercicios de calentamiento para programación

1. Declare un tipo `struct`, `Time`, que represente una cantidad de tiempo, que conste de minutos y segundos.
2. Escriba sentencias que asignen el tiempo 6 minutos y 54 segundos a una variable `someTime`, de tipo `Time`, como se declaró en el ejercicio 1.

3. Declare un tipo `struct`, `Song`, que represente una entrada de canción en una biblioteca MP-3. Ésta debe tener campos para el título, álbum, artista, tiempo que dura la canción en minutos y segundos (use el tipo declarado en el ejercicio 1), y categoría de música. La categoría de música se representa mediante un tipo de enumeración de nombre `Category`.
4. Escriba sentencias para declarar una variable denominada `mySong` de tipo `Song`, y asígnele un conjunto de valores. Para el tiempo que dura la canción, use la variable `someTime` declarada en el ejercicio 2. Determine los valores para los otros campos. Suponga que el tipo de enumeración `Category` incluye alguna categoría de canción que usted desea usar.
5. Escriba una sentencia para producir el tiempo que dura la canción a partir de `mySong`, según se declaró en el ejercicio 4, en el formato `mm:ss`.
6. Escriba una declaración de un tipo `union` denominada `Temporal` que pueda contener un tiempo representado como una cadena, como un entero o como un valor de tipo `Time`, según se declaró en el ejercicio 1.
7. Escriba la declaración de una variable de nombre `shift` de tipo `Temporal`, según se declaró en el ejercicio 6, y una sentencia que asigne el valor de `someTime`, como se declaró en el ejercicio 2, a `shift`.
8. En los problemas de programación 3 y 4 del capítulo 10 se pidió desarrollar programas educacionales para enseñar los periodos de tiempo geológico. Ahora que sabe cómo escribir clases, puede convertir el TDA Period en una clase. Escriba un archivo de especificación para la clase `Period` que tenga un campo privado que sea un tipo de enumeración, el cual represente los nombres de los periodos. Debe tener funciones que devuelvan el periodo como una cadena, que devuelva el periodo como un `int`, que devuelva la fecha de inicio del periodo como una cadena y que incremente el periodo al siguiente periodo más reciente (si el periodo es `CUATERNARIO`, la operación de incremento no debe hacer nada). Debe tener un constructor por omisión que fije el periodo en `PRECÁMBRICO` y un constructor con un parámetro del tipo de enumeración para permitir al usuario crear un objeto Period que contenga cualquier periodo. Para facilidad de referencia, la tabla de los periodos del capítulo 10 se repite aquí. Asegúrese de usar los comentarios de documentación apropiados.

Nombre del periodo	*Fecha de inicio (millones de años)*
Cuaternario	2.5
Terciario	65
Cretáceo	136
Jurásico	192
Triásico	225
Pérmico	280
Carbonífero	345
Devónico	395
Silúrico	435
Ordovícico	500
Cámbrico	570
Precámbrico	4500 o antes

9. Use la clase `Period`, como se especificó en el ejercicio 8, para escribir una sentencia For que itera a través de los periodos de tiempo geológico desde el primero hasta el más reciente, escribiendo el nombre del periodo y su fecha de inicio.
10. Escriba las definiciones de función de los dos constructores para la clase `Period` especificada en el ejercicio 8.
11. Escriba la definición de función para el observador `ToInt` de la clase `Period` especificada en el ejercicio 8.
12. ¿Cuál sería el nombre de archivo para el archivo que contiene la especificación de la clase `Period` del ejercicio 8?

13. Escriba la directiva `include` que aparecería cerca del comienzo del archivo de implementación de la clase `Period` para incluir el archivo de especificación para la clase `Period` según se describe en el ejercicio 8.
14. Como resultado de los errores de representación de punto flotante, las cantidades monetarias que necesitan ser exactas no deben almacenarse en tipos de punto flotante. Codifique la especificación para una clase que represente una cantidad de dinero como dólares y centavos. Ésta debe tener un constructor por omisión que cree un objeto con cero dólares y cero centavos, un observador para dólares, un observador para centavos y un observador que devuelva la cantidad como un valor de punto flotante. También debe tener transformadores que sumen y resten otros valores de la clase `Money`. Asegúrese de usar los comentarios de documentación apropiados.
15. Escriba las definiciones de función de los dos constructores para la clase Money como se especificó en el ejercicio 14.
16. ¿Cuál sería el nombre del archivo que contiene la especificación de la clase Money en el ejercicio 14?
17. Escriba la directiva `include` que aparecería cerca del comienzo del archivo de implementación de la clase `Money` para incluir el archivo de especificación para la clase `Money` como se describe en el ejercicio 14.
18. Escriba las definiciones de función para el observador en la clase `Money` del ejercicio 14 que devuelve los valores como un `float`.

Problemas de programación

1. En el problema 3 de programación del capítulo 10 se pidió desarrollar un programa educacional para enseñar el tiempo geológico. Reescriba el programa usando una clase para poner en práctica un TDA que represente un periodo del tiempo geológico. Si hizo los ejercicios de calentamiento para programación 8 al 13, esto será relativamente fácil. El programa debe permitir al usuario introducir un intervalo de fechas prehistóricas (en millones de años), y después producir los periodos que están incluidos en el intervalo. Cada vez que se hace esto, se pregunta al usuario si quiere continuar. El objetivo del ejercicio es que el alumno intente averiguar cuándo comenzó cada periodo, de modo que pueda hacer una gráfica del tiempo geológico. Véase, en el ejercicio 8 de calentamiento para programación, una lista de los periodos geológicos y sus fechas de inicio.
2. En el problema 4 de programación del capítulo 10 se pidió escribir un segundo programa educacional para aprender acerca del tiempo geológico. Reescriba ese programa con una clase para poner en práctica un TDA que represente un periodo del tiempo geológico. Si hizo los ejercicios de calentamiento para programación 8 al 13, entonces ya ha hecho algo de este trabajo. El TDA en esos ejercicios necesita ser mejorado con un constructor que tome una cadena como un argumento y convierta la cadena en el valor `Period` correspondiente (el constructor debe trabajar con cualquier estilo de uso de mayúsculas de los nombres de periodos). En este programa, la computadora selecciona una fecha en el tiempo geológico y lo presenta al alumno. El alumno conjetura qué periodo corresponde a la fecha. Se permite que el alumno continúe haciendo conjeturas hasta obtener la respuesta correcta. Entonces el programa pregunta al alumno si quiere intentar de nuevo y repite el proceso si la respuesta es "sí". Es posible que también desee escribir una función que devuelva un periodo para una fecha específica; sin embargo, esto no necesita ser parte de la clase.
3. En varios capítulos, desde el 4, se han incluido problemas de programación que piden desarrollar o reescribir un programa que produce el peso del usuario en distintos planetas. El objetivo de hacer esto ha sido que usted vea cómo el mismo programa se puede poner en práctica de varias maneras. Aquí sería deseable que usted reescribiera el programa usando una clase para representar el planeta y su gravedad. La clase debe incluir un constructor que permita que un planeta sea especificado con una cadena, con cualquier uso de mayúsculas (si la cadena no es un nombre de planeta, entonces se debe suponer que se trata de la Tierra). El constructor por omisión para la clase creará un objeto que represente a la Tierra. La clase tiene un operador observador que toma un peso sobre la Tierra como un argumento y devuelve el peso sobre el planeta. Debe tener un

segundo observador que devuelve el nombre del planeta como una cadena con el uso apropiado de mayúsculas.

Para facilidad de referencia, la información para el problema original se repite aquí. En la tabla siguiente se proporciona el factor por el que se debe multiplicar el peso para cada planeta. El programa debe producir un mensaje de error si el usuario no escribe de manera correcta el nombre de un planeta. La leyenda de orientación y el mensaje de error deben aclarar al usuario cómo se debe introducir el nombre de un planeta. Asegúrese de usar el formato y los comentarios apropiados en su código. El resultado debe ser marcado con claridad y tener un formato nítido.

Mercurio	0.4155
Venus	0.8975
Tierra	1.0
Luna	0.166
Marte	0.3507
Júpiter	2.5374
Saturno	1.0677
Urano	0.8947
Neptuno	1.1794
Plutón	0.0899

4. Diseñe, implemente y pruebe una clase (`class`) que represente una cantidad de tiempo en minutos y segundos. La clase debe proporcionar un constructor que establezca el tiempo en un número especificado de minutos y segundos. El constructor por omisión debe crear un objeto para un tiempo de cero minutos y cero segundos. La clase debe proporcionar observadores que devuelvan los minutos y los segundos por separado, y un observador que devuelva el tiempo total en segundos (minutos $\times$ 60 + segundos). Se deben proveer observadores de comparación booleanos que prueben si dos tiempos son iguales, uno es mayor que el otro o uno es menor que el otro. Se deben proporcionar transformadores que agreguen un tiempo a otro y resten un tiempo a otro. La clase no debe permitir tiempo negativo (la resta de más tiempo del que actualmente está almacenado debe dar como resultado un tiempo de 0:00).
5. Diseñe, implemente y pruebe una clase que represente una canción en un CD o en un biblioteca de MP-3. Si hizo los ejercicios de calentamiento para programación 3 al 5, ya sabe cómo representar una canción como una `struct`. El objetivo aquí es hacer del TDA una clase (`class`) encapsulada. Ésta debe tener miembros para título, artista, tiempo que dura la canción en minutos y segundos (si hizo el problema 4, puede usar la clase aquí), y la categoría de música. Esta última se representa mediante un tipo de enumeración denominado `Category`. Establezca una enumeración de sus categorías favoritas. La clase debe tener un constructor que permita establecer todos los miembros de datos, y uno por omisión que los fije en los valores vacíos apropiados. Debe tener una operación observadora para cada miembro y un observador que devuelva todos los datos para la canción como una cadena. Se debe desarrollar un observador que compare si dos canciones son iguales. Los transformadores se deben proporcionar para permitir que cada valor de datos sea cambiado.
6. Diseñe, implemente y pruebe una clase que represente un número telefónico. El número debe ser representado por un código de país, un código de área, un número y un tipo. Los tres primeros pueden ser enteros. El miembro de tipo es una enumeración de HOME, OFFICE, FAX, CELL, PAGER. La clase debe proporcionar un constructor por omisión que establezca todos los valores enteros en cero y el tipo en HOME. Se debe proporcionar un constructor que permita establecer todos los valores. También debe proporcionar un constructor que tome sólo el número y tipo de argumentos, y establezca los códigos de país y área propios de su ubicación. La clase tendrá observadores que permitan recuperar cada miembro de datos, y transformadores que permitan cambiar cada miembro de datos. Se debe proporcionar un observador adicional que compare la igualdad de dos números telefónicos.

Seguimiento de caso práctico

1. Clasifique cada una de las siete funciones miembro de la clase `Name` como un constructor, transformador u operación observador.
2. Escriba un plan de prueba para probar la función `ComparedTo` de la clase `Name`.
3. Escriba un manejador para ejecutar su plan de prueba para la función `ComparedTo`.
4. ¿Qué sucede si el cliente introduce una inicial cuando se solicita el segundo nombre? ¿Qué sucede si el cliente introduce una inicial seguida de un punto cuando se solicita el segundo nombre?
5. Mejore la clase `Name` con un campo de título y una función observadora que devuelva su valor. ¿Se debe usar el título en la función `ComparedTo`? Explique su respuesta.

CAPÍTULO

12

Arrays

Objetivos

Objetivos de conocimiento

- *Comprender la estructura de un array unidimensional.*
- *Saber cómo usar un array unidimensional en la resolución de un problema.*
- *Entender la estructura de arrays de registros y objetos de clase.*
- *Conocer cómo se puede hacer que valores de índice tengan contenidos semánticos.*
- *Entender la estructura de un array bidimensional.*
- *Entender la estructura de un array multidimensional.*

Objetivos de habilidades

Ser capaz de:

- *Declarar un array unidimensional, con y sin inicialización.*
- *Realizar operaciones fundamentales en arrays unidimensionales.*
- *Aplicar el procesamiento de sub-arrays a un array unidimensional.*
- *Declarar un array bidimensional.*
- *Realizar operaciones fundamentales en un array bidimensional.*
- *Usar arrays como parámetros y argumentos.*
- *Declarar y procesar un array multidimensional.*

Las estructuras de datos desempeñan un papel importante en el proceso de diseño. La selección de estructuras de datos afecta el diseño en forma directa porque determina los algoritmos que se usan para procesar los datos. En el capítulo 11 vimos cómo el registro o estructura (struct) y la clase nos dan la habilidad de referirnos a un completo grupo de componentes por medio de un nombre. Esto simplifica el diseño de muchos programas.

Sin embargo, en muchos problemas una estructura de datos tiene tantos componentes que su procesamiento es difícil si cada uno debe tener un nombre de miembro único. Por ejemplo, el tipo de datos abstractos IntList (ADT) que hemos propuesto brevemente en el capítulo 11 representa una colección de hasta 100 valores enteros. Si usáramos una estructura o una clase para conservar estos valores, tendríamos que inventar 100 diferentes nombres de miembros, escribir 100 diferentes instrucciones de entrada para leer valores en los miembros, y escribir 100 diferentes instrucciones de salida para indicar los valores; o sea, ¡una tarea increíblemente tediosa! Un *array* —el cuarto de los tipos de datos estructurados apoyados por C++— es un tipo de datos que nos permite programar operaciones de este tipo con facilidad.

En este capítulo examinaremos los tipos de datos de arrays proporcionados por el lenguaje C++; en el capítulo 13 mostraremos cómo combinar clases y arrays para implementar un ADT, como por ejemplo un IntList.

12.1 Arrays unidimensionales

Si quisiéramos introducir 1 000 valores enteros e imprimirlos en orden inverso, podríamos escribir un programa de esta forma:

```
//***********************************
// Programa Números en orden inverso
//***********************************
#include <iostream>

using namespace std;

int main()
{
    int value0;
    int value1;
    int value2;
      ⋮
    int value999;

    cin >> value0;
    cin >> value1;
    cin >> value2;
      ⋮
    cin >> value999;
    cout << value999 << endl;
    cout << value998 << endl;
    cout << value997 << endl;
      ⋮
    cout << value0 << endl;
    return 0;
}
```

Este programa tiene una longitud de más de 3 000 líneas, y tenemos que usar 1 000 variables separadas. Nótese que todas las variables tienen el mismo nombre, con la excepción de un número agre-

gado que las distingue. ¿No sería conveniente si pudiéramos poner el número en una variable de conteo y usar ciclos For para ir de 0 a 999, y luego devolver nuevamente de 999 hasta 0? Por ejemplo, si la variable de conteo fuera `number`, podríamos remplazar las 2 000 instrucciones originales de entrada/salida con las siguientes cuatro líneas de código (ponemos aquí `number` en corchetes para distinguirlo de `value`):

```
for (number = 0; number < 1000; number++)
    cin >> value[number];
for (number = 999; number >= 0; number--)
    cout << value[number] << endl;
```

Este fragmento de código es correcto en C++ si declaramos que `value` es un *array unidimensional*, que es una colección de variables (todas del mismo tipo) donde la primera parte de cada variable es la misma, y la última es un *valor de índice* entre corchetes. En nuestro ejemplo, el valor guardado en `number` se denomina *índice*.

La declaración de un array unidimensional es similar a la declaración de una variable simple (una variable de tipo de dato simple), con una excepción: también se deberá declarar el tamaño del array. Para tal efecto se debe indicar entre corchetes el número de componentes en el array:

```
int value[1000];
```

Esta declaración crea un array con 1 000 componentes, todos del tipo int. El primer componente tiene un valor de índice 0; el segundo tiene el valor de índice 1, y el último componente tiene el valor de índice 999.

A continuación se encuentra el programa de Reverse`Number`s completo, usando la notación de array. Esto ciertamente es mucho más corto que nuestra primera versión del programa.

```
//************************************
// Programa Números en orden inverso
//************************************
#include <iostream>

using namespace std;
int main()
{
    int value[1000];
    int number;

    for (number = 0; number < 1000; number++)
        cin >> value[number];
    for (number = 999; number >= 0; number--)
        cout << value[number] << endl;
    return 0;
}
```

Como estructura de datos, un array difiere de una estructura o una clase en dos formas fundamentales:

1. Un array es una estructura de datos *homogénea* (todos los componentes son del mismo tipo de datos), mientras que estructuras y clases son tipos heterogéneos (sus componentes pueden ser de diferentes tipos).
2. El acceso a un componente de un array es por medio de su *posición* en el arreglo, mientras que el acceso a un componente de una estructura o clase es por medio de un identificador (el nombre del miembro).

Ahora definiremos los arrays formalmente, y echaremos un vistazo a las reglas para el acceso a componentes individuales.

La declaración de arrays

Array unidimensional Colección estructurada de componentes, todos del mismo tipo, que recibe un solo nombre. Cada componente (elemento de *array*) es accesado por medio de un índice que ubica la posición del componente dentro de la colección.

Un **array unidimensional** es una colección estructurada de componentes (con frecuencia denominados *elementos de array*) que pueden ser individualmente accesados especificando la posición de un componente con un solo valor de índice. (Más adelante en este capítulo introduciremos los arrays multidimensionales, que son arrays que tienen más de un valor de índice.)

A continuación se presenta una plantilla de sintaxis que describe la forma más sencilla de una declaración de un array unidimensional:

ArrayDeclaration

```
DataType ArrayName [ ConstIntExpression ] ;
```

En la plantilla de sintaxis, DataType describe lo que está almacenado en cada componente del array. Los componentes de arrays pueden ser de casi todos los tipos, pero por ahora limitaremos nuestro estudio a componentes atómicos. ConstIntExpression es una expresión de enteros compuesta sólo de constantes literales o nombrados. Esta expresión, que especifica el número de componentes en el array, debe tener un valor mayor a 0. Si el valor es n, el intervalo de valores de índice es de 0 hasta $n - 1$, y no de 1 hasta n. Por ejemplo, las declaraciones

```
float angle[4];
int   testScore[10];
```

crean los arrays que se muestran en la figura 12-1. El array `angle` tiene cuatro componentes, cada uno de los cuales es capaz de sostener un valor `float`. El array `testScore` tiene un total de diez componentes, todos del tipo `int`.

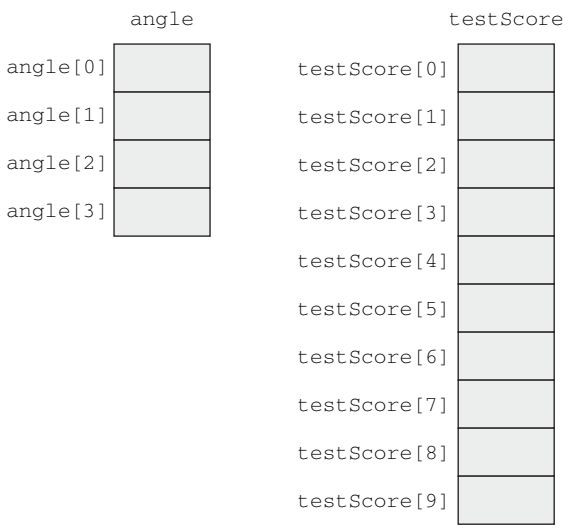


Figura 12-1 *Arrays* `angle` *y* `testScore`

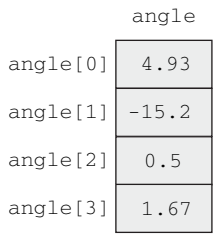


Figura 12–2 *Array* `angle` *con valores*

Acceder a componentes individuales

Recuerde que para acceder a un componente individual de una estructura o una clase usamos la notación de puntos: el nombre de la variable tipo estructura o del objeto de clase, seguido por el nombre del miembro. En contraste, para acceder a un componente individual de array escribimos el nombre del array, seguido por una expresión entre corchetes. La expresión especifica el componente al que se va a acceder. La plantilla de sintaxis para acceder a un componente de array es:

ArrayComponentAccess

```
ArrayName [ IndexExpression ]
```

La expresión de índice puede ser tan simple como una constante o un nombre de variable, o tan compleja como una combinación de variables, operadores y llamadas de función. Sea lo que fuere la forma de la expresión, ella deberá resultar en un valor entero. Expresiones de índice pueden ser del tipo `char`, `short`, `int`, `long` o `bool`, porque todos son tipos enteros. Adicionalmente se pueden usar valores de tipos de enumeración como expresiones de índice, con un valor de enumeración implícitamente forzado a un entero.

La forma más sencilla de una expresión de índice es una constante. Si usamos nuestro array `angle`, la siguiente secuencia de sentencias de asignación

```
angle[0] = 4.93;
angle[1] = -15.2;
angle[2] = 0.5;
angle[3] = 1.67;
```

llena los componentes de arrays uno por uno (véase la figura 12-2).

A cada componente de array (`angle[2]`, por ejemplo) se puede tratar exactamente de la misma manera que cualquier variable sencilla del tipo `float`. Por ejemplo, podemos hacer lo siguiente al ángulo del componente individual `angle[2]`:

```
angle[2] = 9.6;              Asignarle un valor.
cin >> angle[2];             Leer un valor en él.
cout << angle[2];            Escribir sus contenidos.
y = sqrt(angle[2]);          Pasarlo como argumento.
x = 6.8 * angle[2] + 7.5;    Usarlo en una expresión aritmética.
```

Vamos a considerar expresiones de índice que son más complicadas que constantes. Supongamos que declaramos un array de 1 000 elementos de valores `int` con el enunciado

```
int value[1000];
```

y ejecutamos las siguientes dos sentencias.

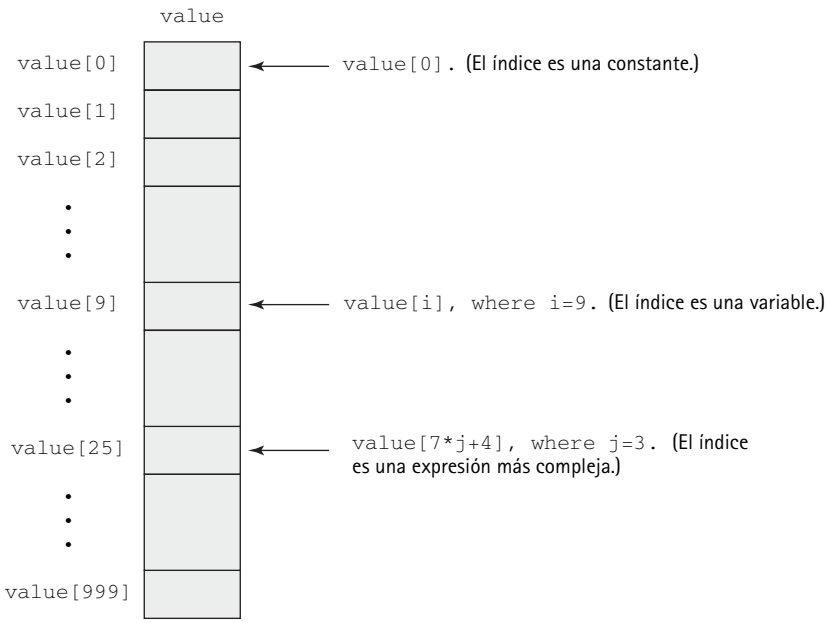


Figura 12-3 *Un índice como Constante, Variable y Expresión Arbitraria*

```
value[counter] = 5;
if (value[number+1] % 10 != 0)
    ⋮
```

En la primera sentencia, 5 es almacenado en un componente de array. Si `counter` es 0, 5 es almacenado en el primer componente del array. Si `counter` es 1, 5 es almacenado en el segundo lugar del array, etcétera.

En la segunda sentencia, la expresión `number+1` selecciona un componente de array. El componente específico de array accedido se divide entre 10 y se verifica si el remanente no es cero. Si `number+1` es 0, comprobamos el valor en el primer componente; si `number+1` es 1, comprobamos el segundo lugar, y así sucesivamente. La figura 12-3 muestra la expresión de índice como una constante, una variable y una expresión más compleja.

Observe que hemos visto el uso de corchetes anteriormente. En capítulos anteriores hemos dicho que la clase de `string` nos permite acceder a un carácter individual dentro de una cadena:

```
string aString;

aString = "Hello";
cout << aString[1];     // Imprime 'e'
```

Aunque `string` es una clase, no un array, la clase de `string` fue escrita usando la técnica avanzada C++ de *sobrecarga de operadores* para dar otro significado al operador `[]` (selección de componentes de cadena) en adición a su significado estándar (selección de elementos de array). El resultado es que un objeto `string` es similar a un array de caracteres, pero con propiedades especiales.

Índices de arrays fuera de límite

Dada la declaración

```
float alpha[100];
```

el intervalo válido de valores de índice es de 0 a 99. ¿Qué pasa si ejecutamos la sentencia

```
alpha[i] = 62.4;
```

cuando `i` es menor que 0, o cuando `i` es mayor que 99? El resultado es que se accede a una ubicación de memoria fuera del array. C++ no revisa índices de array inválidos (**fuera de límite**), ya sea en tiempo de compilación o en tiempo de ejecución. Si `i` resulta ser 100 en la sentencia anterior, la computadora almacena 62.4 en la siguiente ubicación de memoria más allá del final del array, destruyendo cualquier valor que se hubiese encontrado allí. Es responsabilidad exclusiva del programador asegurar que un índice de array no se salga de cualquier extremo del array.

Índice de array fuera de límite Valor de índice que, en C++, es menor a 0 o mayor al tamaño de array menos 1.

Los algoritmos de procesamiento de arrays a menudo usan ciclos For para pasar uno por uno a través de los elementos del array. A continuación se presenta un ciclo para eliminar nuestro array `alpha` de 100 elementos (`i` es una variable `int`):

```
for (i = 0; i < 100; i++)
    alpha[i] = 0.0;
```

Podríamos escribir la primera línea también como

```
for (i = 0; i <= 99; i++)
```

Sin embargo, los programadores de C++ normalmente usan la primera versión, así que el número en la prueba de ciclo (100) corresponde al tamaño del array. Cuando se usa este patrón, es importante recordar que se debe hacer la prueba para *menor que*, y no para menor o igual que.

Inicialización de arrays en declaraciones

En el capítulo 8 hemos aprendido que C++ siempre nos permite inicializar una variable en su declaración:

```
int delta = 25;
```

El valor 25 se denomina como un inicializador. También se puede inicializar un array en su declaración, usando una sintaxis especial para el inicializador. Para ello se especifica una lista de valores iniciales para los elementos del array, se separan por comas y se coloca la lista dentro de corchetes:

```
int age[5] = {23, 10, 16, 37, 12};
```

En esta declaración, `age[0]` es inicializado a 23, `age[1]` es inicializado a 10, etcétera. Debe haber por lo menos un valor inicial entre los corchetes. Si se especifican demasiados valores iniciales, se recibe un mensaje de error de sintaxis. Si se especifican pocos, los restantes elementos de array se inicializan a 10.

Los arrays siguen la misma regla que las variables sencillas en cuanto al tiempo (o los tiempos) en que ocurre la inicialización. Un array estático (un array ya sea global o declarado como `static` dentro de un bloque) sólo es inicializado una vez cuando el control alcance su declaración. Un array automático (un array local y no declarado como `static`) es reinicializado cada vez que el control alcanza su declaración.

Una característica interesante de C++ es que se permite omitir el tamaño de un array cuando se inicializa en una declaración:

```
float temperature[] = {0.0, 112.37, 98.6};
```

El compilador averigua el tamaño del array (en este caso, 3) de acuerdo con el número de valores iniciales de la lista. En general, esta característica no es particularmente útil. Sin embargo, en el capítulo 13 veremos que puede ser útil para inicializar ciertos tipos de arrays `char` denominados cadenas C.

Ausencia de operaciones agregadas en arrays

En el capítulo 11 hemos definido una operación agregada como una operación sobre una estructura de datos. Algunos lenguajes de programación permiten operaciones agregadas sobre arrays, pero C++ no. Si `x` y `y` son declarados como

```
int x[50];
int y[50];
```

no hay ninguna operación de asignación agregada de `y` a `x`:

```
x = y;   // No válida
```

Para copiar el array `y` al array `x`, lo tendrá que hacer uno mismo, elemento por elemento:

```
for (index = 0; index < 50; index++)
    x[index] = y[index];
```

De manera similar, no hay ninguna operación de comparación agregada en los arrays:

```
if (x == y)   // No válida
```

ni se pueden realizar operaciones agregadas* de E/S:

```
cout << x;   // No válida
```

ni aritmética agregada en arrays:

```
x = x + y;   // No válida
```

Finalmente, no es posible devolver un array entero como el valor de una función de devolución de valores;

```
return x;   // No válida
```

Lo único que se le puede hacer a un array como conjunto es pasarlo como un argumento a una función:

```
DoSomething(x);
```

Pasar un array como un argumento le da a la función acceso al array completo. La siguiente tabla compara arrays, estructuras y clases con respecto a operaciones agregadas.

Operación agregada	Arrays	Estructuras y clases
E/S	No (excepto cadenas C)	No
Asignación	No	Sí
Aritmético	No	No
Comparación	No	No
Paso de argumento	Sólo por referencia	Por valor o por referencia
Devolución como valor de devolución de una función	No	Sí

Más adelante, en este capítulo, veremos en detalle el paso de arrays como argumentos.

* C++ permite una sola excepción para E/S, la cual analizaremos en el capítulo 13. E/S agregado se permite para cadenas C, que son tipos especiales de arrays `char`.

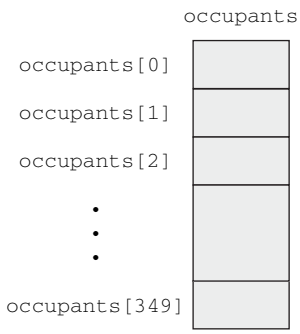


Figura 12-4 *Array de* `occupants`

Ejemplos de declarar y acceder a arrays

Ahora analizaremos algunos ejemplos específicos de declarar y acceder a arrays. A continuación se presentan algunas declaraciones que un programa podrá usar para analizar niveles de ocupación en un edificio de departamentos:

```
const int BUILDING_SIZE = 350;  // Número de departamentos

int occupants[BUILDING_SIZE];   // ocupants[i] es el número de
                                //   ocupantes en el departamento i
int totalOccupants;             // Total del número de ocupantes
int counter;                    // Ciclo de control y variable de índice
```

`occupants` es un array de 350 elementos de enteros (véase la figura 12-4). `occupants[0]` = 3 si el primer departamento tiene tres ocupantes; `occupants[1]` = 5 si el segundo departamento tiene 5 ocupantes, etcétera. Si se han almacenado valores en el array, entonces el siguiente código totaliza el número de ocupantes en el edificio.

```
totalOccupants = 0;
for (counter = 0; counter < BUILDING_SIZE; counter++)
    totalOccupants = totalOccupants + occupants[counter];
```

La primera vez por medio del ciclo, `counter` es 0. Sumamos los contenidos de `totalOccupants` (es decir, 0) a los contenidos de `occupants[0]`, guardando el resultado en `totalOccupants`. Ahora `counter` se vuelve 1 y la prueba del ciclo ocurre. La segunda iteración del ciclo suma los contenidos de `totalOccupants` a los contenidos de `occupants[1]`, guardando el resultado en `totalOccupants`. Ahora `counter` se vuelve 2 y se realiza la prueba del ciclo. Finalmente el ciclo suma los contenidos de `occupants[349]` a la suma e incrementa `counter` a 350. En este punto la condición del ciclo es falsa y el control sale del ciclo.

Observe cómo hemos usado la constante nombrada `BUILDING_SIZE` tanto en la declaración del array como en el ciclo `For`. Cuando las constantes se usan de este modo, es fácil realizar cambios. Si el número de departamentos cambia de 350 a 400, sólo necesitamos cambiar una línea: la declaración `const` de `BUILDING_SIZE`. Si hubiéramos usado el valor literal 350 en lugar de `BUILDING_SIZE`, habríamos necesitado actualizar varias de las sentencias en el código anterior, y tal vez muchas más a través del resto del programa.

A continuación se presenta un programa completo que usa el array `occupants`. Este programa llena el array con datos de ocupantes leídos de un archivo de entrada, y luego permite que el usuario busque en forma interactiva el número de ocupantes en un departamento específico.

```
//**************************************************************************
// Programa Departamentos
// Este programa permite que el dueño de un edificio vea cuántos
// ocupantes están en un departamento específico
// Nota: a este programa le falta el código para comprobar los errores
   introducidos por el usuario
//**************************************************************************
#include <iostream>
#include <fstream>                // Para archivo de E/S

using namespace std;

const int BUILDING_SIZE = 350;    // Número de departamentos

int main()
{
    int occupants[BUILDING_SIZE]; // occupants[i] es el número de ocupantes
                                  //   en el departamento i
    int totalOccupants;           // Número total de ocupantes
    int counter;                  // Control de ciclo y variable de índice
    int apt;                      // Un número de departamento
    ifstream inFile;              // Archivo de datos de ocupante
                                  //   (un entero por departamento)

    inFile.open("apt.dat");
    totalOccupants = 0;
    for (counter = 0; counter < BUILDING_SIZE; counter++)
    {
        inFile >> occupants[counter];
        totalOccupants = totalOccupants + occupants[counter];
    }
    cout << "El no. de departamentos es " << BUILDING_SIZE << endl
         << "El número total de ocupantes es " << totalOccupants << endl;

    cout << "Comenzar la búsqueda de departamentos... " << endl;
    do
    {
        cout << "Número de departamento (1 al " << BUILDING_SIZE
             << ", o 0 para terminar): "
        cin >> apt;
        if (apt > 0)
            cout << "Dep. " << apt << " tiene " << occupants[apt-1]
                 << " ocupantes" << endl;
    } while (apt > 0);
    return 0;
}
```

Observe detalladamente la última instrucción de salida en el programa de departamentos. El usuario introduce un número de departamento [`apt`] en el intervalo de 1 a `BUILDING_SIZE`, pero el array tiene los índices de 0 hasta `BUILDING_SIZE`-1. Por tanto, tenemos que restar 1 de `apt` para que indexemos el lugar indicado en el array.

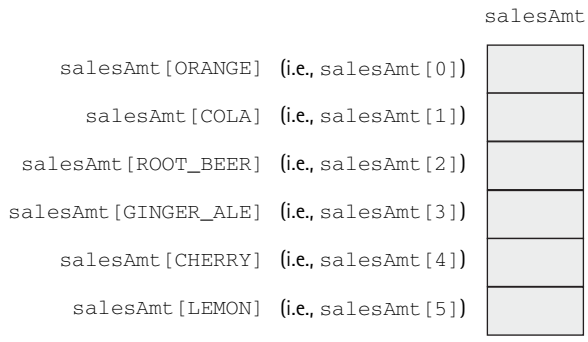


Figura 12-5 *Array* `salesAmt`

Debido a que un índice de array es un valor entero, accedemos a los componentes por medio de su posición en el array; o sea, el primero, el segundo, el tercero, etcétera. El uso de un índice `int` es la forma más común de pensar en un array. Sin embargo, C++ proporciona mayor flexibilidad, permitiendo que un índice sea de cualquier tipo integral o de enumeración. (La expresión del índice aún debe evaluarse a un entero en el intervalo de 0 hasta el tamaño del array menos uno.) El siguiente ejemplo muestra un array en el cual los índices son valores de un tipo de enumeración.

```
enum Drink {ORANGE, COLA, ROOT_BEER, GINGER_ALE, CHERRY, LEMON};

float salesAmt[6]; // Array de 6 flotantes, son indizadoindizados por tipo
                      Drink
Drink flavor;      // Variable de tipo índice
```

`Drink` es un tipo de enumeración en el cual los enumeradores `ORANGE, COLA, ..., LEMON` tienen las representaciones internas de 0 hasta 5, respectivamente. `salesAmt` es un grupo de seis componentes `float`, que representa cifras de ventas en dólares para cada tipo de bebida (véase la figura 12-5). El siguiente código imprime los valores en el array (véase el capítulo 10 para verificar cómo incrementar los valores de tipos de enumeración en ciclos For).

```
for (flavor = ORANGE; flavor <= LEMON; flavor = Drink(flavor + 1))
    cout << salesAmt[flavor] << endl;
```

A continuación un último ejemplo.

```
const int NUM_STUDENTS = 10;

char grade[NUM_STUDENTS];   // Array de las calificaciones de 10 estudiantes
                               usando letras
int  idNumber;              // Número de estudiante de 0 a 9
```

El `array` grade se representa en la figura 12-6. En los componentes aparecen valores, lo cual implica que ya ha ocurrido un determinado procesamiento del array. A esto le siguen algunos ejemplos sencillos que muestran cómo se podrá usar este array.

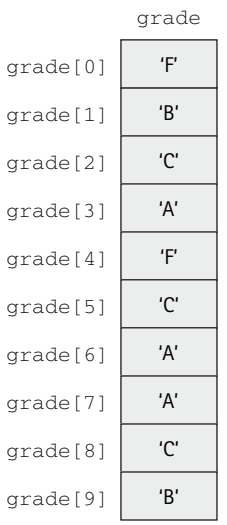


Figura 12-6 *Array* `grade` *con valores*

```
cin >> grade[2];
```
Lee el siguiente carácter sin espacio blanco del flujo de entrada y lo almacena en el componente en grade indizadoindizado por 2.

```
grade[3] = 'A';
```
Asigna el carácter 'A' al componente en grade indizadoindizado por 3.

```
idNumber = 5;
```
Asigna 5 a la variable de índice idNumber.

```
grade[idNumber] = 'C';
```
Asigna el carácter 'C' al componente de grade indizadoindizado por idNumber (o sea, por 5).

```
for (idNumber = 0; idNumber < NUM_STUDENTS;
     idNumber++)
    cout << grade[idNumber];
```
Hace un ciclo a través del array grade, imprimiendo cada componente. Para este ciclo, la salida sería `FBCAFCAACB`.

```
for (idNumber = 0; idNumber < NUM_STUDENTS;
     idNumber++)
    cout << "Student " << idNumber
         << " Grade " << grade[idNumber]
         << endl;
```
Hace un ciclo a través de grade, imprimiendo cada componente en una forma más legible.

En el último ejemplo, `idNumber` se usa como el índice, pero también tiene un contenido semántico: se trata del número de identificación del estudiante. La salida sería

```
Student 0 Grade F
Student 1 Grade B
  ⋮
Student 9 Grade B
```

Pasando arrays como argumentos

En el capítulo 8 hemos dicho que si una variable se pasa a una función y no debe ser cambiada por la función, entonces la variable se debería pasar por medio de valor y no por medio de referencia. Específicamente habíamos excluido de esta regla variables de flujo (como las que representan archivos de datos) y habíamos dicho que iba a haber una excepción más. Esta excepción son los arrays.

Por omisión, variantes simples de C++ siempre son pasadas por valor. Para pasar una variable simple por referencia, se deberá adjuntar un signo `"&"` al nombre del tipo de datos en la lista de parámetros de la función:

```
int SomeFunc( float param1,      // Paso por valor
              char& param2 )     // Paso por referencia
{
    ⋮
}
```

Es imposible pasar un array de C++ por valor; los arrays *siempre* pasan por referencia. Por tanto, nunca se usa el signo `&` cuando se declara un array como un parámetro. Cuando un array pasa como argumento, su **dirección base**, o sea la dirección de memoria del primer elemento del array, es transmitida a la función. La función sabe dónde se ubica el array actual del invocador, y puede acceder a cualquier elemento del array.

Dirección base Dirección de memoria del primer elemento de un array.

A continuación, una función de C++ que elimina un array `float` unidimensional de cualquier tipo:

```
void ZeroOut( /* out */ float arr[],
              /* in */  int   numElements )
{
    int i;

    for (i = 0; i < numElements; i++)
        arr[i] = 0.0;
}
```

En la lista de parámetros, la declaración `arr` no incluye un tamaño entre los corchetes. Si se incluye un tamaño, el compilador lo ignorará. El compilador sólo quiere saber que se trata de un array `float`, pero no un array `float` de algún tamaño particular. Por tanto, se deberá incluir en la función `ZeroOut` un segundo parámetro (el número de elementos del array) a fin de que el ciclo For funcione correctamente.

El código puede invocar la función `ZeroOut` para una array `float` de cualquier tamaño. El siguiente fragmento de código hace llamadas de función para eliminar dos arrays de tamaños distintos. Observe cómo se declara un parámetro de array en un prototipo de función.

```
void ZeroOut( float[], int );   // Prototipo de función
 ⋮
int main()
{
    float velocity[30];
    float refractionAngle[9000];
     ⋮
    ZeroOut(velocity, 30);
    ZeroOut(refractionAngle, 9000);
     ⋮
}
```

Con las variables sencillas, el paso por valor impide que una función modifique el argumento del invocador. Aunque no se pueden pasar arrays por valor en C++, sí se puede impedir que la función modifique el array del invocador. Para efectuar esto, se usa la palabra reservada `const` en la declaración del parámetro. A continuación se presenta una función que copia un array `int` a otro. Se espera que se modifique el primer parámetro, o sea el array de destino, pero no el segundo array.

```
void Copy( /* out */       int destination[],
           /* in */  const int source[],
           /* in */        int size          )
```

```
{
    int i;

    for (i = 0; i < size; i++)
        destination[i] = source[i];
}
```

La palabra `const` garantiza que cualquier intento de modificar el array `source` dentro de la función Copy resulte en un error de tiempo de compilación.

Aquí se presenta una tabla que resume el paso de argumentos para variables sencillas y arrays unidimensionales:

Argumento	Declaración de parámetro para un paso por valor	Declaración de parámetro para un paso por referencia
Variable sencilla	`int cost`	`int& price`
Array	imposible*	`int arr[]`

* Sin embargo, un prefijo a la declaración del array con la palabra `const` impide que la función modifique el parámetro.

Una nota final sobre el paso de argumentos: es un error común pasar un *elemento* de array a una función cuando la intención fue el paso del array completo. Por ejemplo, nuestra función `ZeroOut` espera que la dirección base de un array `float` sea transmitida como el primer argumento. En el siguiente fragmento de código, la llamada de función es un error.

```
float velocity[30];
 ⋮
ZeroOut(velocity[30], 30);    // Error
```

Primero que nada, `velocity[30]` denota un elemento de array sencillo, o sea un solo número de punto flotante, y no un array completo. Además, no hay ningún elemento de array con un índice de 30. Los índices para el array `velocity` son de 0 a 29.

Información básica

C, C++ y arrays como argumentos

Algunos lenguajes de programación permiten que se pasen arrays por valor o por referencia. Recuerde que en el paso por valor, una copia del argumento es transmitida a la función. Cuando un array es pasado por valor, se copia el array completo. No sólo se requiere espacio extra en la función para guardar la copia; el propio copiado toma su tiempo. El paso por referencia sólo requiere que se pase la dirección del argumento a la función; de esta manera, si un array es pasado por referencia, se pasa únicamente la dirección del primer componente del array. Así, el paso de arrays grandes por referencia ahorra espacio de memoria y tiempo.

(continúa) ▼

C, C++ y arrays como argumentos

El lenguaje de programación C, o sea el predecesor directo de C++, fue diseñado como un lenguaje de programación de sistemas. Los programas de sistemas, como compiladores, ligadores y sistemas operativos, tienen que ser rápidos y al mismo tiempo económicos en cuanto al espacio de memoria. En el diseño del lenguaje C se había considerado que el paso de arrays por valor era una propiedad de lenguaje innecesaria. Los programadores de sistemas nunca usaban un paso por valor cuando trabajaban con arrays. Por esta razón, tanto C como C++ pasan los arrays sólo por referencia.

Por supuesto que el uso de un parámetro de referencia puede llevar a errores inadvertidos si los valores son cambiados dentro de la función. En las versiones iniciales del lenguaje C, no había manera de proteger el array del invocador contra una modificación por parte de la función.

C++ (y las versiones recientes de C) agregaron la habilidad de declarar un parámetro de array como `const`. Declarando el array como `const` se produce un error de tiempo de compilación si la función intenta modificar el array. Como resultado, C++ admite la eficiencia de pasar arrays por referencia, pero también establece la protección (mediante `const`) del paso por valor.

Cada vez que su diseño de la interface de una función identifique un parámetro de array como sólo de entrada (a ser examinado, pero no modificado por la función), declare el array como `const` para obtener la misma protección que se da para el paso por valor.

Afirmaciones sobre arrays

En afirmaciones escritas como comentarios, a menudo necesitamos referirnos a un intervalo de elementos de arrays:

```
// Afirmación: alpha[i] hasta alpha[j] tienen que ser impresos
```

Para especificar estos rangos, será más conveniente usar una notación abreviada que consiste en dos puntos suspensivos:

```
// Afirmación: alpha[i] .. alpha[j] tienen que ser impresos
```

o, en forma más corta:

```
// Afirmación: alpha[i..j] tienen que ser impresos
```

Observe que esta notación de punto a punto no es sintaxis válida en sentencias del lenguaje C++. Sólo estamos hablando de comentarios en un programa.

Como ejemplo del uso de esta notación, escribiríamos la condición previa y la condición posterior para nuestra función `ZeroOut` de la siguiente forma:

```
void ZeroOut( /* out */ float arr[],
              /* in */  int   numElements )

// Precondición:
//     numElements es asignada
// Poscondición:
//     arr[0..numElements-1] == 0.0

{
    int i;

    for (i = 0; i < numElements; i++)
        arr[i] = 0.0;
}
```

El uso de Typedef con arrays

En el capítulo 10 hemos estudiado la sentencia Typedef como una forma de asignar un nombre adicional a un tipo de datos existente. Hemos dicho que antes de que `bool` fuera un tipo integrado en C++, con frecuencia los programadores usaban una sentencia Typedef, como por ejemplo la siguiente:

```
typedef int Boolean;
```

También podemos usar Typedef para asignar un nombre a un tipo de array, como en el siguiente ejemplo:

```
typedef float FloatArr[100];
```

Esta sentencia indica que el tipo `FloatArr` es el mismo que el tipo "array de 100 elementos de `float`". (Observe que el tamaño del array entre corchetes aparece al final del enunciado.) Ahora podemos declarar variables del tipo `FloatArr`:

```
FloatArr angle;
FloatArr velocity;
```

El compilador traduce estas declaraciones esencialmente a

```
float angle[100];
float velocity[100];
```

En este libro no solemos usar Typedefs para asignar nombres a tipos unidimensionales de arrays. Sin embargo, cuando analicemos los arrays multidimensionales más adelante en este capítulo, veremos que la técnica puede ser útil.

12.2 Arrays de registros (estructuras) y objetos de clase

Aunque los arrays con componentes atómicos son muy comunes, muchas aplicaciones requieren una colección de registros u objetos de clase. Por ejemplo, una empresa necesita una lista de registros de partes, y un maestro necesita una lista de estudiantes en un grupo de alumnos. Los arrays son ideales para estas aplicaciones. Solamente definimos un array cuyos componentes son registros u objetos de clase.

Arrays de registros (estructuras)

Definamos un libro de calificaciones en forma de una colección de registros de alumnos de la siguiente manera:

```
const int MAX_STUDENTS = 150;

enum GradeType {A, B, C, D, F};

struct StudentRec
{
    string    stuName;
    float     gpa;
    int       examScore[4];
    GradeType courseGrade;
};

StudentRec gradeBook[MAX_STUDENTS];
int        count;
```

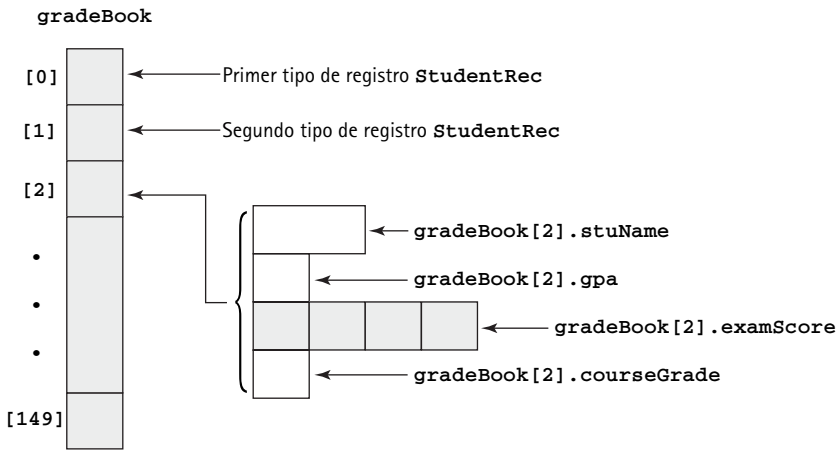


Figura 12-7 Array *gradeBook* con registros como elementos

Esta estructura de datos se puede visualizar como se muestra en la figura 12-7.

Un elemento `gradeBook` es seleccionado por medio de un índice. Por ejemplo, `gradeBook[2]` es el tercer componente en el array `gradeBook`. Cada componente de `gradeBook` es un registro del tipo `StudentRec`. Para acceder a la calificación de curso del tercer alumno, usaremos la siguiente expresión:

```
gradeBook[2].courseGrade
```

Especifica el tercer registro en el array `gradeBook`

Especifica el miembro `courseGrade` en el registro `gradeBook[2]`

El componente de registro `gradeBook[2].examScore` es un array. Podemos acceder a los elementos individuales en este componente de la misma forma en que accederíamos a los elementos de cualquier otro array: asignamos el nombre del array, seguido por el índice entre corchetes.

```
gradeBook[2].examScore[0]
```

Especifica el tercer registro en el array `gradeBook`

Especifica el miembro `examScore` (un array)

Especifica la primera calificación en el miembro `examScore`

El siguiente fragmento de código imprime el nombre de cada alumno del grupo:

```
for (count = 0; count < MAX_STUDENTS; count++)
    cout << gradeBook[count].stuName << endl;
```

Arrays de objetos de clase

La sintaxis para declarar y usar arrays de objetos de clase es la misma que se usa para arrays de estructuras. Dada la clase de `Time` del capítulo 11, podemos mantener una colección de diez tiempos de cita, empezando con la declaración

```
Time appointment[10];
```

Esta sentencia crea un array de diez elementos llamado `appointment`, en el cual cada elemento es un objeto de `TimeType`. Las siguientes sentencias establecen las primeras dos citas a efectuarse a las 8:45:00 y a las 10:00:00.

```
appointment[0].Set(8, 45, 0);
appointment[1].Set(10, 0, 0);
```

Para emitir los diez horarios de citas, podemos escribir

```
for (index = 0; index < 10; index++)
{
    appointment[index].Write();
    cout << endl;
}
```

Recuerde que la clase `Time` tiene dos constructores definidos para ella. Uno es el constructor de omisión (sin parámetro), que establece la hora para un objeto de nueva creación a ser 00:00:00. El otro es un constructor parametrizado por el cual el código de cliente puede especificar una hora inicial cuando el objeto de clase se ha creado. ¿Cómo se manejan los constructores cuando se declara un array de objetos de clase? Ésta es la regla en C++:

> Si una clase tiene por lo menos un constructor, y se declara un array de objetos de clase:
>
> ```
> SomeClass arr[50];
> ```
>
> entonces uno de los constructores *tiene* que ser el constructor por omisión (sin parámetro). Este constructor es invocado para cada elemento del array.

Por tanto, por medio de nuestra declaración del array `appointment`

```
Time appointment[10];
```

se llama el constructor de omisión para los diez elementos, estableciendo cada hora a un valor inicial de 00:00:00.

12.3 Tipos especiales de procesamiento de arrays

Con especial frecuencia ocurren dos tipos de procesamiento de arrays: usar sólo una parte del array declarado (un sub-array) y usar valores de índice que tienen un significado especial dentro del problema (índices con contenido semántico). Aquí describiremos brevemente ambos métodos y daremos otros ejemplos en el resto del capítulo.

Procesamiento de sub-arrays

El *tamaño* de un array (el número declarado de componentes del array) se establece en tiempo de compilación. Tenemos que declararlo tan grande que jamás necesite cambiar. Puesto que el número

exacto de valores que se colocan en el array depende a menudo de los propios datos, posiblemente no llenemos todos los componentes del array con valores. El problema consiste en que para evitar el procesamiento de espacios vacíos, es necesario recordar el número de componentes realmente llenos.

Mientras se colocan valores en el array, mantenemos un conteo del número de componentes llenados. Luego usamos este conteo para procesar sólo componentes con valores almacenados. Los lugares restantes no son procesados. Por ejemplo, si hay 250 alumnos en un grupo, un programa para analizar calificaciones de exámenes apartaría 250 posiciones para las calificaciones. Sin embargo, se tiene que contar el número de calificaciones del examen, y este número, en lugar de 250, se usa para controlar el procesamiento del array.

Si el número de unidades de datos realmente almacenados en un array es menor que su tamaño declarado, las funciones que reciben parámetros de arrays también deberán recibir el número de unidades de datos como parámetro. Por ejemplo,

```
void Print( /* in */ const char grade[],     // Arreglo para hasta
                                             //   250 estudiantes
            /* in */       int  numGrades )  // Número de calificaciones
                                             //   en el arreglo
```

Índices con contenido semántico

En algunos problemas, un índice de array tiene significados más allá de la simple posición; es decir, que el índice tiene *contenido semántico*. Un ejemplo de esto es el array `salesAmt` que hemos visto antes. Este array es indizadoindizado por un valor de tipo enumeración `Drink`. El índice de una cantidad específica de ventas es el tipo de refresco vendido; por ejemplo, `salesAmt [ROOT_BEER]` es el monto de ventas en dólares para "root beer".

En la siguiente sección presentamos ejemplos adicionales de índices con contenido semántico.

12.4 Arrays bidimensionales

Un array unidimensional se usa para representar unidades en una lista o secuencia de valores. En muchos problemas, sin embargo, las relaciones entre unidades de datos son más complejas que una simple lista. Un **array bidimensional** se usa para representar unidades en una tabla con filas y columnas, siempre y cuando cada unidad en la tabla sea del mismo tipo de datos. Los arrays bidimensionales son útiles para representar juegos de mesa, como ajedrez, tic-tac-toe o Scrabble, así como en gráficas de computadora, donde el monitor se considera un array bidimensional. Se accede a un componente por medio de un array bidimensional especificando los índices de fila y columna del objeto en el array. Ésta es una tarea familiar. Por ejemplo, si se busca una calle en un mapa, se busca el nombre de la calle al reverso del mapa para encontrar las coordenadas de la calle, por lo regular una letra y un número. La letra especifica una columna y el número una fila. La calle se encuentra donde se juntan la fila y la columna.

Array bidimensional Colección de componentes, todos del mismo tipo, estructurados en dos dimensiones. Se accede a cada componente por medio de un par de índices que representan la posición del componente en cada dimensión.

La figura 12-8 muestra un array bidimensional con 100 filas y 9 columnas. El acceso a las filas es por medio de un entero entre 0 y 99; el acceso a las columnas es mediante un entero entre 0 y 8. El acceso a cada componente es por medio de un par de fila y columna; por ejemplo 0, 5.

Un array bidimensional es declarado de la misma forma que un array unidimensional, a excepción de que se deben especificar tamaños para dos dimensiones. En la siguiente página se muestra, junto con un ejemplo, la plantilla de sintaxis para declarar un array con más de una dimensión.

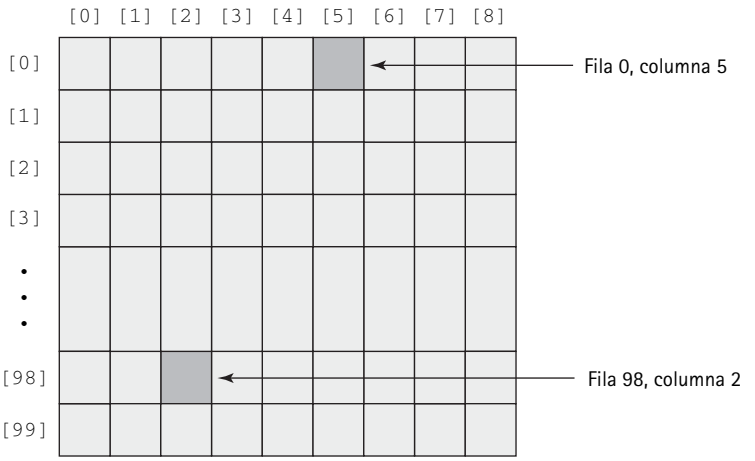


Figura 12-8 *Un array bidimensional*

ArrayDeclaration

```
DataType ArrayName [ ConstIntExpression ] [ ConstIntExpression ] . . . ;
```

```
const int NUM_ROWS = 100;
const int NUM_COLS = 9;
 ⋮
float alpha[NUM_ROWS][NUM_COLS];
```

Primera dimensión

Segunda dimensión

Este ejemplo declara que `alpha` es un array bidimensional, cuyos componentes son todos valores `float`. La declaración crea el array que se representa en la figura 12-8.

Para acceder a un componente individual del array `alpha` se usan dos expresiones (una para cada dimensión) para especificar su posición. Cada expresión se encuentra en su propio par de corchetes junto al nombre del array:

```
alpha[0][5] = 36.4;
```

Número de fila

Número de columna

La plantilla de sintaxis para el acceso a un componente de array es

ArrayComponentAccess

```
ArrayName [ IndexExpression ] [ IndexExpression ] . . .
```

Como en el caso de los arrays unidimensionales, cada expresión de índice debe resultar en un valor entero.

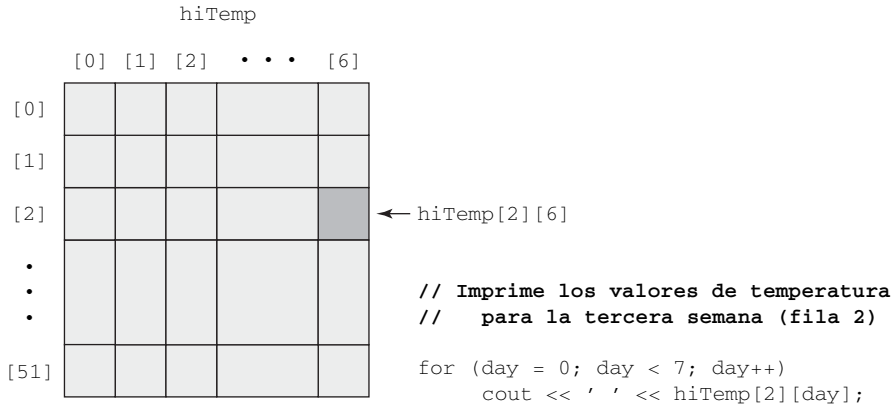


Figura 12-9 *Array* `hiTemp`

Veamos ahora algunos ejemplos. Aquí está la declaración de un array bidimensional con 364 componentes enteros ($52 \times 7 = 364$):

```
int hiTemp[52][7];
```

`hiTemp` es un array con 52 filas y 7 columnas. Cada lugar en el array (cada componente) puede contener cualquier valor `int`. Nuestra intención es que el array contenga las temperaturas máximas para cada día en un año. Cada fila representa una de las 52 semanas de un año, y cada columna representa uno de los 7 días de una semana. (Para mantener el ejemplo sencillo, haremos caso omiso del hecho de que hay 365 —y a veces 366— días en un año.) La expresión `hiTemp[2] [6]` se refiere al valor `int` en la tercera fila (fila 2) y la séptima columna (columna 6). Semánticamente, `hiTemp[2] [6]` es la temperatura para el séptimo día de la tercera semana. El fragmento de código representado en la figura 12-9 imprimiría los valores de temperatura para la tercera semana.

Otra representación de los mismos datos podría ser como sigue:

```
enum DayType
{
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
};

int hiTemp[52][7];
```

Aquí, `hiTemp` se declara del mismo modo que anteriormente, pero podemos usar una expresión tipo `DayType` para el índice de columna. `hiTemp[2] [SUNDAY]` corresponde al mismo componente que `hiTemp[2] [6]` en el primer ejemplo. (Recuerde que los enumeradores como `MONDAY, TUESDAY, . . .` son representados internamente como los enteros 0, 1, 2, ...) Si `day` es del tipo `DayType`, y `week` es del tipo `int`, el fragmento de código que se representa en la figura 12-10 coloca el array completo a 0. (Observe que, mediante el uso de `DayType`, los valores de temperatura en el array comienzan con el primer lunes del año, y no necesariamente con el 1 de enero.)

Otra manera de visualizar un array bidimensional es verlo como una estructura en la cual cada componente tiene dos características. Por ejemplo, en el siguiente código

```
enum Colors {RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET};
enum Makes
{
    FORD, TOYOTA, HYUNDAI, JAGUAR, CITROEN, BMW, FIAT, SAAB
};
const int NUM_COLORS = 7;
```

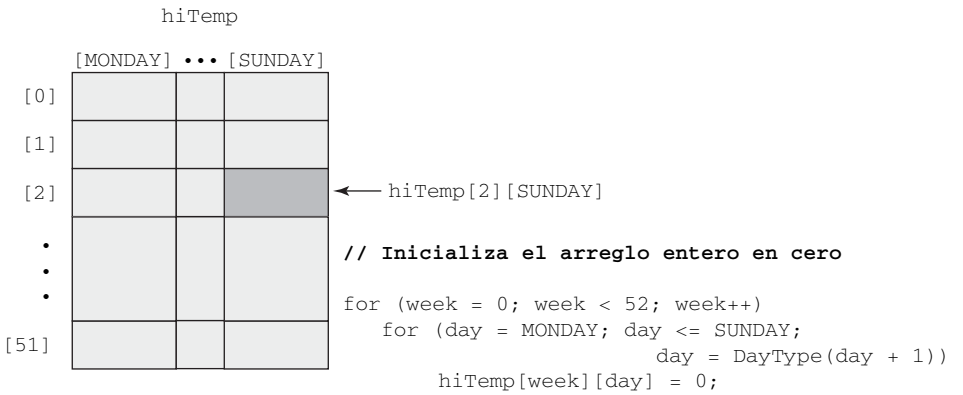


Figura 12-10 *Array `hiTemp` (forma alternativa)*

```
const int NUM_MAKES = 8;

float crashRating[NUM_COLORS][NUM_MAKES];    // Array de choque
                                             // probabilidades por color
                                             // y marca
 ⋮
crashRating[BLUE][JAGUAR] = 0.83;        // Los jaguares azules tienen una
                                         //   probabilidad de choque de 0.83
crashRating[RED][FORD] = 0.19;           // Los fords rojos tienen una
                                         //   probabilidad de choque de 0.19
```

la estructura de datos usa una dimensión para representar el color, y la otra para representar la marca de un automóvil. En otras palabras, ambos índices tienen un contenido semántico; un concepto que hemos analizado en la sección anterior.

12.5 Procesamiento de arrays bidimensionales

El procesamiento de datos en un array bidimensional significa, en general, el acceso al array en uno de cuatro patrones: aleatorio, a lo largo de filas, a lo largo de columnas, o a través del array completo. Cada una de estas maneras podrá también involucrar el procesamiento de sub-arrays.

La forma más sencilla de acceder a un componente es fijarse directamente en una ubicación dada. Por ejemplo, un usuario introduce coordenadas de un mapa, las cuales usamos como índices en un array de nombres de calles para acceder al nombre deseado en estas coordenadas. Este proceso se llama *acceso aleatorio* porque el usuario podrá introducir cualquier combinación de coordenadas al azar.

Hay muchos casos en los cuales quisiéramos realizar una operación en todos los elementos de una fila o columna en particular en un array. Considere el array `hiTemp` que hemos definido previamente, donde las filas representan semanas del año y las columnas representan días de la semana. Si quisiéramos saber el promedio de la temperatura mayor para una semana dada, sumaríamos los valores en esta fila para dividir el total entre 7. Si quisiéramos conocer el promedio para un determinado día de la semana, sumaríamos los valores en esta columna para dividir el total entre 52. El primer caso es el acceso por fila; el segundo caso es el acceso por columna.

Ahora supongamos que quisiéramos determinar el promedio para el año. Tenemos que acceder a cada elemento en el array, sumarlos y dividirlos entre 364. En este caso el orden de acceso por fila o por columna no es importante. (Lo mismo aplica cuando inicializamos cada elemento de un array a cero.) Esto es acceso a través del array.

Hay ocasiones en que tenemos que acceder a cada elemento del array en un cierto orden, ya sea por filas o por columnas. Por ejemplo, si quisiéramos el promedio para cada semana, tendríamos que pasar por todo el array, tomando cada fila a su vez. Sin embargo, si quisiéramos el promedio para cada día de la semana, pasaríamos a través del array tomando cada columna a su vez.

Veamos con más detalle estos patrones de acceso, considerando cuatro ejemplos comunes de procesamiento de arrays.

1. Sumar las filas.
2. Sumar las columnas.
3. Inicializar el array a ceros (o algún valor especial).
4. Imprimir el array.

Primero vamos a definir algunas constantes y variables, usando identificadores generales, como `row` y `col`, en lugar de identificadores dependientes de problemas. Luego consideraremos cada algoritmo en términos del procesamiento bidimensional de arrays.

```
const int NUM_ROWS = 50;
const int NUM_COLS = 50;

int arr[NUM_ROWS][NUM_COLS];      // Un array bidimensional
int row;                          // Un índice de renglón
int col;                          // Un índice de columna
int total;                        // Una variable para sumar
```

Sumar las filas

Supongamos que quisiéramos sumar la fila 3 (la cuarta fila) en el array, e imprimir el resultado. Lo podemos hacer fácilmente con un ciclo For:

```
total = 0;
for (col = 0; col < NUM_COLS; col++)
    total = total + arr[3][col];
cout << "Suma de renglón: " << total << endl;
```

Este ciclo For corre a través de cada columna de `arr`, manteniendo el índice de fila fijo en 3. Cada valor en la fila 3 se suma a `total`.

Ahora supongamos que quisiéramos sumar e imprimir dos filas, la 2 y la 3. Podemos usar un ciclo anidado y hacer del índice de fila una variable:

```
for (row = 2; row < 4; row++)
{
    total = 0;
    for (col = 0; col < NUM_COLS; col++)
        total = total + arr[row][col];
    cout << "Suma de renglón: " << total << endl;
}
```

El ciclo exterior controla las filas y el interior las columnas. Para cada valor de `row` se procesa cada columna; luego el ciclo exterior se mueve hacia la siguiente fila. En la primera iteración del ciclo exterior, `row` se mantiene en 2 y `col` se va de 0 a `NUM_COLS-1`. Por tanto, se accede al array en el siguiente orden:

```
arr[2][0]  [2][1]  [2][2]  [2][3] . . .   [2][NUM_COLS-1]
```

En la segunda iteración del ciclo exterior, `row` se incrementa a 3 y se accede al array como sigue:

```
arr[3][0]  [3][1]  [3][2]  [3][3] . . .   [3][NUM_COLS-1]
```

Podemos generalizar este procesamiento de la fila para correr a través de cada una, haciendo que el ciclo exterior corra de 0 hasta `NUM_ROWS-1`. Sin embargo, si queremos acceder sólo a una parte del array (procesamiento de sub-array), las variables dadas son declaradas como

```
int rowsFilled;   // Los datos están en 0..rowsFilled-1
int colsFilled;   // Los datos están en 0..colsFilled-1
```

luego escribimos el fragmento de código de la siguiente manera:

```
for (row = 0; row < rowsFilled; row++)
{
    total = 0;
    for (col = 0; col < colsFilled; col++)
        total = total + arr[row][col];
    cout << "Suma de renglón: " << total << endl;
}
```

La figura 12-11 ilustra el procesamiento de sub-arrays por fila.

Sumar las columnas

Supongamos que quisiéramos sumar e imprimir cada columna. A continuación se muestra el código para realizar esta tarea. Nuevamente hemos generalizado el código para sumar sólo la parte del array que contiene datos válidos.

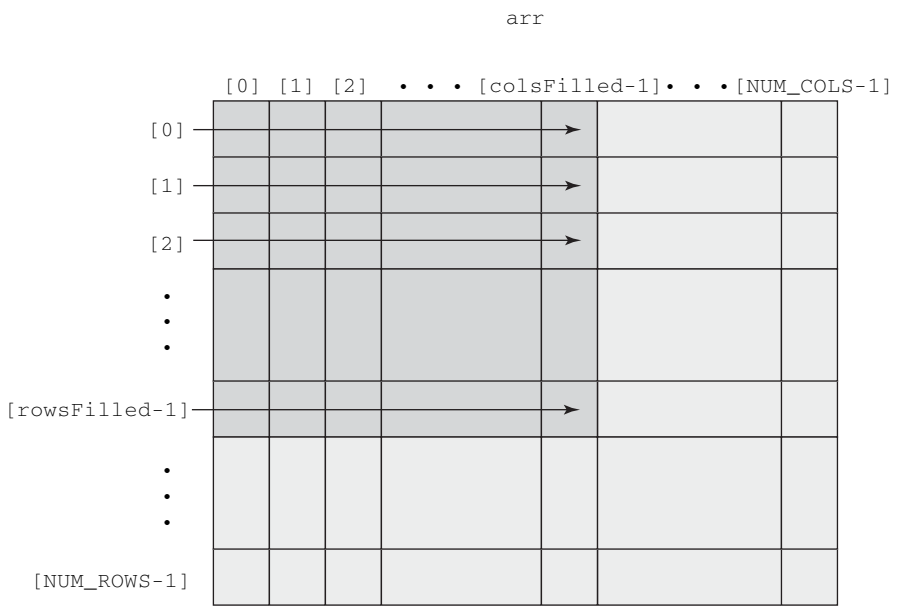


Figura 12-11 *Procesamiento parcial de arrays por fila*

```
for (col = 0; col < colsFilled; col++)
{
    total = 0;
    for (row = 0; row < rowsFilled; row++)
        total = total + arr[row][col];
    cout << "Suma de columna: " << total << endl;
}
```

En este caso el ciclo exterior controla la columna y el interior la fila. Todos los componentes en la primera columna se acceden y se suman antes de que cambie el índice del ciclo exterior y se acceda a los componentes en la segunda columna. La figura 12-12 ilustra el procesamiento de sub-array por columna.

Inicializar el array

Como en el caso de arrays unidimensionales, podemos inicializar un array bidimensional por medio de su declaración o usando sentencias de asignación. Para que la inicialización de un array de dos filas por tres columnas se vea así:

```
14   3  -5
 0  46   7
```

podemos usar la siguiente declaración:

```
int arr[2][3] =
{
    {14, 3, -5},
    {0, 46, 7}
};
```

En esta declaración, la lista de inicializadores consiste en dos elementos, cada uno de los cuales es en sí una lista de inicializadores. La primera lista interior de inicializadores almacena 14, 3 y –5 en la fila 0 del array; la segunda almacena 0, 46 y 7 en la fila 1. El uso de dos listas de inicializadores

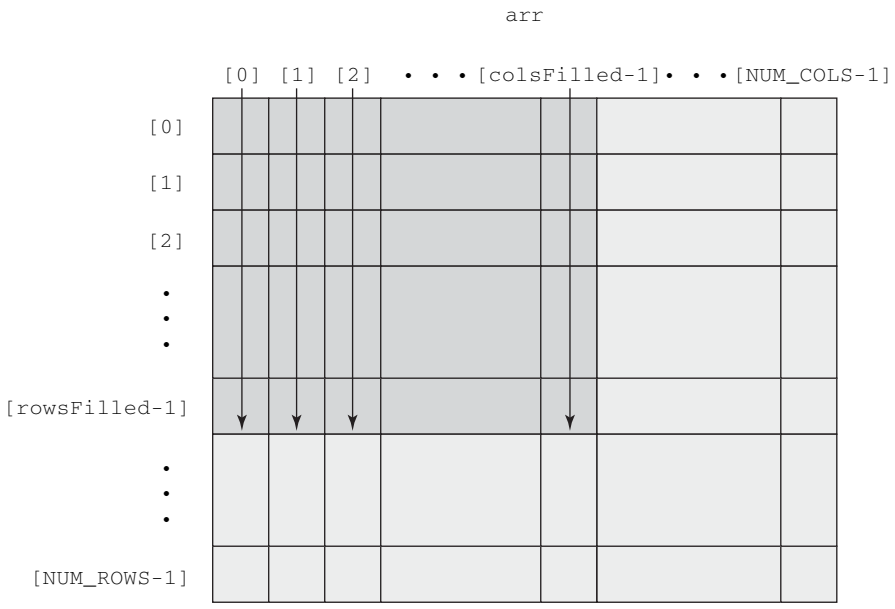


Figura 12–12 *Procesamiento parcial del array por columna*

tiene sentido si se piensa en cada fila del array bidimensional como un array unidimensional de tres `ints`. La primera lista de inicializadores inicializa el primer array (la primera fila) y la segunda lista inicializa el segundo array (la segunda fila). Más adelante en este capítulo volveremos a esta noción de visualizar un array bidimensional como un array de arrays.

La inicialización de un array en su declaración es poco práctico si el array es grande. Para un array de 100 filas por 100 columnas no queremos listar 10 000 valores. Si todos los valores son diferentes, se deberán almacenar en un archivo e ingresarlos en el array al momento de la ejecución. Si todos los valores son iguales, el planteamiento común es usar ciclos For anidados y una sentencia de asignación. A continuación se muestra un segmento de código de propósito general que inicializa a cero un array con filas `NUM_ROWS` y columnas `NUM_COLS`:

```
for (row = 0; row < NUM_ROWS; row++)
    for (col = 0; col < NUM_COLS; col++)
        arr[row][col] = 0;
```

En este caso inicializamos el array fila por fila, pero podríamos haber recorrido por cada fila con la misma facilidad. El orden no importa mientras accedemos a cada elemento.

Imprimir el array

Si queremos imprimir un array con una fila por línea, entonces tenemos otro caso de procesamiento de filas:

```
#include <iomanip>   // For setw()
 ⋮
for (row = 0; row < NUM_ROWS; row++)
{
    for (col = 0; col < NUM_COLS; col++)
        cout << setw(15) << arr[row][col];
    cout << endl;
}
```

Este fragmento de código imprime los valores del array en columnas que tienen el ancho de 15 caracteres. Como una cuestión de estilo correcto, este fragmento deberá estar precedido por un código que imprime encabezados sobre las columnas para identificar su contenido.

No hay regla que nos obligue a imprimir cada fila en una línea. Podríamos voltear el array de lado e imprimir cada columna en una línea, cambiando simplemente los dos ciclos For. Cuando se imprime un array bidimensional, se deberá considerar el orden de presentación más útil, y cómo el array cabe en la página. Por ejemplo, un array con 6 columnas y 100 filas se imprimiría mejor como 6 columnas con un largo de 100 líneas.

Casi todo el procesamiento de datos almacenados en un array bidimensional involucra el procesamiento por filas o por columnas. En la mayoría de nuestros ejemplos, el tipo de índice ha sido `int`, pero el patrón de operación de los ciclos es el mismo, sin importar de qué tipo son los índices.

Los patrones de ciclos para el procesamiento por filas y por columnas son tan útiles que los resumimos a continuación. Para generalizarlos más usamos `minRow` para el primer número de fila y `minCol` para el primer número de columna. Recuerde que el procesamiento de fila tiene un índice de fila en el ciclo exterior, y el procesamiento de columna tiene el índice de columna en el ciclo exterior.

Procesamiento de fila

```
for (row = minRow; row < rowsFilled; row++)
    for (col = minCol; col < colsFilled; col++)
        ⋮          // Cualquier procesamiento que sea requerido
```

Procesamiento de columna

```
for (col = minCol; col < colsFilled; col++)
    for (row = minRow; row < rowsFilled; row++)
        ⋮          // Cualquier procesamiento que sea requerido
```

12.6 Paso de arrays bidimensionales como argumentos

Previamente hemos dicho, en este capítulo, que cuando se declaran arrays unidimensionales como parámetros en una función, el tamaño del array normalmente se omite de los corchetes:

```
void SomeFunc( /* inout */ float alpha[],
               /* in */    int   size    )
{
    ⋮
}
```

Si incluimos un tamaño en los corchetes, el compilador lo ignora. Como hemos aprendido, la dirección base del argumento del invocador (la dirección de memoria del primer elemento del array) se pasa a la función. Ésta funciona para un argumento de cualquier tamaño. Puesto que la función no puede conocer el tamaño del array del invocador, pasamos el tamaño como un argumento (como en `SomeFunc` antes) o usamos una constante nombrada si la función siempre opera en un array de determinado tamaño.

Cuando se pasa un array bidimensional como un argumento, nuevamente se transmite la dirección base del array del invocador a la función. Sin embargo, no podemos omitir los tamaños de ambas dimensiones del array. Se puede omitir el tamaño de la primera dimensión (el número de filas), pero no de la segunda (el número de columnas). La razón es la siguiente:

En la memoria de la computadora, C++ almacena arrays bidimensionales en el orden de filas. Si se piensa en la memoria como una larga línea de células de memoria, la primera fila del array es seguida de la segunda fila, a la que sigue la tercera, etc. (véase la figura 12-13). Para localizar `beta[1] [0]` en esta figura, una función que recibe la dirección base de `beta` debe ser capaz de saber que existen cuatro elementos en cada fila; o sea que el array consiste en cuatro columnas. Por tanto, la declaración de un parámetro siempre deberá indicar el número de columnas:

```
void AnotherFunc( /* inout */ int beta[][4] )
{
    ⋮
}
```

Además, el número de columnas declaradas para el parámetro debe ser exactamente igual al número de columnas en el array del invocador. Como se puede derivar de la figura 12-13, si hay alguna discrepancia en el número de columnas, la función tomará acceso al elemento de array equivocado en la memoria.

Nuestra función `AnotherFunc` funciona para un array bidimensional de cualquier número de filas, siempre que tenga exactamente cuatro columnas. En la práctica rara vez escribimos programas que usan arrays con un número variado de filas pero con el mismo número de columnas. A fin de evitar errores de correspondencia en el tamaño de argumento y parámetro, conviene usar una sentencia Typedef para definir un tipo de array bidimensional, y luego declarar que tanto el argumento como el parámetro sean de este tipo. Por ejemplo, podríamos hacer la declaración

```
const int NUM_ROWS = 10;
const int NUM_COLS = 20;
typedef int ArrayType[NUM_ROWS][NUM_COLS];
```

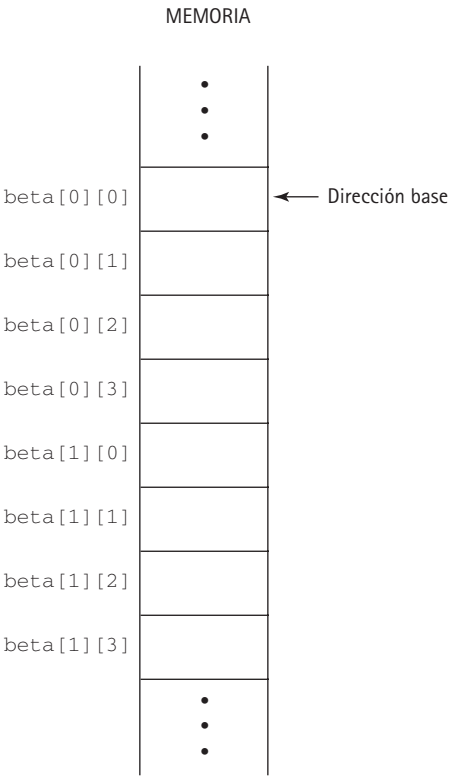


Figura 12-13 *Vista de la memoria para un array de dos filas por cuatro columnas*

y luego escribir la siguiente función de propósito general que inicializa todos los elementos de un array a un valor especificado:

```
void Initialize( /* out */ ArrayType arr,     // Array a inicializar
                 /* in */  int       initVal) // Valor inicial

// Inicializa cada elemento de arr a initVal

// Precondición:
//     initVal es asignada
// Poscondición:
//     arr[0..NUM_ROWS-1][0..NUM_COLS-1] == initVal

{
    int row;
    int col;

    for (row = 0; row < NUM_ROWS; row++)
        for (col = 0; col < NUM_COLS; col++)
            arr[row][col] = initVal;
}
```

El código de llamado podría luego declarar e inicializar uno o varios arrays del tipo `ArrayType`, haciendo llamadas a la función `Initialize`. Por ejemplo,

```
ArrayType delta;
ArrayType gamma;

Initialize(delta, 0);
Initialize(gamma, -1);
 ⋮
```

12.7 Otra forma de definir arrays bidimensionales

Hemos mencionado que un array bidimensional se puede visualizar como un array de arrays. Esta perspectiva es sustentada por C++ en el sentido de que los componentes de un array unidimensional no tienen que ser atómicos. Los componentes en sí pueden ser estructuras estructuradas, objetos de clase, o incluso arrays. Por ejemplo, nuestro array `hiTemp` se podría declarar como sigue:

```
typedef int WeekType[7];    // Tipo array para siete temperaturas leídas

WeekType hiTemp[52];        // Array de 52 arrays del tipo WeekType
```

Por medio de esta declaración, los 52 componentes del array `hiTemp` son arrays unidimensionales del tipo `WeekType`. En otras palabras, `hiTemp` tiene dos dimensiones. Podemos referirnos a cada fila como una entidad: `hiTemp[2]` se refiere al array de temperaturas para la semana 2. También podemos acceder a cada componente individual de `hiTemp` especificando ambos índices: `hiTemp[2] [0]` accede a la temperatura en el primer día de la semana 2.

¿Tiene alguna importancia la manera en que declaramos un array bidimensional? No en C++. La elección se deberá basar en la legibilidad y comprensibilidad. En ocasiones, las características de los datos se muestran con mayor claridad si ambos índices son especificados en una sola declaración. En otras ocasiones, el código es más claro si una dimensión es definida primero como un tipo de array unidimensional.

A continuación se presenta un ejemplo para una situación en la que es una ventaja definir un array bidimensional como un array de arrays. Si las filas fueron definidas primero como un tipo de array unidimensional, cada fila puede ser pasada a la función cuyo parámetro es un array unidimensional del mismo tipo. Por ejemplo, la siguiente función calcula y devuelve el valor máximo en un array del tipo `WeekType`.

```
int Maximum( /* in */ const WeekType data )  // a ser examinado

// Precondición:
//     data[0..6] son asignados
// Postcondition:
//     Poscondición: Valor de la función == valor máximo en data[0..6]

{
    int max;      // Temporalmente el valor máximo
    int index;    // Control del ciclo y variable del índice

    max = data[0];
    for (index = 1; index < 7; index++)
        if (data[index] > max)
            max = data[index];
    return max;
}
```

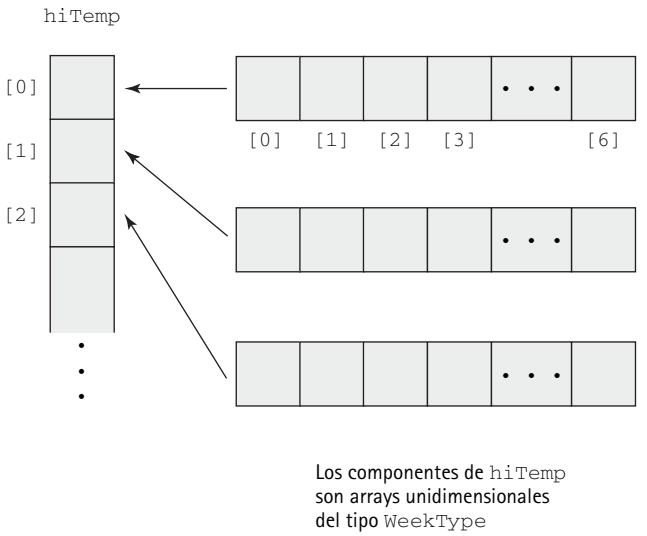


Figura 12–14 *Un array unidimensional de arrays unidimensionales*

Nuestra declaración de dos partes de `hiTemp` nos permite llamar `Maximum` usando un componente de `hiTemp` como sigue:

```
highest = Maximum(hiTemp[20]);
```

La fila 20 de `hiTemp` es pasada a `Maximum`, que lo trata como cualquier otro array unidimensional del tipo `WeekType` (véase la figura 12-14). Es conveniente pasar la fila como un argumento porque tanto ella como el parámetro de función son del mismo tipo nombrado, `WeekType`.

Una vez que `hiTemp` se declara como un array de arrays, podemos emitir la temperatura máxima de cada semana del año con el siguiente código:

```
cout << " Máxima de la semana" << endl
     << "Temperatura número" << endl;
for (week = 0; week < 52; week++)
    cout << setw(6) << week
         << setw(9) << Maximum(hiTemp[week]) << endl;
}
```

12.8 Arrays multidimensionales

C++ no limita el número de dimensiones que un array puede tener. Podemos generalizar nuestra definición de un **array** para que cubra todos los casos.

Array Colección de componentes, todos del mismo tipo, ordenados en *N* dimensiones ($N \geq 1$). Cada componente es accedido por *N* índices, de los que cada uno representa la posición del componente dentro de esta dimensión.

Se podrá ver que, a partir de las plantillas de sintaxis, que podemos tener todas las dimensiones que queramos. ¿Cuántas deberíamos tener en un caso particular? Se usa el mismo número de características que describen los componentes en el array.

Tomamos, por ejemplo, una cadena de tiendas departamentales. Se deberán mantener cifras de ventas mensuales para cada artículo por tienda. Hay tres importantes informaciones sobre cada

artículo: el mes en que fue vendido, la tienda donde se compró y el número de artículo. Podemos definir un tipo de array para resumir estos datos de la siguiente manera:

```
const int NUM_ITEMS = 100;
const int NUM_STORES = 10;
typedef int SalesType[NUM_STORES][12][NUM_ITEMS];

SalesType sales;   // Arrays de cifras de ventas
int       item;
int       store;
int       month;
int       numberSold;
int       currentMonth;
```

Una representación gráfica del array `sales` se muestra en la figura 12-15.

El número de componentes en `sales` es 12 000 (10 × 12 × 100). Si las cifras de ventas sólo están disponibles para los meses de enero a junio, entonces la mitad del array está vacía. Si queremos procesar los datos en el array, tenemos que usar el procesamiento de sub-array. El siguiente fragmento de programa suma e imprime el número total de cada artículo vendido por todas las tiendas en el presente año hasta la fecha.

```
for (item = 0; item < NUM_ITEMS; item++)
{
    numberSold = 0;
    for (store = 0; store < NUM_STORES; store++)
        for (month = 0; month <= currentMonth; month++)
            numberSold = numberSold + sales[store][month][item];
    cout << "Item #" << item << " Ventas a la fecha = " << numberSold
         << endl;
}
```

Puesto que `item` controla el ciclo For exterior, sumamos las ventas de cada artículo por `month` y `store`. Si queremos buscar las ventas totales para cada tienda, usamos `store` para controlar el ciclo For exterior, sumando sus ventas por `month` e `item` con los ciclos interiores.

```
for (store = 0; store < NUM_STORES; store++)
{
    numberSold = 0;
```

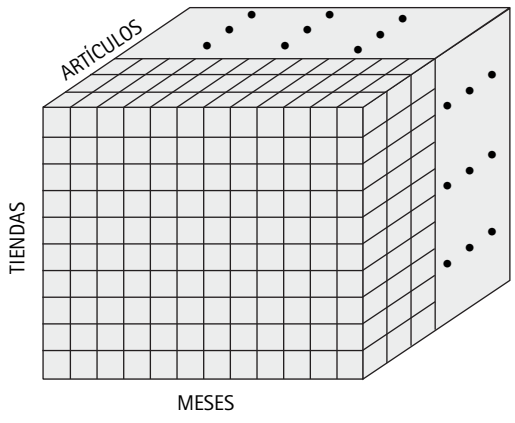


Figura 12-15 *Representación gráfica del array* `sales`

```
    for (item = 0; item < NUM_ITEMS; item++)
        for (month = 0; month <= currentMonth; month++)
            numberSold = numberSold + sales[store][month][item];
    cout << "Store #" << store << " Ventas a la fecha = " << numberSold
         << endl;
}
```

Se necesitan dos ciclos para acceder a cada componente en un array bidimensional; son necesarios tres ciclos para acceder a cada componente en un array tridimensional. La tarea a realizar determina el índice que controla el ciclo exterior, el ciclo medio y el ciclo interior. Si queremos calcular las ventas mensuales por tienda, `month` controla el ciclo exterior y `store` controla el ciclo medio. Si queremos calcular las ventas mensuales por artículo, `month` controla el ciclo exterior e `item` el ciclo medio.

Si queremos dar seguimiento a los departamentos que venden cada artículo, podemos agregar una cuarta dimensión.

```
enum Departments {A, B, C, D, E, F, G};
const int NUM_DEPTS = 7;
typedef int SalesType[NUM_STORES][12][NUM_ITEMS][NUM_DEPTS];
```

¿Cómo visualizaríamos esta nueva estructura? ¡No muy fácilmente! Por fortuna no tenemos que visualizar una estructura a fin de usarla. Si queremos la cifra de ventas en la tienda 1 durante el mes de junio para el artículo número 4 en el departamento C, simplemente accedemos al elemento de array

```
sales[1][5][4][C]
```

Cuando un array multidimensional es declarado como un parámetro en una función, C++ requiere que se indiquen los tamaños de todas las dimensiones con excepción de la primera. Para nuestra versión tetradimensional de `SalesType`, un encabezado de función se podría ver así:

```
void DoSomething( /* inout */ int arr[][12][NUM_ITEMS][NUM_DEPTS] )
```

o, mejor aún, así:

```
void DoSomething( /* inout */ Sales arr )
```

La segunda versión es la más segura (y la menos confusa a la vista). Asegura que los tamaños de todas las dimensiones del parámetro se relacionan exactamente con los del argumento. En el caso de la primera versión, la razón por la cual se deben declarar los tamaños de todas las dimensiones menos la primera es la misma que analizamos para arrays bidimensionales. Puesto que los arrays son almacenados en forma lineal en la memoria (un elemento de array tras otro), el compilador debe usar esta información de tamaño para localizar correctamente un elemento que se ubica dentro del array.

Caso práctico de resolución de problemas

Calcular estadísticas de examen

PROBLEMA Usted es el evaluador en su clase de Política. El maestro le ha pedido que prepare la siguiente estadística para el último examen: promedio de calificación, calificación máxima, calificación mínima, cantidad de calificaciones arriba del promedio, y cantidad de calificaciones debajo del promedio. Puesto que se trata del primer examen, usted decide escribir un programa para calcular estas estadísticas, de modo que pueda usar el programa también para el resto de los exámenes.

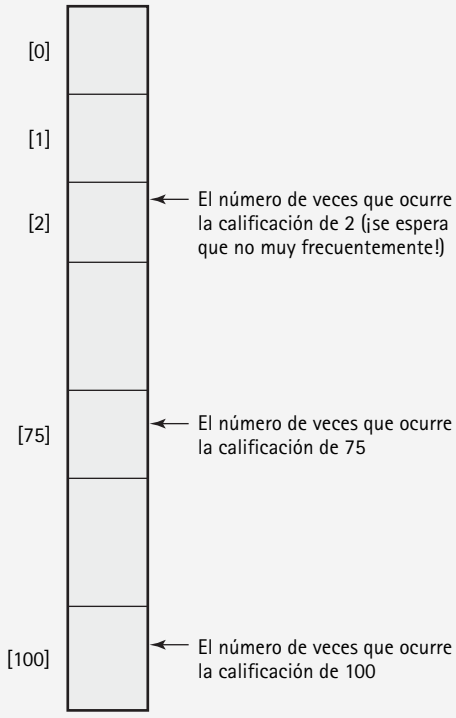


Figura 12–16

ANÁLISIS Vamos a abstraer este problema del contexto dado para asomarnos a las tareas de manera aislada. Hay tres cosas separadas que se deben hacer con este problema: calcular un promedio de valores en un archivo, buscar el valor mínimo y el valor máximo en un archivo, y comparar cada valor con el promedio. Existen diferentes planteamientos para la solución de este problema. En el siguiente capítulo examinaremos una técnica de solución de problemas completamente distinta; sin embargo, aquí basamos nuestra solución sobre el hecho de que los valores en la lista son entre 0 y 100. Usamos un array donde los índices tienen un contenido semántico: cada índice representa una calificación.

La analogía a mano es marcar 101 líneas en una hoja de papel y numerar (o etiquetar) las líneas de 0 a 100. Cada número de línea representa una posible calificación. Conforme se lee una calificación, se asienta el símbolo # en la línea cuyo número es el mismo que la calificación. Después de registrar cada calificación de este modo, se realiza la suma de las calificaciones, sumando los productos de cada calificación (número de línea) multiplicado por el número de símbolos # en esta línea. El número de calificaciones se puede calcular ya durante la lectura o cuando se calcula la suma.

Para calcular la calificación más baja, empiece viendo la línea número 0 y vea hacia delante; el número de línea de la primera línea con un símbolo # es la calificación más baja. Para calcular la calificación más alta, empiece a ver hacia atrás desde la línea número 100, y el número de línea de la primera línea que contenga un símbolo # es la calificación más alta. Para determinar cuántas calificaciones están arriba del promedio, empiece en la línea cuyo número es el promedio de calificación más 1, y cuente los símbolos # en las líneas desde ahí hasta la línea 100. Para determinar cuántas calificaciones están debajo del promedio, sume los símbolos # desde la línea cuyo número es el promedio truncado hasta la línea 0.

El equivalente de la estructura de datos de su hoja de papel es un array de enteros declarado a ser de un tamaño de 101. El índice es el número de línea; el componente corresponde a donde se asientan los símbolos # (incremente el componente) cada vez que ocurra la calificación que corresponde al índice.

INPUT Archivo cuyo nombre es introducido desde el teclado y que contenga calificaciones del examen.

OUTPUT Archivo cuyo nombre es introducido desde el teclado y que muestra las siguientes estadísticas, correctamente etiquetadas.

Número de calificaciones
Calificación promedio
Calificación más baja
Calificación más alta
Número de calificaciones arriba del promedio
Número de calificaciones abajo del promedio

Principal **Nivel 0**

```
Abrir archivos
Introducir calificaciones
Calcular promedio
Calcular más alta
Calcular más baja
Calcular arriba del promedio
Calcular abajo del promedio
Cerrar archivos
```

Podemos utilizar las mismas funciones para abrir archivos, que hemos usado en otros programas. Sin embargo, es necesario cambiar el encabezado impreso en la salida.

Input Grades **Nivel 1**

Esta función debe tener el nombre del archivo y el array como parámetros. El programa debe conocer el número de calificaciones. Como hemos dicho, este valor puede ser calculado mientras se leen las calificaciones o mientras se suman las calificaciones para obtener el promedio. Vamos a calcularlo aquí y pasarlo como un argumento a la función que calcula el promedio.

(Inout: inData, grades, numGrades)

```
Colocar todas las calificaciones en cero
Colocar numGrades en 0
Leer calificación
MIENTRAS QUE NO SEA eof
    Incrementar grades[grade]
    Incrementar numGrades
    Leer calificación
```

CalculateAverage(In: grades, numGrades)
Out: Function value

```
Colocar suma en 0
Índice FOR de 0 a 100
    Colocar suma en sum + grades[index]*index;
Devolver float(sum) / float(numGrades);
```

CalculateHighest(In: grades)
Out: Function value

```
Colocar highGrade a 100;
MIENTRAS grades[highGrade] igual a 0
    Disminuir highGrade
Devolver highGrade;
```

CalculateLowest(In: grades)
Out: Function value

```
Colocar lowGrade a 0
MIENTRAS grades[lowGrade] igual a 0
    Incrementar lowGrade
Devolver lowGrade
```

CalculateAboveAverage(In: grades, average)
Out: Function value

```
Colocar averagePlus a int(average) + 1
Colocar número a cero
Índice FOR de averagePlus a 100
    Colocar número a number + grades[index]
Devolver número
```

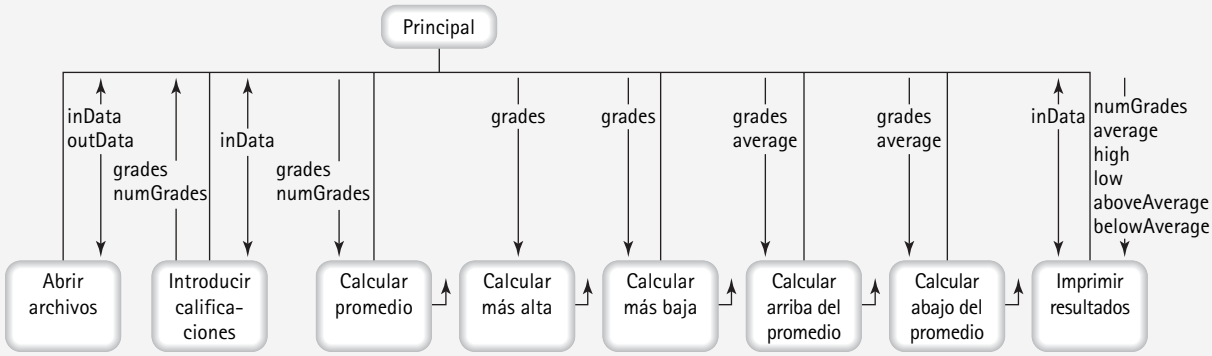
CalculateBelowAverage(In: grades, average)
Out: Function value

```
Colocar truncatedAverage a promedio (int)
Colocar número a cero
Índice FOR de 0 a truncatedAverage
    Colocar número a number + grades[index]
Devolver número
```

PrintResults(Inout: outData; In: numGrades, average, highest, lowest, numberAbove, numberBelow)

```
Imprimir en outData "El número de calificaciones es " numGrades
Imprimir en outData "La calificación promedio es " average
Imprimir en outData "La calificación más alta es " highest
Imprimir en outData "La calificación más baja es " lowest
Imprimir en outData "El número de calificaciones arriba del promedio es " aboveAverage
Imprimir en outData "El número de calificaciones abajo del promedio es " belowAverage
```

DIAGRAMA DE ESTRUCTURA DE MÓDULOS




```
//**************************************************************************
// Programa estadísticas
// Este programa calcula el promedio, la calificación alta, calificación
// baja, el número arriba del promedio y el número abajo del promedio
// para un archivo de calificaciones de prueba.
// Para ahorrar espacio, se omite de cada función los comentarios
// de precondición que documentan las suposiciones hechas acerca de
// datos de parámetros de entrada válidos. Éstos se incluirían
// en un programa dedicado a uso real.
// Suposición: el archivo contiene por lo menos un valor no cero.
//**************************************************************************

#include <iostream>
#include <fstream>
#include <iomanip>

using namespace std;

// Declarar prototipos de función
void OpenFiles(ifstream& inData, ofstream& outData);
void InputGrades(int grades[], int& numGrades, ifstream& inData);
float CalculateAverage(int const grades[], int numGrades);
int CalculateHighest(int const grades[]);
int CalculateLowest(int const grades[]);
int CalculateAboveAverage(int const grades[], float average);
int CalculateBelowAverage(int const grades[], float average);
void PrintResults(ofstream& outData, int numGrades, float average,
    int highest, int lowest, int aboveAverage, int belowAverage);

int main()
{
    int grades[101];          // Array de contadores para cada calificación
    int numGrades;            // Número de calificaciones
    float average;            // Calificación promedio
    int highest;              // Calificación más alta
    int lowest;               // Calificación más baja
    int aboveAverage;         // Número de calificaciones arriba del promedio
    int belowAverage;         // Número de calificaciones abajo del promedio
```

```
    // Declarar y abrir archivos
    ifstream inData;
    ofstream outData;
    OpenFiles(inData, outData);
    if ( !inData || !outData )
    {
        cout << "Los archivos no se abrieron con éxito. " << endl;
        return 1;
    }

    // Leer y procesar las calificaciones
    InputGrades(grades, numGrades, inData);
    average = CalculateAverage(grades, numGrades);
    highest = CalculateHighest(grades);
    lowest = CalculateLowest(grades);
    aboveAverage = CalculateAboveAverage(grades, average);
    belowAverage = CalculateBelowAverage(grades, average);
    PrintResults(outData, numGrades, average, highest, lowest,
        aboveAverage, belowAverage);

    inData.close();
    outData.close();
    return 0;
}

//*********************************************************************

 void OpenFiles( /* inout */ ifstream& text,
                 /* inout */ ofstream& outFile )

// La función OpenFiles lee los nombres del archivo de entrada
// y el archivo de salida y los abre para procesamiento
// Poscondición:
//     Los archivos han sido abiertos Y se ha escrito una etiqueta
//     en el archivo de salida

{
    string inFileName;
    string outFileName;
    cout << "Introduzca el nombre del archivo que será procesado"
         << endl;
    cin >> inFileName;
    text.open(inFileName.c_str());
    cout << "Introduzca el nombre del archivo de salida" << endl;
    cin >> outFileName;
    outFile.open(outFileName.c_str());
    outFile << "Análisis de los exámenes en el archivo " << inFileName
          << endl << endl;
}

//*********************************************************************

float CalculateAverage
    ( /* inout */ int const grades[], // Estructura de calificación
      /* in */     int numGrades )    // Número de calificaciones
```

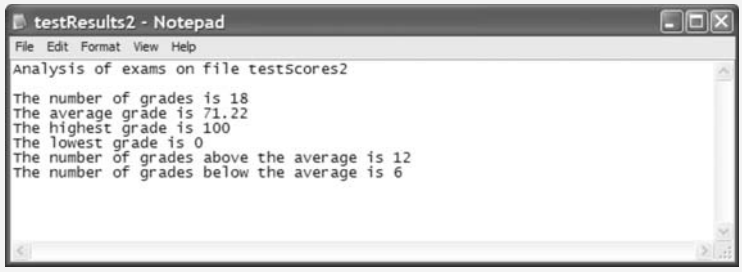
```
// Esta función calcula la calificación de prueba promedio
// Poscondición:
//     El valor de retorno es la calificación promedio

{
    int sum = 0;
    // Sumar el número de calificaciones multiplicadas por el índice
    for (int index = 0; index <= 100; index++)
        sum = sum + grades[index] * index;

    return float(sum) / float(numGrades);
}

//*************************************************************************

void InputGrades
    ( /* inout */ int grades[],     // Estructura de calificación
      /* inout */ int& numGrades,   // Número de calificaciones
      /* inout */ ifstream& inData) // Archivo de entrada

// Las calificaciones son introducidas desde el archivo inData. Cada espacio
// en las calificaciones se establece en cero. A medida que se lee cada
// calificación, ésta se usa como un índice y se incrementa ese espacio.
// Precondición:
//     El archivo no está vacío
// Poscondición:
//     Para cada calificación del archivo, las calificaciones, indizadas
//     por la calificación, han sido incrementadas

{
    int grade;
    // Establezca en cero el array de contadores
    for (int index = 0; index <= 100; index++)
        grades[index] = 0;
    numGrades = 0;

    inData >> grade;                    // Leer cada carácter

    // Incremantar los contadores para las calificaciones
    while (inData)
    { // Process data
        grades[grade]++;
        numGrades++;
        inData >> grade;
    }
}

//*************************************************************************

int CalculateHighest
    ( /* in */ int const grades[] ) // Estructura de calificación
```

```
// Esta función calcula la calificación más alta comenzando en el índice
// 100 y trabajando hacia atrás hasta que se encuentre un espacio no cero.
// El índice de este espacio es la calificación más alta.
// Poscondición:
//     El valor devuelto es la calificación más alta
{
    int highGrade = 100;
    // El índice de la calificación no cero es la calificación alta
    while (grades[highGrade] == 0)
        highGrade--;
    return highGrade;
}

//*************************************************************************

int CalculateLowest
    ( /* in */ int const grades[] ) // Estructura de calificación

// Esta función calcula la calificación mínima comenzando en el índice
// cero y trabajando hacia delante hasta que se encuentra un espacio no cero.
// El índice de este espacio es la calificación mínima.
// Poscondición:
//     El valor devuelto es la calificación mínima

{
    // El índice de la primera calificación no cero es la calificación baja
    int lowGrade = 0;
    while (grades[lowGrade] == 0)
        lowGrade++;
    return lowGrade;
}

//*************************************************************************

int CalculateAboveAverage
  ( /* in */ int const grades[],        // Estructura de calificación
    /* inout */  float average )         // Calificación promedio

// Esta función calcula el número de calificaciones arriba
// del promedio al contar los números del array del promedio
// redondeado al espacio 100.
// Poscondición:
//     El valor devuelto es el número de calificaciones arriba del promedio
{
    int averagePlus = (int)(average) + 1;
    int index;
    int number = 0;
    for (index = averagePlus; index <= 100; index++)
        number = number + grades[index];
    return number;
}

//*************************************************************************
```

```
int CalculateBelowAverage
  ( /* in */ int const grades[],       // Estructura de calificación
   /* inout */  float average )         // Calificación promedio

// Esta función calcula el número de calificaciones abajo del promedio
// contando los números del array del índice cero al índice
// promedio truncado.
// Poscondición:
//     El valor devuelto es el número de calificaciones abajo del promedio

{
    int truncatedAverage = (int) (average);
    int index;
    int number = 0;
    // Sumar el número de calificaciones abajo del promedio
    for (index = 0; index <= truncatedAverage; index++)
        number = number + grades[index];
    return number;
}

//***************************************************************************

void PrintResults
    ( /* inout */ ofstream& outData, // Archivo de salida
      /* in */    int numGrades,     // Número de calificaciones
      /* in */    float average,     // Promedio
      /* in */    int highest,       // Calificación máxima
      /* in */    int lowest,        // Calificación mínima
      /* in */    int aboveAverage,  // Número arriba
      /* in */    int belowAverage)  // Número abajo

// Las estadísticas se imprimen en el archivo outData
// Poscondición:
//     El archivo de salida se ha abierto con éxito
// Poscondición:
//     Las estadísticas se han escrito en outData, marcadas de manera
//     apropiada

{
    outData << "El número de calificaciones es " << numGrades << endl;
    outData << fixed << setprecision(2)
            << "La calificación promedio es " << average << endl;
    outData << "La calificación más alta es " << highest << endl;
    outData << "La calificación mínima es " << lowest << endl;
    outData << "El número de calificaciones arriba del promedio es "
            << aboveAverage  << endl;
    outData << "El número de calificaciones abajo del promedio es "
            << belowAverage << endl;
}
```

PRUEBA Para las pruebas de este programa, son los valores de las calificaciones, mas no el tamaño del archivo, lo que determina los casos de prueba. Los casos de prueba deben incluir valores de MinGrade, MaxGrade y valores entre los dos. A continuación se presenta un juego de datos de muestra que cumple con este criterio. Los resultados aparecen enseguida.

```
88
87
88
66
55
56
75
75
78
80
80
90
99
87
44
34
0
100
```



Estudio de caso de solución de problema

Grupo de rock favorito

PROBLEMA En una pequeña universidad, cuatro grupos de rock del campus han organizado un proyecto de recolección de fondos, donde habrá una competencia entre los grupos. Cada estudiante puede votar a favor de su grupo favorito, y habrá dos premios: el mejor grupo y la clase con la mejor participación obtendrán un premio cada uno.

INPUT Un número arbitrario de votos en un archivo `voteFile`, donde cada voto es representado como un par de números; un número de clase (de 1 a 4) y un número para el grupo de rock (de 1 a 4), así como nombres de grupos, introducidos desde el teclado (que se usará para la impresión del resultado).

OUTPUT Los siguientes tres elementos, escritos en un archivo `reportFile`: un reporte en forma de tabla que muestra cuántos votos ha recibido cada grupo de rock en cada clase, el número total de votos para cada grupo de rock, y el número total de votos emitidos por cada clase.

ANÁLISIS Los datos consisten en un par de números para cada voto. El primer número es el número de clase; el segundo es el número del grupo de rock.

Si hiciéramos el análisis en forma manual, nuestra primera tarea sería repasar los datos, contando cuántas personas en cada clase votó por cada grupo. Probablemente crearíamos una tabla con clases en forma vertical, y nombres de grupos de rock como encabezados en forma horizontal. Cada voto se registraría como un símbolo # en la fila y columna apropiadas (véase la figura 12-17).

	Fish	Snake	Sharks	Leopards
1	//// //	//	//// //// //	////
2	//// ////	//	////	///
3	//	//// ///	//// //// ////	///
4	////	//// ///	//// ////	//

Figura 12-17 *Tabla de conteo de votos*

Después del registro de todos los votos, una suma de cada columna nos diría cuántos votos fueron emitidos para cada grupo. Una suma de cada fila nos diría cuántas personas votaron en cada clase.

Como es el caso en tantas ocasiones, podemos usar este algoritmo manual directamente en nuestro programa. Podemos crear un arrary bidimensional en el cual cada componente es un contador para el número de votos para un grupo particular en cada clase; por ejemplo, el valor indizadoindizado por `[2] [1]` sería el contador para los votos en la clase 2 (segundo año) para el grupo 1. Bueno, no tanto. Los arrays de C++ son indizadoindizados empezando en 0, así que el componente correcto del array sería indizadoindizado por `[1] [0]`. Cuando introducimos un número de clase y un número de grupo, debemos recordar que tenemos que restar 1 de cada uno antes del indizadoindizado en el array. De la misma manera tenemos que sumar 1 a un índice de array que representa un número de clase antes de imprimirlo.

ESTRUCTURAS DE DATOS Un array bidimensional denominado `votes`, donde las filas representan clases y las columnas representan grupos.

Un array unidimensional de cadenas que contiene los nombres de los grupos, a ser usado para la impresión (véase la figura 12-18).

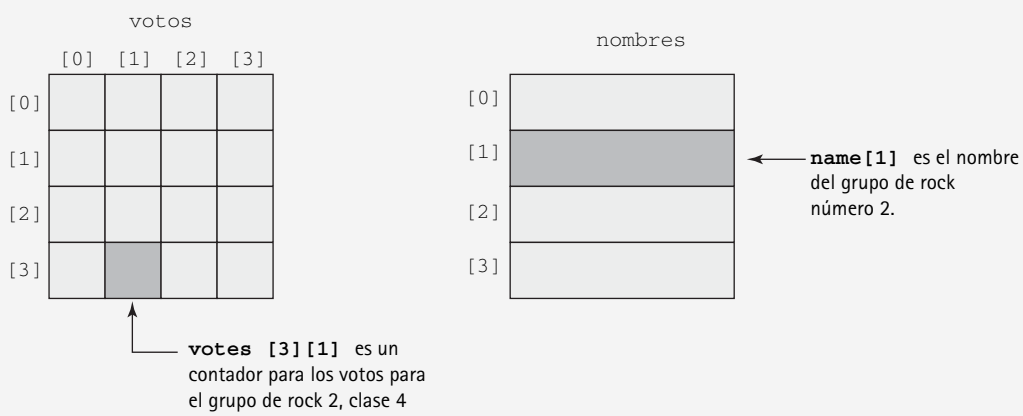


Figura 12-18 *Estructuras de datos para el programa de votación*

En nuestro análisis hemos usado la palabra "class" para representar alumnos del primero, segundo, tercero y cuarto años. Pero debemos tener cuidado. No podemos usar la palabra "class" en nuestro programa porque se trata de una palabra reservada. Vamos a cambiar aquí y usar en su lugar la palabra "nivel". En el diseño que sigue usamos las constantes nombradas `NUM_LEVELS` y `NUM_ROCK_GROUPS` en lugar de las constantes literales 4 y 4.

Principal **Nivel 0**

```
Abrir archivos
Obtener nombres de grupos
Colocar el array de votos a 0
Leer nivel, grupo de voteFile
MIENTRAS QUE NO SEA EOF en voteFile
    Incrementar votes[level-1][group-1] en 1
    Leer nivel, rockGroup de voteFile
Escribir reporte a reportFile
Escribir totales por grupo a reportFile
Escribir totales por nivel a reportFile
```

Get RockGroup Names (Out: name) **Nivel 1**

```
Imprimir "Introducir los nombres de los grupos de rock, uno por línea,
    en el orden de su aparición en el voto".
FOR rockGroup de 0 hasta NUM_ROCK_GROUPS-1
    Leer el name[rockGroup]
```

Observe que el nombre de cada grupo de rock es almacenado en la ranura en el array de nombre que corresponde a su número de grupo (menos 1). Estos nombres son útiles cuando se imprimen los totales.

Set Votes to Zero (Out: votes)

```
FOR cada nivel
    FOR cada grupo
        Colocar votes[level][group] a 0
```

Write Report (In: votes, name; Inout: reportFile)

```
FOR cada rockGroup      // Establecer encabezados
    Escribir name[rockGroup] a reportFile
FOR cada nivel          // imprimir array por fila
    FOR cada rockGroup
        Escribir votes[level][rockGroup] a reportFile
```

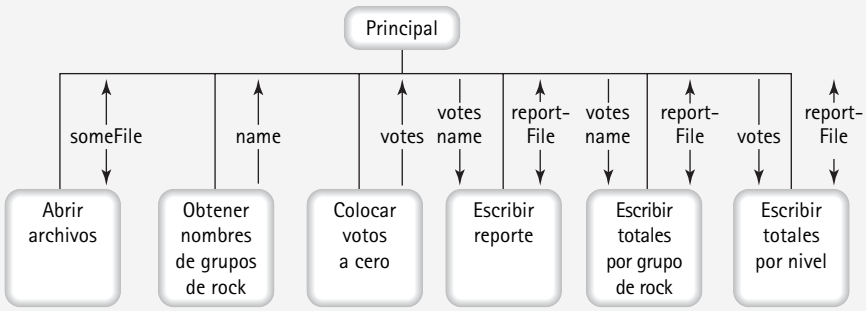
Write Totals per RockGroup (In: votes, name; Inout: reportFile)

```
FOR cada rockGroup
    Colocar el total = 0
FOR cada nivel      // Realizar suma de columna
    Agregar votes[level][rockGroup] al total
    Escribir "Total de votos para", name[rockGroup], total a reportFile
```

Write Totals per Level (In: votes; Inout: reportFile)

```
FOR cada nivel
    Colocar total = 0
    FOR cada rockGroup     // Realizar suma de fila
        Agregar votes[level][rockGroup] al total
    Escribir "Total de votos para nivel", nivel, ':' total a reportFile
```

DIAGRAMA DE ESTRUCTURA DE MÓDULOS




```
//**************************************************************************
// Programa grupo de rock favorito
// Este programa lee los votos representados por número de nivel
// y el número de grupo de rock de un archivo de datos, calcula
// las sumas por nivel y por grupo de rock, y escribe los totales
// en un archivo de salida
//**************************************************************************
#include <iostream>
#include <iomanip>    // Para setw()
#include <fstream>    // Para archivo de E/S
#include <string>     // Para nivel de cadena

using namespace std;

const int NUM_LEVELS = 4;
const int NUM_ROCK_GROUPS = 4;
typedef int VoteArray[NUM_LEVELS][NUM_ROCK_GROUPS];
                                  // tipo array bidimensional
                                  //   para votos
void GetNames( string[] );
void OpenFiles( ifstream&, ofstream& );
void WritePerRockGroup( const VoteArray, const string[],
                        ofstream& );
void WritePerLevel( const VoteArray, ofstream& );
void WriteReport( const VoteArray, const string[], ofstream& );
void ZeroVotes( VoteArray );

int main()
{
```

```
    string    name[NUM_ROCK_GROUPS];  // Array de nombres de grupos de rock
    VoteArray votes;          // Totales para nivel contra grupos de rock
    int       rockGroup;      // Entrada de número de grupo de rock desde el
                                 archivo de votos
    int       level;          // ingreso del número de nivel desde el archivo
                                 de votos
    ifstream  voteFile;       // Introducir el archivo de nivel, rockGroups
    ofstream  reportFile;     // Producir el archivo que recibe resúmenes

    OpenFiles(voteFile, reportFile);
    if ( !voteFile || !reportFile )
    {
        cout << "Los archivos no se abrieron con éxito." << endl;
        return 1;
    }


    GetNames(name);
    ZeroVotes(votes);

    // Leer y contar los votos

    voteFile >> level >> rockGroup;
    while (voteFile)
    {
        votes[level-1][rockGroup-1]++;
        voteFile >> level >> rockGroup;
    }

    // Escribir los resultados en el archivo de informe

    WriteReport(votes, name, reportFile);
    WritePerRockGroup(votes, name, reportFile);
    WritePerLevel(votes, reportFile);

    return 0;
}

//*****************************************************************************

void OpenFiles( /* inout */ ifstream& text,
                /* inout */ ofstream& outFile )

// La función OpenFiles lee los nombres del archivo de entrada
// y el archivo de salida y los abre para procesamiento
// Poscondición:
//     Se han abierto los archivos Y se ha escrito una etiqueta
//     en el archivo de salida

{
    string inFileName;
    string outFileName;
```

```
    cout << "Introduzca el nombre del archivo que será procesado"
              << endl;
    cin >> inFileName;
    text.open(inFileName.c_str());
    cout << "Introduzca el nombre del archivo de salida" << endl;
    cin >> outFileName;
    outFile.open(outFileName.c_str());
    outFile << "Análisis de exámenes del archivo " << inFileName
          << endl << endl;
}

//*****************************************************************************

void GetNames( /* out */ string name[] )     // Array de nombres
                                              //   de grupos de rock
// Lee los nombres de grupos de rock desde la entrada estándar

// Poscondición:
//     Se ha solicitado al usuario que introduzca los nombres de grupos de rock
//  && name[0..NUM_ROCKGROUPS-1] contiene los nombres de entrada
//     truncados a 10 caracteres cada uno

{
    string inputStr;      // Una cadena de entrada
    int    rockGroup;     // Contador de ciclo

    cout << "Introducir los nombres de los grupos de rock, uno por línea,"
         << endl << "en el orden en el que aparecen en el voto."
         << endl;
    for (rockGroup = 0; rockGroup < NUM_ROCK_GROUPS; rockGroup++)
    {
        cin >> inputStr;
        name[rockGroup] = inputStr.substr(0, 10);
    }
}

//*****************************************************************************

void ZeroVotes( /* out */ VoteArray votes )  // Array de totales de votos

// Establecer en cero el array de votos

// Poscondición:
//     Todos los votos[0..NUM_LEVELS-1][0..NUM_rockGroupS-1] == 0
{
    int level;       // Contador de ciclo
    int rockGroup;   // Contador de ciclo

    for (level = 0; level < NUM_LEVELS; level++)
        for (rockGroup = 0; rockGroup < NUM_ROCK_GROUPS; rockGroup++)
            votes[level][rockGroup] = 0;
}
```

```
//*****************************************************************************

void WriteReport(
       /* in */    const VoteArray votes,         // Votos totales
       /* in */    const string    name[],        // Nombres de grupos de rock
       /* inout */       ofstream& reportFile )   // Archivo de salida

// Escribe los totales de votos en forma tabular para el archivo de informe

// Precondición:
//     Se asignan los votos[0..NUM_LEVELS-1][0..NUM_ROCKGROUPS]
//  Sea asignan && nombre[0..NUM_ROCK_GROUPS-1]
// Poscondición:
//     Se ha producido el array de nombres en una línea, seguido del array
//     de votos, un renglón por línea

{
    int level;          // Contador de ciclo
    int rockGroup;      // Contador de ciclo
    // Establecer encabezados

    reportFile << "            ";
    for (rockGroup = 0; rockGroup < NUM_ROCK_GROUPS; rockGroup++)
        reportFile << setw(12) << name[rockGroup];
    reportFile << endl;

    // Imprimir el array por renglón

    for (level = 0; level < NUM_LEVELS; level++)
    {
        reportFile << "nivel" << setw(4) << level + 1;
        for (rockGroup = 0; rockGroup < NUM_ROCK_GROUPS; rockGroup++)
            reportFile << setw(12) << votes[level][rockGroup];
        reportFile << endl;
    }
    reportFile << endl;
}

//*****************************************************************************

void WritePerRockGroup(
       /* in */    const VoteArray votes,         // Votos totales
       /* in */    const string    name[],        // Nombres de grupos de rock
       /* inout */       ofstream& reportFile )   // Archivo de salida

// Suma los votos por persona y escribe los totales para el
// archivo de informe

// Precondición:
//     Se asignan los votos[0..NUM_LEVELS-1][0..NUM_ROCKGROUPS]
//  Sea asignan && nombre[0..NUM_ROCK_GROUPS-1]
// Poscondición:
```

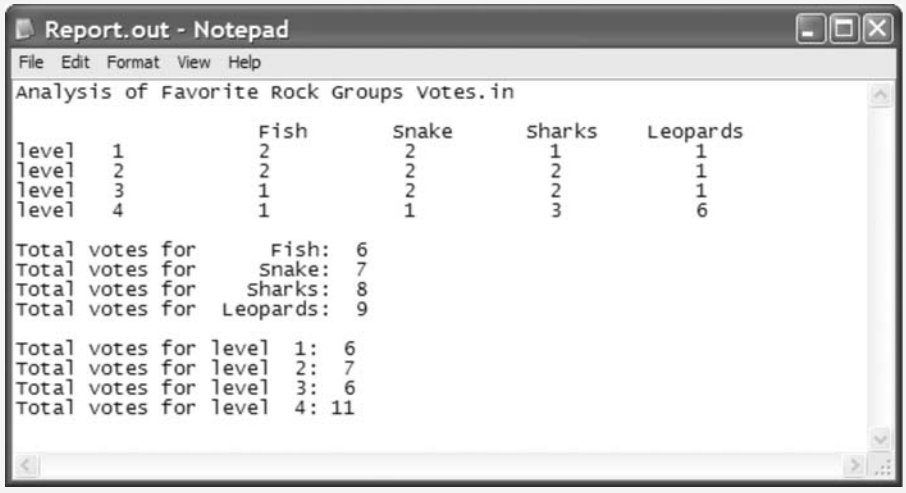
```
//     Para cada persona i, se ha producido el nombre [i],
//     seguido de la suma
//     votos[0][i] + votos[1][i]+ ... +votos[NUM_LEVELS-1][i]

{
    int level;          // Contador de ciclo
    int rockGroup;      // Contador de ciclo
    int total;          // Votos totales para un grupo de rock

    for (rockGroup = 0; rockGroup < NUM_ROCK_GROUPS; rockGroup++)
    {
        total = 0;
        // Calcular la suma de columna

        for (level = 0; level < NUM_LEVELS; level++)
            total = total + votes[level][rockGroup];

        reportFile << "Votos totales para"
                   << setw(10) << name[rockGroup] << ":"
                   << setw(3) << total << endl;
    }
    reportFile << endl;
}

//*************************************************************************

void WritePerLevel(
           /* in */    const VoteArray votes,         // Votos totales
           /* inout */       ofstream& reportFile )   // Archivo de salida

// Suma los votos por nivel y escribe los totales para el
// archivo de informe

// Precondición:
//     se asignan los votos[0..NUM_LEVELS-1][0..NUM_ROCKGROUPS]
// Poscondición:
//     Para cada nivel i, se ha producido el valor i+1,
//     seguido de la suma
//     votos[i][0] + votos[i][1] + ... +votos[i][NUM_LEVELS-1]

{
    int level;          // Contador de ciclo
    int rockGroup;      // Contador de ciclo
    int total;          // Votos totales para un nivel

    for (level = 0; level < NUM_LEVELS; level++)
    {
        total = 0;

        // Calcular la suma de renglón

        for (rockGroup = 0; rockGroup < NUM_ROCK_GROUPS; rockGroup++)
            total = total + votes[level][rockGroup];
```

```
        reportFile << "Votos totales para el nivel"
                   << setw(3) << level + 1 << ':'
                   << setw(3) << total << endl;
    }
}
```

PRUEBA Este programa fue ejecutado con los datos que se muestran en la lista que se presenta enseguida. (Listamos los datos en tres columnas para ahorrar espacio.) Los nombres de los grupos introducidos desde el teclado fueron Fish, Snake, Sharks y Leopards. En este juego de datos hay por lo menos un voto para cada grupo en cada clase. El ejercicio de seguimiento de estudio de caso 7 le pide delinear una estrategia completa de pruebas para este programa.

Datos de entrada

```
1 1      3 1      3 3
1 1      4 3      4 4
1 2      3 4      4 4
1 2      3 2      4 3
1 3      3 3      4 4
1 4      2 1      4 4
2 2      2 3      4 1
2 2      4 3      4 2
2 3      4 4      2 4
2 1      3 2      4 4
```



Prueba y depuración

Arrays unidimensionales

El error más común en el procesamiento de arrays es un índice de array fuera de límites. Esto quiere decir que el programa intenta acceder a un componente usando un índice menor que cero o mayor que el tamaño del array menos 1. Por ejemplo, dadas las declaraciones

```
char line[100];
int  counter;
```

la siguiente sentencia For imprimiría los 100 elementos del array `line` y luego imprimiría un valor 101: el valor que reside en la memoria inmediatamente después del fin del array.

```
for (counter = 0; counter <= 100; counter++)
    cout << line[counter];
```

Este error es fácil de detectar porque se imprimen 101 caracteres en lugar de 100. La prueba del ciclo debe ser `counter < 100`. Pero no siempre se usará una simple sentencia For para acceder a los arrays. Supongamos que leemos datos al primer array `line` en otra parte del programa. Vamos a usar una sentencia While que también se lee al carácter newline:

```
counter = 0;
cin.get(ch);
while (ch != '\n')
{
    line[counter] = ch;
    counter++;
    cin.get(ch);
}
```

Este código parece bastante razonable, pero, ¿qué pasa si la línea de introducción tiene más de 100 caracteres? Después de la lectura y almacenamiento del centésimo carácter en el array, el ciclo sigue ejecutando con el índice del array ya fuera de límites. Se almacenan caracteres en lugares de la memoria más allá del fin del array, borrando otros valores de datos (¡o incluso instrucciones en lenguaje de máquina en el programa!).

La moraleja es la siguiente: cuando se procesan arrays, se deberá poner especial atención al diseño de las condiciones de terminaciones de ciclos. Uno siempre se deberá preguntar si existe la posibilidad de que el ciclo siga trabajando después de que se haya procesado el último componente del array.

Cada vez que un índice de array sale de sus límites, lo primero que se deberá sospechar es un ciclo que no termina correctamente. El segundo aspecto que se debe revisar es algún acceso de array que involucra un índice basado en datos de introducción o un cálculo. Cuando se introduce un índice de array como datos, un chequeo de validación de datos es una necesidad absoluta.

Estructuras complejas

Como hemos demostrado en muchos ejemplos en este capítulo y en el anterior, es posible combinar estructuras de datos de varias maneras: estructuras cuyos componentes son estructuras, estructuras cuyos componentes son arrays, arrays cuyos componentes son estructuras u objetos de clase, arrays cuyos componentes son arrays (arrays multidimensionales), etcétera. Cuando se combinan arrays, estructuras y objetos de clase, puede haber una confusión respecto a la colocación exacta de los operadores para la selección de elementos de arrays (`[]`) y la selección de miembros de clase (`.`).

Para resumir la colocación correcta de estos operadores, vamos a usar el tipo `StudentRec` que hemos introducido en este capítulo:

```
struct StudentRec
{
    string    stuName;
    float     gpa;
    int       examScore[4];
    GradeType courseGrade;
};
```

Si declaramos una variable del tipo `StudentRec`,

```
StudentRec student;
```

entonces, ¿cuál es la sintaxis para seleccionar la primera calificación del examen del estudiante (es decir, para seleccionar el elemento 0 del miembro `examScore` de `student`)? El operador de punto es un operador binario (dos operandos); su operando izquierdo denota una variable tipo estructura, y su operando derecho es un nombre de miembro:

StructVariable . MemberName

El operador `[]` es un operador unario (un solo operando); viene inmediatamente después de una expresión que denota un array:

Array [IndexExpression]

Por tanto, la expresión

```
student
```

denota una variable tipo estructura; la expresión

```
student.examScore
```

denota un array, y la expresión

```
student.examScore[0]
```

denota un entero: el entero ubicado en el elemento 0 del array `student.examScore`.

En el caso de arrays de estructuras u objetos de clase, nuevamente tendremos que estar seguros de que los operadores `[]` y . están en las posiciones correctas. Dada la declaración

```
StudentRec gradeBook[150];
```

podemos acceder al miembro `gpa` del primer elemento del array `gradeBook` mediante la expresión

```
gradeBook[0].gpa
```

El índice `[0]` está correctamente conectado al identificador `gradeBook` porque `gradeBook` es el nombre de un array. Además, la expresión

```
gradeBook[0]
```

denota una estructura, así que el operador de punto selecciona el miembro `gpa` de esta estructura.

Arrays multidimensionales

Los errores con arrays multidimensionales normalmente se dividen en dos categorías principales: expresiones de índice que están fuera de orden y errores de intervalo del índice.

Supongamos que quisiéramos ampliar el programa de los grupos de rock para acomodar diez grupos de rock y cuatro niveles. Vamos a declarar el array `votes` como

```
int votes[4][10];
```

La primera dimensión representa los niveles, y la segunda, los grupos de rock. Un ejemplo del primer tipo de error —el orden incorrecto de las expresiones de índice— sería imprimir el array votes de la siguiente forma:

```
for (level = 0; level < 4; level++)
{
    for (rockgroup = 0; rockgroup < 10; rockgroup++)
        cout << setw(4) << votes[rockgroup][level];
    cout << endl;
}
```

La sentencia de salida especifica los índices del array en el orden equivocado. Los ciclos marchan a través del array con el primer índice teniendo un intervalo de 0 a 9 (en lugar de 0 a 3), y el segundo índice con un intervalo de 0 a 3 (en lugar de 0 a 9). El efecto de la ejecución de este código podrá variar de sistema a sistema. El programa podrá emitir los componentes de array equivocados y continuar ejecutando, o el programa podrá terminar con un error de acceso de memoria.

Un ejemplo del segundo tipo de error —un intervalo de índice incorrecto en un ciclo correcto— se puede ver en el siguiente código:

```
for (level = 0; level < 10; level++)
{
    for (rockgroup = 0; rockgroup < 4; rockgroup++)
        cout << setw(4) << votes[level][rockgroup];
    cout << endl;
}
```

En este caso la sentencia de salida correctamente usa `level` para el primer índice y `rockgroup` para el segundo. Sin embargo, las sentencias For usan límites superiores incorrectos para las variables de índices. Como en el ejemplo anterior, el efecto de ejecutar este código no está definido, pero es absolutamente incorrecto. Una manera valiosa de prevenir este tipo de error es el uso de constantes nombradas en lugar de los literales 10 y 4. En el estudio de caso `hemos usado` `NUM_LEVELS` y `NUM_ROCK_GROUPS`. Es mucho más probable que se detecte un error (o que se evite un error en primer lugar) si se escribe algo similar a lo siguiente:

```
for (level = 0; level < NUM_LEVELS; level++)
```

en vez de usar una constante literal como límite superior para la variable de índice.

Consejos para pruebas y depuración

1. Cuando se accede a un componente individual de un array unidimensional, el índice debe estar dentro del intervalo de 0 hasta el tamaño del array menos 1. Tratando de usar un valor de índice fuera de este intervalo su programa accederá a ubicaciones de memoria fuera del array.
2. Los componentes individuales de un array son en sí variables del tipo componente. Cuando se almacenan valores en un array, deberán ser del tipo componente o explícitamente convertidos al tipo componente; de lo contrario, ocurrirá coerción de tipo implícita.
3. C++ no permite operaciones agregadas en arrays. No hay asignación agregada, comparación agregada, E/S agregadas, ni aritmética agregada. Se debe escribir código para realizar todas estas operaciones, un elemento de array a la vez.
4. La omisión del tamaño de un array unidimensional en su declaración sólo se permite en dos casos: (1) cuando un array es declarado como un parámetro en un encabezado de función y (2) cuando un array es inicializado en su declaración. En todas las demás declaraciones, se *debe* especificar el tamaño del array con una expresión entera constante.
5. Si un array parámetro es únicamente de entrada, declare el parámetro como `const` para impedir que la función modifique accidentalmente el argumento del invocador.
6. No pase un componente individual de array como argumento cuando la función espera recibir la dirección base de un array entero.
7. El tamaño de un array se fija en el momento de la compilación, pero el número de valores realmente almacenados allí es determinado a la hora de la ejecución. Por tanto, un array se deberá declarar lo más grande que jamás podría ser para el problema en cuestión. Se usa el procesamiento de sub-array para procesar sólo los componentes que contienen datos.
8. Cuando las funciones realizan procesamiento de sub-array en un array unidimensional, pase tanto el nombre del array como el número de elementos de datos realmente almacenados en el array.
9. En el caso de arrays multidimensionales, use el número correcto de índices al referenciar un componente de array, y asegúrese de que los índices se encuentren en el orden correcto.

10. En ciclos que procesan arrays multidimensionales, compruebe dos veces los límites superiores e inferiores en cada variable de índice para estar seguro de que están correctos para esta dimensión del array.
11. Cuando se declara un array multidimensional como un parámetro, se deberán indicar los tamaños de todas las dimensiones, menos la primera. Estos tamaños también deberán coincidir exactamente con los del argumento del invocador.
12. Para eliminar la posibilidad de desigualdades de tamaños a los que se refiere el inciso 11, use una sentencia Typedef para definir un tipo de array multidimensional. Declare tanto el argumento como el parámetro de este tipo.

Resumen

El array unidimensional es una estructura homogénea de datos que da nombre a un grupo secuencial de componentes iguales. Se accede a cada componente mediante su posición relativa dentro del grupo (y no por nombre, como en una estructura o una clase) y cada componente es una variable del tipo de componente. Para acceder a un componente particular, damos el nombre del array y un índice que especifica qué componente del grupo queremos. El índice puede ser una expresión de cualquier tipo integral, siempre que evalúe a un entero de 0 hasta el tamaño del array menos 1. Se puede acceder a los componentes de arrays en orden aleatorio de manera directa, o de modo secuencial pasando a través de los valores del índice uno por uno.

Los arrays bidimensionales son útiles para procesar informaciones representadas de manera natural en forma tabular. El procesamiento de datos en arrays bidimensionales normalmente toma una de dos formas: el procesamiento por filas o el procesamiento por columnas. Un array de arrays, que es útil si las filas del array deben ser pasadas como argumentos, es un modo alternativo de definir un array bidimensional.

Un array multidimensional es una colección de componentes iguales ordenados en más de una dimensión. Se accede a cada componente por medio de un juego de índices, uno para cada dimensión, que representan la posición del componente en las diferentes dimensiones. Se podrá imaginar cada índice como la descripción de una característica de un componente de array dado.

Comprobación rápida

1. ¿Por qué decimos que un array es una estructura homogénea de datos? (pp. 488-489)
2. Usted está resolviendo un problema que requiere que almacene 24 valores de temperatura. ¿Qué estructura sería la más apropiada para esto: un registro, una unión, una clase o un array? (pp. 486-488)
3. Cómo accedería usted la 3a. letra en un miembro de cadena de datos (nombrado `street`) de una variable de clase que es el undécimo elemento de un array denominado `mailList`? (pp. 500-501)
4. Explique cómo el índice de un array de 24 lecturas de temperatura en intervalos de una hora tiene un contenido semántico, si el índice es un entero con un intervalo de 0 a 23. (pp. 502-503)
5. ¿Cómo difiere un array bidimensional en el sentido sintáctico de un array unidimensional? (pp. 503-506)
6. ¿Qué aspecto de un problema le llevaría a considerar el uso de un array multidimensional como la representación de su estructura de datos? (pp. 514-516)
7. Escriba una declaración de una variable de array denominado `temps` que contenga 24 valores del tipo `float`. (pp. 488-489)
8. Escriba un ciclo For que llena cada elemento del array `temps` declarado en el ejercicio 7 con el valor 32.0. (pp. 509-510)
9. Escriba un ciclo que lee valores del archivo `indata` al array `temps` del ejercicio 7, hasta que se haya introducido "fin del archivo" o 24 valores. Deberá estar al corriente del número de valores en el array en una variable `int` denominada `count`. (pp. 493-496)

10. Escriba la declaración para un array bidimensional, nombrado `allTemps`, que contenga las 24 lecturas de intervalos de una hora para cada día de un año (hasta 366 días). (pp. 503-506)
11. Escriba un ciclo For anidado que proporcione los contenidos del array `allTemps` declarado en el ejercicio 7, con las 24 temperaturas para un día en una línea y 366 líneas de días. (pp. 506-511)
12. Escriba el encabezado para una función que acepte el array `temps` del ejercicio 7 y su longitud como parámetros. Llame la función `Quick`, y haga que sea una función `void`. (pp. 511-513)
13. Escriba una declaración para un array multidimensional que almacena 24 lecturas de temperatura con un intervalo de una hora para cada día de un año durante una década. Llame el array `decadeTemps`. (pp. 514-516)

Respuestas

1. Porque sus elementos son todos del mismo tipo. **2.** Un array. **3.** `mailList[12],street[2]` **4.** El índice representa la hora durante la cual se realizó la lectura, basado en un reloj de 24 horas. **5.** Es declarado y accedido por medio de dos valores de índice en lugar de uno. **6.** Si el problema tiene una colección de datos homogéneos que es ordenada por más de dos índices.

7.
```
float temps[24];
```

8.
```
for (int count = 0; count <=23; count++)
  temps[count] = 32.0;
```

9.
```
count = 0;
cin >> inTemp;
while (indata && count <= 24)
{
  count++;
  temps[count - 1] = inTemp;
  cin >> inTemp;
}
```

10.
```
float allTemps[24][366];
```

11.
```
for (day = 0; day <= 365; day++)
{
  for (hour = 0; hour <= 23; hour++)
    cout << allTemps[hour][day];
  cout << endl;
}
```

12.
```
void Quick (/* inout */ float arr[],
            /* in */ int numElements)
```

13.
```
float decadeTemps[24][366][10]
```

Ejercicios de preparación para examen

1. Los componentes de un array pueden ser de diferentes tipos. ¿Correcto o falso?
2. Los arrays pueden tener cualquier tipo para sus componentes. ¿Correcto o falso?
3. Los arrays multidimensionales están limitados a no más de cuatro dimensiones. ¿Correcto o falso?
4. Cuando se declara un array unidimensional, su tamaño se tiene que especificar por medio de un entero. ¿Correcto o falso?
5. El tipo de una expresión de índice puede ser cualquiera de los tipos integrales o un tipo de enumeración. ¿Correcto o falso?
6. ¿Qué pasa si un programa intenta acceder a un array usando un índice de –1?
7. ¿Qué operación (operaciones) agregada(s) está(n) permitida(s) en arrays?
8. ¿A qué se refiere el término “dirección base de un array”, y cómo se usa en una llamada de función?

9. ¿Qué tipo especial de array puede ser devuelto por una función de devolución de valores?
10. Si quiere usar un ciclo anidado para procesar un array bidimensional fila por fila, ¿el índice de qué dimensión (fila o columna) incrementaría usted en el ciclo interior, y cuál incrementaría en el ciclo exterior?
11. ¿Cuántos elementos existen en cada uno de los siguientes arrays?
    a) `int x[27];`
    b)
    ```
    const int base = 10;
    int y[base + 5];
    ```
    c) `int z[100][100][100][100];`
12. ¿Cuál es el problema con el siguiente fragmento de código?

```
int prep[100];
   for (int index = 1; index <= 100; index++)
     prep[index] = 0;
```

13. ¿Cuál es el problema con el siguiente fragmento de código?

```
const int limit = 100;
int eprep[limit];
int examp[limit];
for (int index = 0; index <= limit - 1; index++)
{
  eprep[index] = 0;
  examp[index] = 0;
}
if (eprep = = examp)
    cout << "Equal";
```

14. ¿Cuál es el problema con el siguiente fragmento de código?

```
typedef int Exrep[50];

Exrep Init(/* in */ Exrep); // Prototipo
```

15. ¿Cuál es el problema con el siguiente fragmento de código?

```
int prepex[3][4] =
{
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9},
  {0, -1, -2}
};
```

16. ¿Cuál es el peligro potencial en el siguiente fragmento de código?

```
int index;
int value;
int xeperp[100];
cin >> index >> value;
xeperp[index] = value;
```

17. ¿Cuál es el efecto secundario de la siguiente función?

```
int Max (/* in */ int trio[3])
{
  int temp;
```

```
  if (trio[0] > trio[1])
  {
    temp = trio[1];
    trio[1] = trio[0];
    trio[0] = temp;
  }
  if (trio[1] > trio[2])
  {
    temp = trio[2];
    trio[2] = trio[1];
    trio[1] = temp;
  }
  return trio[2];
}
```

18. ¿Qué proporciona como salida el siguiente fragmento de código?

```
char pattern[5][5] =
{
  {'*', ' ', '*', ' ', '*'},
  {' ', '*', ' ', '*', ' '},
  {'*', ' ', '*', ' ', '*'},
  {' ', '*', ' ', '*', ' '},
  {'*', ' ', '*', ' ', '*'}
}
for (int outer = 0; outer < 5; outer++)
{
  for (int inner = 1; inner <= 5; inner++)
    cout << pattern[outer][inner % 5];
  cout << endl;
}
```

19. Dada la declaración de patrón en el ejercicio 18, ¿qué emite el siguiente ciclo?

```
for (int index = 0; index < 5; index++)
  cout << pattern[index][(index+2) % 5];
```

20. Dada la declaración de patrón en el ejercicio 18, ¿qué emite el siguiente ciclo?

```
for (int outer = 0; outer < 5; outer++)
{
  for (int inner = 0; inner < 5; inner++)
    cout << pattern[inner][outer];
  cout << endl;
}
```

21. Dada la siguiente declaración:

```
int examPrep [12][15];
```

a) Escriba un ciclo para imprimir la primera fila del array en una línea en `cout`.
b) Escriba un ciclo para imprimir la primera columna del array en una línea en `cout`.
c) Escriba un ciclo para imprimir las primeras siete filas del array, cada una en una línea individual.
d) Escriba un ciclo para imprimir el array completo hacia atrás y al revés, de modo que la última línea aparezca primero, con la última columna a la izquierda.

Ejercicios de calentamiento de programación

1. Escriba las siguientes declaraciones.
   a) Un array llamado `topTenList`, de 10 valores del tipo `string`.
   b) Un tipo de enumeración de los siete colores principales del espectro, y una declaración de array que puede ser indizadoindizado por el tipo de espectro. El array se deberá llamar `colorMix`, y contiene valores del tipo `float`.
   c) Un array bidimensional que represente los días en una página de calendario hasta con seis semanas. Nombre el array `month` y declare un tipo de enumeración que consista en los nombres de los días, el cual se pueda usar para indexar las columnas del array. Las semanas deben ser indexadas por un `int`. Para el tipo de componente del array, declare una estructura que consista en un campo `int` llamado `day` y un campo `string` llamado `activity`.
2. Escriba una declaración de un tipo de array nombrado y luego declare tres arrays de este tipo. El tipo de array se deberá denominar `DataSet`, y los tres arrays se deberán llamar `input`, `output` y `working`. Cada array deberá tener cinco valores `float`.
3. Usando el tipo `DataSet` declarado en el ejercicio 2, declare un array de tres juegos de datos, llamado `set`.
4. Usando el array `set` declarado en el ejercicio 3, escriba un ciclo anidado For que inicialice todos los valores de los tres juegos de datos a 0.0.
5. Escriba un encabezado de función para una función llamada `Equals` que tome dos arrays del tipo `DataSet` declarado en el ejercicio 2 y devuelva un resultado `bool`. Los parámetros de array deberán ser `const`, puesto que son parámetros de sólo entrada para la función.
6. Escriba el cuerpo de la función `Equals` descrita en el ejercicio 5. Deberá devolver `true` si cada elemento del primer array es igual a su elemento correspondiente en el segundo array.
7. Una galería necesita mantenerse al tanto de sus pinturas y fotografías. Cuando mucho, la galería mantiene 120 obras de arte en sus paredes. Cada una de ellas tiene una descripción con la siguiente información:

   Artista (string)
   Título (string)
   Medio (óleo, acuarela, pastel, acrílico, impreso, fotografía de color, fotografía de blanco y negro)
   Tamaño (struct)
   Alto (int)
   Ancho (int)
   Espacio donde se exhibe (sala principal, verde, azul, norte, sur, entrada, balcón)
   Precio (float)

   Escriba una declaración para una estructura que representa una obra de arte. Declare los tipos estructura y enumeración conforme sea necesario para elaborar los campos de tipo `ArtWork`. Escriba una declaración adicional de un tipo de array nombrado que contenga la lista de todas las obras de arte en la galería. Finalmente, declare un array de este tipo llamado `currentList` y una variable `int` llamada `numPieces`. La variable `numPieces` contiene el número de obras representadas en el array.
8. Escriba expresiones que recuperen los siguientes valores del array declarado en el ejercicio 7.
   a) La obra de arte 37.
   b) El título del la obra de arte 12.
   c) El ancho de la obra de arte 85.
   d) El lugar de exhibición del la obra de arte 120.
   e) La primera letra del nombre del artista de la obra de arte 78.
9. Escriba un ciclo For que imprima una lista del artista, título y precio para cada obra en el array `currentList` definido en el ejercicio 7.
10. Escriba un segmento de código que sume los precios de las obras en la galería, como se describió en el ejercicio 7.
11. Escriba un segmento de código que emita los títulos de las obras de arte en el salón azul de la galería, como se describió en el ejercicio 7.

12. Escriba un segmento de código que sume los precios de las pinturas al óleo en la galería, como se describió en el ejercicio 7, que tengan un tamaño mayor a 2 500 centímetros cuadrados.
13. Un piano es afinado en una escala levemente desigual (lo que se llama una escala bien temperada) en vez de una escala perfectamente científica donde cada nota suena al doble de frecuencia de la misma nota de una octava inferior (lo que se llama una "escala justa"). Por esta razón no siempre podemos calcular la frecuencia de una nota, sino que debemos mantenerla en una tabla. Declare un array bidimensional (`scale`) para mantener las frecuencias de la escala bien temperada. Una frecuencia es representada por un valor `float`. La dimensión de filas es indexada por un valor `int` que representa la octava (hay 8 octavas, numeradas del 0 al 7), y la otra deberá ser indexada por un tipo de enumeración (`Notes`) que consiste en los nombres de las notas. Cuando escribe la declaración del tipo de enumeración, sólo use sostenidos (no bemoles). De este modo, los nombres de las notas de una octava son: DO, DO SOSTENIDO, RE, RE SOSTENIDO, MI, FA, FA SOSTENIDO, SOL, SOL SOSTENIDO, LA, LA SOSTENIDO y SI, en este orden.
14. Escriba un segmento de código que lea una tabla de frecuencias al array `scale` declarado en el ejercicio 13 de un archivo llamado `frequencies.dat`. Los valores de frecuencia están dispuestos uno por línea en el archivo.
15. Escriba un segmento de código que emita las frecuencias de las notas en la cuarta octava del array `scale` declarado en el ejercicio 13.
16. Escriba un segmento de código que emita las frecuencias de todas las notas DO en el array `scale` declarado en el ejercicio 13.
17. Escriba una declaración para un array de cuatro dimensiones que es indizadoindizado por 10 años (0-9), 52 semanas (0-51), los 50 estados (0-49) y un tipo de enumeración que consiste en `MAX`, `MIN` y `AVERAGE`, llamado `Type`. El array contiene mediciones de humedad (cada componente es un `float`).
18. Escriba una función de C++ llamada `Reset`, que pone todos los componentes a cero en el array declarado en el ejercicio 17.
19. Escriba una declaración de tipo estructura llamado `TimePlace` que tiene tres campos, uno para cada uno de los primeros tres valores de índice (año, semana, estado) en el array `humidity` declarado en el ejercicio 17. `TimePlace` deberá tener un cuarto campo llamado `difference`, que es un `float`. Luego escriba una función de C++ llamada `MaxSpread` que toma el array `humidity` como un parámetro y lo recorre para buscar el año, semana y estado con la mayor diferencia de humedad, y devuelve estos valores como una estructura de `TimePlace`. Si hay más de una semana con la mayor diferencia, entonces se deberá devolver la primera.
20. Escriba un segmento de código que emita el componente `AVERAGE` del array `humidity` del ejercicio 17, para todas las semanas en los últimos 5 años para el estado 23.

Problemas de programación

1. Escriba un programa para desarrollar un juego en el cual usted trata de hundir una flota de cinco embarcaciones, adivinando sus ubicaciones en una cuadrícula. El programa usa números aleatorios para posicionar sus barcos en una cuadrícula de 15 × 15. Los barcos tienen las siguientes longitudes diferentes:

   Fragata: 2 espacios
   Barcaza: 2 espacios
   Destructor: 3 espacios
   Crucero: 3 espacios
   Portaaviones: 4 espacios

   El programa debe seleccionar una casilla como ubicación inicial, luego seleccionar la dirección del barco en la tabla y marcar el número de casillas en esta dirección para representar el tamaño del barco. No debe permitir que un barco esté en contacto con otro, ni que salga de la tabla.

   El usuario introduce coordenadas en el intervalo de 1 a 15 para las filas y de A a O para las columnas. El programa verifica esta ubicación y reporta si se trata de un impacto o una falla. Si

se trata de un impacto, el programa también verifica si el barco ha sido impactado en cada ubicación que ocupa. En caso afirmativo, el barco se reporta como hundido y el programa identifica de qué barco se trata.

El usuario tendrá 60 tiros para intentar hundir la flota. Si el usuario hunde a todos los barcos antes de usar los 60 tiros, ganará el juego. Al final del juego, el programa deberá mostrar la cuadrícula para que el usuario pueda ver dónde están ubicados los barcos.

2. Los ejercicios de calentamiento de programación 7 al 12 tienen como tema un array representando el inventario de una galería de arte. Usando la misma representación, escriba un programa C++ que lee el inventario de la galería desde un archivo llamado `art.dat` al array. Luego permita que el usuario busque las obras de arte especificando a cualquier campo del registro. A continuación un recordatorio de los campos:

   Artista (string)
   Título (string)
   Medio (óleo, acuarela, pastel, acrílico, impreso, fotografía de color, fotografía de blanco y negro)
   Tamaño (struct)
   Alto (int)
   Ancho (int)
   Espacio donde se exhibe (sala principal, verde, azul, norte, sur, entrada, balcón)
   Precio (float)

   De este modo, el usuario deberá ser capaz de especificar un campo y un valor para este campo, y el programa devolverá todas las obras que cumplan el criterio. Por ejemplo, el usuario podrá especificar Artista y Smithely, y el programa emitirá toda la información acerca de cada obra de este artista que se encuentre en la galería.

3. El problema de programación 1 en el capítulo 9 le pidió escribir un programa que introduce una cadena y luego emite las palabras correspondientes en el alfabeto de la Asociación Internacional de Aviación Civil que se usaría para deletrearlo fonéticamente. Para este programa, usted deberá haber usado una sentencia Switch grande. Reescriba este programa usando un array de cadenas para contener las palabras del alfabeto, e indexe el array por las posiciones de las letras del alfabeto. Mediante el uso de un índice con contenido semántico, usted puede evitar la necesidad para la sentencia Switch. No trate de indexar en el array usando caracteres no alfabéticos, ya que esto resultará en un acceso fuera de límites. Para la facilidad de referencia, se repite a continuación el alfabeto de la ICAO del capítulo 9:

A	Alpha
B	Bravo
C	Charlie
D	Delta
E	Echo
F	Foxtrot
G	Golf
H	Hotel
I	India
J	Juliet
K	Kilo
L	Lima
M	Mike
N	November
O	Oscar
P	Papa
Q	Quebec
R	Romeo
S	Sierra
T	Tango

U Uniform
V Victor
W Whiskey
X X-ray
Y Yankee
Z Zulu

No olvide usar el formateo correcto y los comentarios apropiados en su código. Proporcione los mensajes apropiados para el usuario. El resultado deberá ser claramente etiquetado y nítidamente formateado.

4. El problema de programación 4 en el capítulo 6 le pidió escribir un programa para verificar si una línea de introducción es un palíndromo (una palabra o frase que se lee igual de izquierda a derecha que de derecha a izquierda). En aquel momento necesitábamos usar la función substring para extraer caracteres individuales de una cadena. Puesto que esto es una manera muy farragosa de trabajar con una cadena, habíamos limitado nuestra definición de palíndromos exclusivamente a los palíndromos perfectos: cada letra y espacio exactamente lo mismo en las versiones hacia delante y hacia atrás. Ahora que sabemos cómo usar arrays, podemos extender la definición para ignorar espacios blancos, puntuación y tamaño de letras. Así que

   ```
   madam I'm adam
   ```

   se convierte en

   ```
   madamimadam
   ```

   que ahora es un palíndromo. Escriba un programa de C++ que lea una línea de introducción y verifique si se trata de un palíndromo sobre la base de esta definición menos restrictiva. Emita la línea en reversa, con todos los espacios blancos y la puntuación eliminada, y todas las letras convertidas en minúsculas, junto con la decisión del programa.
5. En el capítulo 5, el problema 5 de programación le pidió crear una clase representando una canción en un CD o en una biblioteca MP-3. Escriba un programa de C++ usando esta clase, que permita al usuario especificar el nombre de un archivo de datos que contiene canciones (asumimos que hay menos de 200 canciones en este archivo), y que luego lee los datos de las canciones desde el archivo a un array de objetos de canciones. Luego se le deberá permitir al usuario ingresar el nombre de un artista, junto con otras informaciones sobre la canción que está en el array.

Seguimiento de caso práctico

1. No hay verificación de errores en el programa de Cálculo de estadísticas de exámenes. Nombre al menos dos errores que se podrían verificar fácilmente.
2. Todas las funciones, excepto `OpenFiles`, `InputGrades` y `PrintResults`, son funciones de devolución de valores. En lugar de calcular y almacenar los valores que calculan, ¿se podrían calcular conforme se están imprimiendo en `PrintResult`? En caso afirmativo, ¿sería una buena idea hacer esto?
3. Esta solución utiliza una técnica denominada índices con contenido semántico. Explique qué significa esto en relación con este problema.
4. El encabezado para la emisión está codificado directamente en la función `OpenFiles`. Remueva esta instrucción, pida que el usuario introduzca un encabezado, y escriba este encabezado en el archivo de salida.
5. El ejercicio 4 tenía el encabezado escrito en la función `OpenFiles`. ¿Sería mejor que la función `PrintResults` pidiera el encabezado?
6. Reescriba el programa del grupo de rock favorito usando un tipo enumerado para clase (o nivel). ¿Cuál es el código más legible y de mejor autodocumentación?
7. Diseñe una estrategia de prueba completa para el programa del grupo de rock favorito.

CAPÍTULO

13

Listas basadas en arrays

Objetivos

Objetivos de conocimiento

- *Comprender la estructura de un ADT de lista*
- *Conocer las operaciones básicas asociadas con un ADT de lista*
- *Saber cómo funciona un algoritmo de búsqueda lineal*
- *Entender las propiedades de un ADT de lista ordenada*
- *Saber cómo funciona una ordenación de inserción*
- *Saber cómo funciona el algoritmo binario de búsqueda*

Objetivos de habilidades

Ser capaz de:

- *Representar un ADT de lista usando una clase de C++*
- *Insertar y borrar valores de lista*
- *Representar un ADT de lista ordenada usando una clase de C++*
- *Implementar una ordenación de inserción*
- *Implementar una búsqueda binaria en una lista ordenada*

El capítulo 12 introdujo el array, una estructura de datos que es una colección de componentes del mismo tipo que recibe un nombre único. En general un array unidimensional es una estructura que se usa para contener una lista de objetos. En este capítulo examinaremos algoritmos que construyen y manipulan datos almacenados como una lista en un array unidimensional. Estos algoritmos son aplicados como funciones multiuso que pueden ser fácilmente modificadas para trabajar con muchos tipos de listas.

También consideraremos el *C string*, un tipo especial de array unidimensional integrado que se usa para almacenar cadenas de caracteres. Concluiremos con un estudio de caso que usa un array como estructura de datos principal.

13.1 La lista como un tipo de datos abstractos (ADT)

Como hemos definido en el capítulo 12, un array unidimensional es una estructura integrada de datos que consiste en un número fijo de componentes homogéneos. Uno de los usos para un array es el de almacenar una lista de valores. Una lista podrá contener menos valores que el número de lugares reservados en el array.

Por ejemplo, dado un array `firstList` con 500 lugares, sólo los primeros componentes `Length-1` posiblemente contengan datos válidos. En la figura 13-1 se puede ver que el *array* va desde `firstList[0]` hasta `firstList[499]`, pero la *lista* almacenada en el array va desde `firstList[0]` hasta `firstList[Length-1]`. El número de lugares en el array es fijo, pero el número de valores en la lista que se guarda allí podrá variar.

Lista Colección de longitud variada y lineal de componentes homogéneos.

Longitud Número de valores realmente almacenados en la lista.

Pensemos por un momento en el concepto de una lista no en términos de arrays, sino como un tipo de datos separado. Podemos definir una **lista** como una colección de longitud variada y lineal de componentes homogéneos. Eso ya es un bocado grande. Por *lineal* entendemos que cada componente (excepto el primero) tiene un componente único que le precede, y cada componente (excepto el último) tiene un componente único que le sigue. La **longitud** de una lista —el número de valores actualmente almacenados en la lista— puede variar durante la ejecución del programa.

Como todo tipo de datos, una lista debe tener asociado a ella un juego de operaciones permisibles. ¿Qué tipo de operaciones quisiéramos definir para una lista? Aquí hay algunas posibilidades: crear una lista, agregar un elemento a una lista, borrar un elemento de una lista, imprimir una lista, buscar un valor particular en una lista, crear una lista en orden alfabético o numérico, etc. Cuando definimos formalmente un tipo de datos —especificando sus propiedades así como las operaciones

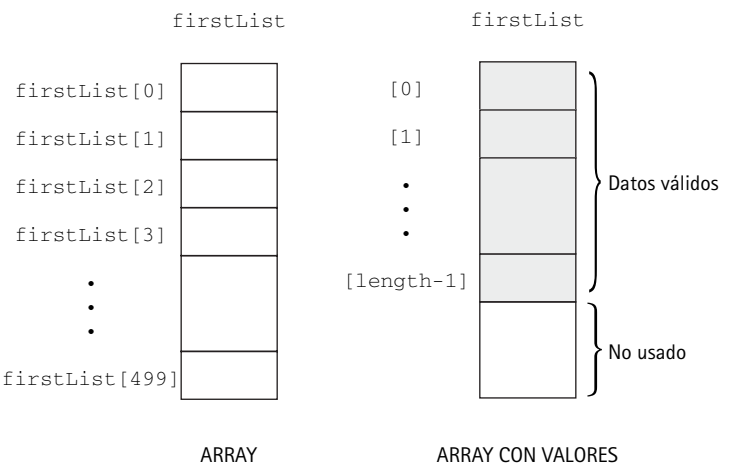


Figura 13-1 *Array* `firstList`

que preservan estas propiedades— estamos creando un tipo de datos abstracto (ADT, por sus siglas en inglés). De hecho, en el capítulo 11 propusimos un ADT llamado IntList, un tipo de datos para una lista de hasta 100 valores enteros. En ese momento no aplicamos este ADT porque no tuvimos a nuestra disposición una representación adecuada de datos concretos. Ahora que estamos familiarizados con la idea de usar un array unidimensional para representar una lista, podemos combinar clases de C++ y arrays para aplicar ADT de lista.

Vamos a generalizar el ADT IntList del siguiente modo: *a*) permitir que los componentes sean de *cualquier* tipo sencillo o del tipo `string`, *b*) remplazar la longitud máxima de 100 por un máximo de `MAX_Length`, una constante definida, y *c*) incluir una variedad más amplia de operaciones permisibles. Aquí mostramos la especificación del ADT más general:

TYPE
  Lista
DOMAIN
  Cada instancia del tipo Lista es una colección de hasta MAX_LENGTH componentes, cada uno del tipo ItemType.
OPERATIONS
  Crear una lista inicialmente vacía.
  Reportar si la lista está vacía (`true` o `false`).
  Reportar si la lista está llena (`true` o `false`).
  Devolver la longitud actual de la lista.
  Insertar un elemento en la lista.
  Borrar un elemento de la lista.
  Buscar un elemento específico, devolviendo `true` o `false` dependiendo si el elemento está presente en la lista o no.
  Crear la lista en orden ascendente.
  Iterar a través de la lista, devolviendo cada elemento en su momento.

Podemos usar una clase de C++ denominada `List` para representar el ADT de lista en nuestros programas. Para la representación concreta de datos usamos dos elementos: un array unidimensional para contener los elementos de la lista, y una variable `int` que almacena la longitud actual de la lista. Cuando compilamos la clase `List`, tenemos que proporcionar definiciones para `MAX_LENGTH` e `ItemType`:

```
const int MAX_LENGTH = [          ];      // Número máximo posible de
                                          //   componentes necesarios
typedef [          ] ItemType;            // Tipo de cada componente
                                          //   (un tipo simple o
                                          //    de clase string)
```

Aquí está el archivo de especificación para nuestro ADT de lista: observe que usamos 50 para `MAX_LENGTH` e `int` para `ItemType`. Nótese asimismo que la operación abstracta *Crear una lista inicialmente vacía* está aplicada como el constructor de clase `List()`.

```
//****************************************************************************
// ARCHIVO DE ESPECIFICACIÓN (list.h)
// Este archivo da la especificación de una lista de tipo de datos abstractos.
// Se supone que los componentes de la lista están en orden por valor.
// Para ahorrar espacio, se omiten de cada función los comentarios de
// precondición que documentan las suposiciones hechas acerca de los datos
// de parámetros de entrada válidos. Éstos se incluirían en un programa
// propio para uso real.
//****************************************************************************
```

```
const int MAX_LENGTH =100;     // Número posible máximo de
                               //   componentes necesarios
typedef int ItemType;          // Tipo de cada componente
                               //   (un tipo simple o clase de cadena)

class List
{
public:
    bool IsEmpty() const;

        // Poscondición:
        //     Valor de retorno == true, si la lista está vacía
        //                      == false, en caso contrario

    bool IsFull() const;

        // Poscondición:
        //     Valor de retorno == true, si la lista está llena
        //                      == false, en caso contrario

    int Length() const;

         // Poscondición:
         //     Valor de retorno == longitud de la lista

    void Insert( /* in */ ItemType item );

        // Precondición:
        //     NO está completa ()
        // Poscondición:
        //     el elemento está en la lista
        //  && Length() == Length()@entry + 1

    void Delete( /* in */ ItemType item );

        // Precondición:
        //     NO está vacía()
        // Poscondición:
        //     SI el elemento está en la lista en la entrada
        //         La primera aparición del elemento ya no está en la lista
        //       && Length() == Lenght()@entry - 1
        //     ELSE
        //         La lista permanece sin cambio

    bool IsPresent( /* in */ ItemType item ) const;

        // Poscondición:
        //     Valor de retorno == true, si el elemento está en la lista
        //                      == false, en caso contrario

    void Reset();

        // Poscondición:
        //     Se inicializa la iteración
```

```
    ItemType GetNextItem();

        // Precondición:
        //     La iteración ha sido inicializada mediante llamada a Reset;
        //     Ningún transformador ha sido invocado desde la última llamada
        // Poscondición:
        //     El valor de retorno es el elemento en la posición actual
        //     en la lista de la entrada;
        //     Si ha sido devuelto el último elemento, la siguiente llamada
        //     devolverá el primer elemento.

    void SelSort();

        // Poscondición:
        //     Los componentes de la lista están en orden ascendente de valor

    List();

        // Constructor
        // Poscondición:
        //     Se crea la lista vacía
private:
    int      length;
    int      currentPos;
    ItemType data[MAX_LENGTH];
};
```

En el capítulo 11 hemos ordenado las operaciones ADT como constructores, transformadores, observadores e iteradores. `IsEmpty`, `IsFull`, `Length` e `IsPresent` son observadores. `Reset`, `Insert`, `Delete` y `SelSort` son transformadores. `GetNextItem` es un iterador. El constructor de clase es una operación de constructor de ADT.

La parte privada de la declaración de clase muestra nuestra representación de datos de una lista: dos variables `int` y un array (véase la figura 13-2). Sin embargo, nótese que las precondiciones y poscondiciones de las funciones de miembros no mencionan nada acerca de un array. La abstracción es una lista, no un array. El usuario de esta clase sólo está interesado en la manipulación de listas de elementos y no le importa cómo aplicamos una lista. Si cambiamos a una representación de datos diferente (como lo hacemos en el capítulo 15), ni se tendrá que cambiar la interfase pública ni el código del cliente.

Veamos un ejemplo de un programa de cliente. Un archivo de datos contiene las lecturas de temperaturas máximas diarias de una estación meteorológica para un mes; un valor entero por día. Desafortunadamente el sensor de temperatura está defectuoso y registra a veces una temperatura de 200 grados. El siguiente programa usa la clase `List` para almacenar las lecturas de temperatura, borrar cualquier lectura falsa de 200 grados, y emitir las lecturas restantes en forma ordenada. Supuestamente el archivo de datos no contiene más de 31 enteros para el mes, que deberá estar muy por debajo de la clase `List MAX_LENGTH` de 50. Sin embargo, sólo en caso de que el archivo erróneamente contenga valores con más de `MAX_LENGTH`, el ciclo de lectura en el siguiente programa no sólo termina si se encuentra “end-of-file”, sino también si la lista se llena (`IsFull`). Otra razón para usar la operación `IsFull` en este ciclo se puede observar en las especificaciones de función en el archivo `list.h`, es decir, debemos garantizar la precondición de `Insert` de que la lista no está llena. De manera similar, en el ciclo que borra las lecturas erróneas de 200 grados, tenemos que usar la operación `IsEmpty` para garantizar la precondición `Delete` de que la lista no está vacía.

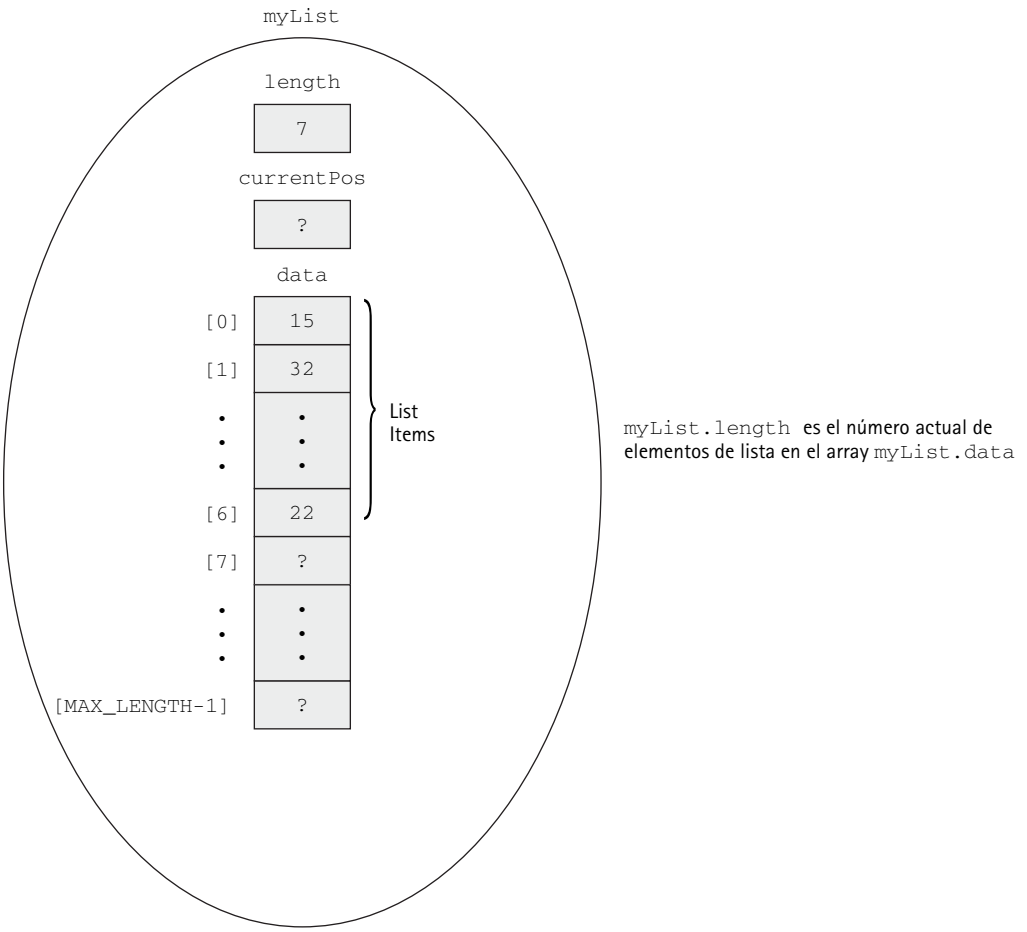


Figura 13-2 *myList, un objeto de clase del tipo* `List`

```
//**************************************************************************
// Programa Temperaturas
// Este programa introduce las lecturas de temperatura de un mes desde
// un archivo, borra las lecturas falsas de 200 grados de un
// sensor defectuoso
//**************************************************************************
#include <fstream>     // Para archivo de E/S
#include "list.h"      // Para la clase List

using namespace std;

int main()
{
    List temps;          // Lista de lecturas de temperatura
    int oneTemp;        // Una lectura de temperatura
    ifstream inData;     // Archivo de lecturas de temperaturas
    int limit;           // El número de lecturas
    ofstream outData;    // Archivo de salida
```

```
    inData.open("temps.dat");
    if ( !inData )
    {
        outData << "No se puede abrir el archivo temps.dat" << endl;
        return 1;
    }

    outData.open("temps.ans");

    // Obtener las lecturas de temperatura del archivo

    inData >> oneTemp;
    while (inData && !temps.IsFull())
    {
        temps.Insert(oneTemp);
        inData >> oneTemp;
    }

    // Producir la lista original

    temps.Reset();                      // Establecer una iteración
    limit = temps.Length();             // Obtener el número de elementos
    outData << "No. de lecturas: " << limit << endl;
    outData << "Lista original:" << endl;

    for (int count = 0; count < limit; count++)
    {
        oneTemp = temps.GetNextItem();
        outData << oneTemp << endl;
    }

    // Desechar las lecturas falsas de 200 grados

    while ( !temps.IsEmpty() && temps.IsPresent(200))
        temps.Delete(200);

    temps.SelSort();                    // Lista de clasificación

    // Producir la lista clasificada

    temps.Reset();                      // Establecer una iteración
    limit = temps.Length();             // Obtener el número de elementos
    outData << "No. de lecturas válidas:_" << limit << endl;
    outData << "Lista clasificada:" << endl;

    for (int count = 0; count < limit; count++)
    {
        oneTemp = temps.GetNextItem();
        outData << oneTemp << endl;
    }
    inData.close();
    outData.close();
    return 0;
}
```

El siguiente archivo de salida se muestra en dos columnas en lugar de en una.

```
No. de lecturas: 30                    No. de lecturas no válidas: 21
Lista original:                        Lista clasificada:
70                                     61
68                                     65
69                                     66
68                                     67
68                                     68
72                                     68
200                                    68
69                                     68
87                                     69
200                                    69
82                                     69
200                                    70
200                                    71
```

Continuación del archivo de salida:

```
No. de lecturas: 30                    No. de lecturas no válidas: 21
Lista original:                        Lista clasificada:
81                                     72
200                                    75
75                                     77
78                                     78
68                                     78
67                                     81
200                                    82
65                                     87
66
200
61
71
200
200
77
78
69
```

Ahora consideraremos cómo aplicar cada una de las operaciones de ADT, puesto que los elementos de la lista están almacenados en el array. Sin embargo, antes de hacerlo, tenemos que distinguir entre listas cuyos componentes siempre se deben mantener en orden alfabético o numérico (*listas ordenadas*) y listas en las cuales los componentes no estén dispuestos en algún orden particular (*listas no ordenadas*). Empezaremos con las listas no ordenadas.

13.2 Listas no ordenadas

Operaciones básicas

Como hemos analizado en el capítulo 11, un ADT normalmente se aplica en C++ usando un par de archivos: el de especificación (como el precedente archivo `list.h`) y el de implementación, que contiene las aplicaciones de las funciones de miembros de clase. A continuación se muestra cómo empieza el archivo de implementación `list.cpp`:

```
//****************************************************************************
// ARCHIVO DE EJECUCIÓN (list.cpp)
// Este archivo pone en práctica las funciones miembros de clases de List
// Representación de lista: un array unidimensional y una longitud
//                          variable
//****************************************************************************

#include "list.h"
#include <iostream>

using namespace std;
// Miembros privados de clase:
//    int      length;             Longitud de la lista
//    int      currentPos;         Posición actual durante la iteración
//    ItemType data[MAX_LENGTH];   Array que contiene la lista
```

Ahora echamos un vistazo a las aplicaciones de las operaciones de listas básicas.

Crear una lista vacía Como muestra la figura 13-2, la lista existe en los elementos de array `data[0]` hasta `data[Length-1]`. A fin de crear una lista vacía, es suficiente `Length` a 0. No necesitamos almacenar ningún valor especial en el array de `data` para vaciar la lista, porque sólo los valores de `data[0]` hasta `data[Length-1]` son procesados por los algoritmos de lista.

En la clase `List`, el lugar apropiado para inicializar la lista a fin de que quede vacía es el constructor de clase:

```
List::List()

// Constructor

// Poscondición:
//     longitud == 0

{
    length = 0;
}
```

Algo que usted notará conforme pasamos a través de las funciones de miembros de `List` es que las *afirmaciones de implementación* (las precondiciones y poscondiciones que aparecen en el archivo de implementación) frecuentemente se indican de modo distinto de las *afirmaciones abstractas* (que se ubican en el archivo de especificación). Las afirmaciones abstractas están escritas en términos que son significativos para el usuario de ADT; no se deberán mencionar detalles de aplicación. A diferencia de esto, se puede hacer que las afirmaciones de implementación sean más precisas refiriendo directamente a variables y algoritmos en el código de implementación. En el caso del constructor de clase `List` la poscondición abstracta es simplemente que se ha creado una lista vacía. Por otro lado, la poscondición de implantación

```
// Poscondición:
// Longitud == 0
```

se expresa en términos de nuestra representación de datos privados.

La operación IsEmpty Esta operación devuelve `true` si la lista está vacía, y `false` si la lista no está vacía. Usando nuestra convención de que `Length` es igual a 0 si la lista está vacía, la aplicación de esta operación es sencilla.

```
bool List::IsEmpty() const

// Informa si la lista está vacía

// Poscondición:
//     Valor devuelto == true, si la longitud == 0
//                    == false, en caso contrario

{
    return (length == 0);
}
```

La operación IsFull La lista está llena si ya no hay lugar en el array que contiene los elementos de la lista; o sea, si la longitud de la lista es igual a `MAX_LENGTH`.

```
bool List::IsFull() const

// Informa si la lista está llena

// Poscondición:
//     Valor devuelto == true, si la longitud == MAX_LENGTH
//                    == false, en caso contrario

{
    return (length == MAX_LENGTH);
}
```

La operación Length Esta operación simplemente devuelve al cliente la longitud actual de la lista.

```
int List::Length() const

// Devuelve la longitud actual de la lista

// Poscondición:
//     Valor de retorno == longitud

{
    return length;
}
```

Inserción y supresión

A fin de elaborar un algoritmo para insertar un nuevo elemento en la lista observamos primero que estamos trabajando con una lista no ordenada y que los valores no se tendrán que mantener en algún orden particular. Por tanto, podemos almacenar un nuevo valor en el array —`data[Length]`— y luego incrementar `Length`. Este algoritmo provoca una pregunta: ¿necesitamos verificar que hay lugar en la lista para el nuevo elemento? Tenemos dos opciones. La función `Insert` puede examinar `Length` comparando con `MAX_LENGTH` y devolver una señal de error si no hay ningún lugar, o podemos permitir que el código de cliente realice la prueba antes de llamar `Insert` (es decir, hacerlo una precondición que la lista no está llena). Si recordamos la declaración de clase `List` en `list.h`, podemos ver que hemos elegido el segundo planteamiento. El cliente puede usar la operación `IsFull` para asegurar que la precondición es verdadera. Si el cliente no satisface la precondición, se rompe el contrato entre cliente y función, y no se requiere que la función satisfaga la poscondición.

```
void List::Insert( /* in */ ItemType item )

// Inserta el elemento en la lista

// Poscondición:
//     length < MAX_LENGTH
// Poscondición:
//     data[length@entry] == item
//  && length == length@entry+1

{
    data[length] = item;
    length++;
}
```

Borrar un componente de una lista consiste en dos partes: encontrar el componente y removerlo de la lista. Antes de poder escribir el algoritmo, tenemos que saber qué hacer si el componente no está allí. *Borrar* puede significar "Borrar si está allí" o "Borrar, sí *está* allí". De acuerdo con la declaración de clase `List` en `list.h`, supongamos el primer significado; el código para la primera definición también funciona para la segunda, pero no es tan eficiente. Tenemos que empezar en el principio de la lista y buscar el valor a ser borrado. Si lo encontramos, ¿cómo lo removemos? Tomamos el último valor en la lista (el que está guardado en `data[Length-1]`, lo colocamos donde está ubicado el elemento a borrar, y luego disminuimos `Length`. Mover el último elemento desde su posición original sólo es apropiado en el caso de una lista no ordenada porque no necesitamos conservar el orden de los elementos en la lista.

La definición "Borrar si está allí" requiere un ciclo de búsqueda con una condición compuesta. Examinamos cada componente a su vez y dejamos de buscar cuando encontramos el elemento a ser borrado o cuando hemos visto todos los elementos y sabemos que no está allí.

```
void List::Delete( /* in */ ItemType item )

// Elimina el elemento de la lista, si está ahí

// Precondición:
//     length > 0
// Poscondición:
//     SI el elemento está en el array de datos de la entrada
//           La primera aparición del elemento ya no está en el array
//        && length == length@entry - 1
//     ELSE
//        la longitud y el array de datos permanecen sin cambio

{
    int index = 0;    // Variable de índice

    while (index < length && item != data[index])
        index++;

    if (index < length)
    {                                  // Eliminar el elemento
        data[index] = data[length-1];
        length--;
    }
}
```

Para observar cómo trabaja el ciclo While y la subsiguiente instrucción If, vamos a ver las dos posibilidades: `item` está en la lista o no está en ella. Si `item` está en la lista, el ciclo termina cuando la expresión `index < length` es verdadera y la expresión `item != data[index]` es `falsa`. Después de la salida del ciclo, la instrucción If concluye que la expresión `index < length` es `true` y elimina el elemento. Por otra parte, si `item` no está en la lista, el ciclo termina cuando la expresión `index < length` es `falsa`; o sea, cuando `index` se vuelve igual a `length`. Subsecuentemente, la condición If es `falsa`, y la función retorna sin cambiar nada.

Búsqueda secuencial

En la función `Delete`, el algoritmo que hemos usado para buscar el elemento a ser borrado es conocido como una *búsqueda secuencial* o *lineal* en una lista no ordenada. Usamos el mismo algoritmo para aplicar la función `IsPresent` de la clase `List`.

```
bool List::IsPresent( /* in */ ItemType item ) const

// Busca en la lista al elemento e informa si se halló

// Poscondición:
//     Valor devuelto == true, si el elemento está en data[0.. Length -1]
//                    == false, en caso contrario

{
    int index = 0;     // Variable de índice

    while (index < length && item != data[index])
        index++;

    return (index < length);
}
```

Este algoritmo se llama búsqueda secuencial porque comenzamos en el inicio de la lista y miramos cada elemento en secuencia. Detenemos la búsqueda tan pronto como encontramos el elemento que estamos buscando (o cuando lleguemos al final de la lista, concluyendo que el elemento deseado no está presente en ella).

Podemos usar este algoritmo en cualquier programa que requiera una búsqueda de lista. En la forma demostrada, el algoritmo buscará en una lista de componentes de `ItemType`, siempre y cuando `ItemType` sea un tipo integral o de la clase `string`. Para usar la función con una lista de valores de punto flotante, tendremos que modificarlo de tal forma que la instrucción While busque la casi igualdad en lugar de la igualdad exacta (debido a las razones que hemos analizado en el capítulo 10). En el siguiente enunciado suponemos que `EPSILON` está definido como una constante global.

```
while (index < length && fabs(item – data[index]) >= EPSILON)
    index++;
```

El algoritmo de búsqueda secuencial encuentra la primera ocurrencia del elemento que se busca. ¿Cómo lo modificaríamos para encontrar la última ocurrencia? Inicializaríamos `index` a `length-1` y disminuiríamos `index` cada vez a través del ciclo, deteniéndonos al encontrar el elemento que deseamos o cuando `index` se vuelve –1.

Antes de dejar atrás este algoritmo de búsqueda, vamos a introducir una variante que hace el programa más eficiente, aunque un poco más complejo. El ciclo While contiene una condición compuesta: se detiene cuando encuentra `item` o cuando llega al final de la lista. Podemos insertar una copia de `item` en `data[length]`, o sea en el componente del array más allá del final de la lista, como un centinela. De este modo garantizamos que vamos a encontrar `item` en la lista. Luego pode-

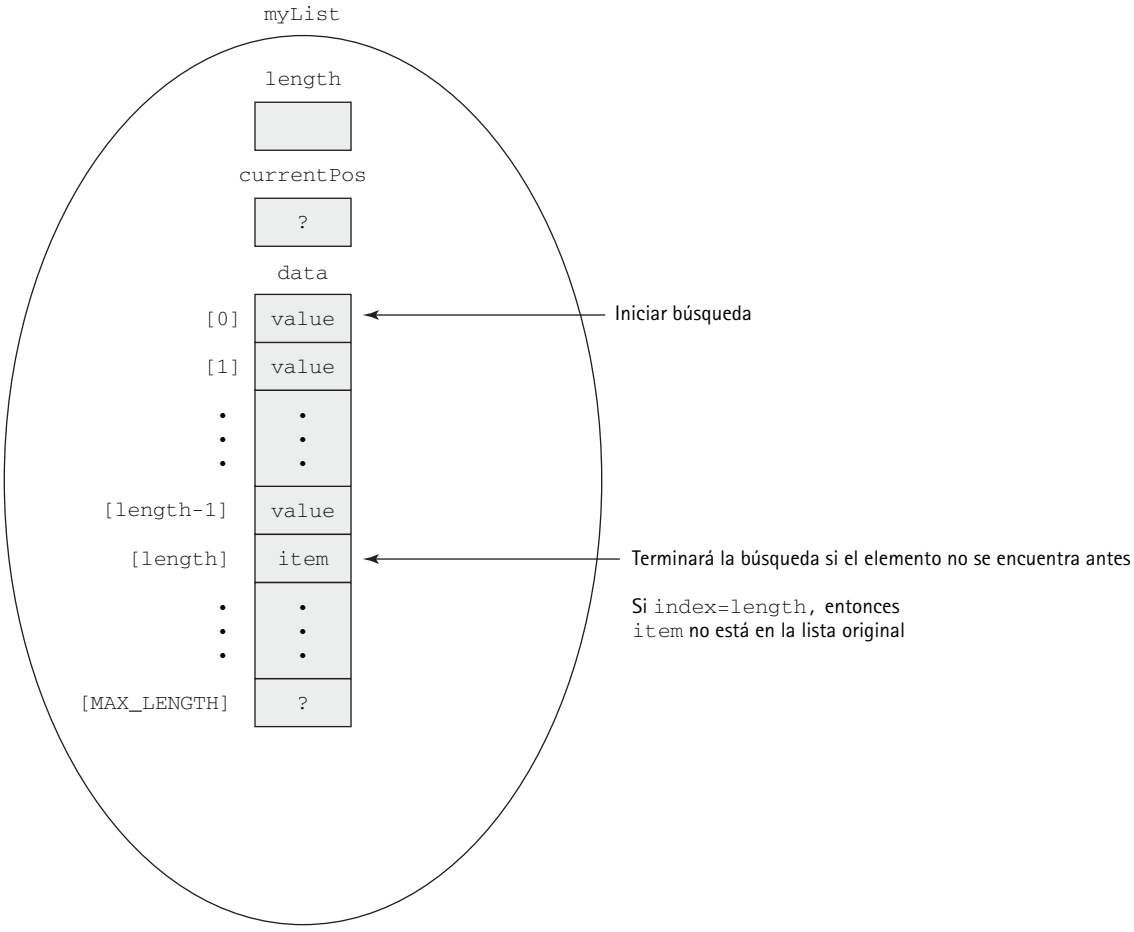


Figura 13-3 *Búsqueda secuencial con una copia de* `item` *en* `data[Length]`

mos eliminar la condición que verifica el final de la lista (`index < length`) (véase la figua 13-3). La eliminación de una condición le ahorra a la computadora el tiempo necesario para comprobarla. En este caso ahorramos tiempo durante cada iteración del ciclo, así que los ahorros se suman rápidamente. Observe, sin embargo, que ganamos eficiencia al costo de espacio. Tenemos que declarar que el tamaño del array sea 1 más que `MAX_LENGTH` para mantener el valor del centinela si la lista se llena. Esto quiere decir que tenemos que cambiar la parte privada de la declaración de clase de la siguiente manera:

```
class List
{
   ⋮
private:
    int      length;
    int      currentPos;
    ItemType data[MAX_LENGTH+1];
};
```

La figura 13-3 refleja este cambio. El último elemento del array muestra un índice de `MAX_LENGTH` en lugar de `MAX_LENGTH-1`, como en la figura 13-2.

La siguiente función, `IsPresent2`, aplica este nuevo algoritmo. Después de que termine el ciclo de búsqueda, la función devuelve `true` si `index` es menos que `length`; en caso contrario, devolverá `false`.

```
bool List::IsPresent2( /* in */ ItemType item ) const

// Busca en la lista al elemento e informa si se encontró

// Poscondición:
//     data[0..length -1] son los mismos que en la entrada
//  && data[length] se sobrescribe para ayudar en la búsqueda
//  && Devolver el valor == true, si el elemento está en data[0..length - 1]
//                       == false, en caso contrario
{
    int index = 0;                  // Variable de índice

    data[length] = item;            // Almacene el elemento en la posición
                                    //   más allá de la lista
    while (item != data[index])
        index++;

    return (index < length);
}
```

Observe que la declaración de nuestra clase `List` en el archivo `list.h` incluye la función de miembro `IsPresent`, pero no `IsPresent2`. Hemos presentado a `IsPresent2` sólo para mostrarle un planteamiento alternativo para aplicar la operación de búsqueda.

Iteradores

Los iteradores se usan con tipos compuestos para permitir al usuario procesar una estructura entera componente por componente. Para dar al usuario acceso a cada elemento en secuencia, proporcionaremos dos operaciones: una para inicializar el proceso de iteración (análogo a Reset u Open con un archivo) y una para devolver una copia del "componente siguiente" cada vez que se le llame. El usuario puede entonces fijar un ciclo que procesa cada componente. Llamamos a estas operaciones Reset y GetNextItem. Observe que Reset en sí no es un iterador, sino un transformador auxiliar que apoya la iteración. Otro tipo de iterador es uno que toma una operación y la aplica a cada elemento en la lista.

La función `Reset` es análoga a la operación de abrir para un archivo donde el apuntador del archivo se posiciona al principio del archivo, así que la primera operación de introducción accede al primer componente del archivo. Esto es, necesitamos una variable que se mantenga al tanto del siguiente elemento a ser devuelto de una iteración. Esta variable la denominamos `currentPos`. `Reset` debe inicializar `currentPos` para que apunte al primer elemento en la lista.

La operación `GetNextItem` es análoga a una operación de salida; accede al siguiente elemento, que está en `currentPos`, incrementa `currentPos`, y devuelve el elemento que accede. ¿Qué pasa si se ha accedido al último elemento? Vamos a poner `currentPos` nuevamente al primer elemento. De esta manera es posible que el programa obtenga la respuesta equivocada, pero no accederá a datos fuera de la lista.

Reset

Inicializar currentPos al primer elemento

GetNextItem

```
Colocar el elemento a data[currentPos]
IF currentPos es Length-1
    Colocar currentPos a 0
ELSE
    Incrementar currentPos
Devolver item
```

`currentPos` es indefinido hasta que sea inicializado por `Reset`. Después de la primera llamada a `GetNextItem`, `currentPos` es la ubicación del siguiente elemento a ser accedido por `GetNextItem`. Por tanto, a fin de aplicar este algoritmo en una lista basada en arrays en C++, `currentPos` debe ser inicializado a 0. Estas operaciones se codifican de la siguiente manera:

```
void List::Reset()
// Post: se ha inicializado currentPos.
{
    currentPos = 0;
}
```

¿Qué pasaría si se ejecuta una operación de transformador entre llamadas a `GetNextItem`? La iteración sería inválida. Por esta razón hay una precondición para impedir que esto suceda.

```
Itemtype List::GetNextItem()

// Precondición:
//     Ningún transformador ha sido ejecutado desde la última
//     llamada
// Poscondición:
//     El valor devuelto es currentPos@entry
//     && se ha actualizado la posición actual
//     Si ha sido devuelto el último elemento, la siguiente llamada
//     devuelve el primer elemento

{
    ItemType item;
    item = data[currentPos];
    if (currentPos == length - 1)
        currentPos = 0;
    else
        currentPos++;
    return item;
}
```

`Reset` y `GetNextItem` están diseñados para ser usados en un ciclo en el programa del cliente que itera a través de todos los elementos en la lista. La precondición en las especificaciones para `GetNextItem` protege contra intentos de acceder a un elemento de array que no está en la lista. Recordemos el programa de temperaturas. `Reset` y `GetNextItem` fueron usados dos veces: una vez para imprimir la lista original y otra para imprimir la lista ordenada corregida.

Ordenamiento

Aunque estemos aplicando un ADT de lista no ordenada, habrá ocasiones en que el usuario de la clase `List` pueda querer reordenar los componentes de la lista para que tengan un cierto orden justo antes de llamar a la función `Print`. Por ejemplo, el usuario podrá querer presentar una lista de números de existencias en un orden ascendente o descendente, o querrá poner una lista de palabras en orden alfabético. En el desarrollo de software, el ordenamiento de elementos de listas es una operación muy común que se llama **ordenamiento.**

Ordenamiento Colocar los componentes de una lista en orden (por ejemplo, palabras en orden alfabético o números en orden ascendente o descendente.

Si usted recibe una hoja con una columna de 20 números, y si se le pide que escriba los números en orden ascendente, usted probablemente haría lo siguiente:

1. Repasar la lista para buscar el número más pequeño.
2. Anotar en la hoja una segunda columna.
3. Tachar el número en la lista original.
4. Repetir este proceso, siempre buscando el número más pequeño que permanezca en la lista original.
5. Detenerse cuando todos los números estén tachados.

Podemos aplicar este algoritmo directamente en C++, pero necesitamos dos arrays: uno para la lista original, y otro para la lista ordenada. Si la lista es grande, posiblemente no tengamos la memoria suficiente para dos copias de ella. También, ¿cómo "tachamos" un componente de array? Podríamos simular el tachado de un valor remplazándolo con un valor simulado, como por ejemplo `INT_MAX`. Esto quiere decir que colocaríamos el valor de la variable tachada a algo que no podría interferir con el procesamiento del resto de los componentes. Sin embargo, una leve variación de nuestro algoritmo manual nos permite ordenar los componentes *en su lugar*. No necesitamos usar un segundo array; podemos colocar un valor en su propio lugar en la lista haciendo que cambie su lugar con el componente que realmente se encuentra en esta posición.

Podemos enunciar el algoritmo del siguiente modo: buscamos el valor más pequeño en la lista y lo cambiamos con el componente en la primera posición en la lista. Buscamos el siguiente valor más pequeño en la lista y lo cambiamos con el componente en la segunda posición. Este proceso continúa hasta que todos los componentes están en el lugar correcto.

```
Conteo FOR desde 0 hasta Length-2
    Encuentre el valor mínimo en data[count..length-1]
    Intercambie el valor mínimo con data[count]
```

La figura 13-4 ilustra cómo funciona este algoritmo.

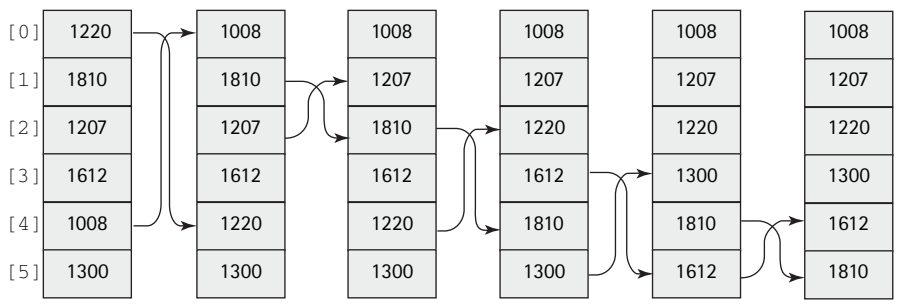


Figura 13-4 *Ordenamiento de selección directa*

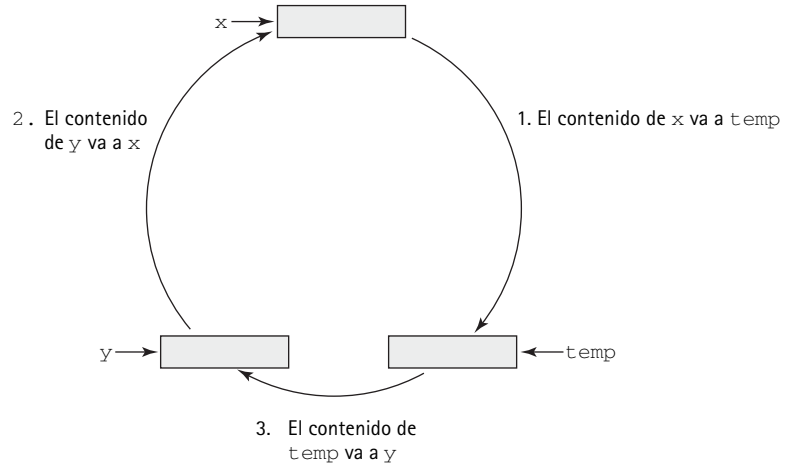


Figura 13-5 *Intercambio de los contenidos de dos variables, x y y*

Observe que realizamos pasadas de `length-1` a través de la lista porque `count` corre desde 0 hasta `length-2`. No es necesario ejecutar el ciclo cuando `count` es igual a `length-1` porque el último valor, `data[length-1]`, está en el lugar correcto después de que se hayan ordenado los componentes precedentes.

Este ordenamiento, llamado *ordenamiento de selección directa*, pertenece a una clase llamada ordenamientos de selección. Existen muchos tipos de algoritmos de ordenamiento. Los ordenamientos de selección se caracterizan por encontrar el valor más pequeño (o más grande) que se quede en la parte no ordenada en cada iteración, e intercambiarlo con el valor indizado por el contador de iteración. El intercambio de contenidos de dos variables requiere una variable temporal para que no se pierdan valores (véase la figura 13-5).

A continuación se encuentra el código para la operación de ordenamiento de la clase `List`:

```
void List::SelSort()

// Clasifica la lista en orden ascendente

// Poscondición:
//     el array de datos contiene los mismos valores que data@entry,
//     reordenados en orden ascendente

{
   ItemType temp;               // Variable temporal
   int      passCount;          // Variable de control de ciclo
   int      searchIndx;         // Variable de control de ciclo
   int      minIndx;            // Índice de mínimo hasta el momento

   for (passCount = 0; passCount < length - 1; passCount++)
   {
      minIndx = passCount;

      // Encontrar el índice del componente más pequeño
      // en data[passCount..length-1]

      for (searchIndx = passCount + 1; searchIndx < length;
                                                  searchIndx++)
```

```
            if (data[searchIndx] < data[minIndx])
               minIndx = searchIndx;

         // Intercambiar data[minIndx] y data[pass Count]

         temp = data[minIndx];
         data[minIndx] = data[passCount];
         data[passCount] = temp;
      }
}
```

Observe que en cada paso a través del ciclo exterior estamos buscando el valor mínimo en el resto de la lista (`data[passCount]` por medio de `data[Length-1]`). Por tanto, `minIndx` es inicializado a `passCount` y el ciclo interior corre de `searchIndx` igual a `passCount+1` hasta `length-1`. Al salir del ciclo interior, `minIndx` contiene la posición del valor más pequeño. (Nótese que la instrucción If es la única instrucción en el ciclo.)

Observe también que podemos intercambiar un componente consigo mismo, lo que ocurre si ningún valor en la lista que queda es menor a `data[passCount]`. Podríamos evitar este intercambio innecesario verificando si `minIndx` es igual a `passCount`. Puesto que esta comparación se realizaría durante cada iteración del ciclo exterior, es más eficiente no verificar esta posibilidad y simplemente intercambiar algo consigo mismo de vez en cuando. Si los componentes que estamos ordenando son mucho más complejos que simples números, podríamos reconsiderar esta decisión.

Este algoritmo ordena los componentes de forma ascendente. Para de forma descendente tendríamos que revisar para buscar el valor máximo en lugar del valor mínimo. Para ello simplemente cambiaríamos el operador relacional en el ciclo interior de <to>. Desde luego que `minIndx` ya no sería un identificador apropiado y debería cambiar a `maxIndx`.

Por el hecho de proporcionar al usuario de la clase `List` una operación de ordenamiento, no hemos convertido nuestro ADT de lista no ordenada en un ADT de lista ordenada. Los algoritmos `Insert` y `Delete` que escribimos no conservan el ordenado por valores. `Insert` coloca un nuevo elemento al final de la lista, independientemente de su valor, y `Delete` mueve el último elemento a una posición diferente en la lista. Después de la ejecución de `SelSort`, los elementos de la lista quedan en orden sólo hasta que se realice la siguiente inserción o eliminación. Ahora vemos un ADT de lista ordenada donde todas las operaciones de lista cooperan para conservar el orden de los componentes de la lista.

13.3 Listas ordenadas

En la clase `List`, los dos algoritmos `IsPresent` e `IsPresent2` suponen que la lista que se ha de revisar no está ordenada. Una desventaja en la búsqueda de una lista no ordenada es que tenemos que revisar la lista completa para descubrir que el elemento de la búsqueda no se encuentra en ella. Piense cómo sería si su directorio telefónico de la ciudad contuviera los nombres de las personas en orden aleatorio en lugar de alfabético. Para buscar el número telefónico de Mary Anthony tendría usted que empezar con el primer nombre en el directorio y revisar en forma secuencial, página tras página, hasta encontrarlo. En el peor de los casos tendría que examinar decenas de miles de nombres, sólo para descubrir que el nombre de Mary no está en el directorio.

Los directorios telefónicos, por supuesto, sí *están* alfabetizados, y el orden alfabético facilita la búsqueda. Si el nombre de Mary Anthony no está en el directorio, se descubre este hecho rápidamente si se empieza a buscar en las "A" y se detiene la búsqueda tan pronto como se pase el lugar donde su nombre debería estar.

Vamos a definir un ADT de lista ordenada donde los componentes siempre permanecen en orden por valores, independientemente de las operaciones que se apliquen. A continuación se encuentra el archivo `slist.h` que contiene la declaración de una clase de `SortedList`.

```
//****************************************************************************
// ARCHIVO DE ESPECIFICACIÓN (SortedList.h)
// Este archivo da la especificación de un tipo de datos abstractos.
// Se supone que los componentes están en orden por valor.
// Para ahorrar espacio, se omite de cada función los comentarios
// de precondición que documentan las suposiciones hechas acerca
// de los datos de parámetros de entrada válidos. Éstos se incluirían
// en un programa propio para uso real.
//****************************************************************************

const int MAX_LENGTH =100;  // Número posible máximo
                            //   de componentes necesarios
typedef int ItemType;       // Tipo de cada componente
                            //   (un tipo simple o clase de cadena)

class SortedList
{
public:
    bool IsEmpty() const;

        // Poscondición:
        //     Valor de retorno == true, si la lista clasificada (SortedList)
                está vacía
        //                      == false, en caso contrario

    bool IsFull() const;

        // Postcondition:
        //     Valor de retorno == si SortedList está llena
        //                      == false, en caso contrario

    int Length() const;

        // Poscondición:
        //     Valor devuelto == longitud de SortedList

    void Insert( /* in */ ItemType& item );

        // Inserta el elemento en SortedList

        // Poscondición:
        //     length < MAX_LENGTH
        //  && data[0..Length - 1] están en orden ascendente
        // Poscondición:
        //     el elemento está en la lista clasificada
        //  && Length = Length@entry + 1
        //  && data[0..Length - 1] están en orden ascendente

    void Delete( /* in */ ItemType item );

        // Precondición:
        //     NOT IsEmpty()
        // Poscondición:
        //     SI el elemento está en SortedList en la entrada
```

```
        //         La primera ocurrencia del elemento ya no es
        //         en SortedList
        //       && Length() == Length@entry - 1
        //     ELSE
        //         SortedList permanece sin cambio

    bool IsPresent( /* in */ ItemType item ) const;

        // Precondición:
        // Poscondición:
        //     Valor de retorno == true, si el elemento está en la lista
        //     clasificada
        //                      == false, en caso contrario

    void Reset();

        // Posconción:
        //     Se inicializó la iteración

    ItemType GetNextItem();

        // Precondición:
        // La iteración se inicializó mediante llamada a Reset;
        //     Ningún transformador ha sido invocado desde la última llamada
        // Poscondición:
        //     Devuelve el elemento en la última posición en la
        //     SortedList

    SortedList();

        // Constructor
        // Poscondición:
        //     Se crea la lista vacía SortedList

private:
    int      length;
    int      currentPos;
    ItemType data[MAX_LENGTH];
    void BinSearch( ItemType, bool&, int& ) const;
};
```

¿Cómo se distingue la declaración de `SortedList` de la declaración de nuestra clase `List` original? Aparte de unos cuantos cambios en los comentarios de documentación, sólo hay dos diferencias:

1. La clase `SortedList` no proporciona una operación de ordenamiento al cliente. Esta operación no hace falta, puesto que se supone que los componentes de la lista en todo momento se mantendrán ordenados.
2. La clase `SortedList` tiene un miembro de clase adicional en la parte privada: una función `BinSearch`. Esta función es una función auxiliar ("ayudante") que sólo es usada por otras funciones de miembros de clase y es inaccesible para clientes. Analizaremos su finalidad cuando examinemos la aplicación de clase.

Vamos a ver qué cambios, si acaso, se requieren en los algoritmos para las operaciones ADT, puesto que ahora estamos trabajando con una lista ordenada.

Operaciones básicas

Los algoritmos para el constructor de clase, `IsEmpty`, `IsFull`, `Length`, `Reset` y `GetNextItem` son idénticos a los que se encuentran en la clase `List`. Los constructores colocan el miembro de datos privados `length` a 0. `IsEmpty` reporta si `length` es igual a 0, `IsFull` reporta si `length` es igual a `MAX_LENGTH`, `length` devuelve el valor de `length`, y `Print` emite los elementos de la lista desde el primero hasta el último.

Inserción

A fin de agregar un nuevo valor a una lista ya ordenada, podríamos almacenar el nuevo valor en `data[length]`, incrementar `length` y volver a ordenar el array. Sin embargo, esta solución *no* es una manera eficiente de resolver el problema. La inserción de cinco elementos nuevos resultará en cinco operaciones de ordenamiento separadas.

Si tuviéramos que insertar un valor en forma manual en una lista ordenada, escribiríamos el nuevo valor afuera al lado y trazaríamos una línea para mostrar dónde permanece. Para encontrar esta posición empezamos en la parte superior y recorremos la lista hasta que encontramos un valor mayor al que insertamos. El nuevo valor entra en la lista justo antes de este punto.

Podemos usar un proceso similar en nuestra función `Insert`. Buscamos el lugar correcto en la lista usando el algoritmo manual. En lugar de escribir el valor al lado, movemos todos los valores mayores al nuevo un lugar hacia abajo para hacer lugar para él. El algoritmo principal se expresa como sigue, donde `item` es el valor a insertar.

```
MIENTRAS no se encuentre lugar Y se vean más espacios
    Si el elemento > componente actual en la lista
        Incrementar la posición actual
    DE OTRO MODO
        Lugar encontrado
Desplace hacia abajo el resto de la lista
Insertar elemento
Incrementar la longitud
```

Suponiendo que `index` es el lugar donde `item` se debe insertar, el algoritmo para recorrer la lista hacia abajo es:

```
Set data[length]     = data[length–1]
Set data[length–1]   = data[length–2]
      ⋮                    ⋮
Set data[index+1]    = data[index]
```

El algoritmo está ilustrado en la figura 13-6.

Este algoritmo está basado en cómo realizaríamos la tarea en forma manual. A menudo una adaptación de este estilo es la mejor manera para resolver un problema. Sin embargo, seguimos pensando si en este caso se revela una manera levemente mejor. Observe que estamos buscando a partir del principio de la lista (la gente siempre lo hace así), y movemos hacia abajo desde el final de la lista hacia arriba. Podemos combinar el buscar y el mover si empezamos al *final* de la lista.

Si `item` es el nuevo elemento a ser insertado, compare `item` con el valor en `data[length-1]`. Si `item` es *menor*, coloque `data[length-1]` en `data[length]` y compare `item` con el valor en `data[length-2]`. Este proceso continúa hasta que encontremos el lugar donde `item` es mayor o igual al elemento en la lista. Guarde `item` directamente debajo de él. A continuación el algoritmo:

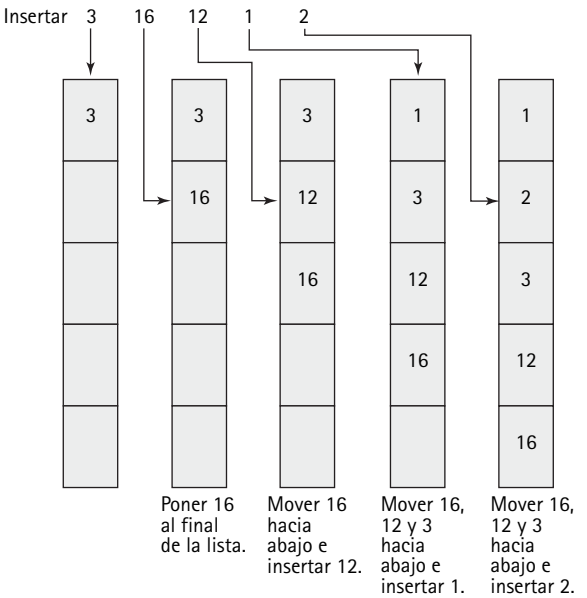


Figura 13-6 *Insertar en una lista ordenada*

Establecer índice = longitud – 1
MIENTRAS índice ≥ 0 Y elemento < data[index]
    Establecer data[index + 1] = data[index]
    Disminuir el índice
Establecer data[index + 1] = elemento
Incrementar la longitud

¿Qué pasa con duplicados? El algoritmo continúa hasta que se encuentre un elemento que es menor al que estamos insertando. Por tanto, el nuevo elemento es insertado debajo de un valor duplicado (si es que hay uno). He aquí el código:

```
void SortedList::Insert( /* in */ ItemType item )

// Insertar el elemento en la lista

// Precondición:
//     longitud < MAX_LENGTH
// Poscondición:
//     el elemento está en la lista

{
    int index;    // Índice y variable de control de ciclo

    index = length - 1;
    while (index >= 0 && item < data[index])
    {
        data[index+1] = data[index];
        index--;
```

```
    }
    data[index+1] = item;          // Insertar elemento
    length++;
}
```

Observe que este algoritmo funciona incluso si la lista está vacía. Cuando la lista está vacía, `length` es 0 y el cuerpo del ciclo while no se ejecuta. De esta manera, `item` se guarda en `data[0]`, y `length` es incrementado a 1. ¿Funcionará el algoritmo si `item` es el más pequeño? ¿O el más grande? Veamos. Si `item` es el más pequeño, el cuerpo del ciclo es ejecutado `length` veces, e `index` es –1. Así que `item` es guardado en la posición 0 donde pertenece. Si `item` es el más grande, no se ejecuta el cuerpo del ciclo. El valor de `index` sigue siendo `length-1`, así que `item` es guardado en `data[length]`, donde pertenece.

¿Está usted sorprendido de que el caso general también se encargue de los casos especiales? Esta situación no se da todo el tiempo, pero ocurre con la frecuencia suficiente para poder decir que es una buena práctica de programación empezar con el caso general. Si empezamos con los casos especiales, generamos por lo regular una solución correcta, pero no necesariamente nos daremos cuenta de que no hay necesidad de tratar los casos especiales por separado. Empiece entonces con el caso general, y luego trate como casos especiales sólo aquellas situaciones que el caso general no maneja correctamente.

Este algoritmo es la base para otro algoritmo de ordenamiento de uso múltiple: un *ordenamiento de inserción*. En un ordenamiento de inserción, los valores son insertados uno por uno en una lista que originalmente estuvo vacía. Un ordenamiento de inserción se usa a menudo cuando los datos de introducción se deben ordenar; cada valor se pone en su lugar correcto conforme se está leyendo. Usaremos esta técnica en el estudio de Caso práctico de resolución de problemas, al final de este capítulo.

Búsqueda secuencial

Cuando buscamos un elemento en una lista no ordenada, no nos daremos cuenta de que el elemento falta hasta que lleguemos al final de la lista. Si la lista ya está ordenada, sabemos que un elemento falta cuando pasamos por el lugar donde se debería encontrar en la lista. Por ejemplo, si una lista contiene los valores

```
 7
11
13
76
98
102
```

y estamos buscando 12, sólo tenemos que comparar 12 con 7, 11 y 13 para saber que 12 no está en la lista.

Si el elemento de búsqueda es mayor que el componente actual de la lista, seguimos al siguiente componente. Si el elemento es igual al componente actual, ya hemos encontrado lo que estamos buscando. Si el elemento es menor que el componente actual, entonces sabemos que no está en la lista. En cualquiera de los últimos dos casos, dejaremos de buscar. Podemos replantear esto en forma algorítmica por medio del siguiente código, donde `found` es colocado a `true` si el elemento de búsqueda fue encontrado.

```
// Búsqueda secuencial en una lista clasificada

index = 0;
while (index < length && item > data[index])
    index++;

found = (index < length && item == data[index]);
```

En promedio, la búsqueda en una lista ordenada de esta manera requiere el mismo número de iteraciones para encontrar un elemento como la búsqueda en una lista no ordenada. La ventaja de este nuevo algoritmo es que descubrimos más pronto si falta algún elemento. Por tanto, es levemente más eficiente; sin embargo, sólo funciona en una lista ordenada.

No usamos este algoritmo para aplicar la función `SortedList::IsPresent`. Para ello existe un algoritmo mejor, como veremos a continuación.

Búsqueda binaria

Hay un segundo algoritmo de búsqueda para una lista ordenada que es considerablemente más rápido, tanto para buscar un elemento como para descubrir la falta de un elemento. Este algoritmo se denomina *búsqueda binaria*. Una búsqueda binaria se basa en el principio de la aproximación sucesiva. El algoritmo divide la lista en dos mitades (divide entre 2; por eso se llama búsqueda *binaria*) y decide cuál es la siguiente mitad donde debe seguir buscando. La división de la parte seleccionada de la lista se repite hasta que se encuentra el elemento o se determina que el elemento no está en la lista.

Este método es análogo a la forma de buscar una palabra en un diccionario. Abrimos el diccionario en el centro y comparamos la palabra con una en la página que hemos abierto. Si la palabra que buscamos viene antes de esta palabra, continuamos nuestra búsqueda en la sección izquierda del diccionario. De lo contrario, continuamos en la sección derecha del diccionario. Repetimos este proceso hasta encontrar la palabra. Si no se encuentra allí, nos damos cuenta de que hemos deletreado mal la palabra o que nuestro diccionario no está completo.

El algoritmo para una búsqueda binaria sigue a continuación. La lista de valores está en el array `data`, y el valor que se busca es `item` (véase la figura 13-7).

1. Compare `item` con `data[middle]`. Si `item` = `data[middle]`, entonces lo hemos encontrado. Si `item < data[middle]`, entonces buscamos en la primera mitad de `data`. Si `item > data[middle]`, entonces buscamos en la segunda mitad de `data`.
2. Redefina `data` para ser la mitad de `data` que buscamos a continuación, y repita el paso 1.
3. Deténgase cuando hayamos encontrado `item` o cuando sepamos que falta. Sabemos que falta cuando no hay más que buscar y aún no lo hemos encontrado.

Este algoritmo deberá tener sentido. En el mejor de los casos, con cada comparación encontramos el elemento que estamos buscando; en el peor de los casos, eliminamos la mitad de la lista restante de la consideración.

Necesitamos estar pendientes del primer lugar posible donde buscar (`first`) y del último lugar posible donde buscar (`last`). En cualquier momento dado estamos buscando sólo en `data[first]` hasta `data[last]`. Cuando empieza la función, `first` es fijado a 0 y `last` es fijado a `length-1` para abarcar la lista completa.

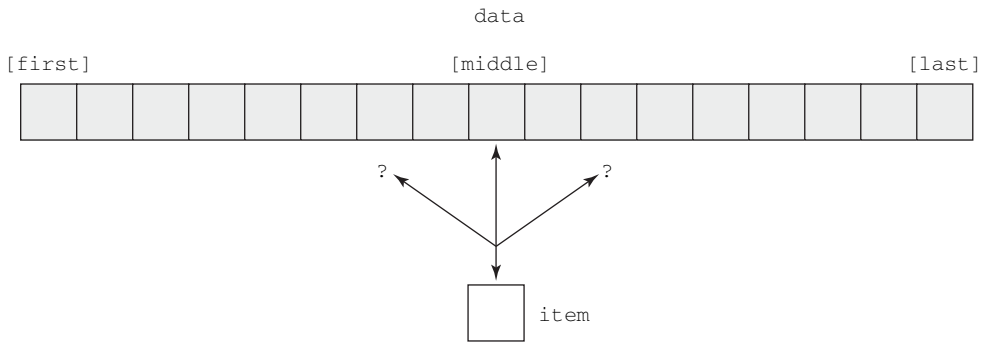


Figura 13-7 *Búsqueda binaria*

Nuestros tres previos algoritmos de búsqueda han sido operaciones booleanas de observador. Ellos simplemente contestan la pregunta: ¿se encuentra este elemento en la lista? Vamos a codificar la búsqueda binaria como una función que no sólo pregunta si el elemento está en la lista, sino también pregunta cuál es (si es que está allí). Para hacer esto tenemos que agregar dos parámetros a la lista de parámetros: un indicador booleano `found` (para decirnos si el elemento está en la lista) y una variable entera `position` (para decirnos de cuál elemento se trata). Si `found` es falso, entonces `position` es indefinido.

```
void SortedList::BinSearch(
      /* in */  ItemType item,              // Elemento por encontrar
      /* out */ bool&    found,             // Verdadero si se halla el
                                               elemento
      /* out */ int&     position ) const   // Ubicación si hallada

// Busca en la lista al elemento y devuelve el índice
// del elemento si se halló éste.

// Precondición:
//     longitud <= INT_MAX / 2
// Poscondición:
//     SI el elemento está en la lista
//         hallado == true && data[position] contiene al elemento
//     ELSE
//         hallado == false && la posición no está definida

{
    int first = 0;              // Límite interior en la lista
    int last = length - 1;      // Límite superior en la lista
    int middle;                 // Índice médio

    found = false;
    while (last >= first && !found)
    {
        middle = (first + last) / 2;
        if (item < data[middle])
            // Afirmación: el elemento no está en data[middle..last]
            last = middle - 1;
        else if (item > data[middle])
            // Afirmación: el elemento no está en data[first..middle]
            first = middle + 1;
        else
            // Afirmación: elemento == data[middle]
            found = true;
    }
    if (found)
        position = middle;
}
```

¿Deberá `BinSearch` ser un miembro público de la clase `SortedList`? No. La función devuelve el índice del elemento de array donde se encontró el elemento. Un índice de array es inútil para un cliente de `SortedList`. El array que contiene los elementos de la lista está encapsulado dentro de la parte privada de la clase y es inaccesible para clientes. Si revisamos la declaración de clase `SortedList`, veremos que `BinSearch` es un miembro de clase *privado*, no público. Nuestra intención es usarlos como una función de ayudante cuando aplicamos las operaciones públicas `IsPresent` y `Delete`.

Vamos a realizar un repaso del algoritmo de búsqueda binaria. El valor que estamos buscando es 24. La figura 13-8*a*) muestra los valores de `first`, `last` y `middle` durante la primera iteración. En esta iteración, 24 se compara con 103, el valor en `data[middle]`. Puesto que 24 es menor que 103, `last` se vuelve `middle-1` y `first` queda igual. La figura 13-8*b*) muestra la situación durante la segunda iteración. Esta vez, 24 se compara con 72, el valor en `data[middle]`. Puesto que 24 es menor que 72, `last` se vuelve `middle-1` y `first` nuevamente queda igual.

En la tercera iteración (figura 13-8*c*), tanto `middle` como `first` son 0. El valor 24 es comparado con 12, el valor en `data[middle]`. Puesto que 24 es mayor que 12, `first` se vuelve `middle+1`. En la cuarta iteración (figura 13-8*d*), `first`, `last` y `middle` son todos iguales. Nuevamente 24 es comparado con el valor en `data[middle]`. Puesto que 24 es menor que 64, `last` se vuelve `middle-1`. Ahora que `last` es menor que `first`, el proceso se detiene; `found` es `false`.

La búsqueda binaria es el algoritmo más complejo que hemos examinado hasta el momento. La siguiente tabla muestra `first`, `last`, `middle` y `data[middle]` para búsquedas de los valores 106, 400 y 406, usando los mismos datos como en el ejemplo anterior. Examine los resultados en esta tabla cuidadosamente.

item	first	last	middle	data[middle]	Termination of Loop
106	0	10	5	103	
	6	10	8	200	
	6	7	6	106	`found = true`
400	0	10	5	103	
	6	10	8	200	
	9	10	9	300	
	10	10	10	400	`found = true`
406	0	10	5	103	
	6	10	8	200	
	9	10	9	300	
	10	10	10	400	
	11	10			`last < first`
					`found = false`

El cálculo

```
middle = (first + last) / 2;
```

explica por qué la precondición de la función limita el valor de `length` a `INT_MAX/2`. Si el elemento que se está buscando reside en la última posición de la lista (por ejemplo cuando `item` es igual a 400 en nuestra lista de muestras), entonces `first + last` es igual a `length` + `length`. Si `length` es más grande que `INT_MAX/2`, la suma `length + length` llegaría a producir un desbordamiento.

Observe en la tabla que independientemente de que buscábamos 106, 400 o 406, el ciclo nunca se ejecutó más de cuatro veces. Nunca se ejecuta más de cuatro veces en una lista de 11 componentes porque la lista es cortada a la mitad en cada paso por el ciclo. La tabla siguiente compara una búsqueda secuencial con una búsqueda binaria en términos del número promedio de iteraciones necesarias para encontrar un elemento.

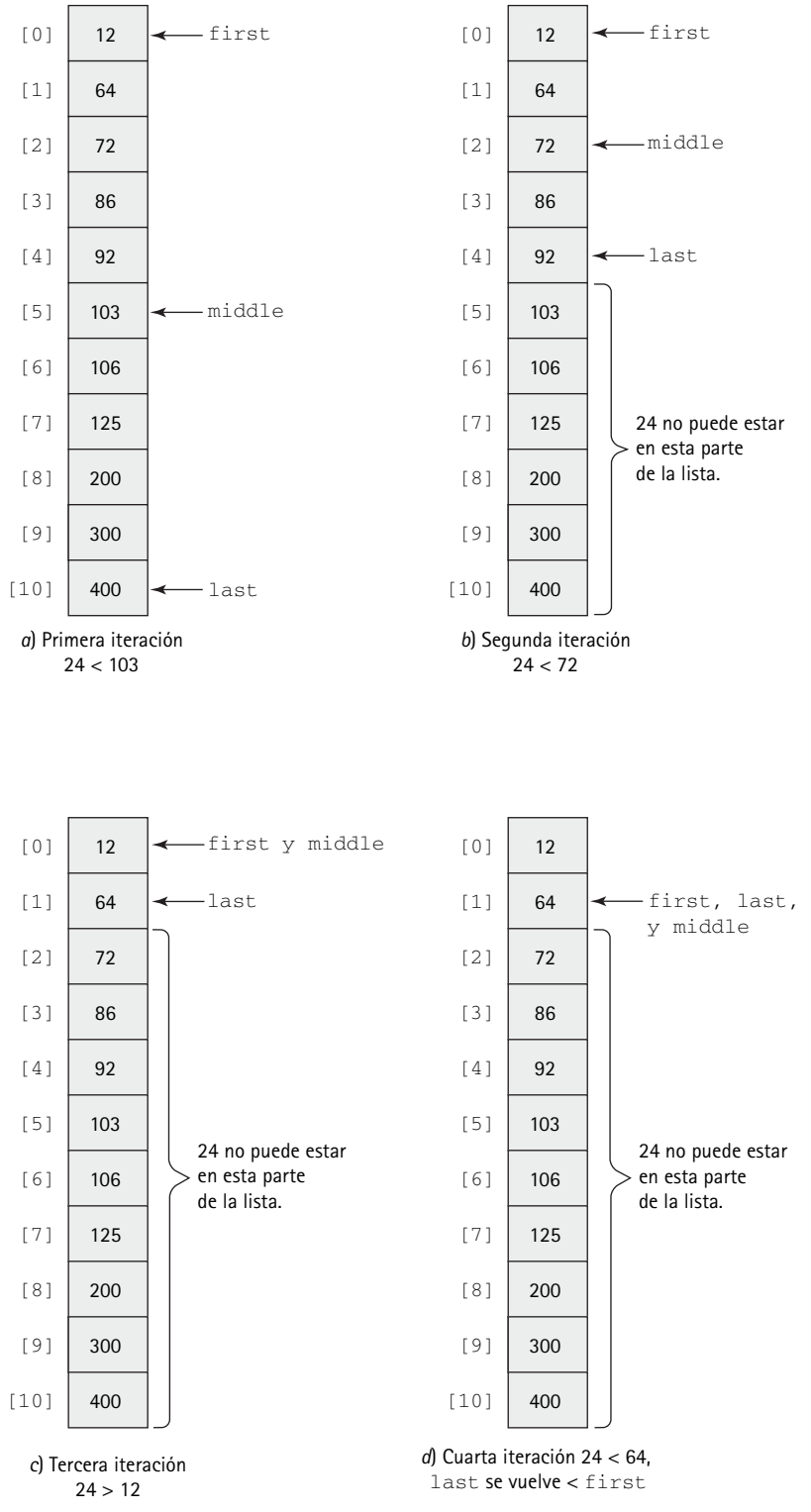


Figura 13-8 *Repaso de código de la función* `BinSearch` *(el elemento de búsqueda es 24)*

	Número promedio de iteraciones	
Longitud de la lista	**Búsqueda secuencial**	**Búsqueda binaria**
10	5.5	2.9
100	50.5	5.8
1 000	500.5	9.0
10 000	5 000.5	12.4

Si la búsqueda binaria es tanto más rápida, ¿por qué no usarla todo el tiempo? Es ciertamente más rápida en términos de veces a través del ciclo, pero se realizan más cálculos dentro del ciclo de búsqueda binaria que en los demás algoritmos de búsqueda. Consecuentemente, si el número de componentes en la lista es pequeño (digamos menos que 20), los algoritmos de búsqueda secuencial son más rápidos porque realizan menos trabajo en cada iteración. Conforme crece el número de componentes en la lista, el algoritmo de búsqueda binaria será relativamente más eficiente. Recuerde, sin embargo, que la búsqueda binaria requiere una lista ordenada, y el ordenamiento consume tiempo. Mantenga tres factores en mente cuando tenga que decidir qué algoritmo usar:

1. La longitud de la lista para la búsqueda
2. Si la lista ya está ordenada o no
3. El número de veces que la lista se tendrá que recorrer

Dada la función `BinSearch` (un miembro privado de la clase `SortedList`), es fácil aplicar la función `IsPresent` (un miembro público de la clase).

```
bool SortedList::IsPresent( /* in */ ItemType item ) const

// Busca en la lista al elemento e informa si se halló

// Precondición:
//     longitud <= INT_MAX / 2
// Poscondición:
//     Valor de retorno == true, si el elemento está en data[0..Length - 1]
//                      == false, en caso contrario
{
    bool found;      // Verdadero si se halló el elemento
    int position;    // Argumento requerido (pero no empleado) para
                     //   la llamada a BinSearch
    BinSearch(item, found, position);
    return found;
}
```

El cuerpo de `IsPresent` llama a `BinSearch`, obteniendo el resultado de la búsqueda en las variables `found` y `position`. Tal como en el juego de niños "Pass It On", `IsPresent` recibe el valor de `found` de `BinSearch` y simplemente lo pasa al cliente (vía la instrucción `return`). El cuerpo de `IsPresent` no está colocado en el lugar donde se encontró el elemento, así que ignorará el valor devuelto en el argumento de `position`. ¿Por qué hemos incluido este tercer argumento cuando diseñamos `BinSearch`? La respuesta es que la operación `Delete`, que veremos a continuación, llama a `BinSearch` y *sí* usa el argumento de `position`.

Borrado

En la función `List::Delete` hemos borrado un elemento moviendo el último componente en la lista hacia arriba para llenar la posición del elemento borrado. Aunque este algoritmo está bien para listas no ordenadas, no funcionará para listas ordenadas. Mover el último componente a una posición arbitraria en la lista significa casi con seguridad que se estropeará el ordenamiento de los componentes. Necesitamos un nuevo algoritmo para listas ordenadas.

Vamos a llamar `BinSearch` para indicar la posición del elemento a ser borrado. Luego podemos "empujar" el elemento borrado hacia fuera, moviendo todos los elementos de array remanentes una posición hacia arriba:

```
BinSearch(item, found, position)
IF found
    Shift remainder of list up
    Decrement length
```

El algoritmo para "Mover lista arriba" es

```
Set data[position]   = data[position+1]
Set data[position+1] = data[position+2]
  ⋮                    ⋮
Set data[length–2]   = data[length–1]
```

A continuación se presenta la versión codificada de este algoritmo:

```
void SortedList::Delete( /* in */ ItemType item )

// Borra al elemento de la lista si está ahí

// Precondición:
//     0 < longitud <= INT_MAX/2
// Poscondición:
//     SI el elemento está en el array de datos de la entrada
//           La primera aparición del elemento ya no está en el array
//        && length == length@entry - 1
//        && data[0..Length – 1] están en orden ascendente
//     ELSE
//        la longitud y el array de datos permanecen sin cambios

{
    bool found;      // Verdadero si se halla el elemento
    int position;    // Posición del elemento si se halla
    int index;       // Índice y variable de control de ciclo

    BinSearch(item, found, position);
    if (found)
    {
        // Desplazar data[position..Length – 1] una posición hacia arriba

        for (index = position; index < length – 1; index++)
            data[index] = data[index+1];
        length--;
    }
}
```

Fundamentos teóricos

La complejidad de buscar y ordenar

Hemos introducido la notación *Big-O* en el capítulo 6 como una forma de comparar el trabajo realizado por diferentes algoritmos. La aplicaremos a los algoritmos que hemos desarrollado en este capítulo, y vamos a ver cómo se comparan entre sí. En cada algoritmo empezamos con una lista que contiene un número de valores, *N*.

En el peor de los casos, nuestra función `List::IsPresent` recorre todos los valores *N* para localizar un elemento. De esta manera requiere *N* pasos para ejecutar. En promedio, `List::IsPresent` necesita aproximadamente *N/2* pasos para localizar un elemento; sin embargo, recuerde que en notación `Big-O` ignoramos factores constantes (igual que términos de orden menor). De modo que la función `List::IsPresent` es un algoritmo de orden *N*, o sea, *O(N)*.

`List::IsPresent` también es un algoritmo *O(N)* porque a pesar de que hemos guardado una comparación en cada iteración del ciclo, se realiza el mismo número de iteraciones. Sin embargo, si hacemos el ciclo más eficiente sin cambiar el número de iteraciones disminuye la constante (el número de pasos) por la que `N` es multiplicado en la fórmula de trabajo del algoritmo. Así, se dice que la función `List::IsPresent2` es un factor de constante más rápido que `List::IsPresent`.

¿Qué hay del algoritmo que hemos presentado para una búsqueda secuencial en una lista ordenada? El número de iteraciones disminuye para el caso en que el elemento falta en la lista. Sin embargo, lo único que hemos hecho es tomar un caso que requeriría `N` pasos y reducir su tiempo, en promedio a *N/2* pasos. Por tanto, este algoritmo también es *O*`(N)`.

Ahora considere `BinSearch`. En el peor de los casos, elimina la mitad de los componentes restantes de la lista en cada iteración. De este modo, el número de peor caso de iteraciones es igual al número de veces que *N* se deberá dividir entre 2 para eliminar todos los valores menos uno. Este número es calculado tomando el algoritmo, base 2, de *N* (escrito $\log_2 N$). A continuación algunos ejemplos de $\log_2 N$ para diferentes valores de *N:*

N	Log_2N
2	1
4	2
8	3
16	4
32	5
1 024	10
32 768	15
1 048 576	20
33 554 432	25
1 073 741 824	30

Como puede ver, para una lista de más de mil millones de valores, `BinSearch` necesita sólo 30 iteraciones. Es definitivamente la mejor opción para búsquedas en listas grandes. De los algoritmos como `BinSearch` se dice que son de *orden logarítmico*.

(continúa) ▼

La complejidad de buscar y ordenar

Ahora abordaremos el ordenamiento. La función `SelSort` contiene ciclos anidados For. El total de iteraciones es el producto de las iteraciones ejecutadas por los dos ciclos. El ciclo exterior se ejecuta $N - 1$ veces. El ciclo interior también inicia ejecutando $N - 1$ veces, pero disminuye paulatinamente hasta que sólo ejecuta una iteración: el ciclo interior ejecuta $N/2$ iteraciones. El total de iteraciones es, entonces,

$$\frac{(N-1)\times N}{2}$$

Ignorando el factor de constante y el término de orden menor, esto significa N^2 iteraciones, y `SelSort` es un algoritmo $O(N^2)$. Mientras que `BinSearch` sólo necesita 30 iteraciones para buscar a través de un array ordenado de mil millones de valores, ¡para poner el array en orden `SelSort` necesita aproximadamente mil millones por mil millones de iteraciones!

Mencionamos que el algoritmo `SortedList::Insert` forma la base para un ordenamiento de inserción, en el cual se insertan valores en una lista ordenada conforme se están introduciendo. En promedio, `SortedList::Insert` debe mover hacia abajo la mitad de los valores ($N/2$) en la lista; así que se trata de un algoritmo $O(N)$. Si `SortedList::Insert` se llama para cada valor de entrada, estamos ejecutando un algoritmo $O(N)$ N veces; por tanto, un algoritmo de ordenamiento de inserción es un algoritmo $O(N^2)$.

¿Todo algoritmo de ordenamiento es $O(N^2)$? La mayoría de los más sencillos sí lo son, pero existen algoritmos de ordenamiento $O(N \times \log_2 N)$. Los algoritmos que son $O(N \times \log_2 N)$ se asemejan mucho más en su rendimiento a algoritmos $O(N)$ que los algoritmos $O(N^2)$. Por ejemplo, si N es 1 millón, entonces un algoritmo $O(N^2)$ necesita un millón por un millón (1 billón) de iteraciones, pero un algoritmo $O(N \times \log_2 N)$ necesita sólo 20 millones de iteraciones, o sea, es 20 veces más lento que el algoritmo $O(N)$, pero 50 000 veces más rápido que el algoritmo $O(N^2)$.

Ahora vamos a dirigir nuestra atención a otro ejemplo de las listas basadas en arrays: un tipo especial de array que es útil cuando se trabaja con datos de caracteres alfanuméricos.

13.4 Entendiendo las cadenas de caracteres

Ya desde el capítulo 2 hemos usado la clase `string` para guardar y manipular cadenas de caracteres.

```
string name;

name = "James Smith";
len = name.length();
 ⋮
```

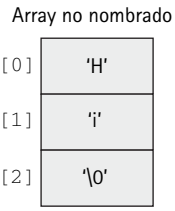
En algunos contextos pensamos en una cadena como una unidad única de datos. En otros, la tratamos como un grupo de caracteres individualmente accesibles. En particular pensamos en una cadena como una colección lineal y de longitud variable de componentes homogéneos (del tipo `char`). ¿Suena familiar? Debería sonar familiar. Como una abstracción, una cadena es una lista de caracteres que, en cualquier momento, tiene una longitud asociada con ella.

Pensando en una cadena como un ADT, ¿cómo aplicaríamos el ADT? Hay muchas maneras de aplicar cadenas. Los programadores han especificado y aplicado sus propias clases de cadenas: la clase `string` de la biblioteca estándar, por ejemplo. El lenguaje C++ tiene su propia noción integrada de una cadena: la **cadena C**. En C++, una constante de cadena (o literal de cadena, o cadena literal) es una secuencia de caracteres entre comillas:

Cadena C En C y en C++, secuencia de caracteres con terminación nula almacenada en un array `char`.

```
"Hi"
```

Una constante de cadena se guarda como un array `char` con componentes suficientes para mantener cada carácter especificado y uno más: el *carácter nulo*. Este carácter nulo, que es el primer carácter en los juegos de caracteres de ASCII y EBCDIC, tiene la representación interna 0. En C++, la secuencia de escape `'\0'` representa el carácter nulo. Cuando el compilador encuentra la cadena `"Hi"` en un programa, guardará los tres caracteres 'H', 'i' y '\0' en un array `char` anónimo (no nombrado) de tres elementos como sigue:



La cadena C es el único tipo de array de C++ para el cual existe una constante agregada: la constante de cadena. Observe que en un programa de C++ los símbolos `'A'` denotan un carácter individual, mientras que los símbolos `"A"` denotan dos: el carácter 'A' y el caracter nulo.*

Además de constantes de cadena C podemos crear *variables* de cadenas C. Para hacer esto declaramos explícitamente un array `char` para guardar en él todos los caracteres que queremos, terminando con el carácter nulo. A continuación presentamos un ejemplo:

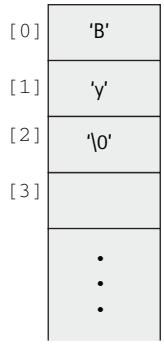
```
char myStr[8];   // Espacio para 7 caracteres significativos más '\0'

myStr[0] = 'H';
myStr[1] = 'i';
myStr[2] = '\0';
```

En C++, todas las cadenas C (constantes o variables) se suponen de terminación nula. Esta convención es un acuerdo entre todos los programadores de C++ y funciones estándares de biblioteca. El carácter nulo sirve como un valor de centinela; permite que los algoritmos localicen el final de la cadena. Por ejemplo, aquí hay una función que determina la longitud de cualquier cadena C, no contando el carácter nulo terminante:

```
int StrLength( /* in */ const char str[] )

// Poscondición:
//     str contiene una cadena con terminación nula
// Posconción:
//     Valor de función == número de caracteres en str (sin incluir a '\0')
{
    int i = 0;   // Variable de índice

    while (str[i] != '\0')
        i++;
    return i;
}
```

* C *string* no es un término oficial usado en manuales del lenguaje C++. Estos manuales normalmente usan el término *string*. Sin embargo, nosotros usamos C *string* para distinguir entre el concepto general de una cadena y la representación integrada de array definida por los lenguajes C y C++.

El valor de `i` es el valor correcto a devolver para esta función. Si el array en análisis es



entonces `i` es igual a 2 en la salida del ciclo. Por tanto, la longitud de la cadena es 2.

El argumento para la función `StrLength` puede ser una variable de cadena C, tal como en la llamada de función

```
cout << StrLength(myStr);
```

o puede ser una constante de cadena

```
cout << StrLength("Hello");
```

En el primer caso, la dirección base del array `myStr` se transmite a la función, como hemos analizado en el capítulo 12. En el segundo caso, una dirección base también se transmite a la función: la dirección base del array no nombrado que el compilador ha reservado para la constante de cadena.

Hay algo más que debemos decir acerca de nuestra función `StrLength`. Un programador de C++ no escribiría realmente esta función. La biblioteca estándar proporciona varias funciones de procesamiento de cadena, una de las cuales se llama `strlen` que hace exactamente lo que nuestra función `StrLength`. Más adelante, en este capítulo, revisaremos `strlen` y otras funciones de biblioteca.

Inicialización de cadenas C

En el capítulo 12 demostramos cómo inicializar un array en su declaración mediante la especificación de una lista de valores iniciales entre corchetes, tal como sigue:

```
int delta[5] = {25, -3, 7, 13, 4};
```

Para inicializar una variable de cadena C en su declaración, se podría usar la misma técnica:

```
char message[8] = {'W', 'h', 'o', 'o', 'p', 's', '!', '\0'};
```

Sin embargo, C++ permite una manera más convencional para inicializar una cadena C. Simplemente se puede inicializar el array usando una constante de cadena:

```
char message[8] = "Whoops!";
```

Esta notación es única para cadenas C porque no hay otro tipo de array para el cual existan constantes agregadas.

En el capítulo 12 comentamos que se puede omitir el tamaño de un array cuando se inicializa en su declaración (en este caso, el compilador determina su tamaño). Esta característica se usa frecuentemente con cadenas C porque previene que uno tenga que contar el número de caracteres. Por ejemplo,

```
char promptMsg[] = "Enter a positive number:";  // El tamaño es 25
char errMsg[] = "Value must be positive.";      // El tamaño es 24
```

Tenga mucho cuidado con una cosa: C++ trata la inicialización (en una declaración) y la asignación (en una sentencia de asignación) como dos operaciones distintas. Se aplican reglas diferentes. Recuerde que la inicialización de arrays es permitida, pero la asignación de arrays agregados no.

```
char myStr[20] = "Hello";    // CORRECTO
  ⋮
myStr = "Howdy";             // No permitido
```

Entrada y salida de cadenas C

En el capítulo 12 destacamos que C++ no proporciona operaciones agregadas en arrays. No hay asignación agregada, comparación agregada, ni aritmética agregada en arrays. También dijimos que la entrada/salida agregada de arrays no es posible, con una excepción. Las cadenas C son esta excepción. Primero veamos la salida.

Para emitir los contenidos de un array que *no* es una cadena C, no se permite lo siguiente:

```
int alpha[100];
  ⋮
cout << alpha;   // No permitido
```

En su lugar se debe escribir un ciclo e imprimir los elementos del array uno por uno. Sin embargo, la salida agregada de un array `char` de terminación nula (o sea, una cadena C) es válida. La cadena C puede ser una constante (como lo hemos hecho desde el capítulo 2):

```
cout << "Results are:";
```

o puede ser una variable:

```
char msg[8] = "Welcome";
  ⋮
cout << msg;
```

En ambos casos el operador de inserción `(<<)` emite cada carácter en el array hasta que se encuentre el carácter nulo. Depende de usted volver a verificar que el carácter nulo esté presente en el array. Si no es el caso, el operador `<<` marchará a través del array y el resto de la memoria, imprimiendo bytes hasta que —sólo por coincidencia— encuentre un byte cuyo valor entero sea el equivalente a nulo.

Para introducir cadenas C tenemos varias opciones. La primera es usar el operador de extracción `(>>)`, que se comporta exactamente igual que en el caso de objetos de clase `string`. Cuando lee caracteres de entrada en una variable de cadena C, el operador `>>` salta los caracteres de espacios en blanco y luego lee caracteres sucesivos del array, deteniéndose en el primer carácter de espacio blanco (lo que no se consume, pero permanece como el primer carácter esperando en el flujo de entrada). El operador `>>` también se encarga de agregar el carácter nulo al final de la cadena. Por ejemplo, supongamos que tenemos el siguiente código:

```
char firstName[31];   // Espacio para 30 caracteres más '\0'
char lastName[31];

cin >> firstName >> lastName;
```

Si el flujo de entrada inicialmente se ve así (donde ␣ denota un blanco):

```
␣␣␣John␣Smith␣␣␣25
```

entonces nuestra instrucción guarda 'J', 'o', 'h', 'n' y '\0' en `firstName[0]` hasta `firstName[4]`; guarda 'S', 'm', 'i', 't', 'h' y '\0' en `lastName[0]` hasta `lastName[5]`; y sale del flujo de entrada como

```
␣␣␣25
```

El operador >>, sin embargo, tiene dos desventajas potenciales.

1. Si el array no es suficientemente grande para guardar la secuencia de caracteres de entrada (y el '\0'), el operador >> continuará guardando caracteres en la memoria más allá del final del array.
2. El operador >> no se puede usar para introducir una cadena que tiene blancos. (Detiene la lectura tan pronto encuentra el primer carácter de espacio blanco.)

A fin de hacer frente a estas limitaciones, podemos usar una variación de la función `get`, un miembro de la clase `istream`. Hemos usado la función `get` para introducir un solo carácter, aunque sea un carácter de espacio blanco:

```
cin.get(inputChar);
```

La función `get` también se puede usar para introducir cadenas C. En este caso la llamada de la función requiere dos argumentos. El primero es el nombre del array, y el segundo es una expresión `int`.

```
cin.get(myStr, charCount + 1);
```

La función `get` no brinca a caracteres de espacios blancos y continúa hasta que haya leído `charCount` caracteres o hasta que llegue al carácter de cambio de línea '\n', lo que ocurra primero. Luego añade el carácter nulo al final de la cadena. Mediante las instrucciones

```
char oneLine[81];   // Espacio para 80 caracteres más '\0'
  ⋮
cin.get(oneLine, 81);
```

la función `get` lee y guarda una línea completa de entrada (hasta un máximo de 80 caracteres), con todo y blancos intercalados. Si la línea tiene menos de 80 caracteres, la lectura se detiene en '\n', pero no lo consume. El carácter de cambio de línea es ahora el primero esperando en el flujo de entrada. Para leer dos líneas consecutivas de cadenas, es necesario consumir el carácter de cambio de línea:

```
char dummy;
  ⋮
cin.get(string1, 81);
cin.get(dummy);          // Comer la línea nueva antes del siguiente "get"
cin.get(string2, 81);
```

La primera llamada de función lee caracteres hasta, pero no incluyendo, '\n'. Si se omitiera la introducción de `dummy`, entonces la introducción de `string2` no leería *ningún* carácter porque '\n' sería inmediatamente el primer carácter esperando en el flujo.

Finalmente, la función `ignore` —que hemos introducido en el capítulo 4— puede ser útil en conjunto con la función `get`. Recuerde que la sentencia

```
cin.ignore(200, '\n');
```

dice que se debe brincar un máximo de 200 caracteres de entrada, pero se debe detener si se lee un cambio de línea. (El carácter de cambio de línea sí *es* almacenado por esta función.) Si un programa introduce una cadena larga del usuario, pero sólo quiere retener los primeros cuatro caracteres de la respuesta, ésta es la forma de hacerlo:

```
char response[5];         // Espacio para 4 caracteres más '\0'

cin.get(response, 5);     // Introducir a lo sumo 4 caracteres
cin.ignore(100, '\n');    // Omitir los caracteres restantes hasta e
                          //   incluso '\n'
```

El valor 100 en el último enunciado es arbitrario. Cualquier número que sea "suficientemente grande" servirá.

A continuación se presenta una tabla que resume las diferencias entre el operador >> y la función `get` durante la lectura de cadenas C:

Sentencia	¿Se omite el espacio en blanco principal?	¿Cuándo se detiene la lectura?
`cin >> inputStr;`	Sí	En el primer carácter de espacio en blanco posterior (el cual *no* se consume
`cin.get(inputStr, 21);`	No	Cuando son leídos 20 caracteres o se encuentra '\n' (que *no* es consumida)

Finalmente, volveremos a abordar un tema que se había presentado en el capítulo 4. Algunas funciones de biblioteca y funciones de miembros de clases proporcionados por el sistema requieren cadenas C como argumentos. Un ejemplo es la función de miembros de clase `ifstream` con el nombre `open`. Para abrir un archivo pasamos el nombre del archivo como una cadena C, ya sea una constante o una variable:

```
ifstream file1;
ifstream file2;
char     fileName[51];           // Máx. 50 caracteres más '\0'

file1.open("students.dat");
cin.get(fileName, 51);           // Leer a lo sumo 50 caracteres
cin.ignore(100, '\n');           // Omitir el resto de la línea de entrada
file2.open(fileName);
```

Como se examinó en el capítulo 4, si nuestro nombre de archivo está contenido en un objeto de clase `string`, todavía podemos usar la función `open`, *siempre y cuando* usemos la función de miembro de clase denominada `c_str` para convertir la cadena en una cadena C:

```
ifstream inFile;
string   fileName;

cin >> fileName;
inFile.open(fileName.c_str());
```

Cuando se comparan estos dos segmentos de código, se puede observar una mayor ventaja de la clase `string` sobre las cadenas C: una cadena en un objeto de clase `string` tiene una longitud ilimitada, mientras que la longitud de una cadena C está limitada por el tamaño del array, lo cual es fijado al momento de compilar.

Rutinas de biblioteca de cadenas C

Por medio del archivo de encabezado `cstring`, la biblioteca estándar de C++ proporciona una gran variedad de operaciones de cadenas C. En esta sección analizaremos tres de estas funciones de biblioteca: `strlen`, que devuelve la longitud de una cadena; `strcmp`, que compara dos cadenas usando las relaciones menor o igual a, igual a, y mayor a; y `strcpy`, que copia una cadena a otra. A continuación se muestra un resumen de `strlen`, `strcmp` y `strcpy`:

Archivo de encabezado	Función	Valor de función	Efecto
<cstring>	strlen(str)	Longitud del entero de str (sin incluir '\0')	Calcula la longitud de str
<cstring>	strcmp(str1, str2)	Un entero < 0, si str1 < str2 El entero 0, si str1 = str2 Un entero > 0, si str1 > str2	Compara a str1 y str2
<cstring>	strcpy(toStr, fromStr)	Dirección base de toStr (por lo común ignorada	Copia fromStr (incluso '\0') a toStr, sobrescribiendo lo que estuviera ahí; toStr debe ser suficientemente grande para contener el resultado

La función `strlen` es similar a la función `StrLength` que escribimos anteriormente. Devuelve el número de caracteres en una cadena C antes del terminador '\0'. A continuación se presenta un ejemplo de una llamada a la función:

```
#include <cstring>
  ⋮
char subject[] = "Computer Science";

cout << strlen(subject);   // Imprime 16
```

La rutina `strcpy` es importante porque no se permite la asignación agregada mediante el operador = en cadenas C. En el siguiente fragmento de código mostramos las formas incorrecta y correcta de realizar una copia de cadena.

```
#include <cstring>
  ⋮
char myStr[100];
  ⋮
myStr = "Abracadabra";          // No
strcpy(myStr, "Abracadabra");   // Sí
```

En la lista de argumentos de `strcpy`, la cadena de destino es la del lado izquierdo, justo como una operación de asignación transfiere datos de la derecha a la izquierda. Es responsabilidad del invocador asegurar que el array de destino sea lo suficientemente grande para retener el resultado.

La función `strcpy` es técnicamente una función de regreso de valores; no sólo copia una cadena C a otra, sino también devuelve como un valor de función la dirección base del array de destino. La razón por la que el invocador querrá usar este valor de función no es inmediatamente obvia, y no la vamos a examinar aquí. Los programadores casi siempre ignoran el valor de función y simplemente invocan `strcpy` como si fuera una función nula ("void" como lo hicimos antes). Se recomienda revisar el recuadro de Información básica en el capítulo 8 con el título "Ignorar un valor de función".

La función `strcmp` se usa para comparar dos cadenas. La función revisa dos cadenas C como parámetros y las compara en orden *lexicográfico* (el orden en que aparecerían en un diccionario), o sea el mismo orden que se usa en la comparación de objetos de clase `string`. Dada la llamada de función `strcmp(strl, str2)`, la función devuelve uno de los siguientes valores `int`: un entero negativo, si `strl < str2` lexicográficamente; el valor 0, si `strl = str2`; o un entero positivo, si `strl > str2`. Los valores precisos del entero negativo y del entero positivo son no especificados. Simplemente se realiza la prueba para ver si el resultado es menor que 0, 0 o mayor que 0. A continuación un ejemplo:

```
if (strcmp(str1, str2) < 0)   // Si str1 es menor que str2 ...
    ⋮
```

Hemos descrito sólo tres de las rutinas de manejo de cadenas que proporciona la biblioteca estándar. Estas tres son las que se usan más comúnmente, pero hay muchas más. Si usted está diseñando o manteniendo programas que usan cadenas C extensamente, deberá leer la documentación sobre cadenas para su sistema C++.

¿Clase de cadena o cadenas C?

Cuando se trabaja con datos de cadenas, ¿se debe usar una clase como `string`, o se deben usar cadenas C? Desde los puntos de vista de claridad, versatilidad y facilidad de uso, no hay duda. Use una clase de cadenas. La biblioteca estándar de clase `string` proporciona cadenas de longitud ilimitada, asignación agregada, comparación agregada, concatenación con el operador +, etcétera.

Sin embargo, aun así es útil estar familiarizado con las cadenas C. Entre los miles de productos de software realmente en uso que están escritos en C y C++, la mayoría (aunque es un porcentaje en declive) usa cadenas C para representar datos de cadenas. En su próximo empleo el entendimiento de cadenas C será esencial si le piden modificar o actualizar este tipo de software. Además, el *uso* de una clase de cadenas es una cosa; *aplicarla* es otra. Alguien debe aplicar la clase usando una representación de datos concretos. ¡En su empleo, ese alguien podría ser usted, y la representación de datos subyacentes bien podría ser una cadena C!

Caso práctico de resolución de problemas

Calcular estadísticas de examen (rediseño)

PROBLEMA Usted es el evaluador en su clase de Política. El maestro le ha pedido que prepare las siguientes estadísticas. (*No, ¡no se trata de un error de impresión!* Vamos a resolver el mismo problema de una manera completamente distinta.)

INPUT Un archivo cuyo nombre es introducido desde el teclado y que contenga calificaciones de examen.

OUTPUT Un archivo cuyo nombre es introducido desde el teclado y que muestre las siguientes estadísticas correctamente etiquetadas.

Número de calificaciones
Calificación promedio
Calificación más baja
Calificación más alta
Número de calificaciones arriba del promedio
Número de calificaciones debajo del promedio

ANÁLISIS Acabamos de diseñar e implantar la lista de ADT. Consideremos este problema en términos de una lista de calificaciones. Son las mismas tres tareas separadas que se deben realizar en este problema, pero vamos a verlas en términos de operaciones de lista. Tenemos que crear un promedio de los valores en la lista, buscar los valores máximos y mínimos en la lista, y pasar por la lista contando los valores arriba del promedio y los valores debajo del promedio. Para encontrar el promedio sumamos la lista de calificaciones y la dividimos entre el número de calificaciones. Hemos encontrado los valores máximo y mínimo en una lista de números en el programa del Estudio de diseño. La tercera tarea significa mirar cada calificación, comparándola con el promedio, e incrementando uno de dos contadores.

Si todo lo que teníamos que hacer era encontrar el promedio y las evaluaciones mínima y máxima, podríamos hacer el procesamiento y la lectura de los valores de datos al mismo tiempo. Sin embargo, la tarea de imprimir el número de evaluaciones arriba y abajo del promedio requiere que cada evaluación sea examinada más de una vez, porque no tenemos el promedio sino hasta que todos los valores se hayan leído. Por tanto, tenemos que crear una lista de las evaluaciones de modo que podamos acceder a cada una más de una vez.

Tenemos que examinar cada evaluación dos veces: una para calcular el promedio, encontrar el valor mínimo y encontrar el valor máximo, y otra para comparar cada evaluación con el promedio. Podríamos realizar todo el

procesamiento, excepto el número arriba y abajo del promedio, cuando leemos las evaluaciones inicialmente. Sin embargo, un mejor estilo es separar las tareas en diferentes funciones.

ESTRUCTURAS DE DATOS

Variables sencillas
Una lista de valores enteros

Principal **Nivel 0**

```
Abrir archivos
Introducir calificaciones
Calcular el promedio
Calcular el valor máximo
Calcular el valor mínimo
Calcular arriba del promedio
Calcular abajo del promedio
Cerrar archivos
```

Usamos la misma función de abrir archivo que hemos usado en el último programa. Sin embargo, es necesario cambiar el encabezado impreso en la salida.

Introducir calificaciones **Nivel 1**

Esta función debe tener el nombre del archivo y la lista. El programa necesita saber el número de evaluaciones, pero a diferencia de la aplicación anterior, la lista tiene una función que devuelve el número de elementos en la lista.

(Entrada-salida: inData, grades)

```
Leer la calificación de inData
WHILE inData && !grades.IsFull()
    grades.Insert(grade);
    Leer grade de inData
```

Calcular el promedio(Entrada: grades, numGrades)
Salida: valor de función

```
Establecer la suma en 0
Establecer el límite en grades.Length();
grades.Reset();
FOR índice que va de 0 al límite
    Establecer grade en grades.GetNextItem();
    Establecer la suma en sum + grade;
Devolver float(sum)/float(limit);
```

Calcular el valor más alto(entrada: grades)
Salida: valor de función

```
Establecer el límite en grades.Length();
grades.Reset()
Establecer maxGrade en 0;
FOR índice que va de 0 al límite
    Establecer grade en grades.GetNextItem();
    IF grade > maxGrade
        Establecer maxGrade en grade;
Devolver maxGrade;
```

Calcular el valor mínimo(Entrada: grades)
Salida: valor de función

```
Establecer el límite en grades.Length()
grades.Reset()
Establecer minGrade en 100
FOR índice que va de 0 al límite
    Establecer grade en grades.GetNextItem()
    IF grade < minGrade
        Establecer minGrade en grade;
Devolver minGrade;
```

Calcular arriba del promedio(Entrada: grades, average)
Salida: valor de función

```
Establecer roundedAverage en (int)(promedio+0.5)
Establecer el límite en grades.Length()
Establecer el número en 0
grades.Reset()
FOR índice que va de 0 al límite
    Establecer grade en grades.GetNextItem()
    IF grade > roundedAverage
        Incrementar el número
Devolver el número
```

Calcular abajo del promedio(Entrada: grades, average)
Salida: valor de función

```
Establecer truncatedAverage en (int)(average)
Establecer el límite en grades.Length()
Establecer el número en 0
grades.Reset()
FOR índice que va de 0 al límite
    Establecer grade en grades.GetNextItem()
    IF grade < truncatedAverage
        Incrementar el número
Devolver el número
```

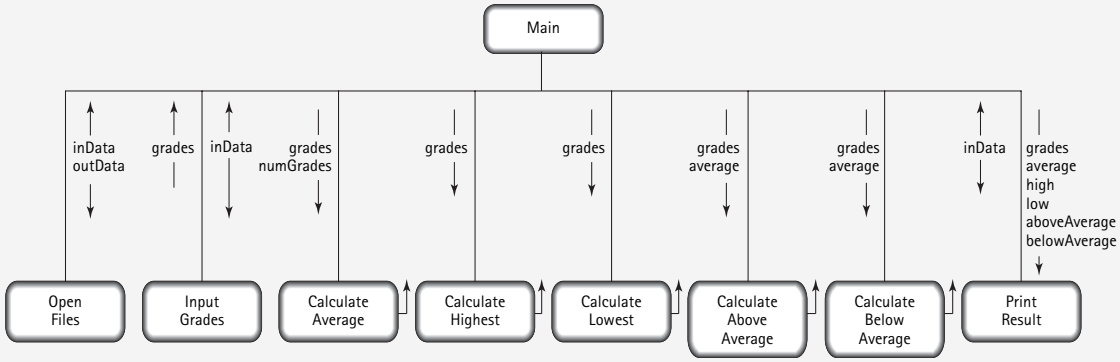
Imprimir resultados(Entrada-salida: outData; entrada: grades, average, highest, lowest, numberAbove, numberBelow)

Imprimir en outData "El número de calificaciones es " grades.`Length`()
Imprimir en outData "La calificación promedio es " average
Imprimir en outData "La calificación máxima es " highest
Imprimir en outData "La calificación mínima es " lowest
Imprimir en outData "El número de calificaciones arriba del promedio es " aboveAverage
Imprimir en outData "El número de calificaciones abajo del promedio es " belowAverage

El módulo principal es idéntico al módulo principal en la otra aplicación. Los cambios están escondidos dentro de las funciones que manipulan la estructura que retiene los datos. La primera aplicación usaba un array de contadores, una ranura para cada resultado de examen posible. El procesamiento involucraba la manipulación del array. La segunda aplicación usaba una lista de calificaciones. El procesamiento involucraba la manipulación de los valores en el ADT de lista.

Incluso los diagramas de estructura de módulo están casi idénticos. Sólo la lista de parámetros de las funciones `InputGrades` y `PrintResults` necesitan una alteración para quitar `numGrades`.

DIAGRAMA DE ESTRUCTURA DE MÓDULOS




```
//**************************************************************************
// Programa Estadísticas
// Este programa calcula el promedio, la puntuación alta, puntuación baja,
// el número arriba del promedio y el número abajo del promedio para
// un archivo de puntuaciones de prueba. Se usa el TDA de lista
// Suposición: el archivo 'testScores' no está vacío y no contiene más
// que valores MAX_GRADES.
// Para ahorrar espacio, se omiten de cada función los comentarios
// de precondición que documentan las suposiciones hechas acerca de
// los datos de parámetros de entrada. Éstos se incluirían en un programa
// propio para uso real.
//**************************************************************************

#include <fstream>
#include <iostream>
#include <iomanip>
#include "list.h"

using namespace std;
```

```
// Prototipos de función

void OpenFiles(ifstream& inData, ofstream& outData);
void InputGrades(List& grades, ifstream& inData);
float CalculateAverage(List grades);
int CalculateHighest(List grades);
int CalculateLowest(List grades);
int CalculateAboveAverage(List grades, float average);
int CalculateBelowAverage(List grades, float average);
void PrintResults(ofstream& outData, List grades, float average,
    int highest, int lowest, int aboveAverage, int belowAverage);

int main()
{
    List grades;           // Una lista de calificaciones

    float average;         // Calificación promedio
    int highest;           // Calificación más alta
    int lowest;            // Calificación más baja
    int aboveAverage;      // Número de calificaciones arriba del promedio
    int belowAverage;      // Número de calificaciones abajo del promedio

    // Declarar y abrir archivos
    ifstream inData;
    ofstream outData;
    OpenFiles(inData, outData);

    // Calificaciones de proceso
    InputGrades(grades, inData);
    average = CalculateAverage(grades);
    highest = CalculateHighest(grades);
    lowest = CalculateLowest(grades);
    aboveAverage = CalculateAboveAverage(grades, average);
    belowAverage = CalculateBelowAverage(grades, average);
    PrintResults(outData,grades, average, highest, lowest,
        aboveAverage, belowAverage);

    inData.close();
    outData.close();
    return 0;
}

//****************************************************************

void OpenFiles( /* inout */ ifstream& text,
                /* inout */ ofstream& outFile )

// La función OpenFiles lee los nombres del archivo de entrada
// y el archivo de salida y los abre para procesamiento
// Poscondición:
//     Los archivos han sido abiertos Y se ha escrito una etiqueta
//     en el archivo de salida
if ( !inData || !outData )
{
```

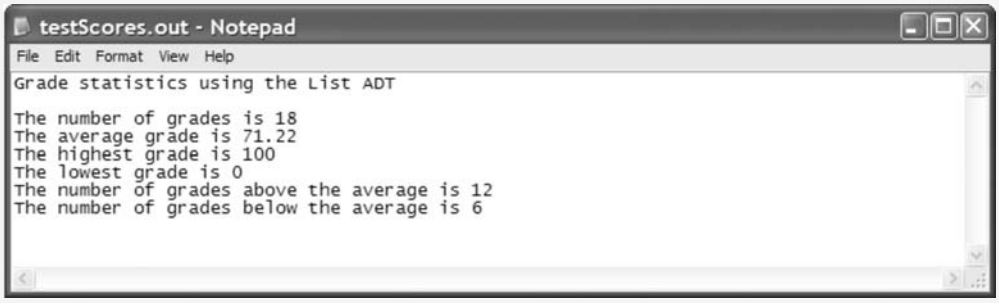
```
    cout << "Los archivos no se abrieron con éxito," << endl;
    return 1;
}

    string inFileName;
    string outDataName;

    cout << "Introduzca el nombre del archivo que será procesado" << endl;
    cin >> inFileName;
    text.open(inFileName.c_str());

    cout << "Introducir el nombre del archivo de salida" << endl;
    cin >> outDataName;
    outFile.open(outDataName.c_str());

    //  Escribir la etiqueta en la salida
    outFile << "Estadísticas de calificación con el TDA de lista" << endl
            << endl;
}

//*************************************************************************

void InputGrades( /* inout */ List& grades,      // Lista de calificaciones
                  /* inout */ ifstream& inData ) // Archivo de entrada

// Las calificaciones se introducen desde el archivo inData y se insertan en
   las calificaciones.
// Precondición:
//     El archivo no está vacío
// Poscondición:
//     Cada calificación del archivo se ha insertado en la lista de
       calificaciones

{
    int grade;
    // Leer las calificaciones e introducirlas en la lista
    inData >> grade;
    while (inData && !grades.IsFull())
    {
        grades.Insert(grade);
        inData >> grade;
    }
}

//*************************************************************************

float CalculateAverage( /* in */ List grades )

// Esta función calcula la puntuación de prueba promedio
// Poscondición:
//     El valor de retorno es la calificación promedio
{
    int sum = 0;
```

```
    int limit = grades.Length();        // el límite es el número de
                                        // calificaciones
    int grade;

    grades.Reset();                     // Preparar para paso

    // Añadir cada calificación a la suma
    for (int index = 0; index < limit; index++)
    {
        grade = grades.GetNextItem();
        sum = sum + grade;
    }

    return float(sum) / float(limit);   // Devolver el promedio
}

//*************************************************************************

int CalculateHighest( /* in */ List grades ) // Lista de calificaciones

// Esta función calcula la calificación más alta en la lista de
// calificaciones
// Poscondición:
//     El valor devuelto es la calificación máxima

{
    int limit = grades.Length();        // Número de calificaciones
    int grade;

    grades.Reset();                     // Preparar para iteración
    int maxGrade = 0;

    // Encontrar la calificación máxima en la lista
    for (int index = 0; index < limit; index++)
    {
        grade = grades.GetNextItem();
        if (grade > maxGrade)
            maxGrade = grade;
    }
    return maxGrade;
}

//*************************************************************************

int CalculateLowest( /* in */ List grades ) // Lista de calificaciones

// Esta función calcula la calificación mínima en la lista de
// calificaciones
// Poscondición:
//     El valor devuelto es la calificación mínima

{
    int limit = grades.Length();       // Número de calificaciones
    int grade;
```

```
    grades.Reset();                    // Preparar para iteración
    int minGrade = 100;

    // Hallar la calificación mínima de la lista
    for (int index = 0; index < limit; index++)
    {
        grade = grades.GetNextItem();
        if (grade < minGrade)
            minGrade = grade;
    }
    return minGrade;
}

//*************************************************************************

int CalculateAboveAverage
    ( /* in */ List grades,         // Lista de calificaciones
      /* inout */  float average )  // Calificación promedio

// Esta función calcula el número de calificaciones arriba del promedio
// Poscondición:
//     El valor devuelto es el número de calificaciones arriba del promedio

{
    int roundedAverage = (int) (average + 0.5);
    int limit = grades.Length();       // Número de calificaciones
    int grade;
    int number = 0;

    grades.Reset();                    // Preparar para iteración

    // Calcular el número de calificaciones arriba del promedio
    for (int index = 0; index < limit; index++)
    {
        grade = grades.GetNextItem();
        if (grade > roundedAverage)
            number++;
    }
    return number;
}

//*************************************************************************

int CalculateBelowAverage
    ( /* in */ List grades,          // Lista de calificaciones
      /* inout */  float average )   // Calificación promedio

// Esta función calcula el número de calificaciones abajo del promedio
// Poscondición:
//     El valor devuelto es el número de calificaciones abajo del promedio

{
    int truncatedAverage = (int) (average);
    int limit = grades.Length();       // Número de calificaciones
```

```
    int grade;
    int number = 0;

    grades.Reset();                  // Preparar para iteración

    // Calcular el número de calificaciones abajo del promedio
    for (int index = 0; index < limit; index++)
    {
        grade = grades.GetNextItem();
        if (grade < truncatedAverage)
            number++;
    }
    return number;
}

//*****************************************************************************

void PrintResults( /* inout */ ofstream& outData, // Archivo de salida
                   /* in */    List grades,       // Lista de calificaciones
                   /* in */    float average,     // Promedio
                   /* in */    int highest,       // Calificación máxima
                   /* in */    int lowest,        // Calificación mínima
                   /* in */    int aboveAverage,  // Número arriba
                   /* in */    int belowAverage ) // Número abajo

// Las estadísticas se imprimen en el archivo outData
// Precondición:
//     El archivo de salida ha sido abierto con éxito
// Poscondición:
//     Las estadísticas han sido escritas en outData, marcadas de manera
//     apropiada

{
    outData << "El número de calificaciones es " << grades.Length()
            << endl;
    outData << fixed << setprecision(2) << "La calificación promedio es "
            << average << endl;
    outData << "La calificación más alta es " << highest << endl;
    outData << "La calificación más baja es " << lowest << endl;
    outData << "El número de calificaciones arriba del promedio es "
            << aboveAverage  << endl;
    outData << "El número de calificaciones abajo del promedio es "
            << belowAverage << endl;
}
```

PRUEBA Esta aplicación está basada en una lista de valores. Tenemos que realizar pruebas para verificar si existen casos donde no hay calificaciones, una calificación, unas cuantas calificaciones, y exactamente el máximo número de calificaciones. Hemos tratado ya el caso donde hay más evaluaciones que almacenar que ranuras en la lista, revisando `IsFull` en la función `InputGrades`. A continuación se ilustra una pasada con unas cuantas calificaciones, las mismas que hemos usado con la última aplicación. Es reconfortante ver que las respuestas son las mismas.



¿Cuál de los algoritmos es más eficiente? No podemos contestar a esta pregunta porque la respuesta depende del intervalo de posibilidades y del tamaño de los conjuntos de datos. La cantidad de trabajo realizado en la segunda solución está basada sobre el número de calificaciones. Los ciclos van de 0 hasta el número de calificaciones, así que la elección del algoritmo depende del contexto en el cual se usará.

Prueba y depuración

En este capítulo hemos analizado, diseñado y codificado algoritmos para construir y manipular elementos en una lista. En adición a las operaciones básicas de lista `IsFull`, `IsEmpty`, `Length` y un par de iteradores, los algoritmos incluyeron tres búsquedas secuenciales, una búsqueda binaria, la inserción en listas ordenadas y no ordenadas, la eliminación en listas ordenadas y no ordenadas, y un ordenamiento de selección. Ya hemos verificado parcialmente `List::Insert`, `List::Delete`, `List::Reset` y `List::GetNextItem` en conjunto con el estudio de caso. Los demás algoritmos de búsqueda y las funciones `SortedList::Insert`, `SortedList::Delete` y `SortedList:SelSort` se tienen que verificar. Los deberíamos verificar con listas que contienen ningún componente, un componente, dos componentes, componentes de `MAX_LENGTH - 1` y componentes de `MAX_LENGTH`.

Cuando escribimos la precondición de que la lista no estaba llena para la operación `List::Insert`, indicamos que podríamos manejar el problema de otro modo; podríamos incluir una bandera de error en la lista de parámetros de la función. La función llamaría a `IsFull` y colocaría la bandera de error. La inserción no sería necesaria si la bandera de error estuviera fijada en `true`. Ambas opciones son formas aceptables de manejar el problema. Lo importante es que indiquemos claramente si el código de llamada o la función llamada son para verificar si hay una condición de error. Sin embargo, es el código de llamada el que debe decidir qué hacer cuando ocurre una condición de error. En otras palabras, si los errores son manejados por medio de precondiciones, entonces el usuario debe escribir el código para garantizar las precondiciones. Si los errores son manejados por banderas, entonces el usuario debe escribir el código para monitorear los banderas de errores.

Consejos para prueba y depuración

1. Revise los consejos para prueba y depuración para el capítulo 12. Ellos se aplican a todos los arrays unidimensionales, incluyendo las cadenas C.
2. Asegúrese de que cada cadena C termina con el carácter nulo. Las constantes de cadenas son automáticamente terminadas a nulo por el compilador. En la entrada, el operador >> y la función `get` automáticamente agregan el carácter nulo. Si se almacenan caracteres en una cadena C individualmente, o si se manipula el array de cualquier manera, asegúrese de que se dé cuenta del carácter nulo.
3. Recuerde que C++ trata la inicialización de cadenas C (en una declaración) de manera distinta de la asignación de cadenas C. La inicialización es permitida, la asignación no.
4. Operaciones input/output agregadas son permitidas para cadenas C, pero no para otros tipos de arrays.

5. Si usted usa el operador >> para introducir una variable de cadena C, asegúrese de que el array es suficientemente grande para retener el carácter nulo más la secuencia más larga de caracteres (de no espacio blanco) en el flujo de entrada.
6. En el caso de introducción de una cadena C, el operador >> se detiene, pero no almacena, en el primer carácter de espacio blanco. Asimismo, si la función `get` deja de leer anticipadamente porque encuentra un carácter de cambio de línea, el carácter de cambio de línea no es almacenado.
7. Cuando usa la función de biblioteca `strcpy`, asegúrese de que el array de destino es por lo menos tan grande como el array del que está copiando.
8. Las funciones de uso múltiple (como operaciones ADT) se deberán verificar fuera del contexto de un programa particular, usando un programa director de pruebas.
9. Elija los datos de prueba cuidadosamente, de modo que se verifiquen todas las condiciones finales y algunas en medio. Las condiciones finales son las que llegan a los límites de la estructura que se usa para almacenarlas. Por ejemplo, en una lista deberán existir datos de prueba en los cuales el número de componentes es 0, 1 y `MAX_LENGTH`, así como entre 1 y `MAX_LENGTH`.

Resumen

Este capítulo ha proporcionado algunas prácticas en el trabajo con arrays unidimensionales. Hemos examinado algoritmos que insertan, eliminan, buscan y ordenan datos guardados en una lista, y hemos escrito funciones para aplicar estos algoritmos. Podemos volver a usar estas funciones una y otra vez en diferentes contextos, porque ellos son miembros de clases de C++ de uso múltiple (`List` y `SortedList`) que representan ADT de lista.

Las cadenas C son una clase especial de arrays `char` en C++. El último carácter significativo debe ser seguido por un carácter nulo para marcar el final de la cadena. Las cadenas C son menos versátiles que una clase de cadenas. Sin embargo, vale la pena saber cómo funcionan porque muchos programas existentes en C y C++ las usan, y las clases de cadenas frecuentemente usan cadenas C como la subyacente representación de datos.

Comprobación rápida

1. ¿Cuáles son las tres propiedades principales de una lista? (pp. 546-552)
2. ¿Dónde insertamos un nuevo valor en una lista que no está ordenada? (pp. 552-556)
3. Si un elemento no está en la lista, ¿cuándo descubre la búsqueda lineal que está faltando? (pp. 556-558)
4. ¿Cómo se distingue un ADT de lista ordenada de un ADT de lista que incluye una operación de ordenamiento? (pp. 562-567)
5. En el caso de un ordenamiento de inserción, ¿qué sucede con los componentes que preceden el punto de inserción, y con los componentes que le siguen? (pp. 565-567)
6. ¿De dónde toma su nombre el algoritmo de búsqueda binaria? (pp. 568-572)
7. ¿Cuáles son los dos miembros de una clase de lista que se usan para representar los datos en una lista? (pp. 546-547)
8. ¿Cuáles son los pasos principales en la eliminación de un valor de una lista? (p. 573)
9. ¿Qué operaciones en una lista ordenada son diferentes de las operaciones en una lista no ordenada, y cómo se distinguen? (pp. 562-564)
10. ¿Cuántos ciclos For existen en una aplicación de un ordenamiento de selección? (pp. 560-562)
11. ¿Qué condición termina una búsqueda binaria? (pp. 567-568)

Respuestas

1. Longitud variable, lineal y homogénea. **2.** Al final de la lista (suponiendo que aún no está llena). **3.** Cuando ha examinado el último elemento de la lista. **4.** Los componentes de una lista ordenada siempre se mantienen en orden, pero los componentes de una lista ordinaria sólo se ordenan después de la ejecución de la operación de ordenamiento, hasta la siguiente inserción o eliminación. **5.** Los componentes que preceden el punto de

inserción permanecen donde están, y los que siguen al punto de inserción se desplazan un lugar para crear espacio. **6.** El nombre viene de su patrón de dividir el área de la búsqueda entre dos en cada iteración. **7.** Un array de datos que contiene el elemento especificado en cada componente, así como una longitud representada por un `int`. **8.** Buscar el valor, borrar el valor, mover todos los valores subsiguientes un lugar hacia arriba. **9.** No hay ninguna operación de ordenamiento en la lista ordenada. La operación de inserción inserta un nuevo valor en su propio lugar en lugar de hacerlo al final de la lista. La operación de borrar mueve los componentes subsiguientes un lugar hacia arriba, en lugar de mover el último componente a la posición vaciada. La búsqueda en la lista ordenada se puede realizar mediante una búsqueda binaria en lugar de hacerlo por medio de una búsqueda lineal. **10.** Dos. **11.** O se encuentra el valor, o el primero y el último índice del área de búsqueda son lo mismo, y el valor no se encuentra.

Ejercicios de preparación para examen

1. ¿Por qué decimos que una lista es lineal?
2. ¿A qué nos referimos cuando decimos que una lista es homogénea?
3. Si la operación `Length` asociada con una lista *no ordenada* devuelve 43, y si luego llamamos la operación de borrar para la lista, pasándole un valor que iguala el elemento número 21 de la lista:
   a) ¿Cuál es el índice del componente que se borra?
   b) ¿Cuál es el índice del componente que toma su lugar?
   c) ¿Qué es lo que devuelve la operación `Length` después de la eliminación?
   d) ¿Cuántos componentes en la lista cambian su posición como resultado de la eliminación?
4. Si `ItemType` es `float`, ¿cómo necesitamos cambiar la siguiente condición de ciclo While, como queda escrito en la búsqueda lineal, y por qué tenemos que cambiarla?

```
while (index < length && item != data[index])
```

5. Las siguientes instrucciones supuestamente intercambian dos valores en un array, pero están en el orden equivocado. Reacomódelas en el orden correcto.

```
data[value2] = temp;
data[value1] = data[value2];
temp = data[value1];
```

6. En un ordenamiento de selección, ¿qué se logra con el ciclo interior cada vez que se ejecuta?
7. Si la operación `Length` asociada con una lista ordenada de elementos devuelve 43, y si luego llamamos la operación de eliminación para la lista, pasándole un valor que equipara al elemento 21 en la lista:
   a) ¿Cuál es el índice del componente que se borra?
   b) ¿Cuál es el índice del componente que toma su lugar?
   c) ¿Qué es lo que devuelve la operación `Length` después de la eliminación?
   d) ¿Cuántos componentes en la lista cambian su posición como resultado de la eliminación?
8. En promedio, una búsqueda secuencial de una lista ordenada requiere el mismo número de iteraciones que la búsqueda de una lista no ordenada. ¿Correcto o falso?
9. Deberemos usar una búsqueda binaria para listas ordenadas grandes, pero una búsqueda secuencial es más eficiente cuando una lista tiene pocos componentes. ¿Correcto o falso?
10. ¿Cuál es el logaritmo (base 2) de 32?
11. ¿Por qué no necesitamos una operación de ordenamiento separado en un ADT de lista ordenada?
12. Una lista ordenada contiene 16 elementos, y se llama la operación de búsqueda binaria con un valor que equipara el valor 12 en la lista. ¿Cuántas iteraciones se requieren para que la búsqueda binaria encuentre este valor?
13. ¿Cuáles son los caracteres guardados en la siguiente cadena C?

```
char exam[4] = "Hop";
```

Ejercicios de calentamiento para programación

1. Escriba una función booleana de C++ denominada `Delete` que tenga tres parámetros: `someItem` (del tipo `ItemType`, como se usa en el presente capítulo), `oldList` y `newList` (ambos del tipo `List`, como se define en este capítulo). La función devuelve `true` si `someItem` se encuentra en `oldList`, pero no está presente en `newList`.
2. El tipo `List` en este capítulo nos permite guardar múltiples copias de un elemento en la lista. En ocasiones es útil tener una lista donde cualquier elemento dado puede aparecer sólo una vez. Cambie la aplicación de la función `Insert` de modo que sólo se agregue un elemento a la lista si el elemento ya no está allí.
3. ¿Cuál es el problema en el siguiente segmento de código?, y ¿qué es lo que se tiene que cambiar en el tipo `List` para que funcione correctamente?

```
List inVals;
for (int count = 1; count <= 100; count++)
{
    cin >> inVal;
    inVals.Insert(inVal);
}
```

4. Nos gustaría agregar una función `DeleteAll` al tipo de `List` que borre todas las ocurrencias de un elemento de una lista. Escriba el código que se deberá agregar al archivo de especificación del tipo para que podamos agregar esta función.
5. Escriba la aplicación de la función `DeleteAll` como se describe en el ejercicio 4.
6. Nos gustaría agregar una función `Replace` al tipo de `List` que requiere dos parámetros, `oldItem` y `newItem` del tipo `ItemType`. La función encuentra `oldItem` en la lista, lo borra e inserta `newItem` en su lugar. La lista no sufre cambios si `oldItem` no está presente en ella. Escriba el código que se deberá agregar al archivo de especificación para que podamos agregar esta función.
7. Escriba la aplicación de la función `Replace` como se describe en el ejercicio 6.
8. El tipo `SortedList` mantiene elementos en orden ascendente. ¿Qué función (o funciones) será necesario cambiar para que la lista mantenga los elementos en orden descendente?
9. Cambie la aplicación de la función `BinSearch` de manera que funcionará con una lista que mantiene los elementos en orden descendente.
10. En el ejercicio 5 usted escribió una función `DeleteAll` para el tipo `List`. Aplique una función `DeleteAll` para el tipo `SortedList`, tomando ventaja de que todas las ocurrencias del elemento están guardadas en ubicaciones adjuntas en el array. Observe que la búsqueda binaria no necesariamente devolverá la posición de la primera ocurrencia de un elemento; puede devolver la posición de cualquier elemento de comparación. Resulta que en este caso podrá ser más eficiente usar una búsqueda lineal para encontrar el inicio de los elementos de comparación. Asegúrese de que su función actualice la longitud de la lista conforme sea necesario.
11. En el ejercicio 7 usted escribió una función `Replace` para el tipo `List` que borra un valor y lo sustituye por otro. Reimplemente la función `Replace` para el tipo `SortedList`. Observe que en este caso el valor de remplazo se deberá insertar en la posición apropiada en la lista a fin de mantener el orden de los elementos.
12. Escriba un segmento de código que llene `SortedList` invocada en `inData` con valores introducidos desde un archivo llamado `unsorted`.
13. Deseamos agregar la función `FilePrint` a `SortedList`. Tal como la operación `Print`, emite el contenido de la lista en orden, pero `FilePrint` requiere un parámetro llamado `outFile` del tipo `ofstream`, y escribe los valores a este archivo. Escriba la aplicación de esta función en el tipo `SortedList`.

Problemas de programación

1. Escriba un programa usando la clase `List` de este capítulo que aplique una lista de cosas que hacer. Los elementos de la lista serán cadenas. Al usuario se le deberá indicar que introduzca un comando (agregue un elemento, marque un elemento como hecho o parcialmente hecho, borre un elemento, e imprima la lista) y los datos necesarios. Almacenar simplemente elementos en la lista es fácil, pero la clase List no admite directamente el registro del estatus de cada tarea. Hay diferentes formas de realizar esto. Una sería aplicar una estructura o una clase que representa un elemento y su estatus, y modificar la clase `List` para que funcione con esa estructura o clase como su tipo de elemento. Otra forma sería mantener tres listas: hecho, parcialmente hecho y abierto. Cuando se crea un elemento, entra en la lista "abierto". Cuando cambia su estatus, se mueve a una de las otras listas, conforme sea apropiado. Elija el planteamiento que usted prefiera, y aplíquelo usando el estilo correcto, indicaciones efectivas y documentación suficiente.
2. A muchos maestros les gusta ver la distribución de resultados de un examen antes de asignar calificaciones. Usted trabaja para un maestro de historia que le ha pedido desarrollar un programa que leerá todos los resultados de un examen e imprimirá un diagrama de barras que muestra la distribución. El intervalo de los resultados varía de un examen a otro, y hay un máximo de 250 estudiantes en la clase. Use o modifique la clase `SortedList` de este capítulo según sea necesario para ayudarle en esta tarea. El intervalo de los resultados se introducen en un archivo llamado `exams.dat` en orden aleatorio. La tarea de su programa es capturar los datos, ordenarlos y emitir un diagrama de barras con un * para cada examen que tenga un resultado particular. La primera barra en el diagrama debe ser el resultado más alto, y la última barra del diagrama debe ser el resultado más bajo. Cada línea de salida deberá empezar con el valor del resultado, seguido por el número apropiado de asteriscos. Cuando haya un valor de resultado que apareció en ningún examen, simplemente emita el valor sin asteriscos, y luego siga a la siguiente línea.
3. Mejore el programa en el problema 2 del siguiente modo: el archivo de datos ahora contiene un resultado y un nombre. Modifique la clase `SortedList` de modo que use una estructura que consista en el resultado y el nombre como sus campos. El programa deberá introducir los datos del archivo en la lista modificada. En adición a visualizar el diagrama de barras, el programa también emitirá la lista ordenada a un archivo llamado `byscore.dat`.
4. Usted ha juntado listas de direcciones de correo electrónico de una variedad de fuentes, y quiere enviar un correo masivo a todas estas direcciones. Sin embargo, no desea enviar mensajes duplicados. Todas las direcciones de correo electrónico (representadas como cadenas) han sido combinadas en un solo archivo llamado `rawlist.dat`. Usted necesita escribir un programa que lea todas las direcciones y que descarte las que ya fueron introducidas. Use una de las clases lista de este capítulo, modificándola según sea necesario para funcionar con datos de cadenas, y para poder manejar hasta 1 000 elementos. Después de que se hayan leído todos los datos, emita la nueva lista de correo a un archivo llamado `cleanList.dat`.
5. Usted está trabajando para el registro estatal de vehículos, y apenas se descubrió que la gente que produce las placas ha hecho duplicados ocasionales erróneamente. Usted tiene un archivo (`platesmade.dat`) que contiene una lista de números de placas que son registradas durante su producción. Usted necesita escribir un programa que lea este archivo e identifique los duplicados en la lista, a fin de que se puedan enviar notificaciones para reclamarlas. Los números de placas, que consisten en letras y números, se deberán guardar como cadenas. Emita los duplicados a un archivo llamado `recallplates.dat`. Use la clase `SortedList` de este capítulo para ayudarle en la escritura de esta aplicación, modificándola donde sea necesario.

Seguimiento de caso práctico

1. En su opinión, ¿cuál de las dos aplicaciones del programa de estadística (capítulos 12 o 13) es más clara?

2. Revise la solución de lista para combinar `CalculateAverage`, `CalculateHighest` y `CalculateLowest` en una sola función `Calculate`.
3. Revise la solución de lista del ejercicio 2 de modo que `CalculateAboveAverage` y `CalculateBelowAverage` sean combinados en una sola función `AboveBelow`.
4. ¿Es mejor tener las funciones combinadas o separadas?

CAPÍTULO

14

Desarrollo de software orientado a objetos

Objetivos

Objetivos de conocimiento

- *Conocer la distinción entre la programación estructurada (procedural) y la programación orientada a objetos.*
- *Conocer las características de un lenguaje orientado a objetos.*
- *Entender la diferencia entre la ligadura estática y la dinámica de operaciones a objetos.*

Objetivos de habilidades

Ser capaz de:

- *Crear una nueva clase de C++ a partir de una clase existente, usando la herencia.*
- *Crear una nueva clase de C++ a partir de una clase existente, usando la composición.*
- *Aplicar la metodología del diseño orientado a objetos para resolver un problema.*
- *Tomar un diseño orientado a objetos y codificarlo en C++.*

En el capítulo 11 hemos introducido el concepto de la abstracción de datos, o sea, separar de las propiedades lógicas de un tipo de datos los detalles de su forma de aplicación. Hemos ampliado este concepto mediante la definición de la idea de un tipo de datos abstractos (ADT) y mediante el uso del mecanismo de clases de C++ para incorporar tanto los datos como las operaciones en un solo tipo de datos. Tanto en el capítulo 11 como en el 13 hemos visto cómo un objeto de una determinada clase mantiene sus propios datos privados y es manipulado por medio de la llamada de sus funciones públicas miembro.

En este capítulo examinaremos cómo clases y objetos de determinada clase se pueden usar para guiar el proceso completo de desarrollo de software. Aunque la fase de diseño precede a la fase de aplicación en el desarrollo de software, invertimos el orden de presentación en este capítulo. Empezamos con la *programación orientada a objetos*, un tema que incluye el diseño pero aborda más bien los asuntos de aplicación. Describiremos los principios básicos, la terminología y propiedades del lenguaje de programación asociadas con el planteamiento orientado a objetos. Después de presentar estos conceptos fundamentales, veremos más detalladamente la fase de diseño, o sea el *diseño orientado a objetos*.

14.1 La programación orientada a objetos

Hasta ahora hemos usado la descomposición funcional (también llamada *diseño estructurado*), en la cual descomponemos un problema en módulos, donde cada módulo es una colección independiente de pasos que resuelve una parte del problema general. El proceso de aplicar una descomposición funcional se denomina **programación estructurada** (o **procedural**). Algunos módulos se traducen directamente a unas cuantas instrucciones del lenguaje de programación, mientras que otros son codificados como funciones con o sin argumentos. El resultado final es un programa que es una colección de funciones de interacción (véase la figura 14-1). Por medio del diseño estructurado y la programación estructurada, los datos son considerados como una cantidad pasiva sobre la cual actúan estructuras y funciones.

Programación estructurada (procedural) Construcción de programas que son colecciones de funciones o procedimientos en interacción.

El diseño estructurado es satisfactorio para la programación en lo pequeño (un concepto que hemos examinado en el capítulo 4), pero con frecuencia no se deja "agrandar" lo suficiente para programar a lo grande. En la elaboración de grandes sistemas de software, el diseño estructurado tiene dos limitaciones importantes. En primer lugar, la técnica produce una estructura inflexible. Si el algoritmo de nivel superior requiere una modificación, los cambios podrán obligar también a una

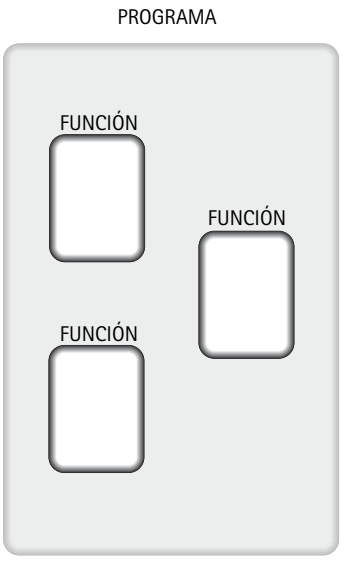


Figura 14-1 *Un programa resultante de la programación estructurada (procedural)*

modificación de muchos algoritmos de nivel inferior. En segundo lugar, la técnica no se presta fácilmente para la reutilización de código. Con el término *reutilización de código* nos referimos a la capacidad de usar partes de código —como son, o levemente modificadas— en otras secciones del programa o en otros programas. Es raro que se pueda tomar una función complicada de C++ y reutilizarla fácilmente en un contexto diferente.

Una metodología que a menudo funciona mejor para crear grandes sistemas de software es el diseño orientado a objetos (OOD), que hemos introducido brevemente en el capítulo 4. OOD descompone un problema en objetos, o sea entidades independientes de datos y operaciones con los datos. El proceso de aplicar un diseño orientado a objetos se denomina **programación orientada a objetos (OOP)**. El resultado final es un programa que es una colección de objetos en interacción (véase la figura 14-2). En OOD y OOP, los datos desempeñan un papel fundamental; la contribución principal de los algoritmos es la de aplicar las operaciones en los objetos. En este capítulo veremos por qué OOD tiende a resultar en programas que son más flexibles y favorables para la reutilización de código que los programas producidos por medio del diseño estructurado.

Programación orientada a objetos (OOP) El uso de la abstracción de datos, de la herencia y de la ligadura dinámica para elaborar programas que son colecciones de objetos en interacción.

Varios lenguajes de programación fueron creados específicamente para admitir OOD y OOP: C++, Java, Smalltalk, Simula, CLOS, Objective-C, Eiffel, Actor, Object-Pascal, versiones recientes de Turbo Pascal, y otros más. Estos lenguajes, llamados lenguajes de *programación orientada a objetos*, cuentan con los medios para

1. Abstracción de datos
2. Herencia
3. Ligadura dinámica

Ya hemos visto que C++ admite la abstracción de datos por medio del mecanismo de clases. Algunos lenguajes que no son OOP también cuentan con los medios para la abstracción de datos. Pero sólo

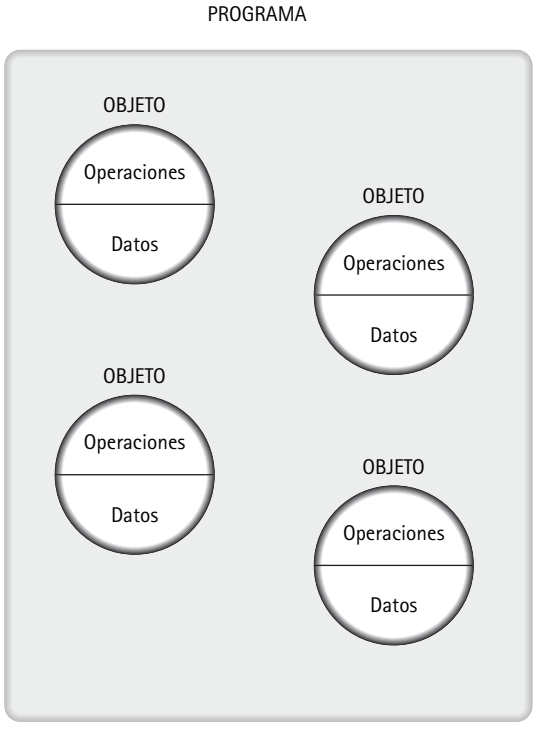


Figura 14–2 *Un programa resultante de la programación orientada a objetos*

los lenguajes OOP admiten los otros dos conceptos: *herencia* y *ligadura dinámica*. Antes de definir estos dos conceptos vamos a examinar algunas de las ideas fundamentales y la terminología de la programación orientada a objetos.

14.2 Objetos

El origen de OOP data de mediados de los años sesenta, en un lenguaje llamado Simula. Sin embargo, gran parte de la terminología actual de OOP se debe a Smalltalk, un lenguaje desarrollado a finales de los años setenta en el Centro de Investigación de Palo Alto de Xerox. En OOP, el término *objeto* tiene un significado muy específico: se trata de una entidad independiente que encapsula datos y operaciones con los datos. En otras palabras, un objeto representa una instancia de un ADT. De manera más específica, un objeto tiene un *estado* interno (los valores actuales de sus datos privados que se denominan *variables de instancia*), y un conjunto de *métodos* (operaciones públicas). Los métodos son el único medio por el cual el estado de un objeto puede ser inspeccionado o modificado por otro objeto. Un programa orientado a objetos consiste en una colección de objetos que se comunican entre sí mediante la *transmisión de mensajes*. Si el objeto A quiere que el objeto B realice alguna tarea, el objeto A transmite un mensaje que contiene el nombre del objeto (B, en este caso) y el nombre del método particular a ejecutar. El objeto B responde mediante la ejecución de este método en su propio modo, posiblemente cambiando su estado y transmitiendo mensajes también a otros objetos.

Como se puede ver, un objeto es muy diferente de una estructura de datos tradicional. Una estructura (struct) de C++ es una estructura pasiva de datos que contiene sólo datos y sobre el cual actúa un programa. En contraste, un objeto es una estructura activa de datos; los datos y el código que manipula están conectados dentro del objeto. En la jerga de OOP, un objeto sabe cómo manipularse a sí mismo.

El vocabulario de Smalltalk ha influido el vocabulario de OOP. La literatura de OOP está llena de frases como "métodos", "variables de instancia" y "transmitir un mensaje a". A continuación se presentan algunos términos de OOP y sus equivalentes en C++:

OOP	C++
Objeto	Objeto de clase o instancia de clase
Variable de instancia	Miembro privado de datos
Método	Función pública miembro
Transmisión de mensajes	Llamada de función (a una función pública miembro)

Vamos a repasar la clase `Time` que hemos definido en el capítulo 11 para representar el Tiempo de ADT, y cambiarla levemente para reflejar la terminología y los principios de orientación a objetos. Había siete operaciones (que se denominan responsabilidades en la terminología de orientación a objetos): `Set`, `Increment`, `Write`, `Equal`, `LessThan`, y dos constructores: uno que toma horas, minutos y segundos como parámetros, y otro que fija el tiempo en cero.

En la terminología de orientación a objetos existen dos tipos de responsabilidades (operaciones): de acción y de conocimiento. Todas las responsabilidades de `Time` son acciones: hacen o calculan algo. Sin embargo, un objeto necesita ser capaz de reportar acerca de su propio estatus; es decir, cada objeto debería tener una función que reporta el estado interior de cada variable apropiada de datos privados. Estas funciones podrían remplazar o suplementar la función `Write`. Si el código del cliente puede *inspeccionar* el estado interno (no *cambiarlo*), puede imprimir el tiempo en forma relevante para el problema usando el ADT. En este caso, vamos a agregar estas funciones en lugar de remplazar `Write`. Las operaciones que devuelven el estado de una variable interna son responsabilidades de conocimiento.

Haremos otro cambio en la clase de `Time`. Vamos a agregar una función que indique y lea los valores del dispositivo estándar de entrada. Aquí, pues, son los archivos revisados de especificación y aplicación para la clase `Time`. Omitimos la documentación que no ha cambiado.

```
//**************************************************************************
// ARCHIVO DE ESPECIFICACIÓN (Time.h)
// Este archivo da la especificación de un TDA Time con responsabilidades
// de acción y responsabilidades de conocimiento
//**************************************************************************

class Time
{
public:
    // Responsabilidades de acción

    void Set( /* in */ int hours,
              /* in */ int minutes,
              /* in */ int seconds );

    void Increment();

    void Write() const;

    bool Equal( /* in */ Time otherTime ) const;

    bool LessThan( /* in */ Time otherTime ) const;

    Time( /* in */ int initHrs,
          /* in */ int initMins,
          /* in */ int initSecs );

    Time();

    void ReadTime();

        // Poscondición:
        //     se han solicitado horas, minutos y segundos para leer
        //     y establecer

    // Responsabilidades de conocimiento

    int Hours() const;

        // Poscondición:
        //     El valor de retorno es el tiempo

    int Minutes() const;

        // Poscondición:
        //     El valor de retorno son minutos

    int Seconds() const;

        // Poscondición:
```

```
        //     El valor de retorno son segundos

private:
    int hrs;
    int mins;
    int secs;
};
```

Aquí está la parte del archivo de aplicación para la clase `Time` que aplica las funciones modificadas.

```
//**************************************************************************
// ARCHIVO DE EJECUCIÓN (Time. cpp)
// Este archivo pone en práctica las funciones miembros Time
//**************************************************************************

#include "Time.h"
#include <iostream>

using namespace std;
⋮

//**************************************************************************

int Time::Hours() const

// Poscondición:
//     El valor de retorno son horas

{
    return hrs;
}

//**************************************************************************

int Time::Minutes() const

// Poscondición:
//     El valor de retorno son minutos

{
    return mins;
}

//**************************************************************************

int Time::Seconds() const

// Poscondición:
//     El valor de retorno son segundos

{
    return secs;
}
```

```
//**************************************************************************

void Time::ReadTime()

// Poscondición:
//     se han solicitado horas, minutos y segundos para leer
//     y guardar en horas, minutos y segundos

{
    cout << "Enter hours (<= 23): " << endl;
    cin >> hrs;
    cout << "Enter minutes (<= 59): " << endl;
    cin >> mins;
    cout << "Enter seconds (<= 59): " << endl;
    cin >>  secs;
}

//**************************************************************************
```

Después de este ejercicio necesitamos agregar tres nuevos términos de vocabulario orientado a objetos.

OOP	C++
Responsabilidad	Operación aplicada como una función
Responsabilidad de acción	Operación que realiza una acción
Responsabilidad de conocimiento	Operación que devuelve el estado de una variable de datos privados

En C++ definimos las propiedades y el comportamiento de objetos usando el mecanismo de clases. Dentro de un programa las clases se pueden relacionar entre sí de varias maneras. Las tres relaciones más comunes son:

1. Dos clases son independientes una de la otra y no tienen nada en común.
2. Dos clases están relacionadas por *herencia*.
3. Dos clases están relacionadas por *composición*.

La primera relación —ninguna— no es muy interesante. Revisemos las otras dos: herencia y composición.

14.3 Herencia

En el mundo en general, con frecuencia es posible arreglar conceptos en forma de una *jerarquía de herencia*, esto es, una jerarquía donde cada concepto hereda las propiedades del concepto inmediatamente superior en la jerarquía. Por ejemplo, podríamos clasificar diferentes tipos de vehículos de acuerdo con la jerarquía de herencia en la figura 14-3. Bajando en la jerarquía, cada tipo de vehículo es más especializado que su *padre* (y todos sus *ancestros*) y es más general que su *hijo* (y todos sus *descendientes*). Un vehículo con ruedas hereda propiedades que son comunes a todos los vehículos (acomoda a una o varias personas y las lleva de un lugar a otro), pero tiene una propiedad adicional que lo hace más especializado (tiene ruedas). Un automóvil hereda propiedades comunes de todos los vehículos con ruedas, pero también tiene propiedades adicionales y más especializadas (cuatro ruedas, un motor, una carrocería, etcétera).

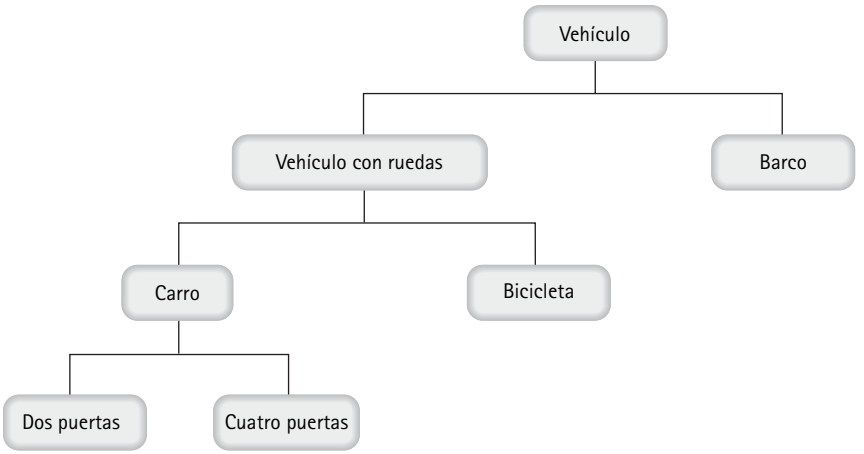


Figura 14–3 *Jerarquía de herencia*

La relación de herencia se puede visualizar como una *relación es*. Cada automóvil de dos puertas es un automóvil, cada automóvil es un vehículo con ruedas, y cada vehículo con ruedas es un vehículo.

Herencia Mecanismo por medio del cual una clase adquiere las propiedades —datos y operaciones— de otra clase.

Clase base (superclase) Clase de la cual se hereda.

Clase derivada (subclase) Clase que obtiene la herencia.

El lenguaje OOP proporciona una manera de crear relaciones de herencia entre clases. En estos lenguajes, **herencia** es el mecanismo por medio del cual una clase adquiere las propiedades de otra. Podemos considerar una clase A (nombrada **clase base** o **superclase**) y crear, a partir de ella, una nueva clase B (denominada **clase derivada** o **subclase**). La clase derivada B hereda todas las propiedades de la clase base A. En particular, los datos y operaciones definidos para A son ahora también definidos para B. (Observe la relación es; cada B es también una A.) La siguiente idea es especializar la clase B, generalmente agregando propiedades específicas a las ya heredadas de A. Veamos un ejemplo en C++.

Derivar una clase de otra

Supongamos que quisiéramos modificar la clase `Time` en la última sección, agregando, como datos privados, una variable de un tipo de enumeración indicando el uso horario (norteamericana): `EST` para Eastern Standard Time, `CST` para Central Standard Time, `MST` para Mountain Standard Time, `PST` para Pacific Standard Time, `EDT` para Eastern Daylight Time, `CDT` para Central Daylight Time, `MDT` para Mountain Daylight Time, o `PDT` para Pacific Daylight Time. Tendremos que modificar la función `Set` y los constructores de clase para acomodar un valor de uso horario. La función `Write` deberá imprimir la hora en la forma

```
12:34:10 CST
```

La función `Increment`, que avanza el tiempo por segundo, no es necesario cambiarla.

Para agregar estas características de usos horarios a la clase `Time`, el planteamiento convencional sería obtener el código fuente que se encuentra en el archivo de implementación `time.cpp`, analizar en detalle cómo se aplica la clase, y después modificar y recompilar el código fuente. Este proceso tiene varias desventajas. Si `Time` es una clase común y corriente en un sistema, el código fuente para la aplicación será posiblemente no disponible. Aunque estuviese disponible, modificarlo podrá introducir defectos en una solución previamente depurada. El acceso al código fuente también viola un beneficio principal de abstracción: Los usuarios de una abstracción no necesitarán saber cómo está aplicado.

En C++, igual que en otros lenguajes OOP, hay una forma mucho más rápida y segura para agregar características de usos horarios: el uso de la herencia. Vamos a derivar una nueva clase de la

clase `Time` y luego especializarla. Esta nueva y extendida clase de tiempo —nombrémosla `ExtTime`— hereda los miembros de su clase base, `Time`. Aquí sigue la declaración de `ExtTime`:

```
enum ZoneType {EST, CST, MST, PST, EDT, CDT, MDT, PDT};

class ExtTime : public Time
{
public:
    void Set( /* in */ int      hours,
              /* in */ int      minutes,
              /* in */ int      seconds,
              /* in */ ZoneType timeZone );
    void Write() const;
    ZoneType zone();  // Devuelve el huso horario
    ExtTime( /* in */ int      initHrs,      // Constructor
             /* in */ int      initMins,
             /* in */ int      initSecs,
             /* in */ ZoneType initZone );
    ExtTime();                                // Constructor por omisión
                                              //   que fija el tiempo en
private:                                      //   0:0:0 EST
    ZoneType zone;
};
```

La línea inicial

```
class ExtTime : public Time
```

indica que `ExtTime` es derivado de `Time`. La palabra reservada `public` declara que `Time` es una *clase base pública* de `ExtTime`. En otras palabras, las funciones miembro de `Time` que son `Set`, `Increment` y `Write`, también pueden ser invocadas para los objetos de `ExtTime`.* Sin embargo, la parte pública de `ExtTime` especializa la clase base mediante la reimplementación (redefinición) de las funciones heredadas `Set` y `Write`, y proporcionando a sus propios constructores una función para devolver el uso horario.

La parte privada de `ExtTime` declara que se agrega un nuevo miembro privado: `zone`. Los miembros privados de `ExtTime` son, por tanto, `hrs`, `mins`, `secs` (todos heredados de `Time`), y `zone`. La figura 14-4 demuestra la relación entre clases de `ExtTime` y `Time`.

Este diagrama también muestra que cada objeto de `ExtTime` tiene un objeto `Time` como *subobjeto*. Cada `ExtTime` es un `Time`, y más. C++ usa los términos *clase base* y *clase derivada* en lugar de *superclase* y *subclase*. Los términos *superclase* y *subclase* pueden ser confusos porque el prefijo *sub* normalmente implica algo más pequeño que el original (por ejemplo, un subjuego de un juego matemático). En contraste, una subclase frecuentemente es "más grande" que su superclase; o sea, tiene más datos y/o funciones.

En la Figura 14-4 vemos una flecha entre dos óvalos etiquetados `Increment`. Puesto que `Time` es una clase base pública de `ExtTime`, y ya que `Increment` no está redefinido por `ExtTime`, la función `Increment` disponible para clientes de `ExtTime` es la misma que la que se heredó de `Time`. Usamos la flecha entre los óvalos correspondientes para indicar este hecho. (Observe en el diagrama

* Si una declaración de clase omite la palabra `public` y empieza como

`class DerivedClass : BaseClass`

o si explícitamente usa la palabra `private`,

`class DerivedClass : private BaseClass`

entonces `BaseClass` se denomina como *clase privada base* de `DerivedClass`. Los miembros públicos de `BaseClass` *no* son miembros públicos de `DerivedClass`. Esto es, que los clientes de `DerivedClass` no pueden invocar operaciones de `BaseClass` en objetos de `DerivedClass`. En este libro no trabajamos con clases privadas base.

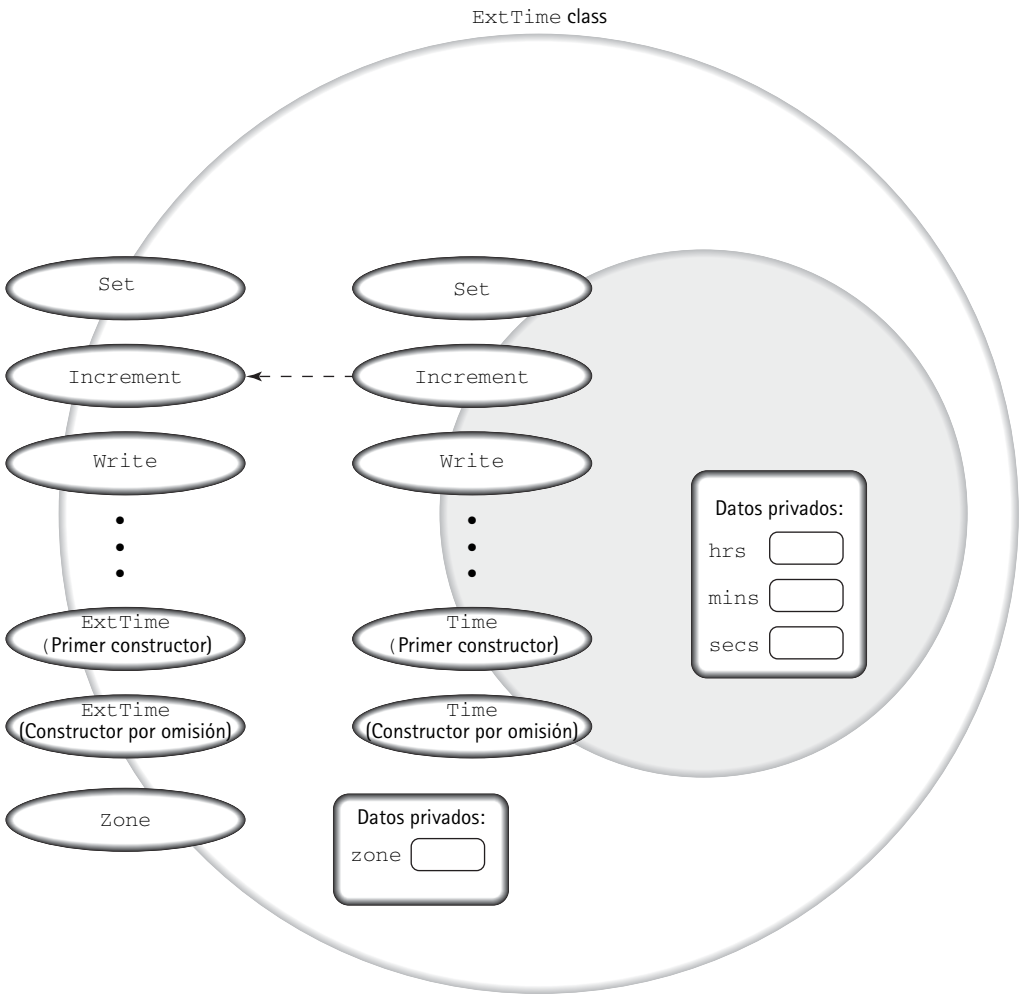


Figura 14–4 *Interfaz de clase para la clase* `ExtTime`

que los constructores de `Time` son operaciones en `Time`, no en `ExtTime`. La clase `ExtTime` debe haber tenido sus propios constructores.)

Consejo práctico de ingeniería de software

Herencia y accesibilidad

Para C++ es importante entender que herencia no implica accesibilidad. Aunque una clase derivada hereda los miembros de su clase base, tanto privados como públicos, no puede acceder a los miembros privados de la clase base. La figura 14-4 muestra que las variables `hrs`, `mins` y `secs` están encapsuladas dentro de la clase `Time`. Ni el código externo del cliente ni las funciones miembro de `ExtTime` pueden referir a estas tres variables en forma directa. Si una clase derivada fuese capaz de acceder a los miembros privados de su clase base, cualquier programador podría derivar una clase de alguna otra, y luego escribir un código para inspeccionar o modificar directamente los datos privados, invalidando así los beneficios del encapsulado y del principio de ocultamiento de la información.

Especificación de la clase `ExtTime`

A continuación se presenta completamente documentada la especificación de la clase `ExtTime`. Observe que la directiva de procesador

```
#include "time.h"
```

es necesaria para que el compilador pueda verificar la consistencia de la clase derivada con la clase base.

```
//**************************************************************************
// ARCHIVO DE ESPECIFICACIÓN (exttime.h)
// Este archivo da la especificación de un tipo de datos abstractos ExtTime.
// La clase Time es una clase base pública de ExtTime, así que las
// operaciones públicas de Time son también operaciones públicas de ExtTime.
//**************************************************************************
#include "time.h"

enum ZoneType {EST, CST, MST, PST, EDT, CDT, MDT, PDT};

class ExtTime : public Time
{
public:
    void Set( /* in */ int      hours,
              /* in */ int      minutes,
              /* in */ int      seconds,
              /* in */ ZoneType timeZone );
        // Precondición:
        //     se asigna 0 <= horas <= 23 && 0 <= minutos <= 59
        //  && 0 <= segundos < = 59 && timeZone
        // Poscondición:
        //     El tiempo se establece de acuerdo con los parámetros entrantes

    void Write() const;
        // Poscondición:
        //     Se ha generado el tiempo en la forma HH:MM:SS ZZZ
        //     donde ZZZ es el huso horario

    ZoneType Zone () const;
        // Poscondición:
        //     Valor de retorno == zona

    ExtTime( /* in */ int      initHrs,
             /* in */ int      initMins,
             /* in */ int      initSecs,
             /* in */ ZoneType initZone );
        // Precondición:
        //     se asigna 0 <= initHrs <= 23 && 0 <= initMins <= 59
        //  && 0 <= initSecs < = 59 && initZone
        // Poscondición:
        //     Se construye el objeto de clase
        //  && Time se establece de acuerdo con los parámetros entrantes

    ExtTime();
        // Poscondición:
```

```
        //     Se construye el objeto de clase
        //  && Time es 0:0:0 hora estándar del este
private:
    ZoneType zone;
};
```

Por medio de esta nueva clase el programador puede fijar el tiempo con un uso horario (vía un constructor de clase o la función redefinida `Set`), emitir el tiempo con su uso horario (vía la función redefinida `Write`), e incrementar el tiempo por un segundo (vía la función heredada `Increment`):

```
//**************************************************************************
// Programa TimeDemo
// Éste es un cliente muy simple de la clase ExtTime
//**************************************************************************
#include <iostream>
#include "exttime.h"   // Para clase ExtTime
using namespace std;

int main()
{
    ExtTime time1(5, 30, 0, CDT);  // Constructor parametrizado empleado
    ExtTime time2;                 // Constructor por omisión empleado
    int     loopCount;

    cout << "time1: ";
    time1.Write();
    cout << endl << "time2: ";
    time2.Write();
    cout << endl;

    time2.Set(23, 59, 55, PST);
    cout << "New time2: ";
    time2.Write();
    cout << endl;

    cout << "Incrementing time2:" << endl;
    for (loopCount = 1; loopCount <= 10; loopCount++)
    {
        time2.Write();
        cout << ' ';
        time2.Increment();
    }
    return 0;
}
```

Cuando se ejecuta, el programa TimeDemo produce el siguiente resultado.

```
time1: 05:30:00 CDT
time2: 00:00:00 EST
New time2: 23:59:55 PST
Incrementing time2:
23:59:55 PST 23:59:56 PST 23:59:57 PST 23:59:58 PST 23:59:59 PST
00:00:00 PST 00:00:01 PST 00:00:02 PST 00:00:03 PST 00:00:04 PST
```

Aplicación de la clase `ExtTime`

La aplicación de la clase `ExtTime` sólo tiene que ocuparse de las nuevas características diferentes de `Time`. Específicamente, tenemos que escribir el código para redefinir las funciones `Set` y `Write`, y tenemos que escribir los dos constructores.

En el caso de las clases derivadas, los constructores están sujetos a reglas especiales. En tiempo de ejecución, el constructor de clase base implícitamente se llama primero, antes de que se ejecute el cuerpo del constructor de la clase derivada. Adicionalmente, si el constructor de clase base requiere argumentos, éstos deben ser transmitidos por el constructor de la clase derivada. Para ver cómo se aplican estas reglas, vamos a examinar el archivo de implementación `exttime.cpp` (véase la figura 14-5).

Figura 14-5 *Archivo de implementación `ExtTime`*

```
//**************************************************************************
// ARCHIVO DE EJECUCIÓN (exttime.cpp)
// Este archivo pone en práctica las funciones miembro ExtTime.
// La clase Time es una clase base pública de ExtTime
//**************************************************************************
#include "exttime.h"
#include <iostream>
#include <string>

using namespace std;

// Miembros de clase privados adicionales:
//     Zona ZoneType;

//**************************************************************************

ExtTime::ExtTime( /* in */ int      initHrs,
                  /* in */ int      initMins,
                  /* in */ int      initSecs,
                  /* in */ ZoneType initZone )

     : Time(initHrs, initMins, initSecs)

 // Constructor

 // Precondición:
//     se asigna 0 <= initHrs <= 23 && 0 <= initMins <= 59
//  && 0 < = initSecs <= 59 && initZone
// Poscondición:
//     El tiempo se establece de acuerdo con initHrs, initMins e initSecs
//     (vía llamada al constructor de clase base)
//  && zona == initZone

{
    zone = initZone;
}
//**************************************************************************

ExtTime::ExtTime()

// Constructor por omisión
```

```
// Poscondición:
//     El tiempo es 0:0:0 (vía llamada implícita al constructor por
//     omisión de la clase base)
//  && zona = EST

{
    zone = EST;
}

//***************************************************************************

void ExtTime::Set( /* in */ int      hours,
                   /* in */ int      minutes,
                   /* in */ int      seconds,
                   /* in */ ZoneType timeZone )

// Preconción:
//     Se asigna 0 < = horas <= 23 && 0 <= minutos <= 59
//  && 0 <= segundos <= && timeZone
// Poscondición:
//     El tiempo se establece de acuerdo con horas, minutos y segundos
//  && zona = timeZone

{
    Time::Set(hours, minutes, seconds);
    zone = timeZone;
}

//***************************************************************************

    ZoneType ExtTime::Zone() const

    // Poscondición:
    //     Valor de retorno == zona

    {
        return zone;
    }

//***************************************************************************

void ExtTime::Write() const

// Poscondición:
//     Se ha generado el tiempo en la forma HH:MM:SS ZZZ
//     donde ZZZ es el huso horario

{
    static string zoneString[8] =
    {
        "EST", "CST", "MST", "PST", "EDT", "CDT", "MDT", "PDT"
    };

    Time::Write();
    cout << ' ' << zoneString[zone];
}
```

En el primer constructor en la figura 14-5, observe la sintaxis mediante la cual un constructor transmite argumentos a su constructor de clase base:

```
ExtTime::ExtTime( /* in */ int      initHrs,
                  /* in */ int      initMins,
                  /* in */ int      initSecs,
                  /* in */ ZoneType initZone )

    : Time(initHrs, initMins, initSecs)     ← Inicializador de constructor

{
    zone = initZone;
}
```

Después de la lista de parámetros para el constructor `ExtTime` (pero antes de su cuerpo), se deberá insertar lo que se denomina un *inicializador de constructor*, o sea un signo de dos puntos, y luego el nombre de la clase base junto con los argumentos para *su* constructor. Cuando se crea un objeto `ExtTime` con una declaración como

```
ExtTime time1(8, 35, 0, PST);
```

el constructor de `ExtTime` recibe cuatro argumentos. Los primeros tres se transmiten simplemente al constructor de clase `Time` por medio del inicializador de constructor. Después de que el constructor de clase `Time` haya ejecutado (creando el subobjeto de la clase base, como se puede ver en la figura 14-4), el cuerpo del constructor `ExtTime` se ejecuta, fijando `zone` igual al cuarto argumento.

El segundo constructor en la figura 14-5 (el constructor por omisión) no necesita un inicializador de constructor; no hay argumentos que transmitir al constructor por omisión de la clase base. Cuando se crea un objeto `ExtTime` mediante la declaración

```
ExtTime time2;
```

el constructor por omisión de la clase base primero implícitamente llama al constructor por omisión de `Time`, y después ejecuta su cuerpo, fijando `zone` a `EST`.

Ahora nos fijamos en la función `Set` en la figura 14-5. Esta función reimplementa la función `Set` heredada de su clase base. En consecuencia, hay dos funciones `Set` distintas: una que es un miembro público de la clase `Time`, y otra que es un miembro público de la clase `ExtTime`. Sus nombres completos son `Time::Set` y `ExtTime::Set`. En la figura 14-5 la función `ExtTime::Set` empieza por "alcanzar" a su clase base, llamando a `Time::Set` para fijar las horas, minutos y segundos. (Recuerde que una clase derivada de `Time` no puede acceder a los datos privados `hrs`, `mins` y `secs` de manera directa; estas variables son privadas de la clase `Time`.) Después, la función termina asignando un valor a los datos privados `ExtTime`, la variable `zone`.

La función `Write` en la figura 14-5 usa una estrategia similar. Alcanza a su clase base e invoca `Time::Write` para emitir las horas, minutos y segundos. Luego emite una cadena correspondiente al uso horario. (Recuerde que un valor de un tipo de enumeración en C++ no se puede emitir de manera directa. Si imprimiéramos el valor de `zone` de modo directo, la salida sería un entero de 0 a 7, o sea las representaciones internas de los valores `ZoneType`.) La función `Write` establece un array de ocho cadenas y selecciona la cadena correcta usando `zone` para indexar al array. ¿Por qué `zoneString` es declarada como `static`? Recuerde que por omisión las variables locales en C++ son variables automáticas, o sea la memoria es asignada para ellas cuando la función inicia la ejecución, y es borrada cuando la función regresa. Con `zoneString` declarada como `static`, el array es asignado sólo una vez cuando el programa comienza la ejecución, y permanecerá asignado hasta que el programa termine. De llamada de función a llamada de función, la computadora no pierde tiempo creando y destruyendo el array.

Ahora podemos compilar el archivo `exttime.cpp` en un archivo de código de objeto, por ejemplo `exttime.obj`. Después de escribir un programa director de pruebas y compilarlo en `test.obj`, obtendremos un archivo ejecutable ligando tres archivos objeto:

1. `test.obj`
2. `exttime.obj`
3. `time.obj`

Ahora podemos verificar el programa que resulta.

Lo extraordinario en cuanto a clases derivadas y herencia es que no es necesario modificar la clase base. El código fuente para la aplicación de la clase `Time` podrá ser no disponible. Pero aun así es posible continuar creando variaciones de este ADT sin este código fuente, en formas que su creador jamás consideraba. Por medio de clases y herencia, los lenguajes OOP facilitan la reutilización de código. Una clase como `Time` se puede usar tal cual en muchos contextos diferentes, o se puede adaptar a un contexto particular usando la herencia. Ésta nos permite crear abstracciones de datos extensibles, esto es, una clase derivada normalmente extiende la clase base mediante la inclusión de datos privados adicionales u operaciones públicas, o ambas cosas.

Evitar inclusiones múltiples de archivos de encabezados

Hemos visto que el archivo de especificación `exttime.h` empieza con una directiva `#include` para inserar el archivo `time.h`:

```
#include "time.h"

enum ZoneType {EST, CST, MST, PST, EDT, CDT, MDT, PDT};

class ExtTime : public Time
{
   ⋮
};
```

Ahora vamos a pensar qué sucede si un programador que usa la clase `ExtTime` ya hubiese incluido `time.h` para otros propósitos, sin darse cuenta del hecho de que `exttime.h` también lo incluye:

```
#include "time.h"
#include "exttime.h"
```

El preprocesador inserta el archivo `time.h`, luego `exttime.h`, y luego `time.h` una segunda vez (porque `exttime.h` también incluye a `time.h`). El resultado es un error en tiempo de compilación, porque la clase `Time` queda definida dos veces.

La solución que por lo común se usa para este problema es escribir `time.h` de esta manera:

```
#ifndef TIME_H
#define TIME_H
class Time
{
   ⋮
};
#endif
```

Las líneas que comienzan con "`#`" son directivas para el preprocesador. `TIME_H` (o cualquier identificador que se quiera usar) es un identificador de preprocesador, y no un identificador de programa en C++. En efecto, estas directivas dicen:

Si el identificador de preprocesador `TIME_H` aún no está identificado, entonces

1. define `TIME_H` como un identificador conocido por el preprocesador,

y

2. permite que la declaración de la clase `Time` pase a través del compilador.

Si se encuentra un subsiguiente `#include "time.h"`, la prueba `#ifndef TIME_H` fallará. La declaración de clase `Time` no pasará a través del compilador una segunda vez.

14.4 Composición

Composición (contención) Mecanismo mediante el cual los datos internos (el estado) de una clase incluyen un objeto de otra clase.

Hemos dicho que dos clases normalmente exhiben una de las siguientes relaciones: son independientes una de la otra, están relacionadas por herencia, o están relacionadas por **composición**. La composición (o **contención**) es la relación en que los datos internos de una clase A incluyen un objeto de otra clase B. Dicho de otra manera, un objeto B está contenido dentro de un objeto A.

C++ no tiene (ni necesita) una notación especial de lenguaje para la composición. Simplemente se declara que un objeto de alguna clase es uno de los miembros de datos de otra clase. Veamos un ejemplo.

Diseño de una clase `Entry`

Usted está desarrollando un programa para representar una agenda de citas. Cada objeto de día será una lista ordenada de objetos de citas. En preparación, usted decide construir un "Entry ADT" con dos campos, un objeto de un ADT de nombre y un objeto de un ADT de hora. Es posible que tenga que agregar más información para un objeto de citas más adelante, pero con esto ya tendrá un inicio. La construcción de este objeto debe ser fácil, porque usted ya tiene una clase `Time` y una clase `Name`. Para los fines de una agenda de citas, usted decide ignorar el segundo nombre o inicial y los segundos en la hora. Siempre podrá agregar el segundo nombre más tarde, si es que lo necesita. Desde luego, usted sí necesita operaciones que devuelven el nombre y la hora al programa cliente.

Las operaciones pueden ser `NameStr`, que devuelve un nombre compuesto del nombre de pila y el apellido, y `TimeStr`, que devuelve las horas y minutos con un signo de dos puntos entre los dos. Por supuesto necesitamos constructores. ¿Cuántos? Bueno, usted sabe que la clase `Name` tiene dos constructores, uno que toma las partes del nombre como parámetros, y otro que lee el nombre del dispositivo estándar de entrada (por omisión). Lo mismo aplica también para la clase `Time`. Ahora necesitamos un constructor por omisión para la clase `Entry`, que llamará a los constructores por omisión para `Name` y `Time`, así como un constructor parametrizado que transmitirá los parámetros a los constructores parametrizados de `Name` y `Time`.

```
//**************************************************************************
// ARCHIVO DE ESPECIFICACIÓN (Entry.h)
// Este archivo contiene la especificación del TDA Entry,
// que tiene dos clases contenidas, Name y Time
//**************************************************************************

#include "Time.h"
#include "Name.h"
#include <string>
```

```
class Entry
{
public:
    string NameStr() const;
        // Devuelve una cadena constituida por nombre, espacio en blanco,
           apellido materno
        // Poscondición:
        //     El valor de retorno es el nombre del objeto Name, espacio
        //     en blanco y el apellido materno del objeto Name
    string TimeStr() const;
        // Devuelve una cadena constituida de hora, dos puntos, minutos
        // Poscondición:
        //     El valor de retorno es minutos del objeto Time, dos puntos y
        //     segundos del objeto Time
    Entry();
        // Constructor por omisión
        // Poscondición:
        //     Se ha construido el objeto Entry

    Entry( (/* in */ string firstName,    // Nombre
            /* in */ string middleName,   // Apellido paterno
            /* in */ string lastName)     // Apellido materno
            /* in */ int  initHours,      // Horas
            /* in */ int  initMinutes,    // Minudos
            /* in */ int  initSeconds )   // Segundos
        // Constructor parametrizado
        // Poscondición:
        //     Se ha construido el objeto Entry con nombre,
        //     apellido paterno y apellido materno como argumentos para
        //     el constructor parametrizado Name; objeto
        //     TimeinitHours, initMinutes e initSeconds como argumentos para
        //     el contrstructor parametrizado Time

private:
    Name name;
    Time time;
}
```

La aplicación de las funciones miembro `Name` y `Time` es fácil: sólo requiere invocar las funciones miembro `Name` y `Time`. Los archivos de especificación para cada una de estas clases proporciona la interfaz que las funciones deben usar. Una versión abreviada de `Name.h` se muestra a continuación; `Time.h` ya se había mostrado antes en este capítulo.

```
// ARCHIVO DE ESPECIFICACIÓN (Name.h)
...
enum RelationType{BEFORE, SAME, AFTER};
class Name
{
public:
    Name();
        // Constructor por omisión
        // Poscondición:
        //     Se solicita el nombre y se lee del dispositivo de entrada
        //     estándar
```

```
    Name( /* in */ string firstName,
          /* in */ string middleName,
          /* in */ string lastName );
        // Constructor parametrizado
        // Poscondición:
        //     primero va firstName, luego middleName y,
        //     por último, lastName
    string FirstName() const;
        // Poscondición:
        //     El valor de retorno es el nombre de esta persona

    string LastName() const;
        // Poscondición:
        //     El valor de retorno es el apellido materno de esta persona

    ...
};
```

Aplicar los constructores de clase por omisión significa invocar los constructores por omisión para los objetos `Name` y `Time`. El constructor por omisión `Entry` no tiene otra cosa que hacer. Así, el cuerpo del constructor por omisión para `Entry` está vacío. El constructor parametrizado deberá transmitir los parámetros a los constructores de `Name` y `Time`. ¿Cómo se realiza esto? Usamos el inicializador de constructor que hemos descrito en la última sección. De esta manera el constructor parametrizado para `Entry` tampoco tiene cuerpo.

```
//**************************************************************************
// ARCHIVO DE EJECUCIÓN (Entry.cpp)
// Este archivo contiene la especificación del TDA Entry, que tiene
// dos clases contenidas, Name y Time
//**************************************************************************

#include "Entry.h"
#include <string>

#include <fstream>
#include <iostream>
using namespace std;
string Entry::NameStr() const

    // Devuelve una cadena constituida por nombre, espacio en blanco,
       apellido materno
    // Poscondición:
    //   El valor de retorno es el nombre del objeto Name, espacio en blanco
    //   y el apellido del objeto Name

{
    return (name.FirstName() + ' ' + name.LastName());
}
string Entry::TimeStr() const

    // Devuelve una cadena constituida por hora, dos puntos, minutos
    // Poscondición:
    //     El valor de retorno es minutos del objeto Time, dos puntos
    //     y segundos del objeto Time
```

```
{
    return "" + time.Hours() + ":" + time.Minutes();
}

bool Entry::LessThan(Entry entry) const

    // Compara el tiempo con entry.time
    // Poscondición:
    //     El valor de retorno es verdadero si el tiempo es menor que
          entry. time;
    //     falso en caso contrario

{
    return time.LessThan(entry.time);
}

bool Entry::Equal(Entry entry) const

    // Compara el tiempo con entry.time
    // Poscondición:
    //     El valor de retorno es verdadero si por sí mismo es igual a
          entry.time;
    //      falso en caso contrario

{
    return time.Equal(entry.time);
}

Entry::Entry()

    // Constructor por omisión
    // Poscondición:
    //     Se ha construido el objeto de entrada
{
}

Entry::Entry( /* in */ string firstName,
              /* in */ string middleName,
              /* in */ string lastName,
              /* in */ int  initHours,
              /* in */ int  initMinutes,
              /* in */ int  initSeconds )

 // Inicializadores de constructor
  : name(firstName, middleName, lastName),
    time(initHours, initMinutes, initSeconds)

     // Constructor parametrizado
     // Poscondición:
     //     El objeto de entrada ha sido construido con firstName,
     //     middleName y lastName como argumentos para el
     //     constructor parametrizado Name; initHours, initMinutes e
     //     initSeconds como argumentos para el constructor
     //     parametrizado Time

{
}
```

Prueba: a continuación se muestra un controlador que crea dos objetos de la clase Entry y aplica las funciones apropiadas para ellos.

```
//**************************************************************************
// DRIVER para la clase Entry
// Este programa prueba los constructores y devuelve funciones
//**************************************************************************

#include <iostream>
#include "Entry.h"

using namespace std;

int main()
{
    Entry entry1("Sally", "Jane", "Smith", 12, 20, 0);
    Entry entry2("Mary", "Beth", "Jones", 10, 30, 0);

    entry1.ReadEntry();

    cout << "Entry 1: " << entry1.NameStr() << " "
         << entry1.TimeStr() << endl;
    cout << "Entry 2: " << entry2.NameStr() << " "
         << entry2.TimeStr() << endl;
    return 0;
}
```

¿Qué? ¿La clase `Entry` no compila? Recibimos el siguiente mensaje de error:

```
Error   : illegal operand
EntryEr.cpp line 33   return "" + time.Hours() + ":" + time.Minutes();
```

¿Cuál podría ser el error en esta instrucción? Acabamos de crear una cadena a partir de una cadena, un valor entero, un signo de dos puntos y otro valor entero. Hemos usado la concatenación para construir cadenas desde el capítulo 2. Los números se convierten en cadenas para la salida. ¿Cuál es la diferencia en esta sentencia? El sistema C++ automáticamente convierte enteros en cadenas *para la salida*, pero no hay ninguna conversión de tipo de enteros a cadenas, excepto para la salida.

Para crear una cadena que incluye enteros podríamos escribirlo en un archivo y luego volver a importarlo como una cadena. Esto sí funcionaría, pero llevaría mucho tiempo (el acceso a disco es un millón de veces más lento que el acceso a memoria). Afortunadamente los diseñadores del lenguaje se dieron cuenta de que puede haber ocasiones en que una conversión de enteros a cadenas puede ser útil, así que derivaron una clase llamada `stringstream` de la clase `iostream`, que nos permite escribir valores a una cadena como si estuviéramos escribiéndolos a un archivo. Puesto que los datos no se transmiten realmente a un archivo sino que son mantenidos en la memoria, la conversión se realiza rápidamente.

Declaramos un objeto de la clase `ostringstream` y usamos el operador de inserción (<<) para enviar los enteros a objetos. Para que sean devueltos en forma de una cadena, usamos la función miembro `str` de la clase `ostringstream`. Aquí, pues, está la versión correcta de `TimeStr`, usando un objeto de la clase `ostringstream`, seguido por la salida del controlador.

```
#include <sstream>                  // ostringstream
string Entry::TimeStr() const
```

```
// Devuelve una cadena constituida por hora, dos puntos, minutos, dos puntos,
   segundos
// Poscondición:
//     El valor de retorno es minutos del objeto Time, dos puntos
//     y segundos del objeto Time

{
    string outStr;
    ostringstream tempOut;  // ostringstream de acceso

    if (time.Hours() < 10)
        tempOut << '0';
    tempOut << time.Hours() << ":";
    if (time.Minutes() < 10)
        tempOut << '0';
    tempOut << time.Minutes() << ":";
    if (time.Seconds() < 10);
        tempOut << '0';
    tempOut << time.Seconds();
    outStr = tempOut.str();
    return outStr;
}
```

Una vez que esta función está insertada, el controlador produce el siguiente resultado:

Inicializador de constructor

Dos veces hemos visto la notación poco usual —el inicializador de constructor— insertada entre la lista de parámetros y el cuerpo de un constructor. La primera vez fue cuando aplicamos el constructor parametrizado de clase `ExtTime` (figura 14-5). En esa ocasión usamos el inicializador de constructor para transmitir algunos de los argumentos a constructores de objetos de miembros (`name` y `time`). Si estamos usando herencia o composición, el propósito de un inicializador de constructor es el mismo: transmitir argumentos a otro constructor. La única diferencia es: en el caso de herencia, se especifica el nombre de la *clase base* antes de la lista de argumentos, de la siguiente manera:

```
ExtTime::ExtTime( /* in */ int      initHrs,
                  /* in */ int      initMins,
                  /* in */ int      initSecs,
                  /* in */ ZoneType initZone )

    : Time(initHrs, initMins, initSecs)
```

En el caso de composición, se especifica el nombre del objeto miembro antes de la lista de argumentos:

```
Entry::Entry( /* in */ string firstName,
              /* in */ string middleName,
              /* in */ string lastName,
              /* in */ int  initHours,
              /* in */ int  initMinutes )
// Inicializadores de constructor
  : name(firstName, middleName, lastName),
    time(initHours, initMinutes)
```

Puesto que hemos analizado tanto la herencia como la composición, ahora podemos dar una descripción completa del orden de ejecución de los constructores:

> *Dada una clase X, si X es una clase derivada, su constructor de clase base se ejecuta primero. Luego se ejecutan los constructores para objetos miembro (si es que hay). Finalmente se ejecuta el cuerpo del constructor de X.*

Cuando se crea un objeto `Entry`, primero se invocan los constructores para sus miembros `name` y `time`. Después de construir los objetos `name` y `time`, se ejecuta el cuerpo del constructor `Entry`, pero en este caso está vacío.

14.5 Ligadura dinámica y funciones virtuales

Al principio del capítulo dijimos que los lenguajes de programación orientada a objetos proporcionan características de lenguaje que admiten tres conceptos: abstracción de datos, herencia y ligadura dinámica. El término *ligadura dinámica* significa, de manera más específica, la *ligadura dinámica de una operación a un objeto*. Para explicar este concepto, comencemos con un ejemplo.

Dadas las clases `Time` y `ExtTime` de este capítulo, el siguiente código crea dos objetos de clase y emite el tiempo representado por cada uno.

```
Time    startTime(8, 30, 0);
ExtTime endTime(10, 45, 0, CST);

startTime.Write();
cout << endl;
endTime.Write();
cout << endl;
```

Este fragmento de código invoca dos diferentes funciones `Write`, aunque las funciones parecerán tener el mismo nombre. La primera llamada de función invoca la función `Write` de la clase `Time`, imprimiendo tres valores: horas, minutos y segundos. La segunda llamada invoca la función `Write` de la clase `ExtTime`, imprimiendo cuatro valores: horas, minutos, segundos y usos horarios. En este fragmento de código el compilador usa la **ligadura estática** (tiempo de compilación) de la operación (`Write`) a objetos apropiados. El compilador puede determinar fácilmente cuál función `Write` llama, verificando el tipo de datos del objeto asociado.

Ligadura estática Determinación en tiempo de compilación, qué función se deberá llamar para un objeto particular.

En algunas situaciones el compilador no puede determinar el tipo de un objeto, y la ligadura de una operación a un objeto debe ocurrir en tiempo de ejecución. Una situación, que veremos a continuación, involucra la transmisión de objetos de clase como argumentos.

La regla básica de C++ para transmitir objetos de clase como argumentos es que el argumento y su correspondiente parámetro deben ser de tipo idéntico. En el caso de herencia, sin embargo, C++ cede a esta regla. Se puede transmitir un objeto de una clase hijo *C* a un objeto de su clase padre *P*,

pero no al revés; esto es, no se puede transmitir un objeto del tipo *P* a un objeto del tipo *C*. De modo más general, se puede transmitir un objeto de una clase descendente a un objeto de cualquiera de sus clases de antepasados. Esta regla tiene un beneficio tremendo: nos permite escribir una sola función que se aplica a cualquier clase descendente, en lugar de escribir una función diferente para cada una. Por ejemplo, podríamos escribir una función `Print` modificada que toma como argumento un objeto del tipo `Time` o de cualquier clase descendente de `Time`:

```
void Print( /* in */ Time someTime )
{
    cout << "************************" << endl;
    cout << "** The time is ";
    someTime.Write();
    cout << endl;
    cout << "************************" << endl;
}
```

Dado el fragmento de código

```
Time    startTime(8, 30, 0);
ExtTime endTime(10, 45, 0, CST);

Print(startTime);
Print(endTime);
```

el compilador nos permite transmitir un objeto `Time` o un objeto `ExtTime` a la función `Print`. Desafortunadamente el resultado no es lo que nos gustaría. Cuando se imprime `endTime`, faltará el uso horario `CST` en el resultado. Veamos por qué.

El problema de corte

Nuestra función `Print` usa la transmisión por valor para el parámetro `someTime`. La transmisión por valor manda una copia del argumento al parámetro. Cada vez que transmitimos un objeto de una clase hijo a un objeto de su clase padre usando una transmisión por valor, sólo se copian los miembros de datos que tienen en común. Recuerde que una clase hijo a menudo es "más grande" que su padre; es decir, que contiene miembros de datos adicionales. Por ejemplo, un objeto `Time` tiene tres miembros de datos (`hrs`, `mins` y `secs`), pero un objeto `ExtTime` tiene cuatro miembros de datos (`hrs`, `mins`, `secs` y `zone`). Cuando el objeto de clase más grande se copia al parámetro más pequeño usando una transmisión por valor, se descartan o se "cortan" los miembros de clase extra. Esta situación se denomina *problema de corte* (véase la figura 14-6).

(El problema de corte también ocurre en el caso de operaciones de asignación. En la sentencia

```
parentClassObject = childClassObject;
```

sólo se copian los miembros de datos que los dos objetos tienen en común. Los miembros de datos adicionales incluidos en `childClassObject` no son copiados.)

En el caso de la transmisión por referencia, el problema de corte no ocurre porque la *dirección* del argumento del solicitante es transmitido a la función. Vamos a cambiar el encabezado de nuestra función `Print`, de manera que `someTime` sea un parámetro de referencia:

```
void Print( /* in */ Time& someTime )
```

Ahora, cuando transmitimos `endTime` como el argumento, su dirección es transmitida a la función. Su miembro de uso horario no es cortado porque no se realiza un proceso de copiado. Pero para nuestra consternación la función `Print` *sigue* imprimiendo sólo tres de los miembros de datos `endTime`: horas, minutos y segundos.

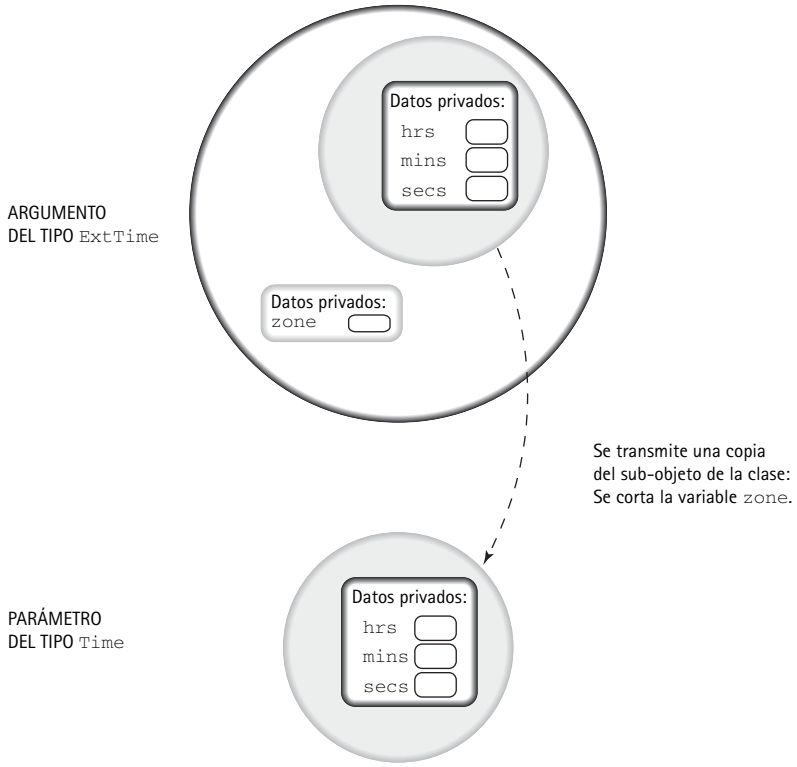


Figura 14–6 *El problema de corte que resulta de la transmisión por valor*

Dentro de la función `Print`, la dificultad es que se usa la ligadura estática en la sentencia

```
someTime.Write();
```

El compilador debe generar código de lenguaje de máquina para la función `Print` en tiempo de compilación, pero el tipo del argumento real (`Time` o `ExtTime`) no se conoce hasta en tiempo de ejecución. ¿Cómo puede el compilador saber cuál función `Write` deberá usar: `Time::Write` o `ExtTime::Write`? El compilador no lo puede saber, y, por ende, usa `Time::Write` porque el parámetro `someTime` es del tipo `Time`. Por tanto, la función `Print` siempre imprime solo tres valores —horas, minutos y segundos— sin considerar el tipo de argumento. Por fortuna C++ proporciona una solución muy sencilla para nuestro problema: *funciones virtuales*.

Funciones virtuales

Supongamos que hiciéramos una pequeña modificación a nuestra declaración de clase `Time`: empezamos la declaración de la función `Time` con la palabra reservada `virtual`.

```
class Time
{
public:
    ⋮
    virtual void Write() const;
    ⋮
private:
    ⋮
};
```

Ligadura dinámica Determinación en tiempo de ejecución de qué función se deberá llamar para un objeto particular.

La declaración de que una función miembro debe ser `virtual` instruye al compilador que debe generar código que garantiza la **ligadura dinámica** (en tiempo de ejecución) de una función a un objeto. Esto significa que la determinación de qué función se deberá llamar se pospone hasta el tiempo de ejecución. (Observe que para hacer que `Write` sea una función virtual, la palabra `virtual` aparece sólo en un lugar, o sea en la declaración de la clase `Time`. No aparecerá en la definición de la función `Write` que se ubica en el archivo `time.cpp`, ni aparecerá en cualquier clase descendente —como `ExtTime`— que redefine la función `Write`.)

Las funciones virtuales funcionan del siguiente modo: si un objeto de clase se transmite por referencia a alguna función, y si el cuerpo de esta función contiene una sentencia

```
param.MemberFunc( . . .  );
```

donde

1. Si `MemberFunc` no es una función virtual, el tipo del *parámetro* determina qué función se deberá llamar. (Se usa la ligadura estática.)
2. Si `MemberFunc` es una función virtual, el tipo del *argumento* determina qué función se deberá llamar. (Se usa la ligadura dinámica.)

Sólo con una palabra —`virtual`—, las dificultades que hemos encontrado con nuestra función `Print` desaparecen por completo. Si declaramos que `Write` sea una función virtual en la clase `Time`, entonces la función

```
void Print( /* in */ Time& someTime )
{
    ⋮
    someTime.Write();
    ⋮
}
```

trabaja correctamente para argumentos tanto del tipo `Time` como del tipo `ExtTime`. La función correcta `Write` (`Time::Write` o `ExtTime::Write`) es invocada porque el argumento porta la información necesaria en tiempo de ejecución para elegir la función apropiada. Derivar una nueva y no anticipada clase de `Time` no presenta complicaciones. Si esta nueva clase redefine la función `Write`, entonces nuestra función `Print` aún trabaja correctamente. La ligadura dinámica asegura que cada objeto sabe cómo imprimirse, y se invocará la versión apropiada. En la terminología OOP, `Write` es una **operación polimórfica**, esto es, una operación que tiene múltiples significados, dependiendo del tipo de objeto que responde a ella en tiempo de ejecución.

Operación polimórfica Operación que tiene múltiples significados, dependiendo del tipo de objeto al que se atribuye en tiempo de ejecución.

A continuación algunos aspectos que se deben saber acerca del uso de funciones virtuales en C++:

1. Para obtener la ligadura dinámica, se debe usar la transmisión por referencia cuando se transmite un objeto de clase a una función. Si se transmite por valor, el compilador no usará el mecanismo `virtual`; en su lugar ocurrirá el corte de miembros y la ligadura estática.
2. En la declaración de una función virtual la palabra `virtual` aparecerá sólo en la clase base, y no en una clase derivada.
3. Si una clase base declara una función virtual, entonces *debe* aplicar esta función, incluso si el cuerpo está vacío.

4. No se requiere una clase derivada para proporcionar su propia reaplicación para una función virtual. En este caso se usa la versión de la clase base por omisión.
5. Una clase derivada no puede redefinir el tipo de devolución de función de una función virtual.

14.6 Diseño orientado a objetos

Hemos visto características de lenguaje que nos permiten aplicar un diseño orientado a objetos. Ahora nos dirigimos a la fase que precede la aplicación; esto es, el propio OOD.

Por lo regular, un programa de computación modela alguna actividad o concepto de la vida real. Un programa de manejo de cuentas de un banco modela las actividades de la vida real asociadas con un banco. Un programa de hoja de cálculo modela una verdadera hoja de cálculo, una hoja de papel grande que usan los contadores y planeadores financieros. Un programa de robótica modela la percepción humana y el movimiento humano.

Casi siempre el aspecto del mundo que estamos modelando (el *dominio de aplicación* o *dominio del problema*) consiste en objetos, o sea cuentas de cheques, cajeros de banco, filas de una hoja de cálculo, columnas de una hoja de cálculo, brazos y piernas de un robot. El programa de computación que resuelve el problema de la vida real también incluye objetos (el *dominio de solución*), o sea contadores, listas, menús, ventanas, etc. OOD está basado en la filosofía de que es más fácil escribir y entender programas si sus objetos principales corresponden estrechamente a los objetos en el dominio del problema.

Existen muchas maneras de realizar un diseño orientado a objetos. Diferentes autores abogan por otras técnicas. Nuestro propósito no es elegir una técnica en particular o presentar un resumen de todas las técnicas. Más bien, queremos describir un proceso de tres pasos que capte la esencia de OOD:

1. Identificar los objetos y operaciones.
2. Determinar las relaciones entre objetos.
3. Diseñar el controlador.

En esta sección no mostraremos un ejemplo completo de un diseño orientado a objetos de una solución de problema. Esto lo dejaremos para el Caso práctico de resolución de problemas, al final del capítulo. En su lugar describiremos los problemas importantes involucrados en cada uno de los tres pasos.

Paso 1: Identificar los objetos y operaciones

Recuerde que el diseño estructurado (la descomposición funcional) empieza con la identificación de las acciones principales que el programa debe realizar. A diferencia de esto, OOD empieza con la identificación de los objetos principales y las operaciones asociadas a estos objetos. En ambos métodos de diseño con frecuencia es difícil decidir dónde se debe empezar.

Para identificar objetos del dominio de solución, una buena forma de iniciar es fijarse en el dominio del problema. De manera más específica, vamos a la definición del problema y buscamos los sustantivos y verbos importantes. Los sustantivos (y sintagmas nominales) podrán sugerir operaciones. Por ejemplo, la definición de problema para un programa de banco podría incluir las siguientes oraciones:

... El programa debe manejar la cuenta de ahorros de un cliente. Se le permite al cliente depositar fondos en la cuenta y retirar fondos de la cuenta, y el banco debe pagar intereses en forma trimestral...

En estas oraciones, los sustantivos clave son

Cuenta de ahorros
Cliente

y los sintagmas verbales clave son

Depositar fondos
Retirar fondos
Pagar intereses

Aunque estamos trabajando con una parte muy pequeña de la definición completa del problema, la lista de sustantivos sugiere dos objetos potenciales: `savingsAccount` y `customer`. La lista de sintagmas verbales sugiere las operaciones en un objeto `savingsAccount` , a saber, `Deposit`, `Withdraw` y `PayInterest`. ¿Cuáles son las operaciones en un objeto de `customer`? Necesitaríamos más información del resto de la definición del problema para contestar a esta pregunta. De hecho, es posible que `customer` no sea de ninguna manera un objeto útil. La técnica de sustantivos y verbos sólo es un punto de partida que nos dirige hacia *potenciales* objetos y operaciones.

La determinación de cuáles son los sustantivos y verbos significativos es uno de los aspectos más difíciles de OOD. No hay una receta para fórmulas de estilo, y tal vez nunca la habrá. No todos los sustantivos se convierten en objetos, y no todos los verbos se convierten en operaciones. La técnica de sustantivos y verbos es imperfecta, pero nos da un primer acercamiento a una solución.

El dominio de solución incluye no sólo objetos tomados del dominio del problema, sino también objetos de *nivel de aplicación*. Éstos son los objetos que no modelan el dominio del problema, pero se usan en la construcción del propio programa. En sistemas con interfaces gráficas de usuario —Microsoft Windows o el sistema operativo de Macintosh, por ejemplo—, un programa quizá necesite varios tipos de objetos de nivel de aplicación: objetos de ventanas, objetos de menú, objetos que responden a un clic del mouse, etc. Este tipo de objetos con frecuencia están disponibles en bibliotecas de clase, así que no tenemos que diseñar y aplicarlos partiendo de cero cada vez que los necesitamos en diferentes programas.

Paso 2: Determinar las relaciones entre objetos

Después de seleccionar los objetos y operaciones potenciales, el siguiente paso es examinar las relaciones entre los objetos. En particular queremos ver si es posible relacionar determinados objetos ya sea por herencia o por composición. Las relaciones por herencia o composición no sólo preparan el camino para la reutilización de código —como hemos destacado en nuestro análisis de OOP—, sino también simplifican el diseño y nos permiten modelar el dominio del problema con mayor precisión. Por ejemplo, es posible que el problema del banco requiera varios tipos de cuentas de ahorro; una para clientes en general, otra para clientes preferidos, y otra para niños menores de 12 años. Si todas estas cuentas son variaciones de una cuenta básica de ahorros, la relación de *es* (y, por tanto, herencia) es probablemente apropiada. Empezando con una clase `SavingsAccount` que proporciona operaciones comunes de cualquier cuenta de ahorro, podríamos diseñar cada una de las otras cuentas como una clase hijo de `SavingsAccount`, concentrando nuestros esfuerzos sólo en las propiedades que hacen a cada una diferente de la clase padre.

Por otro lado, la composición fue la única opción para la clase `Entry`. Un objeto `Entry` contiene un objeto `Time` y un objeto `Name`.

Paso 3: Diseñar el controlador

El paso final es el diseño del controlador, o sea el algoritmo de nivel superior. En OOD, el controlador es el pegamento que une los objetos (junto con sus operaciones). Cuando se aplica el diseño en C++, el controlador se convierte en la función `main`.

Observe que el diseño estructurado *empieza* con el diseño del algoritmo de nivel superior, mientras que OOD *termina* con el algoritmo de nivel superior. En OOD, la mayoría del flujo de control ya fue diseñado en los pasos 1 y 2: los algoritmos están ubicados dentro de las operaciones sobre objetos. Como resultado, con frecuencia el controlador tiene poco que hacer, salvo procesar comandos de usuario o introducir algunos datos, y luego delegar tareas a varios objetos.

Consejo práctico de ingeniería de software

La naturaleza iterativa del diseño orientado a objetos

Algunos desarrolladores de software, investigadores y autores han propuesto muchas estrategias diferentes para realizar OOD. Casi todas tienen en común tres pasos fundamentales:

1. Identificar los objetos y operaciones.
2. Determinar las relaciones entre objetos.
3. Diseñar el controlador.

La experiencia con grandes proyectos de software ha mostrado que estos tres pasos no necesariamente son secuenciales; esto es: paso 1, paso 2, paso 3, y ya está. En la práctica, el paso 1 ocurre primero, pero sólo como un primer acercamiento. Durante los pasos 2 y 3 tal vez se descubren nuevos objetos u operaciones que nos llevan de regreso al paso 1. Es realista pensar en los pasos 1 a 3 no como una secuencia, sino como un ciclo.

Además, cada paso es un proceso iterativo dentro de sí mismo. El paso 1 podrá implicar mucho trabajo para nuestra vista con los objetos y operaciones. De manera similar, con frecuencia los pasos 2 y 3 involucran experimentación y revisión. En cualquier paso podremos concluir que un objeto potencial al final de cuentas no sea útil. O podremos decidir que debamos agregar o eliminar operaciones sobre un determinado objeto.

Siempre hay más de una manera de resolver un problema. La iteración y reiteración a lo largo de la fase de diseño lleva a ideas que producen una mejor solución.

14.7 Implementar el diseño

Cuando identificamos un objeto por primera vez en OOD, se trata de un *objeto abstracto*. No elegimos inmediatamente una representación de datos exacta para este objeto. De modo similar, las operaciones sobre objetos empiezan como *operaciones abstractas* porque no hay ningún intento inicial de proporcionar algoritmos para estas operaciones.

Al final tenemos que aplicar los objetos y operaciones. Para cada objeto abstracto tenemos que

- Elegir una representación de datos adecuada.
- Crear algoritmos para las operaciones abstractas.

Para seleccionar una representación de datos para un objeto, el programador de C++ tiene tres opciones:

1. Usar un tipo de datos integrado.
2. Usar un ADT existente.
3. Crear un nuevo ADT.

Para un objeto dado, una buena regla general es considerar estas tres opciones en el orden enumerado. Un tipo integrado es el tipo más directo para el uso y la comprensión, y operaciones sobre estos tipos ya están definidas por el lenguaje. Si un tipo integrado no es adecuado para representar un objeto, se deberán analizar los ADT disponibles en una biblioteca de clases (ya sea del sistema o la suya propia) para ver si algunos son buenas correspondencias para el objeto abstracto. Si no existe ningún ADT adecuado, se deberá diseñar y aplicar uno nuevo para representar el objeto.

Por fortuna, aun si es necesario hacer a mano la opción 3, los mecanismos de herencia y composición permiten la combinación de las opciones 2 y 3. Cuando necesitábamos una clase `ExtTime`

anteriormente en este capítulo, usábamos la herencia para construir sobre una clase `Time` existente. Y cuando creamos una clase `Entry`, usamos la composición para incluir un objeto `Time` y un objeto `Name` en los datos privados.

En adición a la elección de una representación de datos para el objeto abstracto tenemos que aplicar las operaciones abstractas. En OOD los algoritmos que aplican las operaciones abstractas con frecuencia son cortos y directos. Hemos visto numerosos ejemplos en este capítulo y en los capítulos 11 y 13, donde el código para operaciones ADT consiste sólo en unas cuantas líneas. Pero esto no siempre es el caso. Si una operación es extremadamente compleja, será mejor tratarla como un nuevo problema y usar la descomposición funcional en el flujo de control. En esta situación es apropiado aplicar tanto la descomposición funcional como las metodologías orientadas a objetos en conjunto. Los programadores experimentados están familiarizados con ambas metodologías y las usan o en forma independiente o en combinación. Sin embargo, la comunidad de desarrollo de software está cada vez más convencida de que, a pesar de que la descomposición funcional es importante para diseñar algoritmos de bajo nivel y operaciones en ADT, el futuro en el desarrollo de sistemas de software gigantescos está en OOD y OOP.

Caso práctico de resolución de problemas

Creación de una agenda de citas

PROBLEMA Crear una agenda de citas mediante las siguientes operaciones:

Dada una fecha, inserte un par de hora y nombre.
Dada una fecha, organice los pares de hora y nombre en orden de tiempo.
Dada una fecha y hora, devuelva true si la hora está disponible, y false en caso contrario.

ANÁLISIS ¿Cuáles son los objetos en esta descripción? Una fecha, una hora, un nombre y una colección de pares de hora/nombre para cada fecha. Todas estas operaciones empiezan con "Dada una fecha" e involucran hacer algo con pares de nombre/fecha. Simplifiquemos este problema aplicando antes que nada una lista de pares de nombre/hora. El siguiente paso sería combinar la lista de pares de nombre/hora con una fecha para crear una lista diaria de citas.

Ya tenemos una clase `Name` y una clase `Entry` que consiste en objetos de `Name` y `Time`. Colección es un nombre diferente para una lista. Tenemos una clase `List`. Parece que la solución de este problema es simplemente juntar clases a manera de bloques de construcción. Podemos usar cada una de estas clases directamente, con excepción de la clase `List`.

La clase `List` es una lista de objetos `ItemType`, en la cual `ItemType` es fijado por una instrucción Typedef. En el caso práctico del capítulo anterior hemos usado la clase `List` para retener una lista de enteros. Aquí tenemos una lista de objetos `Entry` que debe ser ordenado por tiempo. Eso está bien; tenemos una clase `SortedList` que podemos usar. Pero no está bien. Las operaciones de `SortedList` que son `Insert` y `BinSearch` involucran la comparación de dos elementos usando los operadores relacionales. Nuestra clase `Time` tiene operadores de comparación `Equal` y `LessThan`; los operadores relacionales no están definidos en objetos `Time`. Esto significa que tenemos que agregar dos funciones a la clase `Entry`: `LessThan` y `Equal`.

Mientras aumentamos la clase `Entry`, vamos a agregar una función `ReadEntry` a la clase. Un objeto `Entry` deberá tener una responsabilidad de acción para leer valores a sí mismo.

He aquí los prototipos para las nuevas funciones:

```
bool LessThan( Entry entry ) const;

      // Compara el tiempo con entry.time
      // Poscondición:
      //     El valor de retorno es verdadero si el tiempo es menor que
      //     entry.time; falso en caso contrario
```

```
bool Equal( Entry entry ) const;

    // Compara el tiempo con entry.time
    // Poscondición:
    //     El valor de retorno es verdadero si el tiempo es igual a
    //     entry.time; falso en caso contrario

void ReadEntry();

    // Poscondición:
    //     Se ha leído los valores desde el teclado y se han guardado
    //     en name y time
```

LessThan(In: entry)

Return time.LessThan(entry.time)

Equal(In: entry)

Return time.Equal(entry.time)

ReadEntry()

time.ReadName()
name.ReadTime()

LISTA ORDENADA ¿Es inútil nuestra clase actual `SortedList`? No; tenemos que reescribir dos de las funciones usando las comparaciones apropiadas `LessThan` y `Equal` de la clase `Entry`, especializando de esta forma la lista, pero podemos usar las operaciones que no requieren una comparación. Hay varias características avanzadas de C++ que nos permiten solucionar este problema por medio de la creación de una lista general. Analizaremos una de ellas en el capítulo 17.

Las operaciones que es necesario reescribir son `Insert` y `BinSearch`. ¿Qué pasa con `Delete`? Nuestro problema no indica la necesidad de borrar una cita, pero deberíamos anticipar esta operación para el futuro. Sin embargo, puesto que `Delete` usa `BinSearch`, no es necesario cambiar. Observe que "item" es de la clase `Entry`, y nuestras comparaciones son del tiempo, así que `LessThan` y `Equal` en la clase `Entry` simplemente llaman a `LessThan` y `Equal` en la clase `Time`.

Insertar(entrada: elemento)

Establecer índice en longitud – 1
WHILE índice ≥ 0 AND item.LessThan(data[index])
    Establecer data[index + 1] en data[index]
    Disminuir el índice
Establecer data[index + 1] en elemento
Incrementar la longitud

BinSearch(entrada: elemento; entrada-salida: hallada, posición)

```
Establecer hallado en falso
Establecer primero en 0
Establecer último en longitud – 1
WHILE último >= primero AND NOT hallado
    Establecer intermedio en (primero + último)/2
    IF item.LessThan(data[middle]
        Establecer último en intermedio – 1
    ELSE IF data[middle].LessThan(elemento)
        Establecer primero en intermedio + 1
    ELSE
        Establecer hallado en verdadero
IF se halló
    Establecer posición en intermedio
```

CONTROLADOR Las llamadas a las funciones `SortedList` para aplicar las tareas van al controlador al mismo instante. Cuando incorporamos lo que hemos hecho aquí en la tarea general, el controlador volverá a ser parte de la clase que manipula los eventos de un día.

Principal

```
Obtener los pares tiempo/nombre
Devolver los pares tiempo/nombre
¿Está libre la hora?
Eliminar
Devolver los pares tiempo/nombre
```

Obtención de pares

Necesitamos un ciclo para la entrada de los pares tiempo/nombre. Dado que es un controlador y no un programa real, podemos establecer un ciclo arbitrario. Digamos cinco pares de entrada.

```
FOR cuenta que va de 1 a 5
    ReadEntry()
    Insertar el elemento en la lista
```

Retorno de pares

Esta opción da acceso a los elementos de la lista. No tenemos idea de qué hará el programa final con los datos, pero podemos permitir al controlador imprimir los pares.

```
Establecer el límite en list.length()
Restablecer la lista
FOR cuenta que va de 1 al límite
    Establecer la entrada en list.GetNextItem()
    Imprimir la entrada
```

Hora libre(entrada: elemento)
Salida: valor de función)

Devolver NOT list.IsPresent(item)

Eliminar(entrada: entry)

list.Delete(entry)

RECAPITULACIÓN La mayor parte del trabajo para este programa fue hecha en otro lado. Nosotros sólo armamos las piezas. La clase `Name` y la clase `Time` ya habían sido verificadas y no fueron alteradas. Hemos agregado tres nuevas funciones a la clase `Entry`, y hemos cambiado dos funciones en la clase `SortedList`. Tenemos que volver a verificar la versión actual de la clase `Entry` antes de intentar ejecutar el controlador. Aquí está el archivo de implementación para la clase extendida `Entry`, el controlador para verificarlo, y la salida del controlador.

```
//**************************************************************************
// ARCHIVO DE EJECUCIÓN (Entry.cpp)
// Este archivo contiene la especificación del TDA Entry,
// que tiene dos clases contenidas, Name y Time
//**************************************************************************

#include "Entry.h"
#include <string>
#include <fstream>
#include <iostream>
#include <sstream>              // ostringstream

using namespace std;

string Entry::NameStr() const

// Devuelve una cadena conformada por nombre, espacio en blanco, apellido
   materno
// Poscondición:
//     El valor de retorno es el nombre del objeto Name, espacio en blanco
//     y el apellido materno del objeto Name

{
    return (name.FirstName() + ' ' + name.LastName());
}

string Entry::TimeStr() const

// Devuelve una cadena constituida por hora, dos puntos, minutos, dos puntos,
   segundos
// Poscondición:
//     El valor de retorno es minutos del objeto Time, dos puntos y
//     segundos del objeto Time
```

```
{
    string outStr;
    ostringstream tempOut;     // Clase ostringstream de acceso
    if (time.Hours() < 10)
        tempOut << '0';
    tempOut << time.Hours() << ":";
    if (time.Minutes() < 10)
        tempOut << '0';
    tempOut << time.Minutes() << ":";
    if (time.Seconds() < 10)
        tempOut << '0';
    tempOut << time.Seconds();
    outStr = tempOut.str();
    return outStr;
}

void Entry::ReadEntry()

// Poscondición:
//     Los valores han sido leídos desde el teclado hacia nombre y tiempo.

{
    name.ReadName();
    time.ReadTime();
}

bool Entry::LessThan(Entry entry) const

// Compara el tiempo con entry.time
// Poscondición:
//     El valor de retorno es verdadero si el tiempo es menor que entry.time;
//     falso en caso contrario

{
    return time.LessThan(entry.time);
}

bool Entry::Equal(Entry entry) const

// Compara el tiempo con entry.time
// Poscondición:
//     El valor de retorno es verdadero si por sí mismo es igual a
       entry.time;
//     falso en caso contrario

{
    return time.Equal(entry.time);
}

Entry::Entry()

// Constructor por omisión
// Poscondición:
//     Se ha construido el objeto de entrada
```

```
{
}

Entry::Entry( /* in */ string firstName,
              /* in */ string middleName,
              /* in */ string lastName,
              /* in */ int  initHours,
              /* in */ int  initMinutes,
              /* in */ int  initSeconds )
    // Inicializadores de constructor
    : name(firstName, middleName, lastName),
      time(initHours, initMinutes, initSeconds)

// Constructor parametrizado
// Poscondición:
//     El objeto de entrada ha sido construido con firstName,
//     middleName y lastName como argumentos para el constructor
//     parametrizado Name; initHours, initMinutes e initSeconds
//     como argumentos para el constructor parametrizado Time

{
}

//**************************************************************************
// DRIVER para la clase extendida Entry (File ExpEntryDr.cpp)
// Este programa prueba los constructores y funciones de retorno
//**************************************************************************

#include <iostream>
#include "Entry.h"
using namespace std;

int main()
{
    Entry entry1;
    Entry entry2("Mary", "Beth", "Jones", 10, 30, 0);

    entry1.ReadEntry();

    // Probar los constructores y ReadEntry
    cout << "Entrada 1: " << entry1.NameStr() << " "
         << entry1.TimeStr() << endl;
    cout << "Entrada 2: " << entry2.NameStr() << " "
         << entry2.TimeStr() << endl;

    // Probar LessThan
    if (entry1.LessThan(entry2))
        cout << "La entrada 1 es menor que la entrada 2" << endl;
    else
        cout << "La entrada 1 no es menor que la entrada 2" << endl;
    if (entry2.LessThan(entry1))
        cout << "La entrada 2 es menor que la entrada 1" << endl;
```

```
    else
        cout << "La entrada 2 no es menor que la entrada 1" << endl;

    // Probar Equal
    if (entry1.Equal(entry2))
        cout << "La entrada 1 es igual a la entrada 2" << endl;
    else
        cout << "La entrada 1 no es igual a la entrada 2" << endl;

    entry1 = entry2;
    cout << "entry1 se establece en entry2" << endl;
    if (entry1.Equal(entry2))
        cout << "La entrada 1 es igual a la entrada 2" << endl;
    else
        cout << "La entrada 1 no es igual a la entrada 2" << endl;

    return 0;
}
```

```
C:\PPSC++\2-22-04\Ch14\EntryTest.exe
Enter first name: Susy
Enter middle name: Sunshine
Enter last name: Smith
Enter hours (<= 23):
9
Enter minutes (<= 59):
30
Enter seconds (<= 59):
0
Entry 1: Susy Smith 09:30:00
Entry 2: Mary Jones 10:30:00
Entry 1 is less than Entry 2
Entry 2 is not less than Entry 1
Entry 1 does not equal Entry 2
entry1 set to entry2
Entry 1 equals Entry 2
```

Ahora que se ha verificado la clase extendida Entry, podemos continuar con el resto del problema. Aquí están las aplicaciones de las funciones cambiadas en la clase `SortedList`.

```
//****************************************************************************

void SortedList::Insert( /* in */ ItemType& item )

// Inserta el elemento en SortedList

// Precondición:
//     longitud < MAX_LENGTH
//  && data[0..length - 1] están en orden ascendente
// Poscondición:
//     el elemento está en SortedList
//  && longitud == length@entry + 1
//  && data[0..length - 1] están en orden ascendente

{
    int index;    // Índice y variable de control de bucle
    index = length - 1;
```

```
    while (index >= 0 && item.LessThan(data[index]))
    {
        data[index+1] = data[index];
        index--;
    }
    data[index+1] = item;          // Insertar elemento
    length++;
}

//****************************************************************************

void SortedList::BinSearch(
      /* in */  ItemType item,              // Elemento que se hallará
      /* out */ bool&    found,             // Verdadero si se halla el
                                               elemento
      /* out */ int&     position ) const   // Ubicación si se halla

// Busca en la lista al elemento y devuelve el índice del elemento si se
// halló.

// Precondición:
//     longitud <= INT_MAX / 2
//  && data[0.. length - 1] están en orden ascendente
// Poscondición:
//     IF el elemento está en la lista
//         hallado = true && data[position] contiene al elemento
//     ELSE
//         hallado = falso && posición indefinida

{
    int first = 0;             // Cota inferior en la lista
    int last = length - 1;     // Cota superior en la lista
    int middle;                // Índice medio

    found = false;
    while (last >= first && !found)
    {
        middle = (first + last) / 2;
        if (item.LessThan(data[middle]))
            // Afirmación: el elemento no está en data[middle..last]
            last = middle - 1;
        else if (data[middle].LessThan(item))
            // Afirmación: el elemento no está en data[first..middle]
            first = middle + 1;
        else
            // Afirmación:_elemento == data[middle]
            found = true;
    }
    if (found)
        position = middle;
}
```

PRUEBA Para verificar la lista de los elementos de la clase `Entry` tenemos que crear otro controlador. A continuación se muestra el controlador que incorpora llamadas a la clase `SortedList`. Las entradas se capturan, se insertan en la lista, se acceden una por una y se imprimen, se comparan con otro Entry y se borran. La pantalla de resultados de este controlador aparece debajo del código.

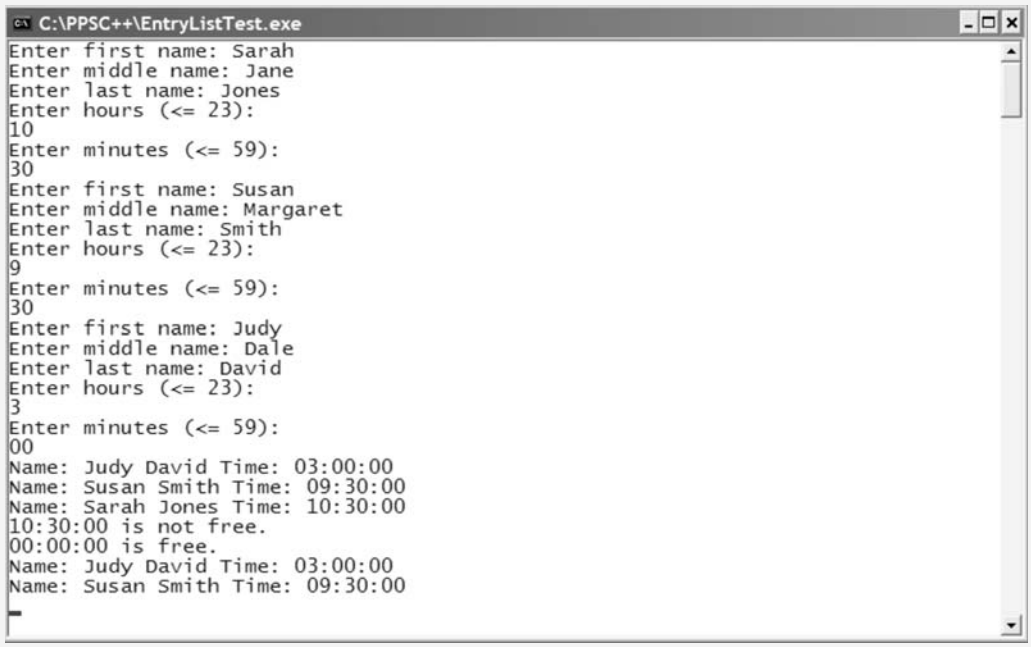
```
//**************************************************************************
// DRIVER para la lista de objetos de clase Entry
// Se leen cuatro entradas desde el teclado y se insertan en la lista
// de entradas. Se tiene acceso a las entradas una a la vez y se imprimen.
// Se comprueba el tiempo que está en la lista; se comprueba un tiempo
// que no está en la lista. Se borra un elemento y se imprime la lista
// para mostrar que se fue el elemento.
//**************************************************************************

#include <iostream>
#include "SortedList.h"
using namespace std;

int main()
{
    int limit;            // Número de elementos en la lista
    Entry nameTime;
    SortedList list;
    for (int count = 1; count <= 3; count++) // Introducir datos
    {
        nameTime.ReadEntry();
        list.Insert(nameTime);
    }

    // Preparar e iterar en la lista, imprimir los elementos
    limit = list.Length();
    list.Reset();         // Preparar para iteración
    for (int count = 0; count < limit; count++)
    {
        nameTime = list.GetNextItem();
        cout << "Nombre: " << nameTime.NameStr() << " Tiempo: "
             << nameTime.TimeStr() << endl;
    }

    // El elemento está en la lista
    if (list.IsPresent(nameTime))
        cout << nameTime.TimeStr() << " no está libre." << endl;
    else
        cout << nameTime.TimeStr() << " está libre. " << endl;

    // El elemento no está en la lista
    Entry entry2("Nell", "Boylan", "Dale", 0, 0, 0);
    if (list.IsPresent(entry2))
        cout << entry2.TimeStr() << " no está libre." << endl;
    else
        cout << entry2.TimeStr() << " está libre. " << endl;

    // Borrar e iterar en la lista, imprimir los elementos
```

```
    list.Delete(nameTime);
    list.Reset();                       // Preparar para iteración
    limit = list.Length();
    for (int count = 0; count < limit; count++)
    {
        nameTime = list.GetNextItem();
        cout << "Nombre: " << nameTime.NameStr() << " Tiempo: "
             << nameTime.TimeStr() << endl;
    }
    return 0;
}
```

```
C:\PPSC++\EntryListTest.exe
Enter first name: Sarah
Enter middle name: Jane
Enter last name: Jones
Enter hours (<= 23):
10
Enter minutes (<= 59):
30
Enter seconds (<= 59):
00
Enter first name: Susan
Enter middle name: Margaret
Enter last name: Smith
Enter hours (<= 23):
9
Enter minutes (<= 59):
30
Enter seconds (<= 59):
00
Enter first name: Judy
Enter middle name: Dale
Enter last name: David
Enter hours (<= 23):
3
Enter minutes (<= 59):
00
Enter seconds (<= 59):
0
Name: Judy David Time: 03:00:00
Name: Susan Smith Time: 09:30:00
Name: Sarah Jones Time: 10:30:00
10:30:00 is not free.
00:00:00 is free.
Name: Judy David Time: 03:00:00
Name: Susan Smith Time: 09:30:00
```

PROBLEMA Cuando tecleamos los datos para el controlador, nos percatamos de que tenemos un problema. La clase `Time` representa horas, minutos y segundos; esto es, representa el tiempo *activo*. Para crear o fijar o leer una hora, el usuario debe capturar horas, minutos y segundos. Existe una operación para incrementar el tiempo por segundo. Para nuestra agenda de citas, ni nos importan los segundos ni el incremento del tiempo por segundo. Los tiempos de citas son diferentes; son tiempos *estáticos*. Sólo estamos interesados en horas y minutos. ¿Cómo resolvemos este problema? Podríamos definir una clase derivada y redefinir las operaciones para trabajar sólo con horas y minutos. Sin embargo, esto no es el uso natural para la herencia. Un tiempo estático no es un tiempo dinámico *y más*. Simplemente vamos a definir una clase `AptTime`, en la que eliminamos la operación de incremento y cambiamos las otras operaciones para que sólo funcionen con horas y minutos. ¿Tenemos que hacer cambios correspondientes en la clase `Entry`? Sí; es necesario eliminar los segundos del constructor y de la cadena de tiempo, y cambiar la sentencia `"Include"`. Dejaremos la codificación y verificación de la clase `AptTime` para los ejercicios de Seguimiento de caso práctico.

A continuación se muestra una imagen de la pantalla de la misma entrada, usando `AptTime` en lugar de `Time`. La única diferencia es que no se le indica al usuario que debe capturar segundos.



Prueba y depuración

La prueba y depuración de un programa orientado a objetos es sobre todo un proceso de verificar y depurar las clases C++ con las cuales está elaborado el programa. El controlador de nivel superior también necesita una verificación, pero en general ésta no es complicada; OOD es un controlador sencillo.

Para repasar cómo verificar una clase de C++, regrese a la sección Pruebas y depuración del capítulo 11. En esa ocasión analizamos detalladamente el proceso de verificar cada función miembro de una clase. Hicimos la observación de que se podía escribir un programa director de pruebas separado para cada función miembro, o se puede escribir sólo un programa director de pruebas que verifica todas las funciones miembro. Este último planteamiento sólo es recomendado para clases que tienen unas cuantas funciones miembro sencillas.

Cuando un programa orientado a objetos usa herencia y composición, el orden en que se verifican las clases está, en cierto sentido, predeterminado. Si la clase X es derivada de la clase Y o contiene un objeto de la clase Y, no se podrá verificar X hasta que se haya diseñado y aplicado Y. De este modo tiene sentido verificar y depurar la clase de nivel inferior (clase Y) antes de verificar la clase X. El Caso práctico de resolución de problemas de este capítulo demostró esta secuencia de verificación. En prmer lugar, verificamos la clase de nivel más bajo, la clase `Time`; luego verificamos la clase `TimeCard`, que contiene un objeto `Time`; por último, verificamos la clase `TimeCardList`, que contiene un array de objetos `TimeCard`. El principio general es que si la clase X está construida sobre la clase Y (mediante herencia o composición), la verificación de X es simplificada si Y ya fue verificada y se sabe que se está comportando correctamente.

Consejos para prueba y depuración

1. Repase los consejos para Prueba y depuración del capítulo 11. Éstos aplican al diseño y verificación de clases C++, que son el corazón de OOP.
2. Cuando use herencia, no olvide incluir la palabra `public` cuando declara la clase derivada:

```
class DerivedClass : public BaseClass
{
 ⋮
};
```

La palabra `public` convierte `BaseClass` en una clase base pública de `DerivedClass`. Esto es que clientes de `DerivedClass` pueden aplicar cualquier operación pública de `BaseClass` (excepto constructores) a objetos `DerivedClass`.

3. El archivo de encabezado que contiene la declaración de una clase derivada debe incluir (`#include`) el archivo de encabezado que contiene la declaración de la clase base.
4. Aunque una clase derivada hereda los miembros privados y públicos de su clase base, no puede acceder directamente a los miembros privados heredados.
5. Si una clase base tiene un constructor, se invocará antes de que se ejecute el cuerpo del constructor de la clase derivada. Si el constructor de clase base requiere argumentos, entonces estos `arguments` se deberán transmitir usando un inicializador de constructor:

```
DerivedClass::DerivedClass( ... )
    ⋮ BaseClass(arg1, arg2)
{
 ⋮
}
```

Si no se incluye un inicializador de constructor, se invocará el constructor por omisión de la clase base.

6. Si una clase tiene un miembro que es un objeto de otra clase, y si el constructor del objeto miembro requiere argumentos, entonces estos argumentos se deberán transmitir usando un inicializador de constructor:

```
SomeClass::SomeClass( ... )
    ⋮ memberObject(arg1, arg2)
{
 ⋮
}
```

Si no hay ningún inicializador de constructor, se invocará el constructor por omisión de la clase base.

7. Para obtener ligadura dinámica de una operación a un objeto cuando se transmiten objetos de clase como argumentos, se debe
   - transmitir el objeto por referencia, no por valor;
   - declarar la operación a ser `virtual` en la declaración de la clase base.
8. Si una clase base declara una función virtual, *debe* aplicar esta función, incluso si el cuerpo está vacío.
9. Una clase derivada no puede redefinir el tipo de devolución de función de una función virtual.

Resumen

El diseño orientado a objetos (OOD) descompone un problema en objetos, o sea entidades independientes en las cuales están vinculados datos y operaciones. En OOD, los datos se tratan como una cantidad activa y no pasiva. Cada objeto es responsable de una parte de la solución, y los objetos comunican invocando las operaciones del otro.

OOD empieza con la identificación de objetos potenciales y sus operaciones. Examinar objetos en el dominio del problema es una buena forma de empezar el proceso. El siguiente paso es determinar las relaciones entre los objetos, usando herencia (para expresar una relación *es*) y composición

(para expresar una relación *tiene*). Finalmente, se diseña un algoritmo de controlador para coordinar el flujo de control en general.

La programación orientada a objetos (OOP) consiste implantar un diseño orientado a objetos mediante el uso de mecanismos de lenguaje para la abstracción de datos, herencia y ligadura dinámica. La herencia permite a cualquier programador tomar una clase existente (clase base) y crear una nueva clase (clase derivada) que hereda los datos y operaciones de la clase base. La clase derivada luego especializa a la clase base, agregando nuevos datos privados, nuevas operaciones, o reimplementando operaciones heredadas; todo esto sin análisis ni modificación de la aplicación en alguna forma de la clase base. La ligadura dinámica de operaciones a objetos permite que objetos de muchos diferentes tipos derivados respondan a un solo nombre de función, cada uno en su propia manera (polimorfismo). En conjunto, la herencia y la ligadura dinámica han demostrado que se reduce de modo drástico el tiempo y esfuerzo requeridos para adaptar ADT existentes. El resultado son componentes de un software verdaderamente reutilizables cuyas aplicaciones y tiempos de vida exceden a los que fueron concebidos por su creador original.

Comprobación rápida

1. ¿En qué se distinguen programas estructurados de programas orientados a objetos en términos de los componentes que interactúan uno con otro para resolver un problema? (pp. 598-600)
2. ¿Cuáles son los tres medios principales que proporciona un lenguaje orientado a objetos que habilitan la programación orientada a objetos? (pp. 598-600)
3. ¿Es una función virtual ligada en forma estática o dinámica? (pp. 619-623)
4. Si usted tiene una clase denominada `Phone` y quiere extenderla para que admita un código de país, nombrando a la nueva clase `InternationalPhone`, ¿cómo escribiría el encabezado de clase para la nueva subclase? (pp. 603-606)
5. Suponga que en la Comprobación rápida 4 usted quería crear `InternationalPhone` como una clase separada que contuviera una instancia de `Phone` en lugar de que fuera una subclase de phone. Describa cómo haría esto. (pp. 612-619)
6. ¿Cuáles son los tres pasos principales en el diseño de una solución orientada a objetos para un problema? (p. 623)
7. Dado el diseño para un objeto que enlista los datos que contiene y proporciona algoritmos para las operaciones que realiza, ¿qué concepto de C++ usaría usted para representar el objeto, sus datos y sus operaciones? (pp. 600-601)

Respuestas

1. En un programa estructurado, las funciones interactúan entre sí y con los datos en el programa. En un programa orientado a objetos, son los objetos (que conectan a las operaciones y datos) los que interactúan. **2.** Abstracción de datos, herencia y ligadura dinámica. **3.** En forma dinámica. **4.** `class InternationalPhone : public Phone`. **5.** Colocando un campo miembro dentro de `InternationalPhone`, que es de la clase `Phone`. 6. Identificar los objetos y operaciones, determinar las relaciones entre los objetos, diseñar el controlador. 7. El objeto sería representado por una clase. Sus datos serían representados por campos miembro de la clase. Las operaciones del objeto serían representados por funciones miembro de la clase.

Ejercicios de preparación para examen

1. Relacione los siguientes términos con las definiciones que se dan abajo.
   - ***a***) Programa estructurado
   - ***b***) Programa orientado a objetos
   - ***c***) Herencia
   - ***d***) Superclase
   - ***e***) Subclase
   - ***f***) Composición
   - ***g***) Ligadura estática

h) Ligadura dinámica
i) Polimorfismo
  i) Determinar, en tiempo de ejecución, desde qué clase llamar una función.
  ii) Una clase de la cual se adquieren propiedades.
  iii) Una colección de clases diseñadas usando abstracción, herencia y polimorfismo.
  iv) Una operación que tiene diferentes significados dependiendo de su ligadura a un objeto.
  v) Incluir un objeto de una clase dentro de otra clase.
  vi) Una clase que adquiere propiedades de otra clase.
  vii) Determinar, en tiempo de compilación, desde qué clase llamar una función.
  viii) Una colección de funciones, diseñada usando la descomposición funcional.
  ix) Adquirir las propiedades de otra clase.

2. La programación estructurada es más apropiada para desarrollar programas pequeños, mientras que la programación orientada a objetos es mejor para escribir programas grandes. ¿Verdadero o falso?
3. La herencia nos permite reutilizar funciones de una clase base y agregar funciones nuevas. Pero no podemos remplazar funciones en la clase base con nuevas aplicaciones. ¿Verdadero o falso?
4. Para resolver el problema del corte usamos una combinación de paso por referencia y funciones virtuales. ¿Verdadero o falso?
5. Si queremos que una función sea virtual, ¿dónde escribimos la palabra clave? ¿En el archivo de declaración de la clase base, en el archivo de definición de la clase base, en el archivo de declaración de la clase derivada, o en el archivo de definición de la clase derivada?
6. ¿Qué mecanismo de lenguaje de C++ aplica la composición?
7. Suponga que tiene una subclase que contiene varios miembros de datos que no están definidos en su superclase. ¿Qué pasa con estos miembros de datos si usted asigna un objeto de la subclase a una variable de la superclase?
8. A un cliente se le entrega la siguiente declaración para una clase base y una clase derivada.

```
class BaseClass
    {
    public:
        void PrintFields() const;
          ⋮
    };

    class DerivedClass : BaseClass
    {
    public:
        void NewFunction();
        DerivedClass( /* in */ int StartValue);
          ⋮
    };
```

El cliente escribe entonces el siguiente código para llamar el constructor para un objeto del tipo `DerivedClass` y luego imprime los campos en el objeto de nueva creación.

```
DerivedClass anObject(10);
    anObject.PrintFields();
```

Pero el compilador reporta un error para la segunda sentencia. ¿Cuál es el problema? ¿Cómo lo resolvería usted?

9. Considere las siguientes declaraciones de clases de base y derivada.

```
class BaseClass
    {
    public:
```

```
    void BaseAlpha();
private:
    void BaseBeta();
    float baseField;
};

class DerivedClass : public BaseClass
{
public:
    void DerivedAlpha();
    void DerivedBeta();
private:
    int derivedField;
};
```

Para cada clase, haga lo siguiente:

a) Liste todos los miembros de datos privados.
b) Liste todos los miembros de datos privados que las funciones miembro de la clase pueden referenciar de manera directa.
c) Liste todas las funciones que las funciones miembro de la clase podrán invocar.
d) Liste todas las funciones miembro que un cliente de la clase pueda invocar.

10. Una clase denominada `DerivedClass` es una subclase de una clase nombrada `BaseClass`. `DerivedClass` también tiene un campo miembro que es un objeto de la clase `ComposedClass`.
a) ¿Al constructor de qué clase se le llama primero cuando se crea un objeto de la clase `DerivedClass`?
b) ¿Al constructor de qué clase se le llama al último cuando se crea un objeto de la clase `DerivedClass`?

11. ¿Por qué ocurre el corte en el paso por valor, pero no en el paso por referencia cuando se pasa un objeto de una clase derivada a un parámetro de su clase base?

12. ¿Cuál es el problema en las siguientes declaraciones de clase?

```
class BaseClass
{
public:
    virtual void BaseAlpha();
private:
    float baseField;
};

class DerivedClass : public BaseClass
{
public:
    virtual void BaseAlpha();
private:
    int derivedField;
};
```

13. Explique la diferencia entre una relación *es* y una relación *tiene*.

14. Cuando codificamos una clase derivada en archivos separados, ¿cuál de las siguientes actividades realizamos?
a) Incluir la especificación de clase base y el archivo de implementación tanto en el archivo de especificación de la clase derivada como en sus archivos de implementación.
b) Incluir el archivo de especificación de la clase base en el archivo de especificación de la clase derivada, e incluir el archivo de implementación de la clase base en el archivo de implementación de la clase derivada.

c) Incluir el archivo de especificación de la clase base en el archivo de especificación de la clase derivada.
d) Incluir el archivo de implementación de la clase base en el archivo de implementación de la clase derivada.

Ejercicios de calentamiento para programación

1. Dada la siguiente declaración para una clase de resultados de exámenes, escriba una declaración de clase derivada llamada `IDScore`, que agrega un número entero de identificación de estudiante como un miembro privado y que proporciona un constructor con parámetros correspondientes a los tres campos miembro, así como un observador que devuelve el número de identificación.

```
class TestScore
    {
    public:
        TestScore(/* in */ string name,
                  /* in */ int score);
        string GetName() const;
        int GetScore() const;
    private:
        string studentName;
        int studentScore;
    };
```

2. Escriba el archivo de implementación para la clase `TestScore` en el ejercicio 1. El constructor sólo asigna sus parámetros a los miembros de datos privados, y el observador simplemente devuelve el miembro correspondiente.
3. Escriba el archivo de implementación para la clase `IDScore` en el ejercicio 1. El constructor sólo asigna sus parámetros a los miembros de datos privados, y el observador simplemente devuelve el miembro correspondiente.
4. Escriba el archivo de especificación para una clase llamada `Exam`, que usa la composición para crear un array de 100 objetos de la clase `IDScore`, como se define en el ejercicio 1. La clase puede usar el constructor por omisión y debe tener una función que asigna un objeto `IDScore` a una ubicación en el array, dados los objetos y las ubicaciones como parámetros. La clase también deberá tener un observador que devuelve el objeto `IDScore` a la posición especificada por su parámetro.
5. Escriba el archivo de implementación para la clase `Exam`, como se define en el ejercicio 4.
6. La siguiente clase representa un número de teléfono en Estados Unidos.

```
// SPECIFICATION FILE (phone.h)

    enum PhoneType (HOME, OFFICE, CELL, FAX, PAGE);

    class Phone
    {
    public:
          Phone( /* in */ int newAreaCode,
                 /* in */ int newNumber,
                 /* in  */ PhoneType newType);
        void Write () const;
    private:
        int areaCode;
        int number;
        PhoneType type;
    }
```

Usando la herencia, queremos derivar una clase de números internacionales de teléfono, `InternPhone`, de la clase `Phone`. Para este ejercicio supondremos que el único cambio necesario es agregar un código de país (un entero) que identifica el país o región. Las operaciones públicas de `InternPhone` son `Write`, que reimplementa la función `Write` de la clase base, y un constructor de clase que lleva cuatro parámetros correspondientes a los cuatro campos miembro en la clase. Escriba la declaración de clase para la clase `InternPhone`.

7. Implemente el constructor de clase `InternPhone` como se describe en el ejercicio 6.
8. Implemente la función Write de la clase `InternPhone`, como se describe en el ejercicio 6.
9. Escriba una función global `WritePhone` que toma un solo parámetro y que usa la ligadura dinámica para imprimir ya sea un número de teléfono de Estados Unidos (`PhoneClass`) o un número de teléfono internacional (clase `InternPhone`). Haga el cambio (los cambios) necesario(s) en la declaración de la clase `Phone` del ejercicio 6, de modo que `WritePhone` ejecute correctamente.
10. Dada la siguiente declaración para una clase que representa una computadora en el inventario de una empresa, escriba una declaración de clase derivada (para una clase llamada `InstallRecord`) que agrega un campo de cadena representando la ubicación de la computadora, así como un campo de la clase `SimpleDate` que contiene la fecha de instalación. La nueva clase deberá proporcionar observadores para cada uno de los nuevos campos. También deberá reimplementar la función `Write`.

```
class Computer
    {
    public:
        Computer(
          /* in */ string newName,
          /* in */ string newBrand,
          /* in */ string newModel,
          /* in */ int    newSpeed,
          /* in */ string newSerial,
          /* in */ int    newNumber);
        string GetName() const;
        string GetBrand() const;
        string GetModel() const;
        int GetSpeed() const;
        string GetSerial() const;
        int GetNumber() const;
        void Write() const;
    private:
        string name;
        string brand;
        string model;
        int speed;
        string serialNumber;
        int inventoryNumber;
    };
```

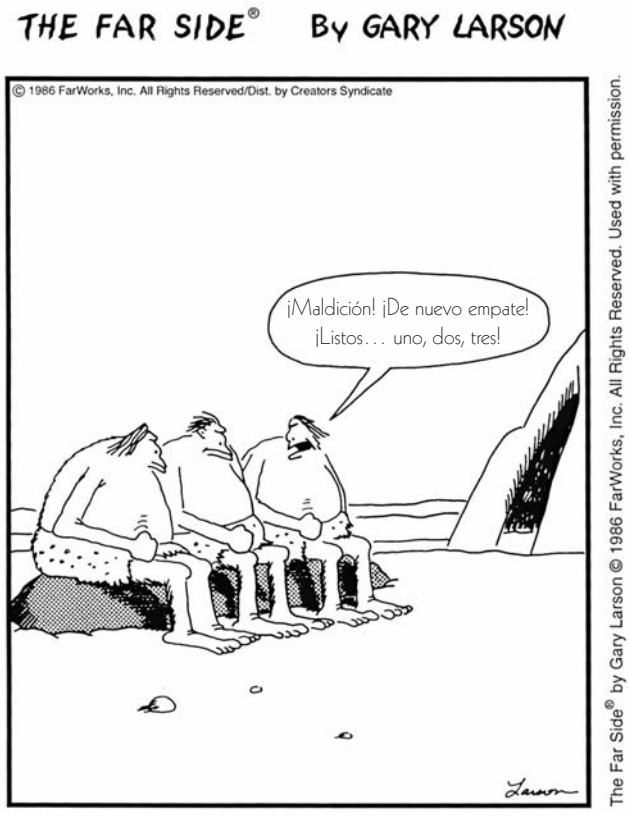
11. Implemente el constructor para la clase `Computer` declarada en el ejercicio 10.
12. Implemente el constructor para la clase `InstallRecord` declarada en el ejercicio 10.
13. Implemente la función `Write` para la clase `Computer` declarada en el ejercicio 10. Deberá emitir cada campo miembro en una línea separada.
14. Implemente la función Write para la clase `InstallRecord` declarada en el ejercicio 10. Deberá emitir cada campo miembro en una línea separada. Suponga que la clase `SimpleDate` proporcione una función `void` llamada `Write ( )` que emite la fecha en un formato estándar.
15. Extienda la clase `ExtTime` mediante la redefinición de la función `ReadTime`.

Problemas de programación

1. Use la programación orientada a objetos para desarrollar una aplicación de juego que simule una mesa de ruleta. La mesa de ruleta tiene 36 números (1 a 36) que están arreglados en tres columnas de 12 filas. La primera fila tiene los números 1 a 3, la segunda fila contiene 4 a 6, etc. También existe un número 0 que se encuentra fuera de la tabla de números. Los números en la tabla son de color rojo y negro (0 es verde). Los números rojos son 1, 3, 5, 7, 9, 12, 14, 16, 18, 19, 21, 23, 25, 27, 30, 32, 34, 36. La otra mitad de los números es negra. En un juego de reglas simplificadas, los jugadores pueden apostar a un número individual (incluyendo 0), los números rojos, los números negros, los números pares, los números impares, los números de 1 a 18, los números de 19 a 36, así como a cualquiera de las columnas o filas en la mesa.

   Al usuario se le deberá permitir que entre a una de las apuestas, y la aplicación usa la función rand de `<cstdlib>` como la base para calcular el número que resultaría en la rueda. Luego compare este número con la apuesta y reporte si ganó o perdió. El proceso se repite hasta que el usuario ingrese un comando de cancelación.
2. Use la programación orientada a objetos para desarrollar una extensión de la aplicación del Problema 1. La nueva aplicación también deberá permitir que el usuario introduzca una cantidad inicial de dinero en una cuenta. Además de jugar una apuesta, el usuario especifica una cantidad para la apuesta. Esta cantidad se deduce de la cuenta. Todas las ganancias se suman a la cuenta. Las ganancias o pérdidas actuales (la diferencia del monto original) se deberá visualizar además del valor de la cuenta. Las ganancias se deberán calcular del siguiente modo:

   Apuestas a un número inicial pagan 36 veces el monto jugado
   Apuestas a filas pagan 12 veces
   Apuestas a columnas pagan 3 veces
   Apuestas a par/impar, rojo/negro, mitad superior/inferior pagan 2 veces el monto jugado
   No se debe permitir que el usuario apueste más de lo que tiene en su cuenta.



Antes de papel y tijeras

3. Use la programación orientada a objetos para desarrollar una aplicación de juego para el juego de niños "piedra, papel, tijeras". El usuario introduce una letra que indica su elección. Cuando se introduce una elección, la función `rand` de `<cstdlib>` se usa para escoger un valor en el rango de 1 a 3, donde 1 corresponde a piedra, 2 a papel, y 3 a tijeras. La elección de la computadora es comparada con la elección del usuario, de acuerdo con estas reglas: la piedra rompe las tijeras, las tijeras cortan el papel, el papel envuelve la piedra. Elecciones iguales son empate. Emita un conteo de las victorias por parte del usuario y de la computadora, así como de los empates. La aplicación termina cuando el usuario introduce una elección inválida.
4. Use la programación orientada a objetos para desarrollar una extensión del Problema 3. La nueva aplicación deberá aceptar la original letra individual o las palabras completas (piedra, papel, tijeras). No importa si se usan mayúsculas o minúsculas. También aquí la aplicación extendida sólo deberá terminar cuando el usuario introduzca "q" o "quit" como elección, y deberá pedir que el usuario introduzca una de las elecciones válidas si se introduce un valor inválido.
5. Use las clases `Computer` e `InstallRecord` declaradas en los Ejercicios de calentamiento para programación 10 a 14, junto con la clase `SortedList` desarrollada en el capítulo 13 como base para un programa orientado a objetos que mantiene el inventario de las computadoras de una empresa. La empresa tiene un máximo de 500 computadoras. Se deberán admitir las siguientes operaciones:

   Agregar una nueva computadora a la lista.
   Borrar una computadora de la lista.
   Cambiar la ubicación de una computadora.
   Imprimir una lista de todas las computadoras en el orden de números de inventario.
   Imprimir una lista de todas las computadoras en una determinada ubicación.
   Imprimir una lista de todas las computadoras de una determinada marca.
   Imprimir una lista de todas las computadoras instaladas antes de una determinada fecha.

   La aplicación deberá mantener la lista ordenada por números de inventario. Deberá leer una lista inicial de un archivo denominado `original.dat`. Al final del procesamiento deberá escribir los datos actuales en la lista en un archivo nombrado `update.dat` en un formato que el programa podrá volver a capturar como una lista original. También será necesario desarrollar la clase `SimpleDate` hasta el grado que admita la aplicación.

Seguimiento de caso práctico

1. Implemente y verifique la clase `AptTime`.
2. No es apropiado que `AptTime` (un tiempo estático) herede de `Time` (un tiempo activo). ¿Sería apropiado que un tiempo activo herede de un tiempo estático? Explique.
3. Cuando examina el resultado del controlador, usted puede ver que las horas se ven extrañas. 03:00 es técnicamente menos que 09:30, pero seguramente la hora de la cita es 03:00 de la tarde y debería venir después de las 09:30. El tiempo se especifica como un reloj de 24 horas. Se le podría decir al usuario que introduzca las horas basándose en un reloj de 24 horas, o se le podría cambiar la clase `AptTime` para tener un campo adicional que contenga p.m. o a.m., o se podría derivar una clase de `AptTime` que tenga un campo de p.m. o a.m. Piense en esta solución y explique qué se debería cambiar para implantar el tiempo en un reloj de 12 horas.
4. Implemente su solución del Problema 3.
5. Escriba un controlador y verifique la nueva clase desarrollada en el Problema 4.
6. Sustituya la nueva clase del Problema 4 en el controlador para el Caso práctico y vuelva a realizar esta prueba.

CAPÍTULO

15

Apuntadores, datos dinámicos y tipos de referencia

Objetivos

Objetivos de conocimiento

- *Entender cómo se pueden usar apuntadores para mejorar la eficiencia de los programas.*
- *Entender la diferencia entre operaciones de copiado profundas y superficiales.*
- *Entender cómo C++ define el término de inicialización.*
- *Conocer y entender las cuatro funciones miembro que deberán estar presentes en todas las clases que manipulan datos dinámicos.*

Objetivos de habilidades

Ser capaz de:

- *Declarar variables de tipos de apuntadores.*
- *Tomar las direcciones de variables y acceder a las variables por medio de apuntadores.*
- *Escribir una expresión que seleccione un miembro de una clase, estructura o unión indicada por un apuntador.*
- *Crear y acceder a datos dinámicos.*
- *Destruir datos dinámicos.*
- *Declarar e inicializar variables de tipos de referencia.*
- *Acceder a variables que son referenciadas por tipos de referencia.*

Tipo de apuntador Tipo de datos simples que consiste en un conjunto ilimitado de valores, cada uno de los cuales dirige o indica de otra manera la ubicación de una variable de un tipo dado. Entre las operaciones definidas sobre variables de apuntadores se encuentra la asignación y la prueba para igualdad.

En los capítulos anteriores hemos visto los tipos simples y los tipos estructurados en C++. Sólo quedan por abordar dos tipos de datos integrados: **tipos de apuntador** y tipos de referencia (véase la figura 15-1). Estos son tipos de datos simples, pero en la figura 15-1 los enlistamos en forma separada de los otros tipos simples, porque su propósito es especial. Nos referimos a tipos de apuntadores y tipos de referencia como *tipos de direcciones*. Una variable de uno de estos tipos no contiene ningún valor de datos, sino que contiene la *dirección de memoria* de otra variable o estructura. Los tipos de dirección tienen dos propósitos principales: pueden hacer un programa más eficiente —ya sea en términos de velocidad o en términos de uso de memoria— y se pueden usar para construir estructuras de datos complejas. En este capítulo demostraremos de qué modo pueden hacer un programa más eficiente. El capítulo 16 explica cómo construir estructuras complejas usando tipos de direcciones.

15.1 Apuntadores

En muchos sentidos hemos dejado lo mejor para el final. Los tipos de apuntadores son los tipos de datos más interesantes. Los apuntadores son lo que su nombre indica: variables que indican dónde encontrar otra cosa; esto es, los apuntadores contienen las direcciones o ubicaciones de otras variables.

Empezaremos este análisis viendo cómo se declaran variables de apuntadores en C++.

Variables de apuntadores

Sorprendentemente la palabra *pointer* no se usa en la declaración de variables de apuntadores. En su lugar se usa el símbolo *. La declaración

```
int* intPtr;
```

enuncia que `intPtr` es una variable que puede apuntar a (es decir, contener la dirección de) una variable `int`. A continuación se muestra la plantilla de sintaxis para declarar variables de apuntadores:

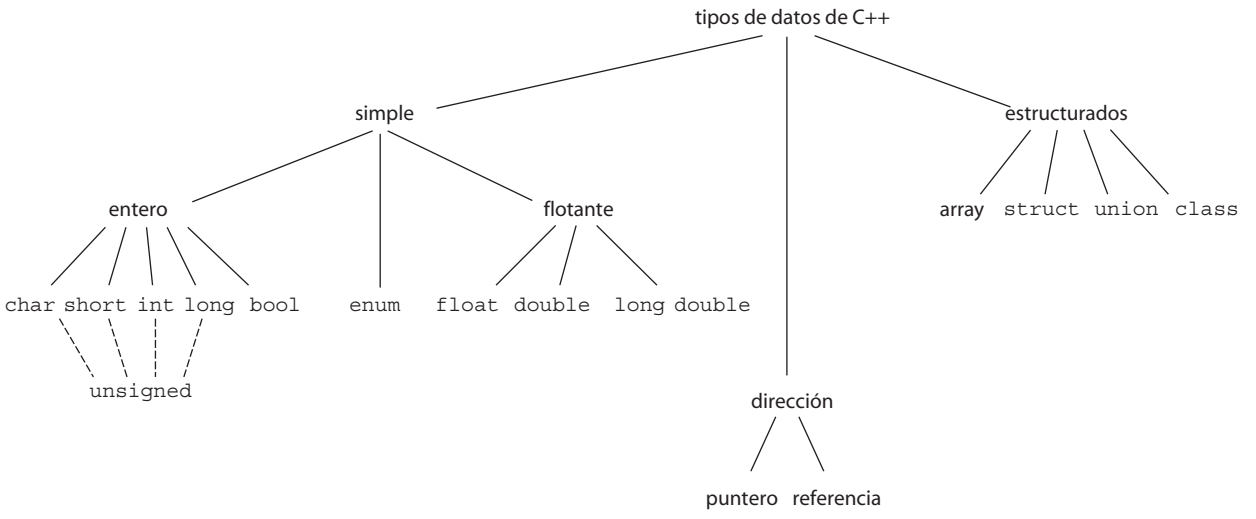


Figura 15-1 *Tipos de datos en C++*

PointerVariableDeclaration

```
DataType* Variable ;

DataType *Variable , *Variable . . . ;
```

La plantilla de sintaxis muestra dos formas: una para declarar una variable individual y otra para declarar varias variables. En la primera forma, no le importa al compilador dónde se coloca el asterisco. Las dos declaraciones siguientes son equivalentes:

```
int* intPtr;
int *intPtr;
```

Aunque los programadores de C++ usan ambos estilos, nosotros preferimos el primero. Agregar el asterisco al nombre del tipo de datos en lugar de al nombre de la variable sugiere en seguida que `intPtr` es del tipo "apuntador a `int`".

De acuerdo con la plantilla de sintaxis, si se declaran varias variables en una sentencia, a cada nombre de variable le deberá preceder un asterisco. En caso contrario, sólo la primera variable se tomará como una variable de apuntador. El compilador interpreta la sentencia

```
int* p, q;
```

como si estuviera escrito de esta manera:

```
int* p;
int  q;
```

Para evitar errores no intencionados en la declaración de variables de apuntadores, la forma más segura es declarar cada variable en una sentencia separada.

Dadas las declaraciones

```
int  beta;
int* intPtr;
```

Podemos hacer que `intPtr` apunte a `beta`, usando el operador unario `&`, lo que se llama operador *address-of* (*dirección de*). Al tiempo de ejecución, la proposición de asignación

```
intPtr = &beta;
```

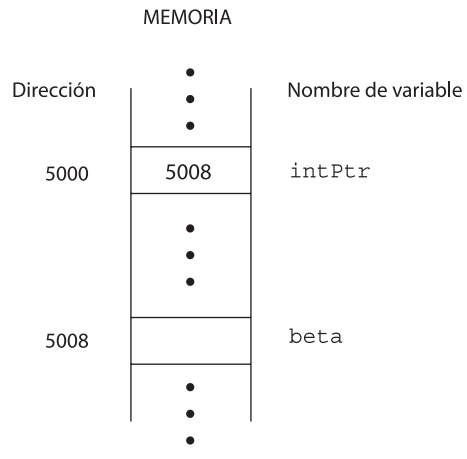


Figura 15–2 *Vista de nivel de máquina de una variable de apuntador*

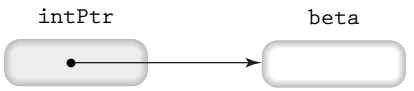


Figura 15–3 *Diagrama abstracto de una variable de apuntador*

toma la dirección de memoria de `beta` y la guarda en `intPtr`. De manera alternativa podríamos inicializar `intPtr` en su declaración del siguiente modo:

```
int  beta;
int* intPtr = &beta;
```

Supongamos que `intPtr` y `beta` estuviesen ubicados en las direcciones de memoria 5000 y 5008, respectivamente. En este caso, guardar la dirección de `beta` en `intPtr` resulta en la relación que se ve representada en la figura 15-2.

Puesto que las verdaderas direcciones numéricas son, en general, desconocidas por el programador de C++, es más común visualizar la relación entre un apuntador y una variable a la que se apunta usando rectángulos y flechas, como se ve en la figura 15-3.

Para acceder a una variable a la que indica un apuntador, usamos el operador unario *, o sea el operador de *desreferencia* o de *indirección*. La expresión

```
*intPtr
```

denota la variable a la que apunta `intPtr`. En nuestro ejemplo, `intPtr` realmente apunta a `beta`, así que la sentencia

```
*intPtr = 28;
```

Direccionamiento indirecto Acceso a una variable en dos pasos, usando primero un apuntador que proporciona la localización de la variable.

Direccionamiento directo Acceso a una variable en un paso, usando el nombre de la variable.

desreferencia `intPtr` y guarda el valor 28 en `beta`. Esta sentencia representa un **direccionamiento indirecto** de `beta`; la máquina accede primero a `intPtr`, y luego usa su contenido para localizar a `beta`. En contraste, la sentencia

```
beta = 28;
```

representa el **direccionamiento directo** de `beta`. El direccionamiento directo es como abrir un buzón apartado de la oficina de correos (por ejemplo el buzón de A.P. 15) para encontrar un paquete, mientras que el direccionamiento indirecto es como abrir el buzón de A.P. 15 para encontrar una nota que dice que tu paquete está esperando en el buzón de A.P. 23.

Continuamos con nuestro ejemplo. Si ejecutamos las instrucciones

```
*intPtr = 28;
cout << intPtr << endl;
cout << *intPtr << endl;
```

entonces el resultado es

```
5008
28
```

La primera sentencia de resultado imprime el contenido de `intPtr` (5008); la segunda imprime el contenido de la variable a la que apunta `intPtr` (28).

Veamos un ejemplo más elaborado para declarar apuntadores, tomar direcciones y desreferenciar apuntadores. El siguiente fragmento de programa declara varios tipos y variables. En su código, la clase de `TimeType` es la clase de C++ que hemos desarrollado en el capítulo 11, con las funciones miembro `Set`, `Increment`, `Write`, `Equal` y `LessThan`.

```
#include "timetype.h"   // Para clase TimeType
   ⋮
enum ColorType {RED, GREEN, BLUE};
struct PatientRec
{
    int idNum;
    int height;
    int weight;
};

int        alpha;
ColorType  color;
PatientRec patient;
TimeType   startTime(8, 30, 0);
int*        intPtr = &alpha;
ColorType*  colorPtr = &color;
PatientRec* patientPtr = &patient;
TimeType*   timePtr = &startTime;
```

Las variables `intPtr`, `colorPtr`, `patientPtr` y `timePtr` son todas variables de apuntador. `intPtr` apunta a (contiene la dirección de) una variable del tipo `int`, `colorPtr` apunta a una variable del tipo `Color`, `patientPtr` apunta a una variable de estructura del tipo `PatientRec`, y `timePtr` apunta a un objeto de clase del tipo `TimeType`.

La expresión *`intPtr` denota la variable a la que apunta `intPtr`. La variable a la que se apunta puede contener cualquier valor `int`. La expresión *`colorPtr` denota una variable del tipo `ColorType`. Puede contener `RED`, `GREEN` o `BLUE`. La expresión `*patientPtr` denota una variable de estructura del tipo `PatientRec`. Adicionalmente, las expresiones `(*patientPtr).idNum`, `(*patientPtr).height` y `(*patientPtr).weight` denotan los miembros `idNum`, `height` y `weight` de *`patientPtr`. Observe cómo la expresión está construida.

`patientPtr`	Una variable apuntador del tipo "apuntador a `PatientRec`"
`*patientPtr`	Una variable estructura del tipo `PatientRec`
`(*patientPtr).weight`	El miembro `weight` de una variable estructura del tipo `PatientRec`

La expresión `(*patientPtr).weight` es una mezcla de desreferenciado de apuntadores y selección de miembros de estructura. Los paréntesis son necesarios porque el operador punto tiene una precedencia más alta que el operador de desreferencia (véase el Apéndice B para la precedencia de operadores en C++). Sin los paréntesis, la expresión `*patientPtr.weight` se interpretaría mal como *(`patientPtr.weight`).

Cuando un apuntador apunta a una variable estructura (o a una clase, o a una unión), encerrar la desreferencia del apuntador entre paréntesis puede ser tedioso. En adición al operador punto, C++ proporciona otro operador de selección de miembros: ->. Este *operador de flecha* consiste en dos símbolos consecutivos: un guión y un símbolo de mayor que. Por definición,

PointerExpression -> MemberName

es equivalente a

(*PointerExpression) . MemberName

Por tanto, podemos escribir `(*patientPtr).weight` como `patientPtr ->weight`.

El lineamiento general para elegir entre los dos operadores de selección de miembros (punto y flecha) es:

Use el operador punto si el primer operando denota una variable de estructura, clase o unión; use el operador flecha si el primer operando denota un apuntador a una variable de estructura, clase o unión.

Si se quiere incrementar e imprimir el objeto de clase `TimeType` apuntado por `timePtr`, sería posible usar ya sea las sentencias

```
(*timePtr).Increment();
(*timePtr).Write();
```

o las sentencias

```
timePtr->Increment();
timePtr->Write();
```

Y si hubiésemos declarado un arreglo de apuntadores

```
PatientRec* patPtrArray[20];
```

y si hubiésemos inicializado los elementos del arreglo, entonces podríamos acceder al miembro `idNum` del cuarto paciente como sigue:

```
patPtrArray[3]->idNum
```

Las variables a las que se apunta se pueden usar de la misma manera que cualquiera otra variable. Todas las sentencias siguientes son válidas:

```
*intPtr = 250;
*colorPtr = RED;
patientPtr->idNum = 3245;
patientPtr->height = 64;
patientPtr->weight = 114;
patPtrArray[3]->idNum = 6356;
patPtrArray[3]->height = 73;
patPtrArray[3]->weight = 185;
```

En la figura 15-4 se muestran los resultados de estas asignaciones.

En este momento usted tal vez se estará preguntando por qué se deben usar los apunadores. En lugar de hacer que `intPtr` apunte a `alpha` y de guardar 250 en `*intPtr`, ¿por qué no sólo guardar 250 directamente en `alpha`? La verdad es que no hay ninguna buena razón para programar de esta forma; al contrario, los ejemplos que se han mostrado harían que un programa fuese más tortuoso y confuso. El principal uso de apuntadores en C++ es para manipular *variables dinámicas*, o sea variables que llegan a existir en el momento de ejecución, y sólo cuando se necesitan. Más adelante, en este capítulo, mostraremos cómo usar apuntadores para crear variables dinámicas. Mientras tanto, continuemos con algunos de los aspectos básicos de los apuntadores mismos.

Expresiones con apuntadores

En los primeros capítulos hemos aprendido que una expresión aritmética está compuesta por variables, constantes, símbolos de operadores y paréntesis. De modo similar, las expresiones con apuntadores están compuestas por variables apuntadores, constantes apuntadores, ciertos operadores permisibles, y paréntesis. Ya hemos analizado las variables apuntadores, o sea variables que retienen direcciones de otras variables. Ahora abordaremos las constantes apuntadores.

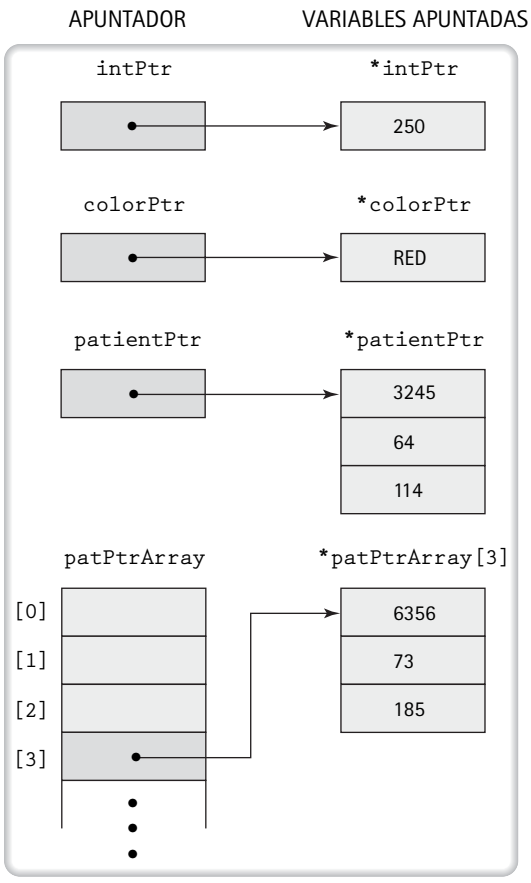
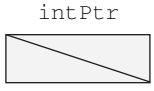


Figura 15–4 *Resultados de declaraciones de asignación*

En C++ sólo existe una constante literal con apuntador: la variable 0. La constante apuntador, llamada *apuntador nulo*, apunta a absolutamente nada. La sentencia

```
intPtr = 0;
```

guarda el apuntador nulo en `intPtr`. Esta sentencia *no* ocasiona que `intPtr` apunte a la ubicación de memoria cero; se garantiza que el apuntador nulo es distinto de cualquier dirección de memoria real. Puesto que el apuntador nulo no apunta a nada, diagramamos el apuntador nulo como sigue, en lugar de usar una flecha para apuntar a algún lugar:



En lugar de usar la constante 0, muchos programadores prefieren usar la constante nombrada `NULL`, que es proporcionada por el archivo de encabezado estándar `cstddef`:

```
#include <cstddef>
  ⋮
intPtr = NULL;
```

Como cualquier constante nombrada, el identificador `NULL` hace que un programa tenga mayor autodocumentación. Su uso también reduce la posibilidad de confundir el apuntador nulo con la constante entera 0.

Es un error dereferenciar el apuntador nulo, ya que no apunta a nada. El apuntador nulo se usa sólo como un valor especial para la verificación de un programa:

```
if (intPtr == NULL)
    DoSomething();
```

Más adelante, en este capítulo, y en el capítulo 16, veremos ejemplos para el uso del apuntador nulo.

Aunque 0 es la única constante literal del tipo apuntador, hay otra expresión con apuntador que es considerada como una expresión con apuntador de constante: un nombre de arreglo sin ningún corchete de índice. El valor de esta expresión es la dirección base (la dirección del primer elemento) del arreglo. Dadas las declaraciones

```
int  arr[100];
int* ptr;
```

la sentencia de asignación

```
ptr = arr;
```

tiene exactamente el mismo efecto que

```
ptr = &arr[0];
```

Ambas sentencias guardan la dirección base de `arr` en `ptr`.

Aunque en su momento no lo explicamos, usted ya usó un nombre de arreglo sin corchetes, que es una expresión con apuntador. Considere el siguiente código que llama una función `ZeroOut` para eliminar un arreglo cuyo tamaño se da como segundo argumento:

```
int main()
{
    float velocity[30];
      ⋮
    ZeroOut(velocity, 30);
      ⋮
}
```

En la llamada de función, el primer argumento —un nombre de arreglo sin corchetes de índice— es una expresión de apuntador. El valor de esta expresión es la dirección base del arreglo `velocity`. Esta dirección base se transmite a la función. Podemos escribir la función `ZeroOut` en una de dos formas. El primer planteamiento —que usted ha visto muchas veces— declara que el primer parámetro de un arreglo sea de un tamaño no especificado.

```
void ZeroOut( /* out */ float arr[],
              /* in */  int   size  )
{
    int i;

    for (i = 0; i < size; i++)
        arr[i] = 0.0;
}
```

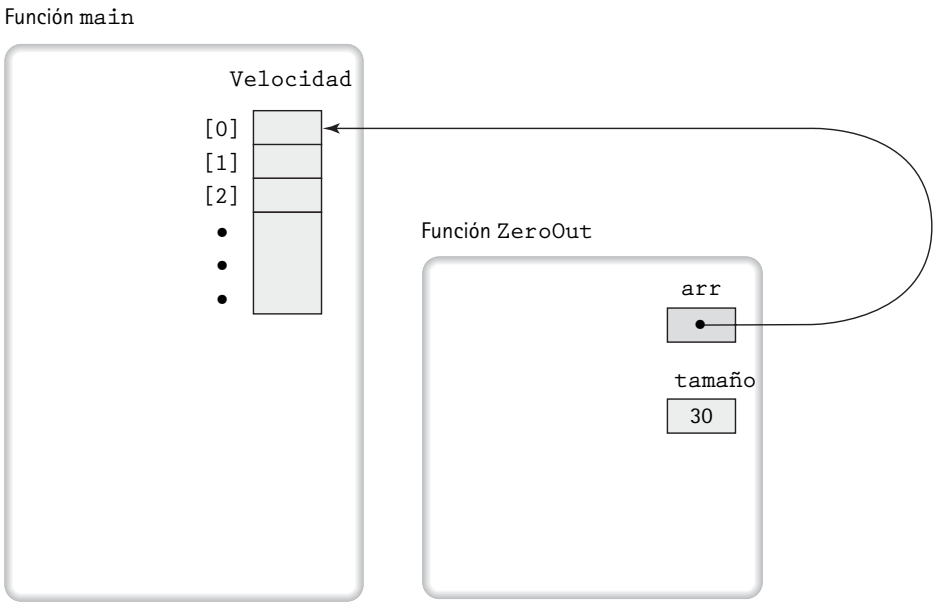


Figura 15-5 *Un parámetro apuntando al argumento del invocador*

Podemos declarar también que el parámetro sea del tipo `float*` porque el parámetro sólo retiene la dirección de una variable `float` (la dirección del primer elemento del arreglo).

```
void ZeroOut( /* out */ float* arr,
              /* in */  int    size )
{
    ⋮   // El cuerpo de la función permanece sin cambio
}
```

Si declaramos el parámetro como `float arr[ ]` o como `float* arr`, el resultado es el mismo para el compilador de C++: dentro de la función `ZeroOut`, `arr` es una simple variable que apunta al inicio del arreglo real del invocador (véase la figura 15-5).

A pesar de que `arr` es una variable apuntador dentro de la función `ZeroOut`, se nos permite agregar una expresión de índice al nombre `arr`:

```
arr[i] = 0.0;
```

La siguiente regla en C++ hace posible el indizado de una variable apuntador:

> El indizado es válido para toda expresión con apuntador, no sólo un nombre de arreglo. (Sin embargo, el indizado de un apuntador sólo tiene sentido si el apuntador apunta a un arreglo.)

Ahora hemos visto cuatro operadores de C++ válidos para apuntadores: =, *, -> y [ ]. La siguiente tabla contiene las operaciones más comunes que pueden ser aplicadas a apuntadores.

Operador	Significado	Ejemplo	Notas

`=`	Asignación	`ptr = &someVar;`
		`ptr1 = ptr2;`
		`ptr = 0;`
			Con excepción del apuntador nulo, ambos operandos tienen que ser del mismo tipo de datos.

`*`	Desreferencia	`*ptr`	
`==, !=, <, <=, > y >=`	Operadores relacionales	`ptr1 == ptr2`	Los dos operandos tienen que ser del mismo tipo de datos.
`!`	Lógico NO	`!ptr`	El resultado es `true` si el operando es 0 (el apuntador nulo); en caso contrario, el resultado es `false`.
`[]`	Índice (o subíndice)	`ptr[4]`	El apuntador indizado deberá apuntar a un arreglo.
`->`	Selección de miembro	`ptr->height`	Selecciona un miembro de la variable de clase, estructura o unión a que se apunta.

Observe que el operador de NO lógico se puede usar para la prueba de existencia del apuntador nulo:

```
if ( !ptr )
    DoSomething();
```

Hay personas que piensan que esta notación es confusa porque `ptr` es una expresión con apuntador, no una expresión booleana. Nosotros preferimos expresar la prueba de la siguiente manera por motivos de claridad:

```
if (ptr == NULL)
    DoSomething();
```

Cuando observamos la tabla, es importante recordar que las operaciones enumeradas son operaciones acerca de apuntadores, *no* sobre las variables que se apuntan. Por ejemplo, si `intPtr1` e `intPtr2` son variables del tipo `int*`, la prueba

```
if (intPtr1 == intPtr2)
```

compara los apuntadores, y no lo que apuntan. En otras palabras, estamos comparando direcciones de memoria, no `int`s. Para comparar los enteros a los que apuntan `intPtr1` e `intPtr2`, tendríamos que escribir:

```
if (*intPtr1 == *intPtr2)
```

En adición a los operadores que hemos enlistado en la tabla, se pueden aplicar los siguientes operadores de C++ a apuntadores: `++`, `--`, `+`, `-`, `+=` y `-=`. Estos operadores realizan aritmética en apuntadores que apuntan a arreglos. Por ejemplo, la expresión `ptr++` hace que `ptr` apunte al siguiente elemento del arreglo, independientemente del tamaño en bytes de cada elemento del arreglo. Y la expresión `ptr+5` accede al elemento del arreglo que está cinco elementos más allá del elemento realmente apuntado por `ptr`. No vamos a decir más acerca de estos operadores ni sobre la aritmética de los apuntadores; el tema de la aritmética de apuntadores está fuera de lo que queremos destacar en el presente capítulo. Ahora procederemos a examinar uno de los usos más importantes de los apuntadores: la creación de datos dinámicos.

15.2 Datos dinámicos

En el capítulo 8 hemos descrito dos categorías de datos de programa en C++: datos estáticos y datos automáticos. Toda variable global es estática, como lo es también toda variable local explícitamente declarada como `static`. El tiempo de vida de una variable estática es el tiempo de vida del programa completo. En contraste, una variable automática, o sea una variable local no declarada como `static`, es asignada (creada) cuando el control llega a su declaración, y borrada (destruida) cuando el control sale del bloque en el cual se declara la variable.

Datos dinámicos Variables creadas durante la ejecución de un programa por medio de operaciones especiales. En C++, estas operaciones son `new` y `delete`.

Con la ayuda de apuntadores, C++ proporciona una tercera categoría de datos de programa: los **datos dinámicos**. Las variables dinámicas no se declaran mediante declaraciones de variables usuales; ellas son explícitamente asignadas y borradas al tiempo de ejecución por medio de dos operadores especiales: `new` y `delete`. Cuando un programa requiere una variable adicional, usa `new` para asignarla. El tiempo de vida de una variable dinámica es, por ende, el tiempo entre la ejecución de `new` y la ejecución de `delete`. La ventaja de poder crear nuevas variables a la hora de la ejecución es que no tenemos que crear más de las que necesitamos.

La operación `new` tiene dos formas, una para asignar una sola variable, y otra para asignar un arreglo. A continuación se muestra la plantilla de sintaxis:

AllocationExpression

```
{ new DataType
{ new DataType [ IntExpression ]
```

La primera forma se usa para crear una sola variable del tipo DataType. La segunda crea un arreglo cuyos elementos son del tipo DataType; el número deseado de elementos del arreglo es dado por IntExpression. A continuación un ejemplo que demuestra ambas formas de la operación `new`:

```
int*  intPtr;
char* nameStr;

intPtr = new int;        Crea una variable del tipo int y guarda su dirección en intPtr.
nameStr = new char[6];   Crea un arreglo char de seis elementos y guarda la dirección base del
                         arreglo en nameStr.
```

Normalmente el operador `new` hace dos cosas: crea una variable no inicializada (o un arreglo) del tipo designado, y devuelve un apuntador a esta variable (o a la dirección base de un arreglo). Sin embargo, si el sistema de la computadora ya no tiene espacio disponible para datos dinámicos, el programa termina con un mensaje de error.*

Se dice que las variables creadas por `new` se encuentran en el **almacenamiento libre** (o **montículo**), una región de memoria apartada para variables dinámicas. El operador `new` obtiene una sección de memoria del almacenamiento libre y, como veremos, el operador `delete` lo devuelve al almacenamiento libre.

Almacenamiento libre (montículo) Agrupación de ubicaciones de memoria reservada para la asignación y omisión de datos dinámicos.

* Técnicamente hablando, si el operador `new` concluye que ya no hay más memoria disponible, crea lo que se llama una excepción `bad_alloc`, un tema que abordaremos en el capítulo 17. A menos que escribamos un código de programa adicional para ocuparnos explícitamente de esta excepción de `bad_alloc`, el programa simplemente termina con un mensaje como "ABNORMAL PROGRAM TERMINATION".

En C++ pre-estándar, se usaba un planteamiento completamente distinto. Si `new` ya no encontraba más memoria disponible, devolvía el apuntador nulo en lugar de un apuntador a un objeto de asignación nueva. El código que invoca a `new` podía entonces verificar el apuntador devuelto para ver si la asignación tenía éxito.

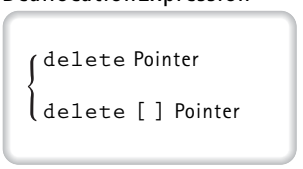
Una variable dinámica no está nombrada y no puede ser direccionada de manera directa. Se tiene que dirigir de modo indirecto por medio del apuntador devuelto por el operador `new`. A continuación se presenta un ejemplo para la creación de datos dinámicos y luego el acceso a los datos por medio de apuntadores. El código empieza por la inicialización de las variables apuntadores en sus declaraciones.

```
#include <cstring>    // Para strcpy()
  ⋮
int*  intPtr = new int;
char* nameStr = new char[6];

*intPtr = 357;
strcpy(nameStr, "Ben");
```

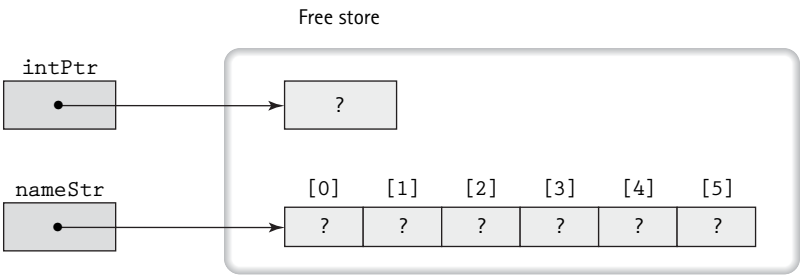
Recuerde, del capítulo 13, que la función de biblioteca `strcpy` requiere dos argumentos, cada uno de los cuales es la dirección base de un arreglo `char`. Para el primer argumento transferimos la dirección base del arreglo dinámico al almacenamiento libre. Para el segundo argumento el compilador transfiere la dirección base del arreglo anónimo donde se ubica la cadena C "Ben" (incluyendo el carácter nulo de fin). Las figuras 15-6*a* y *b* muestran el efecto de la ejecución de este segmento de código.

Los datos dinámicos pueden ser destruidos en cualquier momento durante la ejecución de un programa, cuando ya no se necesitan más. El operador integral `delete` se usa para destruir una variable dinámica. La operación `delete` tiene dos formas, una para borrar una sola variable, y otra para borrar un arreglo:

DeallocationExpression



a.
```
int*  intPtr = new int;
char* nameStr = new char[6];
```



b.
```
*intPtr = 357;
strcpy(nameStr, "Ben");
```

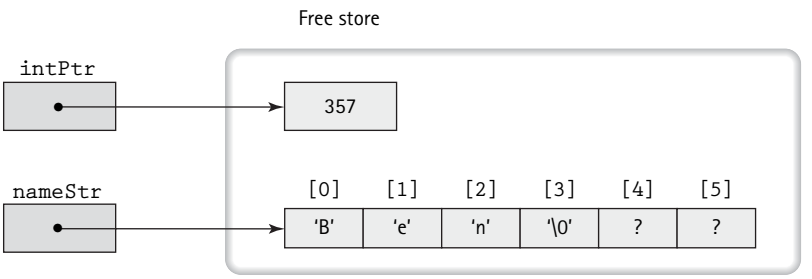


Figura 15-6 *Asignación dinámica de datos en el montículo*

Retomando el último ejemplo, se pueden omitir los datos dinámicos a los que apuntan `intPtr` y `nameStr` por medio de las sentencias siguientes.

`delete intPtr;` Devuelve la variable a que apunta `intPtr` al almacenamiento libre para ser reutilizada. El valor de `intPtr` es entonces indefinido.

`delete [] nameStr;` Devuelve la variable a que apunta `nameStr` al almacenamiento libre para ser reutilizada. El valor de `nameStr` es entonces indefinido.*

Los valores de `intPtr` y `nameStr` están indefinidos después de ejecutar estas instrucciones; es posible que todavía apunten o no a los datos borrados. Antes de volver a usar estos apuntadores, usted les debe asignar nuevos valores (esto es, guardar nuevas direcciones de memoria en ellos).

Hasta que usted adquiera un poco de experiencia con los operadores `new` y `delete`, es importante que pronuncie correctamente la sentencia

```
delete intPtr;
```

En lugar de decir "`Borrar intPtr`", es mejor decir "Borrar la variable a la que *apunta* `intPtr`". La operación `delete` no borra el apuntador, sino borra la variable a la que se apunta.

Cuando se usa el operador `delete`, se deberán recordar dos reglas.

1. No hace ningún daño cuando se aplica `delete` al apuntador nulo; la operación simplemente queda sin efecto.
2. Con excepción de la regla 1, el operador `delete` sólo se debe aplicar a un valor de apuntador que fue obtenido previamente del operador `new`.

Es importante recordar la segunda regla. Si se aplica `delete` a una dirección de memoria arbitraria que no se encuentra en el almacén libre, el resultado es indefinido y podría ser muy desagradable.

Por último, recuerde que una razón importante para usar datos dinámicos es la de economizar el espacio en memoria. El operador `new` permite la creación de variables sólo cuando se necesitan. Cuando se deja de usar una variable dinámica, se deberá `borrar`. Es contraproducente mantener variables dinámicas cuando ya no se necesitan, situación que se denomina **fuga de memoria**. Si esto se hace con excesiva frecuencia, se podrá quedar sin memoria.

Fuga de memoria Pérdida de espacio de memoria disponible que ocurre cuando los datos dinámicos son asignados, pero nunca borrados.

Veamos otro ejemplo del uso de datos dinámicos.

```
int* ptr1 = new int;     // Crear una variable dinámica
int* ptr2 = new int;     // Crear una variable dinámica

*ptr2 = 44;              // Asignar un valor a una variable dinámica
*ptr1 = *ptr2;           // Copiar una variable dinámica en otra
ptr1 = ptr2;             // Copiar un apuntador en otro
delete ptr2;             // Destruir una variable dinámica
```

A continuación se muestra una descripción más detallada del efecto de cada sentencia:

`int* ptr1 = new int;` Crea un par de variables dinámicas del tipo `int` y guarda sus ubicaciones en `ptr1` y `ptr2`.

`int* ptr2 = new int;` Los valores de las variables dinámicas son indefinidos pese a que las variables apuntadores ahora tienen valores (véase la figura 15-7*a*).

* Es posible que la sintaxis para borrar un arreglo, `delete [ ] nameStr`, no sea aceptada por algunos compiladores pre-estándar. Las primeras versiones del lenguaje C++ requerían que el tamaño del arreglo se incluyera entre los corchetes: `delete [6] nameStr`. Si su compilador no acepta los corchetes vacíos, incluya el tamaño del arreglo.

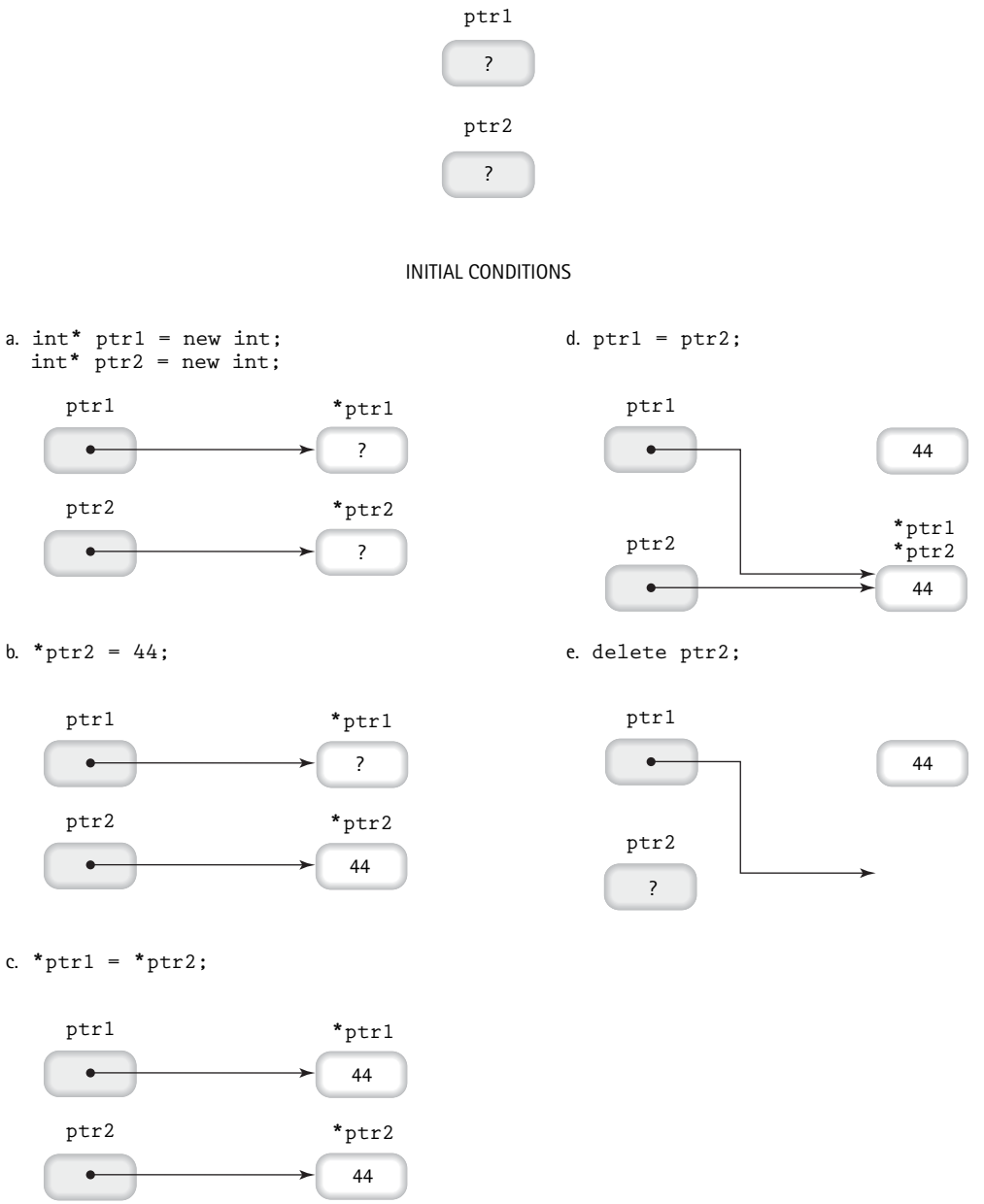


Figura 17–7 *Resultado de la muestra de segmento de código*

`*ptr2 = 44;`	Guarda el valor 44 en la variable dinámica a la que apunta `ptr2` (véase la figura 15-7*b*).
`*ptr1 = *ptr2;`	Copia el contenido de la variable dinámica `*ptr2` a la variable dinámica `*ptr1` (véase la figura 15-7*c*).
`ptr1 = ptr2;`	Copia el contenido de la variable apuntador `ptr2` a la variable apuntador `ptr1` (véase la figura 15-7*d*).
`delete ptr2;`	Devuelve la variable dinámica `*ptr2` al almacén libre para ser reutilizada. El valor de `ptr2` es indefinido (véase la figura 15-7*e*).

En la figura 15-7*d*, observe que la variable a la que apunta `ptr1` antes de su sentencia de asignación sigue ahí. No se puede acceder, sin embargo, porque no hay apuntador que apunte a ella. Esta

variable aislada se llama **objeto inaccesible**. Dejar objetos inaccesibles en el almacén libre podría ser considerado como un error de lógica, y es una causa de fugas de memoria.

Objeto inaccesible Variable dinámica en el almacén libre sin ningún apuntador que le apunte.

Apuntador suspendido Apuntador que apunta a una variable que ha sido borrada.

Observe también que en la figura 15-7*e*, `ptr1` está ahora apuntando a una variable que, por principio, ya no existe. A `ptr1` le nombramos **apuntador suspendido**. Si el programa posteriormente borra `ptr1`, el resultado es imprevisible. Es posible que el valor a que se apunta siga siendo el original (44), o se puede tratar de un valor diferente almacenado ahí como resultado de reutilización de este espacio en el almacén libre.

Las dos situaciones que se muestran en la figura 15-7*e* —un objeto inaccesible y un apuntador suspendido— se pueden evitar borrando *`ptr1` antes de asignar `ptr2` a `ptr1`, y poniendo `ptr1` a `NULL` después de borrar *`ptr2`. (Véase el código a continuación.)

```
#include <cstddef>     // Para NULL
  ⋮
int* ptr1 = new int;
int* ptr2 = new int;

*ptr2 = 44;
*ptr1 = *ptr2;
delete ptr1;           // Evitar un objeto inaccesible
ptr1 = ptr2;
delete ptr2;
ptr1 = NULL;           // Evitar un apuntador suspendido
```

La figura 15-8 muestra los resultados de ejecución de este segmento de código revisado.

15.3 Tipos de referencia

De acuerdo con la figura 15-1, sólo resta un tipo integral: el **tipo de referencia**. A igual que las variables de apuntadores, las variables de referencia contienen las direcciones de otras variables. La sentencia

Tipo de referencia Tipo de datos simples que consiste en un conjunto ilimitado de valores, cada uno de los cuales es la dirección de una variable de un tipo dado. La única operación definida en una variable de referencia es la inicialización, después de la cual cada apariencia de la variable es implícitamente desreferenciada.

```
int& intRef;
```

declara que `intRef` es una variable que puede contener la dirección de una variable `int`. A continuación se muestra la plantilla de sintaxis para declarar variables de referencia:

ReferenceVariableDeclaration

```
DataType& Variable ;

DataType &Variable , &Variable ... ;
```

A pesar de que tanto las variables de referencia como las variables de apuntadores contienen direcciones de objetos de datos, hay dos diferencias fundamentales. En primer lugar, los operadores de desreferencia y de dirección (* y `&`) no se usan con variables de referencia. Después de que una variable de referencia ha sido declarada, el compilador desreferencia de manera *invisible* cada aparición de esta variable de referencia. Esta diferencia se ilustra en la página 660.

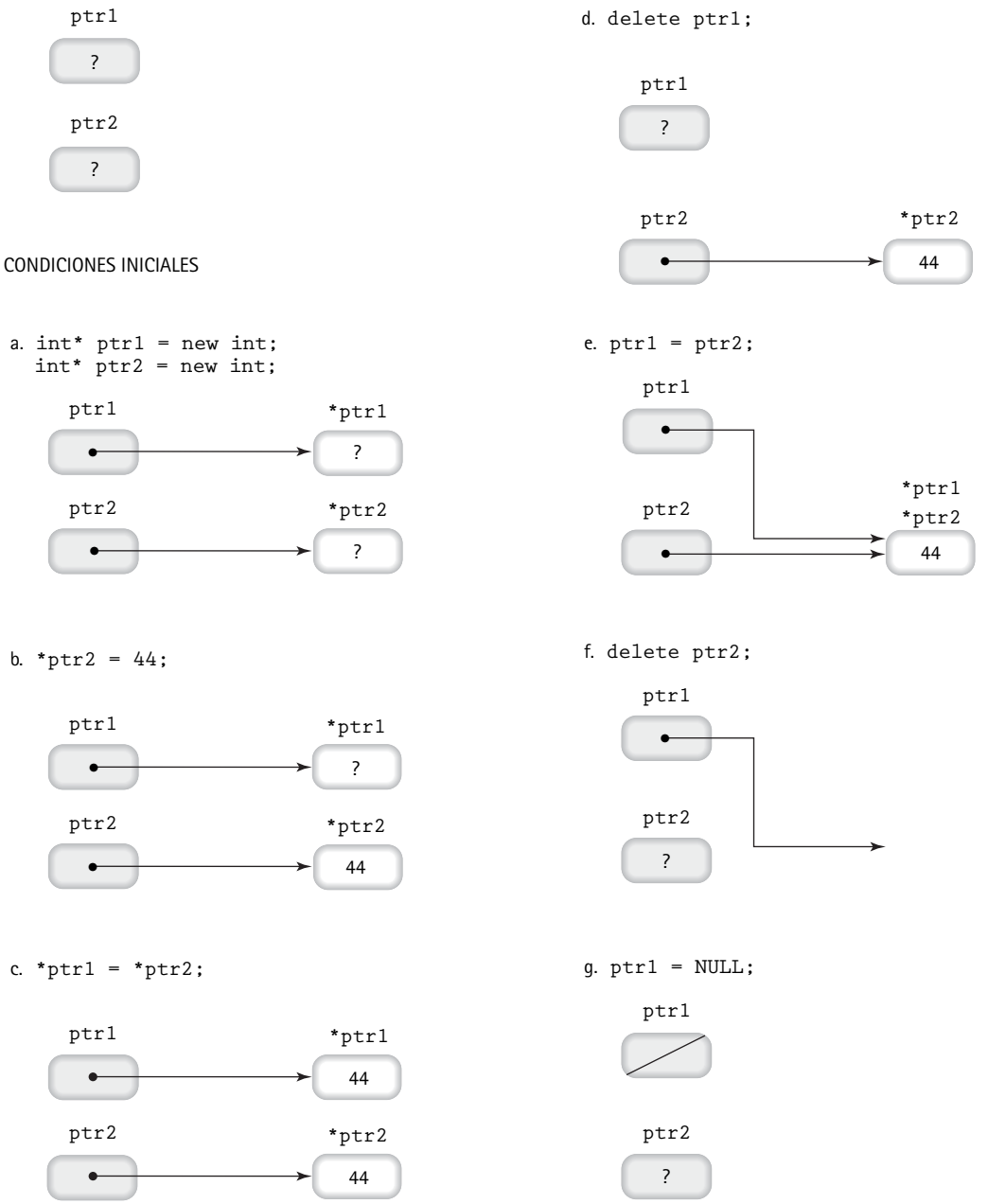


Figura 15-8 *Resultados del segmento de código de muestra después de su modificación*

Usar una variable de referencia

```
int  gamma = 26;
int& intRef = gamma;
// Afirmación: intRef apunta
//          a gamma

intRef = 35;
// Afirmación: gamma == 35

intRef = intRef + 3;
// Afirmación: gamma == 38
```

Usar una variable de apuntador

```
int  gamma = 26;
int* intPtr = &gamma;
// Afirmación: intPtr apunta
//          a gamma

*intPtr = 35;
// Afirmación: gamma == 35

*intPtr = *intPtr + 3;
// Afirmación: gamma == 38
```

Algunos programadores piensan en una variable de referencia como un *alias* para otra variable. En el código que antecede, podemos pensar en `intRef` como un alias para `gamma`. Después de que `intRef` es inicializado en su declaración, todo lo que hacemos a `intRef` en realidad le está sucediendo a `gamma`.

La segunda diferencia entre variables de referencia y variables de apuntadores es que el compilador trata a una variable de referencia como si fuera un apuntador *constante*. No es posible reasignar después de ser inicializado. De hecho, absolutamente ninguna operación aplica de modo directo a una variable de referencia, excepto la inicialización. (En este contexto, C++ define que la inicialización significa *a*) inicialización explícita en una declaración; *b*) inicialización implícita mediante la transferencia de un argumento a un parámetro, o *c*) inicialización implícita mediante la devolución de un valor de función. Por ejemplo, la sentencia

```
intRef++;
```

no incrementa `intRef`; más bien incrementa la variable a la que apunta `intRef`. ¿Por qué? Porque el compilador implícitamente desreferencia cada apariencia del nombre `intRef`.

La ventaja principal de las variables de referencia, pues, es la conveniencia de notación. A diferencia de las variables de apuntadores, las variables de referencia no requieren que el programador prefije constantemente la variable con un asterisco para acceder a la variable a que se apunta.

Un uso común de las variables de referencia es transmitir los argumentos que no son de arreglos por referencia en lugar de transmitirlos por valor (como lo hemos hecho desde el capítulo 7). Supongamos que el programador quiera intercambiar el contenido de dos variables `float` mediante la llamada de función

```
Swap(alpha, beta);
```

Puesto que C++ normalmente transmite las variables simples por valor, el siguiente código fracasará:

```
void Swap( float x, float y )
// Caution: This routine does not work
{
    float temp = x;

    x = y;
    y = temp;
}
```

Por omisión, C++ transmite los dos argumentos por valor. Esto quiere decir que se transmiten a la función *copias* de los valores de `alpha` y `beta`. Los contenidos locales de `x` y `y` son intercambiados dentro de la función, pero los argumentos del invocador `alpha` y `beta` permanecen sin cambios. Para corregir esta situación tenemos dos opciones. La primera es transmitir las direcciones de `alpha` y `beta` explícitamente, usando la dirección del operador (`&`):

```
Swap(&alpha, &beta);
```

Ahora la función tendrá que declarar los parámetros como variables de apuntadores:

```
void Swap( float* px, float* py )
{
    float temp = *px;

    *px = *py;
    *py = temp;
}
```

Este planteamiento es necesario en el lenguaje C, que sí tiene variables de apuntadores, pero no tiene variables de referencia.

La otra opción es usar variables de referencia para eliminar la necesidad para el desreferenciado explícito:

```
void Swap( float& x, float& y )
{
    float temp = x;

    x = y;
    y = temp;
}
```

En este caso, la llamada de función no requiere la dirección del operador (`&`) para los argumentos:

```
Swap(alpha, beta);
```

El compilador implícitamente genera código para transmitir las direcciones, mas no los contenidos, de `alpha` y `beta`. Este método de transmitir argumentos de no arreglos por referencia es el que hemos estado usando desde el principio y que continuaremos usando en todo el libro.

Ya se habrá dado cuenta de que el símbolo "`&`" tiene varios significados en el lenguaje C++. Para evitar errores, vale la pena mantener la separación de estos significados. A continuación se presenta una tabla que resume los diferentes usos del símbolo "`&`". Observe que un operador de *prefijo* es el que precede a su(s) operando(s); un operador *infijo* se encuentra entre sus operandos, y un operador *sufijo* viene después de su(s) operando(s).

Posición	Uso	Significado
Prefijo	&Variable	Dirección de operación
Infijo	Expresión & Expresión	Operación lógica AND a nivel de bits (mencionado, pero no abordado en el capítulo 10)
Infijo	Expresión && Expresión	Operación lógica AND
Sufijo	Tipos de datos &	Tipo de datos (en forma específica, un tipo de referencia) *Excepción*: Para declarar dos variables del tipo de referencia, el `&` debe ser agregado a cada nombre de variable: `int &var1, &var2;`

15.4 Clases y datos dinámicos

Cuando los programadores usan clases de C++, con frecuencia es útil para objetos de clase que crean datos dinámicos en el almacén libre. Consideremos una clase `Message` que consiste en un objeto `Time` y un mensaje.

Para mantener el mensaje simple, proporcionaremos sólo un mínimo de funciones miembro públicas. Empezamos con la declaración de clase para una clase `Message`, abreviado por omitir las precondiciones y poscondiciones de función.

```
class Message
{
public:
    void Print() const;                    // Operación de salida
    Message( /* in */ Time        time,    // Constructor
             /* in */ const char* msgStr );
```

```
private:
    Time time;
    char* msg;
};
```

En la lista de parámetros del constructor de clase igual podríamos haber declarado `msgStr` como

```
const char[] msgStr
```

en lugar de

```
const char* msgStr
```

Recuerde que ambas declaraciones son equivalentes en cuanto se refiere al compilador C++. Ambas significan que el parámetro que se recibe es la dirección base de una cadena C.

Como se puede ver en la parte privada de la declaración de clase, la variable privada `msg` es un apuntador, no un arreglo `char`. Si declarásemos que `msg` fuese un arreglo de tamaño fijo de 30, podría ser que el array fuera demasiado largo o demasiado pequeño para retener la cadena `msgStr` que el cliente transmite a través de la lista de argumentos del constructor. En su lugar, el constructor de clase dinámicamente asignará un arreglo `char` exactamente del tamaño correcto en el almacén libre y hará que `msg` apunte a este arreglo. Aquí está la ejecución del constructor de clase como aparecería en el archivo de ejecución:

```
#include <cstring>   // Para strcpy() y strlen()
  :
 Message::Message( /* in */ Time        newTime,    // Constructor
                   /* in */ const char* msgStr );
{
    time = newTime
    msg = new char[strlen(msgStr) + 1];
    // Afirmación:
    //     El almacenamiento para la cadena C dinámica está ahora en
    //     almacenamiento libre y su dirección base es msg
    strcpy(msg, msgStr);
    // Afirmación:
    //     La cadena entrante ha sido copiada al almacenamiento libre
}
```

El constructor empieza copiando el primer parámetro de entrada en la variable privada apropiada. Luego usamos el operador `new` para asignar un arreglo `char` en el almacén libre. (Sumamos 1 a la longitud de la cadena de entrada para dejar espacio para el carácter final `'\0'`.) Por último, usamos `strcpy` para copiar todos los caracteres del arreglo `msgStr` al nuevo arreglo dinámico. Si el código del cliente declara dos objetos de clase mediante las sentencias

```
Message msg1(10, 30, 0, "Call Bobby");
Message msg2(10, 35, 30, "Call Sue");
  :
```

entonces los dos objetos de clase apuntan a arreglos char dinámicos, como se muestra en la figura 15-9.

La figura 15-9 ilustra un concepto importante: un objeto de clase `Message` no encierra un arreglo; sólo encierra el *acceso* al arreglo. El arreglo propio está ubicado externamente (en el almacén libre), no dentro de la barrera protectora de abstracción del objeto de clase. No obstante, este arreglo no viola el principio de ocultación de información. El único acceso al arreglo es a través de la variable de apuntador `msg`, que es un miembro de clase privado y, por tanto, es inaccesible para clientes.

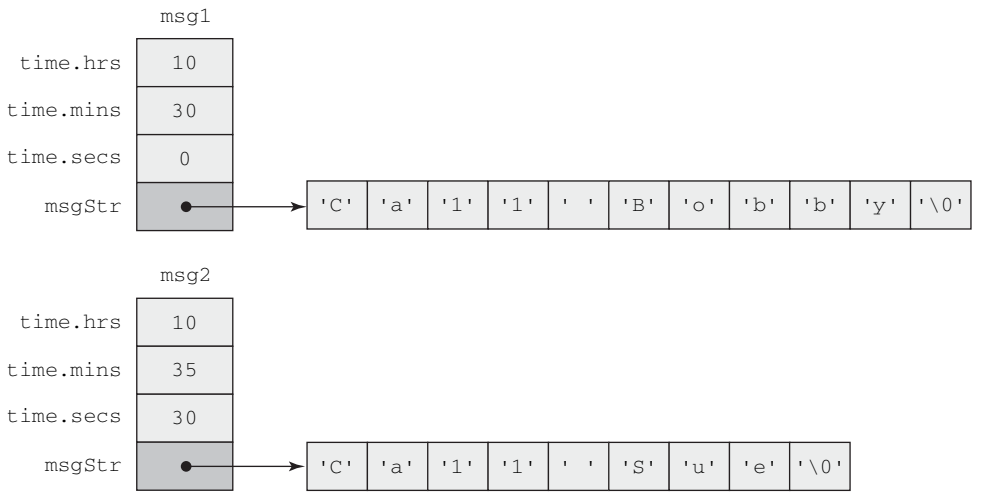


Figura 15-9 *Objetos de clase que apuntan a cadenas C dinámicamente asignadas*

Observe que la clase `Message` asigna datos dinámicos, pero no hemos previsto para la omisión de los datos dinámicos. Para tratar adecuadamente con objetos de clase que apuntan a datos dinámicos sólo necesitamos un constructor de clase. Es necesario un grupo completo de funciones miembro de clases: un *destructor de clase*, una *operación de copia profunda*, y un *constructor de copia de clase*. Explicaremos todas estas nuevas funciones una por una. Pero antes que nada, aquí presentamos el aspecto global que tendrá nuestra nueva declaración de clase:

```
class Message
{
public:
    void Print() const;                       // Operación de salida

    void CopyFrom( /* in */ Message otherMsg );
    // Deep copy operation
    Message( /* in */ Time newTime;
             /* in */ const char* msgStr );
    Message( const Message& otherMsg );       // Constructor de copia
    ~Message();                               // Destructor
private:
    Time  time;
    char* msg;
};
```

Esta declaración de clase incluye prototipos de funciones para las cuatro funciones miembro que vamos a necesitar: constructor, destructor, operación de copia profunda, y constructor de copia. Antes de proceder, definamos de modo más preciso la semántica de cada función de miembro, proporcionando las precondiciones y poscondiciones de función.

```
//*************************************************************************
// ARCHIVO DE ESPECIFICACIÓN (Message.h)
// Este archivo da la especificación de un tipo de datos abstractos Message
// que representa un tiempo y un mensaje
//*************************************************************************
```

```
#include "Time.h"

class Message
{
public:
    void Print() const;
        // Poscondición:
        //     El mensaje y el tiempo han sido producidos en la forma
        //        horas, minutos, segundos y mensaje

    void CopyFrom( /* in */ Message otherMsg );
        // Poscondición:
        //     Este mensaje es una copia de otherMsg, que incluye
        //     la cadena de mensaje

    Message( /* in */ Time newTime,
             /* in */ const char* msgStr  );
        // Constructor
        // Poscondición:
        //     El nuevo objeto de clase se construye con el tiempo
        //     establecido en newTime y msg establecido en msgStr

    Message( const Message& otherMsg );
        // Constructor de copia
        // Poscondición:
        //     El nuevo objeto de clase se construye con el tiempo y
        //     la cadena de mensaje iguales que los de otherMsg
        // Nota:
        //     Este constructor se invoca de forma implícita siempre
        //     que un objeto de mensaje sea pasado por valor, devuelto
        //     como un valor de función o inicializado por otro objeto
        //     de Message en una declaración

    ~Message();
        // Destructor
        // Poscondición:
        //     Message string is destroyed
private:
    Time time;
    char* msg;
};
```

A continuación se presenta un programa de cliente que demuestra llamadas a funciones miembro de la clase `Message`.

```
//**************************************************************************
// MessageDemo program
// This is a very simple client of the Message class
//**************************************************************************
#include <iostream>
#include <string>        // For string class
```

```
#include "Message.h"     // For Message class

using namespace std;

int main()
{
    Time time;         // Time object
    string msg;        // Input message

    // Construct object msg1, and print it
    time.ReadTime();
    cout << "Enter message " << endl;
    cin >> msg;
    Message msg1(time, msg.c_str());
    cout << "Message 1: ";
    msg1.Print();
    cout << endl;

    // Construct object msg2, then make it a copy of msg2
    time.Set(5, 30, 0);
    Message msg2(time, "Old");

    cout << "Old Message 2: ";
    msg2.Print();
    cout << endl;

    msg2.CopyFrom(msg1);
    cout << "New Message 2: ";
    msg2.Print();    // Should be same as msg1
    cout << endl;

    return 0;
}
```

Destructores de clase

La clase `Message` proporciona una función de destructor llamada `~Message`. Un destructor de clase, identificado por una tilde (~) que precede al nombre de la clase se podrá considerar como el contrario de un constructor. Así como un constructor es implícitamente invocado cuando el control alcanza la declaración de un objeto de clase, un destructor es implícitamente llamado cuando el objeto de clase es destruido. Un objeto de clase es destruido cuando "sale del alcance". (Un objeto automático sale del alcance cuando el control abandona el bloque en el cual está declarado. Un objeto estático sale del alcance cuando termina la ejecución del programa.) El siguiente bloque —que podría ser un cuerpo de función, por ejemplo— incluye notas en las ubicaciones donde tanto el constructor como el destructor son invocados:

```
{
    Time time;
    time.ReadTime();
    Message myMsg(time, "Remember meeting");    ← El constructor se invoca aquí
    ⋮
}    ← El destructor se invoca aquí debido a que myMsg sale del alcance
```

En el archivo de ejecución `Message.cpp`, la ejecución de la clase destructor es muy sencilla:

```
Message::~Message()

// Destructor

// Poscondición:
//     El array señalado por msg ya no está en el almacenamiento libre
{
    delete [] msg;
}
```

No se pueden transmitir argumentos a un destructor y, como es el caso del constructor de clase, no se deberá declarar el tipo de datos de la función.

En el caso de la clase `Message`, están encerrados cuatro elementos de datos dentro de la barrera de abstracción: horas, minutos y segundos, y el apuntador al mensaje, mas no el arreglo (véase la figura 15-9). Sin la función del destructor `~Message`, la destrucción de un objeto de clase borraría el apuntador en el arreglo dinámico, pero no borraría el arreglo propio. El resultado sería una fuga de memoria: el arreglo dinámico permanecería asignado, pero ya no sería accesible.

Copiado superficial *vs.* copiado profundo

Veamos ahora la función `CopyFrom` de la clase `Message`. Esta función está diseñada para copiar un objeto de clase a otro, *incluyendo el arreglo de mensaje dinámico*. Por medio del operador de asignación integral (=), la asignación de un objeto de clase a otro sólo copia los miembros de clase; *no* copia ningún dato a que apuntan los miembros de clase. Por ejemplo, dados los objetos `msg1` y `msg2` de la figura 15-9, el efecto de la sentencia de asignación

```
msg1 = msg2;
```

se muestra en la figura 15-10. El resultado se denomina operación de **copiado superficial**: se copia el apuntador, mas no los datos a que apunta.

Copiado superficial Operación que copia un objeto de clase a otro sin copiar ningún dato a que se apunta.

El copiado superficial es perfecto si ninguno de los miembros de clase es apuntador. Pero si uno o más miem-

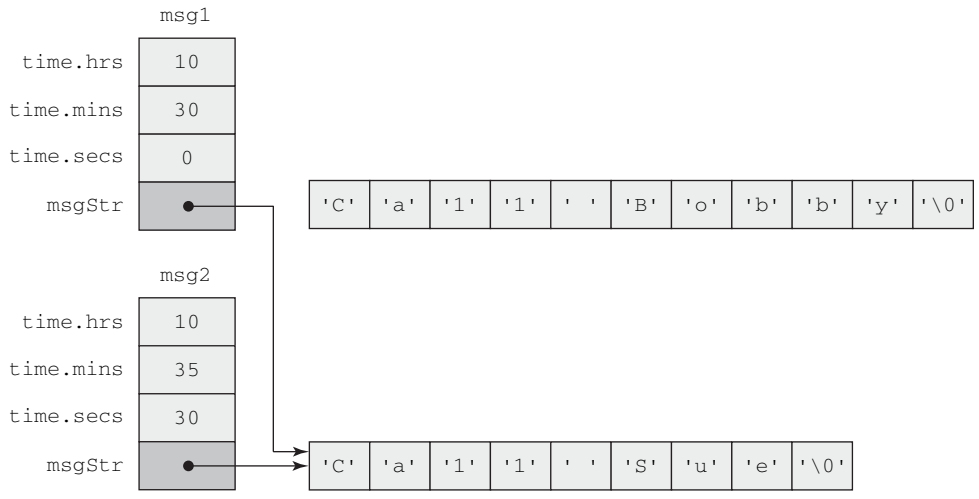


Figura 15–10 *Una copia superficial causada por la asignación msg1 = msg2*

Copiado profundo Operación que no sólo copia un objeto de clase a otro, sino que también hace copias de todos los datos a que se apunta.

bros de clase son apuntadores, entonces el copiado superficial podrá ser erróneo. En la figura 15-10 el arreglo dinámico al que el objeto `msg1` apuntaba originalmente ha quedado inaccesible.

Lo que queremos es una operación de **copiado profundo**, esto es, un copiado que no sólo duplique los miembros de clase, sino también los datos a que se apunta. La función `CopyFrom` de la clase `Message` realiza un copiado profundo.

Lo que sigue es la implementación de la función:

```
void Message::CopyFrom( /* in */ Message otherMsg )

// Poscondición:
//     tiempo == otherMsg.time
//  && msg apunta a un duplicado de la cadena de mensaje de otherMsg
//     en el almacenamiento libre

{
    time = otherMsg.time;
    delete [] msg;                          // Desasignar el original
    msg = new char[strlen(otherMsg.msg) + 1];
    // Allocate new array
    strcpy(msg, otherMsg.msg);              // Copiar los caracteres
}
```

Primero, la función copia el objeto `Time` del objeto `otherMsg` al objeto actual. Luego la función borra el arreglo dinámico del objeto actual del almacén libre, asigna un nuevo arreglo dinámico y copia todos los elementos del arreglo de `otherMsg` al nuevo arreglo. El resultado es, por ende, una copia profunda: dos objetos de clase idénticos apuntando a dos arreglos idénticos (pero *separados*). Dados nuestros objetos `msg1` y `msg2` de la figura 15-9, la sentencia

```
msg1.CopyFrom(msg2);
```

da el resultado que se muestra en la figura 15-11. Compare esta figura con la copia superficial que se presenta en la figura 15-10.

Constructores de copia de clase

Como hemos analizado, el operador de asignación integral (=) conduce a una copia superficial cuando objetos de clases apuntan a datos dinámicos. El problema del copiado profundo *vs*. copiado superficial también puede aparecer en otro contexto: la inicialización de un objeto de clase por otro. C++ define que inicialización significa lo siguiente:

1. Inicialización en una declaración de variable

   ```
   Message msg1 = msg2;
   ```

2. Transmisión de una copia de un argumento a un parámetro (es decir, paso por valor)
3. Devolución de un objeto como el valor de una función

   ```
   return someObject;
   ```

Por omisión, C++ realiza estas inicializaciones usando la semántica del copiado superficial. En otras palabras, el objeto de clase de nueva creación es inicializado vía una copia de miembro por miembro del objeto viejo sin considerar datos apuntados por los miembros de clase. Para nuestra clase `Message`, el resultado sería nuevamente dos objetos de clase apuntando a los mismos datos dinámicos.

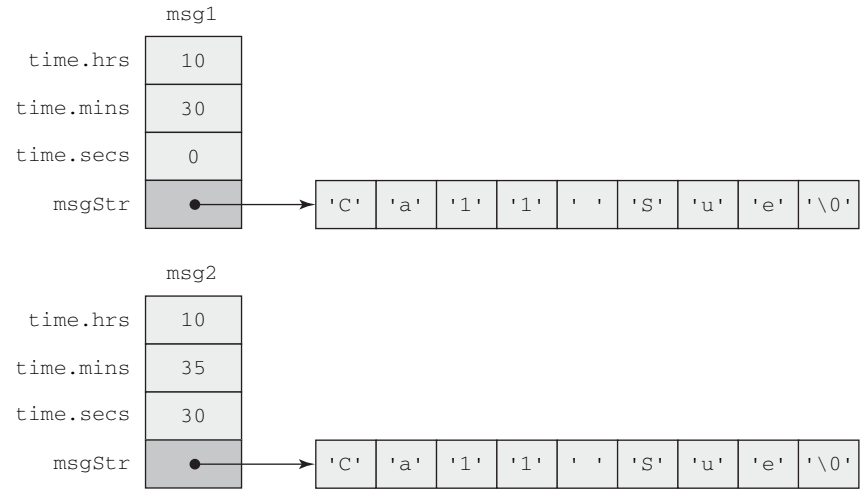


Figura 15–11 *Una copia profunda*

Para controlar esta situación, C++ tiene un tipo especial de constructor conocido como *constructor de copia*. En una declaración de clase, su prototipo tiene la siguiente forma:

```
class SomeClass
{
public:
     :
    SomeClass( const SomeClass& someObject );  // Constructor de copia
     :
};
```

Observe que el prototipo de función no usa ninguna palabra especial para sugerir que esto es un constructor de copia. Sólo se tiene que reconocer el patrón de símbolos: el nombre de clase seguido por una lista de parámetros que contiene un solo parámetro del tipo

```
const SomeClass&
```

Por ejemplo, nuestra declaración de la clase `Message` muestra que el prototipo del constructor de copia es

```
Message( const Message& otherMsg );
```

Si un constructor de copia está presente, el método por omisión de inicialización (copiado miembro por miembro) está inhibido. En su lugar, el constructor de copia es implícitamente invocado cada vez que un objeto de clase es inicializado por otro. La siguiente ejecución del constructor de copia para la clase `Message` muestra los pasos a seguir:

```
Message::Message( const Message& otherMsg )

// Constructor de copia

// Poscondición:
//     tiempo == otherMsg.time
//  && msg apunta a un duplicado de la cadena de mensaje de otherMsg
//     en el almacenamiento libre
```

```
{
    time = otherMsg.time;
    msg = new char[strlen(otherMsg.msg) + 1];
    strcpy(msg, otherMsg.msg);
}
```

El cuerpo de la función del constructor de copia se distingue del cuerpo de la función `CopyFrom` sólo en una línea de código. La función `CopyFrom` ejecuta

```
delete [] msg;
```

antes de asignar un nuevo arreglo. La diferencia entre estas dos funciones es que `CopyFrom` copia un objeto de clase *existente* (que ya está apuntando a un arreglo dinámico que debe ser borrado), mientras que el constructor de copia está creando un nuevo objeto de clase que aún no existe.

Observe el uso de la palabra reservada `const` en la lista de parámetros del constructor de copia. La palabra `const` asegura que la función no podrá alterar `otherMsg`, aunque `otherMsg` sea transmitido por referencia.

Como es el caso con cualquier variable de no arreglo en C++, un objeto de clase se puede transmitir a una función ya sea por valor o por referencia. Puesto que C++ define que la inicialización incluye el paso por valor, los constructores de copia son de vital importancia cuando objetos de clase apuntan a datos dinámicos. Supongamos que no hemos incluido un constructor de copia para la clase `Message`, y supongamos que la siguiente llamada a la función `DoSomething` usa un paso por valor:

```
int main()
{
    Time time;
    time.Set(10, 30, 0);
    Message quizMsg(time, "Geography quiz");
      :
    DoSomething(quizMsg);
      :
```

Sin constructor de copia, `quizMsg` sería copiado al parámetro de la función `DoSomething` usando una copia superficial. Una copia del arreglo dinámico de `quizMsg` ***no*** se crearía para el uso dentro de `DoSomething`. Tanto `quizMsg` como el parámetro dentro de `DoSomething` apuntarían al mismo arreglo dinámico (véase la figura 15-12). Si la función `DoSomething` modificara el arreglo dinámico (pensando que estuviese trabajando en una *copia* del original), entonces después del retorno de la función `quizMsg` apuntaría a un arreglo dinámico corrupto.

En resumen, las operaciones de asignación e inicialización por omisión podrán ser peligrosas cuando objetos de clase apuntan a datos dinámicos en el almacén libre. La asignación e inicialización de miembro por miembro causan que sólo se copien apuntadores, y no los datos a los cuales se apunta. Si una clase asigna y borra datos en el almacén libre, casi con certeza necesitará la siguiente sucesión de funciones miembro para asegurar el copiado profundo de datos dinámicos:

```
class SomeClass
{
public:
      ⋮
void CopyFrom( SomeClass anotherObject );
        // Una operación de copia profunda

    SomeClass( ... );
        // Constructor, para crear datos en el almacenamiento libre
```

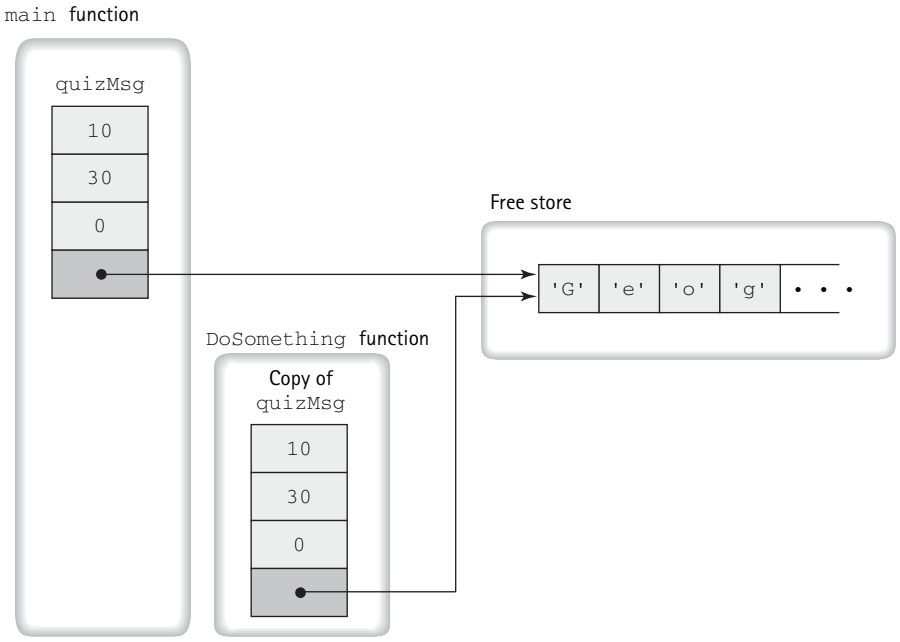


Figura 15-12 *Copia superficial causada por un paso por valor sin constructor de copia*

```
    SomeClass( const SomeClass& anotherObject );
        // Constructor de copia, para el copiado profundo en inicializaciones

    ~SomeClass();
        // Destructor, para limpiar el almacenamiento libre
private:
    ⋮
};
```

Al principio de este capítulo comentamos que los apuntadores se usan por dos razones: para hacer que un programa sea más eficiente –ya sea en velocidad o utilización de memoria– y para crear estructuras complejas de datos (las llamadas *estructuras ligadas*). Daremos ejemplos del uso de apuntadores para hacer un programa más eficiente en el Caso práctico en este capítulo. Las estructuras ligadas se abordarán en el capítulo 16.

Caso práctico de resolución de problemas

Creación de un calendario de citas, continuación

PROBLEMA En el capítulo anterior creamos una parte de un futuro calendario de citas. El calendario contenía una lista de pares de nombres/horas, ordenados por tiempo. El siguiente paso es ampliar el calendario de modo que represente un día completo que deberá contener una fecha y una lista de registros. Posteriormente se podrá realizar la expansión del calendario de citas a una lista de días.

ANÁLISIS Ya habremos adivinado que en algún momento íbamos a necesitar un tipo de datos abstractos Fecha. Una fecha es un objeto como nombre y tiempo que ocurre con frecuencia en problemas. Así que la creación de una clase `Date` es el primer punto en el orden del día.

Crear una clase `Date` ¿Qué tipos de operaciones necesitaremos en nuestro ADT Date? Por supuesto, necesitamos un constructor por omisión y un constructor parametrizado que toma mes, día y año como parámetros. Tal vez deberíamos tener una operación de transformador para reiniciar la fecha. Deberíamos tener operaciones de observadores que nos permitan ver los valores para mes, día y año. También deberíamos tener una operación que nos permita la comparación de dos fechas. La especificación informal del ADT se demuestra a continuación.

TIPO
    Fecha
DOMINIO
    Cada valor Date es una fecha individual después del año de 1582 D.C. en la forma de mes, día y año.
OPERACIONES
    Construir una nueva instancia de Date.
    Fijar la fecha.
    Inspeccionar el mes de la fecha.
    Inspeccionar el día de la fecha.
    Inspeccionar el año de la fecha.
    Comparar dos fechas para "antes", "igual" o "después".

ESPECIFICACIÓN DEL ADT El dominio del ADT es el conjunto de todas las fechas después del año de 1582 D.C. en la forma de un mes, un día y un año. Limitamos el año a la era después de 1582 D.C. a fin de simplificar las operaciones de ADT (en el año de 1582 se saltaron 10 días en el cambio del calendario juliano al calendario gregoriano).

Para representar el ADT de fecha como código de programa usamos una clase de C++ denominada `Date`. Las operaciones de ADT se convierten en funciones miembro públicas de la clase. Ahora vamos a especificar las operaciones más detenidamente.

Construir una nueva instancia de Fecha Para esta operación usamos un constructor por omisión de C++ que inicia el calendario el 1 de enero del año de 1583. El código del cliente podrá reiniciar la fecha en cualquier momento usando la operación "Fijar la fecha".

Fijar la fecha El cliente debe proporcionar tres argumentos para esta operación: mes, día y año. Aunque aún no hemos determinado una representación de datos concreta para una fecha, tenemos que decidir qué tipos de datos el cliente deberá usar para estos argumentos. Elegimos enteros donde el mes debe estar en el intervalo de 1 a 12, el día debe estar en el intervalo de 1 hasta el número máximo de días en el mes, y el año debe ser mayor que 1582. Observe que estas restricciones de intervalo serán la precondición para invocar esta operación.

Inspeccionar el mes, inspeccionar el día, e inspeccionar el año de la fecha Las tres operaciones son operaciones de observadores. Le dan al cliente acceso, de manera indirecta, a los datos privados. En la clase Fecha representamos estas operaciones como funciones de devolución de valores con los siguientes prototipos:

```
int Month();
int Day();
int Year();
```

¿Por qué necesitamos estas operaciones de observadores? ¿Por qué no simplemente dejar que la representación de datos del mes, día y año sean públicas en lugar de privadas, para que el cliente pueda acceder a los valores de manera directa? Hemos visto esta pregunta antes, y la respuesta es la misma: se deberá permitir que el cliente inspeccione *pero no que modifique* estos valores. Si los datos fueren públicos, un cliente podría manipularlos en forma incorrecta (por ejemplo, incrementar el 31 de enero al 32 de enero), comprometiendo de este modo el comportamiento correcto del ADT.

Comparar dos datos Esta operación compara dos datos y determina si el primero viene antes del segundo (BEFORE), si son iguales (EQUAL), o si el primero viene después del segundo (AFTER). Para indicar el resultado de la comparación, definimos un tipo de enumeración con tres valores:

```
enum RelationType {BEFORE, SAME, AFTER};
```

Ahora podemos codificar la operación de comparación como una función de miembros de clase que devuelve un valor del tipo `RelationType`. A continuación se encuentra el prototipo de función:

```
RelationType ComparedTo( /* in */ Date otherDate ) const;
```

Puesto que ésta es una función de miembros de clase, la fecha que se compara con `otherDate` es el objeto de clase para el cual se invoca la función de miembros. Por ejemplo, el siguiente código de cliente verifica para ver si `date1` viene antes de `date2`.

```
Date date1;
Date date2;
 ⋮
if (date1.ComparedTo(date2) == BEFORE)
    DoSomething();
```

Ahora casi estamos en condiciones para escribir el archivo de especificación de C++ para nuestra clase `Date`. La representación más sencilla para una fecha es de tres valores `int`, uno para el mes, uno para el día, y uno para el año. A continuación se encuentra el archivo de especificación que contiene la declaración de la clase `Date` (junto con la declaración del tipo de enumeración `RelationType`).

```
//**************************************************************************
// ARCHIVO DE ESPECIFICACIÓN (Date.h)
// Este archivo da la especificación de un tipo de datos
// abstractos Date y provee un tipo de enumeración para comparar
// fechas. Para ahorrar espacio, se omiten de cada función los comentarios
// de precondición que documentan las suposiciones hechas acerca de
// los datos de parámetros de entrada válidos. Éstos se incluirían en un
// programa destinado a uso real.
//**************************************************************************

enum RelationType {BEFORE, SAME, AFTER};

class Date
{
public:
    void Set( /* in */ int newMonth,
              /* in */ int newDay,
              /* in */ int newYear  );
        // Precondición:
        //     1 <= newMonth <= 12
        //  && 1 <= newDay <= núm. máximo de días en el mes newMonth
        //  && newYear > 1582
        // Poscondición:
        //     La fecha se establece de acuerdo con los parámetros entrantes

    int Month() const;
        // Poscondición:
        //     El valor de retorno es el mes de esta fecha

    int Day() const;
        // Poscondición:
        //     El valor de retorno es el día de esta fecha
```

```
    int Year() const;
        // Poscondición:
        //     El valor de retorno es el año de esta fecha

    RelationType ComparedTo( /* in */ Date otherDate ) const;
        // Poscondición:
        //     El valor de retorno es
        //        BEFORE, si esta fecha está antes de otherDate
        //        SAME, si esta fecha es igual a otherDate
        //        AFTER, si esta fecha está después de otherDate

    Date();
        // Poscondición:
        //     El objeto de la nueva fecha se construye con un
        //     mes, día y año de 1, 1 y 1583

    Date( /* in */ int newMonth,
          /* in */ int newDay,
          /* in */ int newYear  );
        // Precondición:
        //     1 <= newMonth <= 12
        //  && 1 <= newDay <= núm. máximo de días en el mes newMonth
        //  && newYear > 1583
        // Poscondición:
        //     La fecha se establece de acuerdo con los parámetros entrantes

private:
    int month;
    int day;
    int year;
};
```

APLICACIÓN DEL ADT Ya hemos seleccionado una representación de datos concretos para una fecha, que aparece en el archivo de especificación como las variables `int` de mes, día y año. Ahora tenemos que aplicar cada función de miembro de clase, colocando las definiciones de funciones en un archivo de aplicación C++ denominado `Date.cpp`. Conforme aplicamos las funciones miembro, también analizaremos las estrategias de pruebas que pueden ayudar a verificar que las aplicaciones son correctas.

Las funciones de constructor de clase, Fijar, Mes, Día y Año Las aplicaciones de estas funciones son tan obvias que no se requiere análisis alguno.

Date

Poner mes a 1
Poner día a 1
Poner año a 1583

Date(In: newMonth, newDay, newYear)

Poner mes a newMonth
Poner día a newDay
Poner año a newYear

Set(In: newMonth, newDay, newYear)

```
Poner mes a newMonth
Poner día a newDay
Poner año a newYear
```

Month( )
Salida: valor de función

```
Devolver mes
```

Day( )
Salida: valor de función

```
Devolver día
```

Year ( )
Salida: valor de función

```
Devolver año
```

PRUEBAS Las funciones de observador `Month`, `Day` y `Year` se pueden usar para verificar que las funciones de observador de clases y `Set` funcionen correctamente. El código

```
Date someDate;
Date otherDate(12, 24, 2004);

cout << someDate.Month() << ' ' << someDate.Day() << ' '
     << someDate.Year() << endl;
cout << otherDate.Month() << ' ' << otherDate.Day() << ' '
     << otherDate.Year() << endl;
```

deberá imprimir `1 1 1583` y `12 24 2004`. Para verificar la función `Set` es suficiente con poner un objeto `Date` a unos cuantos valores diferentes (obedeciendo la precondición para la función `Set`), para luego imprimir el mes, día y año como se indica arriba.

La función ComparedTo Si tuviéramos que comparar dos fechas en nuestra mente, primero consideraríamos los años. Si los años fueran distintos, sabríamos inmediatamente cuál de ellos viene antes. Si los años fueran iguales, verificaríamos los meses. Si los meses fueran iguales, verificaríamos los días. Como sucede tantas veces, podemos usar este algoritmo directamente en nuestra función.

ComparedTo (Entrada: otherDate)
Salida: valor de función

```
IF año < otherDate.year
    Devolver BEFORE
IF año > otherDate.year
    Devolver AFTER

// Los años son iguales. Comparar meses
IF mes < otherDate.month
    Devolver BEFORE
IF month > otherDate.month
    Devolver AFTER
```

```
// Los años y meses son iguales. Comparar los días
IF día < otherDate.day
    Devolver BEFORE
IF día > otherDate.day
    Devolver AFTER

// Los años, meses y días son iguales
Devolver SAME
```

PRUEBA Para la prueba de esta función debemos asegurar que cada ruta se tome por lo menos una vez. El ejercicio 1 del Seguimiento de caso práctico pide que usted diseñe datos de prueba para esta función y que escriba un manejador que haga la prueba.

```
//*****************************************************************************
// ARCHIVO DE EJECUCIÓN (Date.cpp)
// Este archivo pone en práctica las funciones miembro Date
//*****************************************************************************

#include "Date.h"
#include <iostream>

using namespace std;

// Miembros de clase privada:
//     int month;
//     int day;
//     int year;

int DaysInMonth( int, int );  // Prototipo para función auxiliar

//*****************************************************************************

Date::Date()

// Constructor

// Poscondición:
//     el mes es 1 && el día es 1 && el año es 1583

{
    month = 1;
    day   = 1;
    year  = 1583;
}

//*****************************************************************************

void Date::Set( /* in */ int newMonth,
                /* in */ int newDay,
                /* in */ int newYear  )

// Precondición:
//     1 <= newMonth <= 12
```

```
//  && 1 <= núm. máximo de días en el mes newMonth
//  && newYear > 1582
// Poscondición:
//     el mes es newMonth y el día es newDay y el año es newYear

{
    month = newMonth;
    day   = newDay;
    year  = newYear;
}

//****************************************************************************

int Date::Month() const

// Poscondición:
//     El valor de retorno es el mes

{
    return month;
}

//****************************************************************************

int Date::Day() const

// Poscondición:
//     El valor de retorno es el día

{
    return day;
}

//****************************************************************************

int Date::Year() const
// Poscondición:
//     El valor de retorno es el año

{
    return year;
}

RelationType Date::ComparedTo( /* in */ Date otherDate ) const

// Poscondición:
//     El valor de retorno es BEFORE, si esta fecha está
//                               antes que otherDate
//                          es SAME si esta fecha es igual a otherDate
//                          es AFTER si esta fecha está después
//                               de otherDate

{
```

```
    if (year < otherDate.year)        // Comparar años
        return BEFORE;
    if (year > otherDate.year)
        return AFTER;

    if (month < otherDate.month)     // Los años son iguales. Comparar
        return BEFORE;               //   meses
    if (month > otherDate.month)
        return AFTER;

    if (day < otherDate.day)         // Los años y meses son iguales.
        return BEFORE;               //   Comparar días
    if (day > otherDate.day)
        return AFTER;

    return SAME;                     // Los años, meses y días son
                                     //   iguales
}
```

Cambios en la clase SortedList Nuestra clase SortedList tenía el máximo número de elementos puestos por una constante. Ahora que sabemos cómo crear una estructura en el momento de ejecución que usa `new`, vamos a rescribir la clase de modo que haya un constructor por omisión que fije el número de células a un tamaño asignado, por ejemplo 20, y un constructor que tome el tamaño del arreglo como un parámetro. A continuación siguen los cambios en la parte privada de la clase.

```
class SortedList
{
...
    SortedList( /* in */ int numElements );

    // Poscondición:
    //     han sido creados los datos
    //     el tamaño ha sido establecido en numElements
    //     la longitud ha sido establecida en 0

private:
    int       length;
    int       currentPos;
    int       size;
    ItemType* data;
    void      BinSearch( ItemType, bool&, int& ) const;
};
```

Observe que `data` es ahora una variable de apuntador, no un nombre de arreglo. Apunta al primer elemento de un arreglo dinámicamente asignado. Sin embargo, recuerde que en C++ se puede agregar una expresión de índice a cualquier apuntador —no sólo un nombre de arreglo— siempre y cuando el apuntador apunte a un arreglo. De este modo `data` puede ser indizado exactamente como estaba cuando fue definido como un arreglo del tipo `ItemType`.

Puesto que la función `IsFull` necesita saber el tamaño del arreglo, tenemos que almacenar el valor del parámetro o el valor por omisión. Esta variable la denominamos `size` (véase la figura 15-13). Las funciones cambiadas en la clase `SortedList` se muestran en la siguiente página.

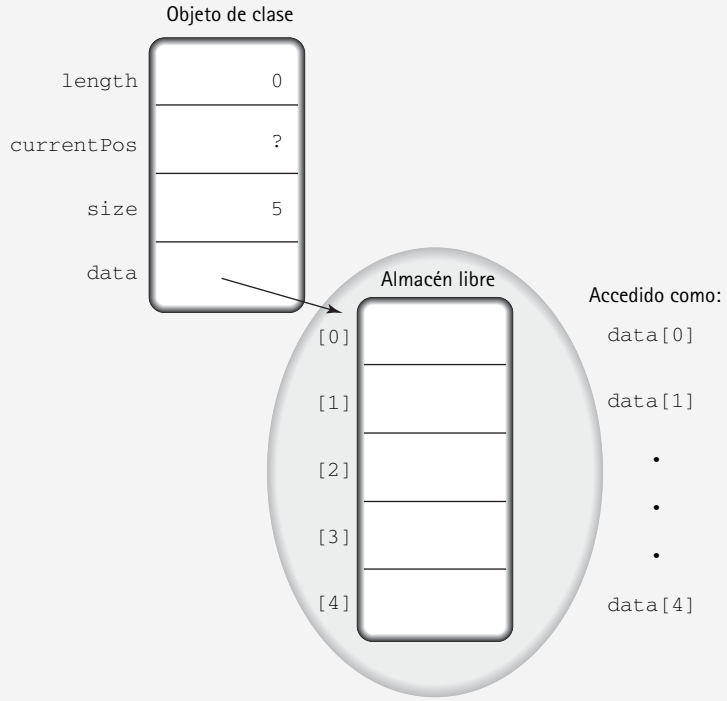


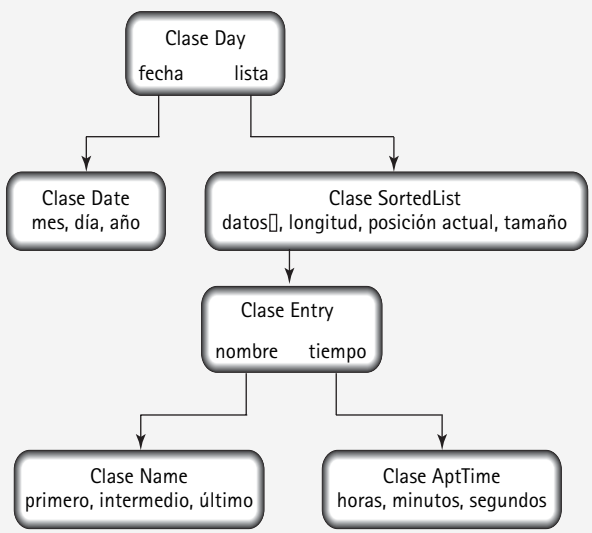
Figura 15–13 *Una lista vacía en un espacio dinámicamente asignado*

```
//*****************************************************************************

SortedList::SortedList()

// Constructor

// Poscondición:
//     la longitud se establece en cero
//     el tamaño se establece en 20
//     los datos se crean con 20 espacios

{
    length = 0;
    size = 20;
    data = new ItemType[size];
}

//*****************************************************************************

SortedList::SortedList( /* in */ int numElements )

// Constructor

// Poscondición:
//     los datos han sido creados con espacios de numElements
//     el tamaño ha sido establecido en numElements
//     la longitud ha sido establecida en 0
```

```
{
    length = 0;
    size = numElements;
    data = new ItemType[numElements];
}

//****************************************************************************

bool SortedList::IsFull() const

// Informa si SortedList está llena

// Poscondición:
//     El valor de retorno es verdadero, si la longitud es igual tamaño;
//     es falso en caso contrario

{
    return (length == size);
}

//****************************************************************************
```

Lista de pares de entradas de fechas Ahora tenemos todas las piezas: objetos de clase `Entry`, objetos de clase `Date` y un objeto `SortedList`, por lo que podemos pasar el número de lugares en el arreglo a la hora de ejecución. Necesitamos combinarlos para formar un ADT de Día. ¿Qué operaciones necesitamos? Dada una fecha, tenemos que aplicar cada una de las operaciones que fueron aplicadas en el controlador en el Caso práctico del capítulo 14: insertar una entrada, borrar una entrada, determinar si un tiempo está libre, e iterar por medio de la lista de entradas. Puesto que sabemos que la siguiente fase en el procesamiento incluye una lista de objetos de clase `Day` ordenada por fechas, deberemos incluir una operación de comparación que compara dos objetos por fecha.

La representación interna de la clase `Day` contiene un objeto `Date` y una `SortedList` de objetos `Entry`, compuestos por objetos `AptTime` y objetos `Name`. Esto se torna muy complicado. Veamos la estructura antes de empezar a escribir especificaciones. La cola de la flecha es una variable, y la cabeza es su tipo. Los tipos de las variables sin flechas que vienen de ellas son tipos integrales `int` o `string`.



Cuando tantas clases están agrupadas vía contención, siempre existe la posibilidad de que habrá alguna duplicación. En este caso, tanto la clase `Date` como la clase `Name` definen el tipo enumerado `RelationTime`. Esta duplicación impide la compilación de la clase `Day`. Podemos resolver este problema pasando la definición de `RelationType` a un encabezado por las directivas de compilador que hemos estudiado en el capítulo 14. Tanto la clase `Name` como la clase `Date` pueden incluir este archivo. Las directivas del compilador prohíben la duplicación. De hecho, sería buena idea proteger cada una de nuestras clases generales, como `Name`, `AptTime` y `Time`, por medio de estas directivas.

```
//*********************************************************************
// ARCHIVO DE ESPECIFICACIÓN para RelationType (relType.h)
// No hay archivo de ejecución de datos
//*********************************************************************
#ifndef ENUM
#define ENUM
enum RelationType{BEFORE, SAME, AFTER};
#endif
```

Aquí, pues, está el archivo de especificación para la clase `Day`. Se nota que hay tres constructores: un constructor por omisión, uno que toma una fecha y el número de elementos de arreglo, y uno que toma una fecha, una entrada y el número de elementos del arreglo.

```
//*********************************************************************
// ARCHIVO DE ESPECIFICACIÓN (Day.h)
// Este archivo contiene la especificación del TDA Day,
// que tiene dos clases contenidas, Date y una SortedList
// de objetos de la clase Entry
// Para ahorrar espacio, se omiten de cada función los comentarios
// de precondición que documentan las suposiciones hechas acerda de
// los datos de parámetros de entrada válidos. Éstos se incluirían en un
// programa propio para uso real.
//*********************************************************************
#include "SortedList.h"
#include "Date.h"
#include "relType.h"

class Day
{
public:
    //  Constructores

    Day();

        // Poscondición:
        //     la fecha y la lista se han inicializado a 20 espacios

    Day( /* in */ Date newDate,
         /* in */ int numAppts );

        // Poscondición:
        //     la fecha se estableció en newDate
        //     la lista se crea con espacios numAppts

    Day( /* in */ Date newDate,
         /* in */ Entry newEntry,
         /* in */ int numAppts );
```

```
    // Poscondición:
    //     la fecha se establece en newDate
    //     se inserta newEntry en la lista

// Otras funciones miembro

void InsertEntry( /* in */ Entry newEntry );

    // Precondición:
    //     el campo de tiempo en newEntry está libre
    // Poscondición:
    //     se inserta newEntry en la lista

void Delete( /* inout */ Entry entry);

    // Precondición:
    //     la entrada está en la lista
    // Poscondición:
    //     la entrada ya no está en la lista

Date DateIs() const;

    // Poscondición:
    //     El valor de retorno es la fecha de este día

RelationType ComparedTo( /* in */ Day otherDay ) const;

    // Poscondición:
    //     El valor de retorno es BEFORE, si la fecha de este día está
    //                                antes que la fecha de otherDay
    //                           es SAME, si la fecha de este día es igual
    //                                a la fecha de otherDay
    //                           es AFTER si la fecha de este día está
    //                                después de la fecha de otherDay

int NumberOfEntries();

    // Poscondición:
    //    El valor de retorno es la longitud de la lista de entradas

void ResetEntries();

    // Poscondición:
    //     Se inicializa la iteración

Entry GetNextItem();

    // Poscondición:
    //     Se inicializó la iteración mediante llamada a
    //     ResetEntries;
    //     Ningún transformador ha sido invocado desde la última llamada
    // Poscondición:
    //     Devuelve el elemento en la posición actual de la lista
    //     en la entrada;
```

```
        //     si ha sido devuelto el último elemento, la siguiente llamada
        //     devolverá el primer elemento.

    bool TimeFree(AptTime time);

        // Poscondición:
        //     El valor de retorno es verdadero si no está en la lista una
        //     entrada con tiempo; falso en caso contrario

private:
    Date       date;
    SortedList list;
};
```

Todas estas operaciones, a excepción del observador que devuelve el miembro de `Date`, involucran manipulaciones de listas. De hecho, ellas sólo llaman las funciones de listas paralelas. La única que es levemente distinta es `TimeFree`, que toma un objeto `AptTime` en lugar de un objeto `Entry`. Un objeto `Entry` debe ser construido para pasar a la función `SortedList IsPresent`. Enseguida se muestra el archivo de ejecución. Observe que hemos usado un inicializador constructor para pasar el tamaño del arreglo al constructor `SortedList`.

```
//**************************************************************************
// ARCHIVO DE EJECUCIÓN (Day.cpp)
// Este archivo contiene la ejecución del TDA Day, que tiene
// dos clases contenidas, Date y SortedList
//**************************************************************************
#include "Day.h"

void Day::InsertEntry( /* in */ Entry newEntry )

// Precondición:
//     está libre el campo de tiempo en newEntry
// Poscondición:
//     newEntry se inserta en la lista

{
    list.Insert(newEntry);
}

Day::Day() : list(24)

//     la fecha y la lista han sido inicializadas en 24 espacios

{
}

Day::Day( /* in */ Date newDate,
          /* in */ int numAppts ) : list(numAppts)

// Poscondición:
//     la fecha se establece en newDate
//     se crea la lista con espacios numAppts

{
```

```
    date = newDate;
}

Day::Day( /* in */ Date newDate,
          /* in */ Entry newEntry,
          /* in */ int numAppts )  : list(numAppts)

// Poscondición:
//     se establece la fecha en newDate; newEntry se inserta en la lista

{
    list.Insert(newEntry);
    date = newDate;
}

void Day::Delete( /* inout */ Entry entry)

// Precondición:
//     la entrada está en la lista
// Poscondición:
//     la entrada ya no está en la lista

{
    list.Delete(entry);
}

Date Day::DateIs() const

// Poscondición:
//     El valor de retorno es la fecha de este día

{
    return date;
}

RelationType Day::ComparedTo( /* in */ Day otherDay ) const

// Poscondición:
//     El valor de retorno es BEFORE, si la fecha de este día está antes
//                                que la fecha de otherDay
//                          es SAME si la fecha de este día es igual
//                                a la fecha de otherDay
//                          es AFTER si la fecha de este día está después
//                                de la fecha de otherDay

{
    return date.ComparedTo(otherDay.date);
}

int Day::NumberOfEntries()

// Poscondición:
//     El valor de retorno es la longitud de la lista de entradas
```

```
{
    return list.Length();
}

void Day::ResetEntries()

// Poscondición:
//     Se inicializa la iteración

{
    list.Reset();
}

Entry Day::GetNextItem()

// Precondición:
//     La iteración ha sido inicializada mediante llamada a ResetEntries;
//     Ningún transformador ha sido invocado desde la última llamada
// Poscondición:
//     Devuelve el elemento en la posición actual en lista
//     de la entrada;
//     Si ha sido devuelto el último elemento, la siguiente llamada
//     devolverá el primer elemento

{
    return list.GetNextItem();
}

bool Day::TimeFree(AptTime time)

// Poscondición:
//     El valor de retorno es verdadero si no está en la lista una entrada
//     con el tiempo; falso en caso contrario

{
    Entry entry(" ", " ", " ", time.Hours(),
                time.Minutes());
    return !(list.IsPresent(entry));
}
```

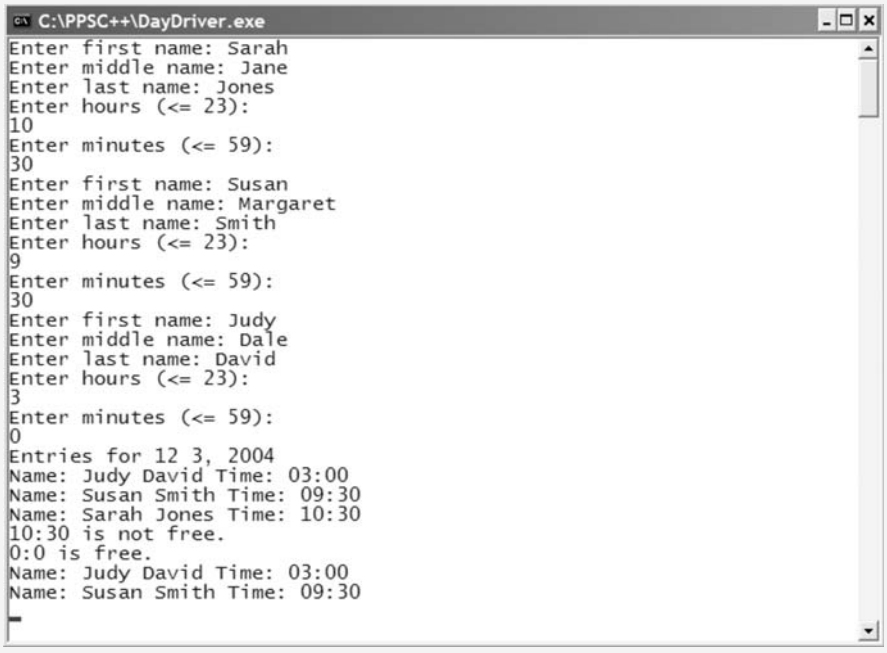
Controlador de clase `Day` El controlador para la clase `Day` debe verificar cada una de las funciones. Puesto que ninguna de las funciones contiene ciclos o bifurcaciones, el plan de pruebas puede ser muy sencillo. A continuación se presenta un controlador que llama todas las funciones miembro `Day`. Examine este controlador con mucho cuidado. La estructura compleja de este problema significa que se debe poner mucha atención cuando se elaboran las expresiones de acceso.

```
//**************************************************************************
// Se leen tres entradas desde el teclado y se insertan en la lista
// de entradas. El acceso a las entradas es una a la vez y se
// imprimen. Se comprueba un tiempo que está en la lista;
// se comprueba un tiempo que no está en la lista.
// Se borra un elemento y se imprime la lista para mostrar
// que se ha ido el elemento.
//**************************************************************************
```

```
#include <iostream>
#include "Day.h"
using namespace std;

int main()
{

    Entry nameTime;

    AptTime time(10, 30);
    Date date;
    date.Set(12, 3, 2004);    // Establecer la fecha en 12 3 2004
    Day day(date, 5);         // Establecer el día en la fecha con citas

    SortedList list;
    for (int count = 1; count <= 3; count++)
    {
        nameTime.ReadEntry();
        day.InsertEntry(nameTime);
    }
    // Preparar e iterar a través de la lista, imprimir los elementos
    int limit = day.NumberOfEntries();
    day.ResetEntries();                    // Preparar para iteración

    cout << "Elementos para " << day.DateIs().Month() << " "
         << day.DateIs().Day()
         << ", " << day.DateIs().Year()  << endl;
    for (int count = 0; count < limit; count++)
    {
        nameTime = day.GetNextItem();
        cout << "Nombre: " << nameTime.NameStr() << " Tiempo: "
             << nameTime.TimeStr() << endl;
    }

    // El elemento está en la lista
    if (day.TimeFree(time))
        cout << time.Hours() << ":" << time.Minutes()
             << " está libre." << endl;
    else
        cout << time.Hours() << ":" << time.Minutes()
             << " no está libre. " << endl;
    // El elemento no está en la lista
    time.Set(0, 0);
    if (day.TimeFree(time))
        cout << time.Hours() << ":" << time.Minutes()
             <<" está libre." << endl;
    else
        cout << time.Hours() << ":" << time.Minutes()
             << " no está libre. " << endl;
    // Borrar e iterar a través de la lista, imprimir los elementos
    day.Delete(nameTime);
    day.ResetEntries();          // Preparar para iteración
    limit = day.NumberOfEntries();
```

```
    for (int count = 0; count < limit; count++)
    {
        nameTime = day.GetNextItem();
        cout << "Nombre: " << nameTime.NameStr() << " Tiempo: "
             << nameTime.TimeStr() << endl;
    }
    return 0;
}
```

El controlador fue ejecutado con los mismos datos que se usaron en el capítulo 14. A continuación se muestra una imagen de pantalla.



¿En qué se distingue esta salida de la última? Tenemos una fecha impresa como encabezado de la salida. Al menos esto es lo que se ve. La estructura de lista subyacente se cambió de modo que el arreglo que contiene los elementos de la lista se genere a la hora usando el operador `new`, y se pide que el usuario introduzca el máximo número de citas para una determinada fecha.

Prueba y depuración

Es más difícil escribir y depurar programas que usan apuntadores que programas sin apuntadores. El direccionamiento indirecto nunca parece tan "normal" como el direccionamiento directo cuando se quiere llegar a los contenidos de una variable.

Los errores más comunes asociados con el uso de variables con apuntadores son:

1. Confundir la variable con apuntador con la variable a la que apunta
2. Tratar de desreferenciar el apuntador nulo o un apuntador no inicializado
3. Objetos inaccesibles
4. Apuntadores suspendidos (*dangling pointers*)

Consideremos cada uno de estos errores.

Si `ptr` es una variable con apuntador, se deberá cuidar que no se confundan las expresiones `ptr` y `*ptr`. La expresión

```
ptr
```

accede a la variable `ptr` (que contiene una dirección de memoria). La expresión

```
*ptr
```

accede a la variable a la que apunta `ptr`.

`ptr1 = ptr2`	Copia el contenido de `ptr2` a `ptr1`
`*ptr1 = *ptr2`	Copia el contenido de la variable a la que apunta `ptr2` a la variable a la que apunta `ptr1`.
`*ptr1 = ptr2`	Imposible; una es un apuntador y otra es una variable a la que se apunta.
`ptr1 = *ptr2`	Imposible; una es un apuntador y otra es una variable a la que se apunta.

El segundo error común es el de desreferenciar el apuntador nulo o un apuntador no inicializado. En algunos sistemas el intento de desreferenciar el apuntador nulo produce un mensaje de error de ejecución, como "NULL POINTER DEREFERENCE", inmediatamente seguido por la terminación del programa. Cuando esto ocurre, se tiene por lo menos alguna idea de lo que falló en el programa. Sin embargo, la situación puede empeorar si el programa desreferencia un apuntador no inicializado. En el fragmento de código

```
int  num;
int* intPtr;

num = *intPtr;
```

a la variable `intPtr` no se ha asignado ningún valor antes de que la desreferenciemos. Inicialmente contiene algún valor sin sentido, por ejemplo 315988, pero la computadora no sabe que este valor no tiene sentido. La máquina sólo accede a la ubicación de memoria 315988 y copia todo lo que encuentra allí a `num`. No hay manera de verificar si una variable con apuntador contiene un valor no definido. El único consejo que podemos dar es revisar el código cuidadosamente para asegurase de que se le haya asignado un valor a cada variable con apuntador antes de desreferenciarla.

El tercer error —dejar objetos inaccesibles en el almacén libre— en general resulta de una operación de copiado superficial o del uso incorrecto del operador `new`. En la figura 15-11 hemos mostrado cómo el operador de asignación integral causa una copia superficial; el objeto de datos dinámicos al que apunta originalmente una variable con apuntador permanece asignado, pero inaccesible. El uso equivocado del operador `new` también puede dejar datos dinámicos inaccesibles. La ejecución del fragmento de código

```
float* floatPtr;

floatPtr = new float;
*floatPtr = 38.5;
floatPtr = new float;
```

crea un objeto inaccesible: la variable dinámica que contiene 38.5. El problema es que hemos asignado un nuevo valor a `floatPtr` en la última sentencia, pero no hemos borrado la variable a la que apuntaba. Para protegerse contra este tipo de error, hay que examinar cada uso del operador `new` en el código. Si la variable asociada apunta expresamente a datos, borre esos datos antes de ejecutar la operación `new`.

Por último, los apuntadores suspendidos son una fuente de errores y pueden ser difíciles de detectar. Una causa para apuntadores suspendidos es borrar un objeto de datos dinámicos al que apun-

ta más de un apuntador. Las figuras 15-7*d* y 15-7*e* ilustran esta situación. Una segunda causa de apuntadores suspendidos es devolver un apuntador a una variable automática desde una función. La siguiente función, que devuelve un valor de función del tipo `int*`, es errónea.

```
int* Func()
{
    int n;
      ⋮
    return &n;
}
```

Recuerde que las variables automáticas son implícitamente creadas en la entrada del bloque e implícitamente destruidas en la salida del bloque. La función que arriba se describe devuelve un apuntador a la variable local `n`, pero `n` desaparece tan pronto como control sale de la función. Por tanto, el invocador de la función recibe un apuntador suspendido. Los apuntadores suspendidos son peligrosos por la misma razón que lo son los apuntadores no inicializados: cuando su programa desreferencia valores incorrectos de apuntadores, tendrá acceso a ubicaciones de memoria cuyos contenidos son desconocidos.

Sugerencias de prueba y depuración

1. Para declarar dos variables con apuntador en la misma sentencia, se deberá usar

   ```
   int *p, *q;
   ```

   No se puede usar

   ```
   int* p, q;
   ```

   Del mismo modo, se debe usar

   ```
   int &m, &n;
   ```

   para declarar dos variables de referencia en la misma sentencia.
2. No confunda un apuntador con la variable a la que apunta.
3. Antes de desreferenciar una variable con apuntador, asegúrese de que se le ha asignado un valor significativo diferente de `NULL`.
4. Para que se puedan comparar o asignar una a la otra, las variables con apuntadores deben ser del mismo tipo.
5. En una expresión, un nombre de arreglo sin corchetes de índice es una expresión con apuntador; su valor es la dirección base del arreglo. El nombre de arreglo es considerado como una expresión de constante, así que nada se le puede asignar. El siguiente código muestra asignaciones correctas e incorrectas.

   ```
   int  arrA[5] = {10, 20, 30, 40, 50};
   int  arrB[5] = {60, 70, 80, 90, 100};
   int* ptr;

   ptr = arrB;      // CORRECTO, puede asignar a una variable
   arrA = arrB;     // Erróneo, no puede asignar a una constante
   ```

6. Si `ptr` apunta a una estructura (*struct*), unión o variable de clase que tiene un miembro `int` llamado `age`, la expresión

   ```
   *ptr.age
   ```

es incorrecta. O se deberá colocar la operación de desreferencia entre paréntesis:

```
(*ptr).age
```

o se deberá usar el operador de flecha:

```
ptr->age
```

7. El operador `delete` se deberá aplicar a un apuntador cuyo valor fue previamente devuelto por `new`. La operación `delete` también deja indefinido el valor de la variable con apuntador; no vuelva a usar esta variable hasta que le haya asignado un nuevo valor.
8. Una función no debe devolver un apuntador a datos locales automáticos. En caso contrario, resultará un apuntador suspendido.
9. Si `ptrA` y `ptrB` apuntan al mismo objeto de datos dinámicos, la sentencia

```
delete ptrA;
```

convierte `ptrB` en un apuntador suspendido. Ahora deberá asignar a `ptrB` el valor `NULL` en vez de dejarlo suspendido.
10. Borre los valores dinámicos cuando ya no se necesiten. Las fugas de memoria pueden causar que se le agote el espacio de memoria.
11. Objetos inaccesibles —otra causa de fugas de memoria— son causados por

    a) copiado superficial de apuntadores que apuntan a datos dinámicos. Cuando se diseñan clases de C++ cuyos objetos apuntan a datos dinámicos, tenga cuidado de proporcionar una operación de copiado profundo y un constructor de copia.
    b) usar la operación `new` cuando la variable asociada ya está apuntando a datos dinámicos. Antes de ejecutar `new`, use `delete` para borrar los datos a los que se apunte expresamente.

Resumen

Los tipos de apuntadores y tipos de referencia son simples tipos de datos para almacenar direcciones de memoria. Las variables de estos tipos no contienen datos; en su lugar contienen las direcciones de otras variables o estructuras de datos. Las variables con apuntadores requieren el explícito desreferenciado usando el operador *. Las variables de referencia son desreferenciadas de manera implícita y en general se usan para pasar argumentos de no arreglo por referencia.

Un uso poderoso de apuntadores es para la creación de variables dinámicas. El apuntador se crea al tiempo de compilar, pero los datos a los cuales apunta el apuntador son creados al tiempo de la ejecución. El operador integral `new` crea una variable en el almacén libre (montículo) y devuelve un apuntador a esta variable. Una variable dinámica no recibe ningún nombre; se accede a ella por medio de una variable con apuntador.

El uso de datos dinámicos ahorra espacio de memoria porque una variable sólo se crea cuando se necesita en el momento de la ejecución. Cuando una variable dinámica ya no se necesita, puede ser borrada (usando `delete`), y el espacio de memoria se puede volver a usar. El uso de datos dinámicos también podrá ahorrar tiempo de máquina cuando se ordenan estructuras grandes. Los apuntadores a las estructuras grandes, en lugar de las propias estructuras grandes, se pueden volver a arreglar.

Cuando los objetos de clase de C++ apuntan a datos en el almacén libre, es importante distinguir entre operaciones de copiado superficial y profundo. Una copia superficial de un objeto de clase a otro, sólo copia los apuntadores y resulta en dos objetos de clase que apuntan a la misma variable dinámica. Una copia profunda resulta en dos copias distintas de los datos a los que se apunta. Por tanto, clases que manipulan datos dinámicos por lo general requieren una colección completa de rutinas de soporte: uno o varios constructores, un destructor (para limpiar el almacén libre), una operación de copiado profundo, y un constructor de copia (para el copiado profundo durante la inicialización de un objeto de clase por otro).

Comprobación rápida

1. ¿Por qué es más eficiente crear una lista de estructuras (*structs*) usando apuntadores en lugar de crearla en forma directa? (pp. 655-657)
2. Si usted quiere copiar todos los datos a los que se apunta en una estructura (*struct*), ¿deberá usar el copiado profundo o el superficial? (pp. 667-671)
3. Dos de las formas en que C++ define la inicialización son la inicialización en una declaración de variables y pasar un argumento por valor. ¿Cuál es la tercera forma que C++ usa para definir la inicialización? (pp. 668-671)
4. Cada clase que manipula datos dinámicos deberá tener un constructor, un destructor, y ¿cuáles son los otros dos tipos de funciones de miembro? (pp. 664-666)
5. Al declarar una variable con apuntador denominada `compass` que apunta a un `int`, hay dos formas de escribir la declaración en C++. ¿Cuáles son? (pp. 646-647)
6. Si quiere que `compass` apunte a la variable `north`, ¿cómo escribiría usted la asignación que realiza esto? (pp. 647-648)
7. Si `installRecord` apunta a un estructura con un miembro llamado `location`, ¿cuáles son las dos formas que C++ le permite para escribir un acceso directo a este miembro? (pp. 648-650)
8. ¿Cuál es la palabra clave que usamos al escribir en C++ una expresión de asignación? (pp. 655-656)
9. ¿Cuál es la palabra clave que usamos al escribir en C++ una expresión de omisión? (pp. 657-658)
10. En la declaración de una variable de referencia denominada `dictionary` que refiere a un `string`, hay dos formas para escribir la declaración en C++. ¿Cuáles son? (pp. 659-662)
11. Si `dictionary` refiere a una variable denominada `firstWord`, ¿cómo usaría usted la variable de referencia para asignar el valor `"aardvark"` a `firstWord`? (pp. 659-662)

Respuestas

1. Porque los apuntadores nos permiten crear de manera dinámica la lista, de modo que sólo tenga la longitud necesaria. **2.** Copiado profundo. **3.** Devolver un objeto como valor de una función. **4.** Una copia profunda y un constructor de copia. **5.** `int* compass; int *compass`. **6.** `compass = &north`. **7.** `(*installRecord).location` o `installRecord -> location` **8.** `new` **9.** `delete` **10.** `string& dictionary; string &dictionary`. **11.** `dictionary = "aardvark"`.

Ejercicios de preparación para examen

1. Relacione los siguientes términos con las definiciones que se dan enseguida.
   - ***a***) Tipo de apuntador
   - ***b***) Direccionamiento indirecto
   - ***c***) Direccionamiento directo
   - ***d***) Tipo de referencia
   - ***e***) Copia superficial
   - ***f***) Copia profunda
     - ***i***) Acceder a una variable usando su nombre.
     - ***ii***) Asignar el valor de un objeto a otro, incluyendo la duplicación de cualquier dato al que se apunte.
     - ***iii***) Acceder a una variable usando una dirección almacenada en un apuntador.
     - ***iv***) Un tipo simple que sólo se puede inicializar mediante la dirección de una variable.
     - ***v***) Un tipo simple al que se le puede asignar la dirección de una variable.
     - ***vi***) Asignar el valor de un objeto a otro sin duplicar datos a los que se apunta.
2. Relacione los siguientes términos con las definiciones que se dan abajo.
   - ***a***) Montículo (Almacén libre o *heap*)
   - ***b***) Fuga de memoria
   - ***c***) Objeto inaccesible
   - ***d***) Apuntador suspendido

e) Destructor
f) Datos dinámicos
  i) La pérdida de espacio disponible que ocurre cuando los datos dinámicos no son correctamente borrados.
  ii) El área de memoria que se usa para la asignación y omisión de datos dinámicos.
  iii) Un objeto que fue asignado pero no tiene apuntador que apunte a él.
  iv) Variables creadas por medio de la operación `new`.
  v) Un apuntador que apunta a un objeto borrado.
  vi) Una función de miembro que se invoca cuando un objeto sale de alcance.

3. Asignar `NULL` a un apuntador causa que no apunte a nada. ¿Verdadero o falso?
4. Una expresión de índice (como las que se usan con arreglos) puede ser agregada a cualquier variable con apuntador, aun si no apunta a un arreglo. ¿Verdadero o falso?
5. Una variable de referencia se puede volver a asignar a un nuevo valor de dirección en cualquier momento. ¿Verdadero o falso?
6. Cuando un objeto sale de alcance, sus variables con apuntador se borran automáticamente, pero sus valores a los que se apunta no son automáticamente borrados. ¿Verdadero o falso?
7. ¿`Delete` devuelve el apuntador al montículo (*heap*) o a los datos a que se apunta?
8. ¿Qué pasa con el(los) apuntador(es) y los datos a los que se apunta con un objeto si sale de alcance sin aplicar delete al apuntador (a los apuntadores) correspondiente(s)?
9. ¿Cuál es el problema en el siguiente segmento de código?

```
int* needle;
needle = new int;
*needle 100;
cout << *needle;
needle = new int;
*needle = 32;
cout << * needle;
```

10. ¿Cuál es el problema en el siguiente segmento de código?

```
int* birdDog;
int* germanShortHair;
birdDog = new int;
germanShortHair = birdDog;
*birdDog = 42;
cout << *birdDog;
delete birdDog;
cout << *germanShortHair;
```

11. ¿Qué hace el siguiente código?

```
int number;
int& atlas = number;
number = 212;
atlas++;
```

a) Incrementa el contenido de atlas.
b) Incrementa el contenido de number.
c) Incrementa el contenido tanto de atlas como de number.
d) Suma dos a number.
e) Produce un apuntador suspendido.

12. ¿Qué hace el siguiente código?

```
int number;
int* weathervane;
```

```
weathervane = &number;
number = 180;
(*weathervane)++;
```

a) Incrementa el contenido de `weathervane`.
b) Incrementa el contenido de `number`.
c) Incrementa el contenido tanto de `weathervane` como de `number`.
d) Suma 2 a `number`.
e) Produce un apuntador suspendido.

13. ¿Qué hace el siguiente código?

```
int number;
int* finger;
finger = &number;
number = 2;
*finger++;
```

a) Incrementa el contenido de `finger`.
b) Incrementa el contenido de `number`.
c) Incrementa el contenido tanto de `finger` como de `number`.
d) Suma 2 a `number`.
e) Produce un apuntador suspendido.

14. El constructor de copia para la clase `faxLog` deberá tener un parámetro. ¿De qué tipo debería ser?
15. ¿Por qué es un problema si usted realiza una copia superficial de un objeto a otro para luego borrar el primero y todos los datos dinámicos a los que se apunta?
16. ¿Qué tipo de función de miembro declara el siguiente encabezado?

```
~phoneTree();
```

17. Aplicar `delete` al apuntador nulo produce un mensaje de error. ¿Verdadero o falso?
18. En la siguiente expresión, `index` es un apuntador a un arreglo y `book` es un miembro de una `estructura` del tipo `libraryRecord`.

```
index[12] -> book[5]
```

¿Cuál es el tipo de los componentes contenidos en el arreglo al que apunta `index`?

Ejercicios de calentamiento de programación

1. Declare una variable con apuntador `intPointer` e inicialícela para que apunte a una variable `int` denominada `someInt`. Escriba dos sentencias de asignación, la primera de las cuales va a almacenar el valor 451 directamente en `someInt`, y la segunda, 451 indirectamente en la variable a la que apunta `intPointer`.
2. Declare una variable con apuntador `charArrPointer` e inicialícela para que apunte al primer elemento de un arreglo `char` de 4 elementos denominado `initials`. Escriba sentencias de asignación para almacenar `'A'`, `'E'` y `'W'` indirectamente en los primeros tres elementos del arreglo al que apunta `charArrPointer`.
3. Declare una variable con apuntador `structPointer` e inicialícela para que apunte a una variable `struct` denominada `newPhone`, del tipo `Phone`, que tiene tres campos `int` nombrados `country`, `area` y `number`. Escriba también la declaración del tipo `Phone`. Luego escriba sentencias de asignación para almacenar de manera indirecta los valores 1, 888 y 5551212 en estos campos.
4. Declare una variable de referencia `structReference` e inicialícela para apuntar a una variable `struct` denominada `newPhone`, del tipo `Phone`, que tiene tres campos `int` nombrados `country`, `area` y `number`. Escriba también la declaración del tipo `Phone`. Luego escriba senten-

cias de asignación para almacenar de manera indirecta los valores 1, 888 y 5551212 en estos campos.

5. Escriba una función booleana de devolución de valores denominada `ShallowCompare` que toma dos variables del tipo `structPointer`, como queda definido en el ejercicio 3, y que devuelve true si apuntan a la misma estructura, y false en caso contrario.
6. Escriba una función booleana de devolución de valores llamada `DeepCompare` que toma dos variables del tipo `structPointer`, tal como se describe en el ejercicio 3, y que devuelve true si las estructuras a las que apunta tienen valores idénticos en sus campos correspondientes.
7. Escriba un ciclo que recorra a través de un arreglo `int` dinámicamente asignado, al que apunta una variable denominada `data`, dando seguimiento del valor más grande en una variable `int` estática `max`. El arreglo contiene 100 valores. Al final del ciclo se deberá borrar el arreglo.
8. Coloque el ciclo que se escribió en el ejercicio 7 en una función de devolución `int` denominada `Greatest` que toma el arreglo y su tamaño como parámetros y que devuelve el valor en `max` después de que se ha borrado el arreglo. Tenga cuidado en cambiar el ciclo para que funcione con el tamaño dado del arreglo en lugar de la constante 100.
9. Escriba una función de devolución `int` que pida del usuario el número de valores a introducir, luego determine un arreglo `int` de este tamaño, y que memorice dicho número de valores en él. La función luego pasa este arreglo a la función `Greatest` que se define en el ejercicio 8, y que devuelve el resultado de esta función como su propio resultado. Llame la función `GetGreatestInput`.
10. Una clase denominada `Circuit` tiene dos miembros de datos privados y dinámicos de arreglo nombrados `source` y `sink`. Escriba un destructor para la clase que asegura que los datos dinámicos son destruidos.
11. Escriba un segmento de código que verifica si el apuntador `oldValue` realmente apunta a una ubicación de memoria válida. Si éste es el caso, se asigna su contenido a `newValue`, entonces `newValue` se asigna a una nueva variable `int` desde la pila.
12. Escriba un segmento de código que verifica si el apuntador `oldValue` y `newValue` apuntan a diferentes localidades. Si es así entonces borre el valor apuntado por `oldValue`. De lo contrario no haga nada.

Problemas de programación

1. Usted está trabajando para la oficina de registro de automóviles de su estado, y se ha detectado que algunas personas en la base de datos del departamento de licencias de conducir tienen registros múltiples. Los registros de licencias están guardados en orden alfabético en un conjunto de archivos, uno por letra del alfabeto. El primer archivo es `licensesa.dat`, y el último es `licensesz.dat`. Los archivos tienen tamaños distintos. La primera línea de cada uno es un entero que especifica el número de registros en el archivo. Para este problema, sólo nos vamos a enfocar en que el programa funcione para el archivo `licensesa.dat`. Algunos de los duplicados de registros se deben a ligeras diferencias en la ortografía de nombres, así que cada archivo se tiene que ordenar por número de licencia (un entero de 8 dígitos) para encontrar los duplicados. Cada registro consiste en un número de licencia, un nombre y una dirección, todo en una sola línea. Para los fines de este problema, el nombre y la dirección se pueden guardar en una sola cadena porque no se procesan por separado. El número de licencia y la cadena correspondiente se deberán mantener juntos en una estructura (`struct`).

   Cambie la clase `SortedList` del capítulo 13 de modo que el arreglo de lista esté creado dinámicamente para tener la longitud necesaria para los datos en el archivo particular. Cada componente de la lista debe ser un tipo de licencia `struct`. También agregue a la clase `SortedList` una función que devuelve valores de la lista en orden sucesivo (denominado `GetNext`), y una función compañera (nombrada `Restart`) que reinicia la otra función para que vuelva a empezar en el primer componente de la lista.

   Una vez que se hayan leído los datos desde un archivo a la `SortedList`, use las nuevas funciones para revisar la lista, comparando cada número de licencia con el número que antecede para verificar si son idénticos. Mantenga un conteo del número de registros duplicados que se

descubran. Luego emita este conteo a un archivo de salida de nombre `duplicensesa.dat`. Reinicie el recorrido de la lista desde el principio, y esta vez cada registro duplicado que se encuentre se deberá escribir en el archivo de salida. Recuerde que debe borrar el arreglo dinámico y cerrar el archivo al final del programa.

2. Extienda el programa de Problema 1 en dos formas. La primera extensión debe lograr que el programa procese automáticamente los 26 archivos de datos. La segunda extensión debe lograr que los registros duplicados se extraigan en orden alfabético, de modo que el archivo de salida tenga la misma organización que el archivo de entrada. Para realizar esto, usted deberá crear, después del primer paso a través de la lista, una segunda `SortedList` que tenga la longitud adecuada para retener todos los registros duplicados, y esta lista debe ordenar en el campo de cadenas de la estructura de licencias. Una vez que se hayan copiado todos los registros duplicados a esta lista, usted podrá borrar la primera y luego extraer la segunda lista al archivo.
3. Usted trabaja para una empresa que quiere obtener la información de contacto de clientes tal como las introduce un empleado de ventas desde una pila de tarjetas de presentación, para luego extraer las informaciones a un archivo (`contacts.dat`) en orden alfabético. Se supone que nunca hay más de 100 tarjetas que introducir. El programa deberá pedir al empleado de ventas cada uno de los siguientes valores para cada tarjeta:

   Apellido materno
   Nombre
   Apellido paterno
   Título
   Nombre de la compañía
   Dirección de calle
   Ciudad
   Estado
   Código postal
   Número de teléfono
   Número de fax
   Dirección electrónica

   Después de introducir los datos para cada tarjeta, se deberá preguntar al usuario si hay otra tarjeta que introducir. Estos datos se deberán ordenar por apellido, usando una versión de la clase `SortedList` del capítulo 13 que puede retener hasta 100 valores `struct`, cada uno con un miembro correspondiente a un elemento en la lista arriba mencionada. Debido a que `struct` es tan grande, será más eficiente si cada elemento del arreglo de la lista es un apuntador a uno de los valores `struct`, y la clase sólo reorganiza los apuntadores dentro del arreglo para hacer la clasificación. De manera dinámica usted deberá crear una nueva estructura para cada componente del arreglo de la lista a que apunta conforme se introduzcan las tarjetas. De este modo el programa no asignará memoria para más tarjetas de las que se introducen. Será necesario modificar la función `Print` de la clase para extraer la lista al archivo `contacts.dat` en lugar de `cout`. Cada miembro de la estructura se deberá escribir en una línea separada. La función también deberá escribir el número de tarjetas que están en la lista, como la primera línea del archivo.
4. Usted trabaja para una empresa que tiene una colección de archivos, cada uno de los cuales contiene información de hasta 100 tarjetas de presentación. Otro programa crea los archivos (véase el problema 3 para una descripción de lo que emite este programa). No existen más de 100 de estos archivos. La empresa desea unir los archivos para formar uno solo, ordenado de manera alfabética por apellido materno. Al usuario se le deberá pedir que introduzca los nombres de los archivos hasta que aparezca un archivo con el nombre "done". Para cada nombre de archivo que se introduce, determine de manera dinámica una estructura (del tipo `fileRecord`) cuyos miembros sean un `int length` y un objeto `ifstream`. Asigne un apuntador a `struct` para el próximo componente disponible de un arreglo. Cada componente del arreglo es un `fileRecord*`. Abra el `ifstream` en la estructura de creación más reciente, usando el nombre proporcionado por el

usuario, y luego extraiga la primera línea al miembro `length` de `struct`. Una vez que se hayan introducido todos los nombres de archivo y los archivos estén abiertos, recorra todas las estructuras a las que apunta el arreglo, sumando el número de tarjetas en todos los archivos como lo dan sus miembros `length`.

Modifique la `SortedList` del capítulo 13 para usar un arreglo dinámico que contenga un número específico de componentes. Cada componente de la lista deberá ser un apuntador a un tipo `struct`, como se describe en el problema 3. Estos apuntadores son los únicos valores que serán reagrupados por el proceso de ordenación. Use el conteo de tarjetas de entre todos los archivos, como el tamaño del arreglo en la `SortedList`. Luego repase el arreglo de valores `fileRecord`, extrayendo todos los datos de cada archivo a la `SortedList`. Una vez que todos los datos se hayan introducido, cópielos a un archivo de nombre `mergedcontacts.dat`. Será necesario modificar la función `Print` de la clase `SortedList` para extraer la lista al archivo `mergedcontacts.dat` en lugar de `cout`. Cada miembro de `struct` se deberá escribir en una línea separada. La función también deberá escribir como primera línea del archivo el número de tarjetas que se encuentran en la lista combinada.

5. Modifique el programa en el problema 4 de modo que después de la introducción de los datos el programa recorra la lista y borre todos los registros duplicados antes de extraerla. Un registro es un duplicado si el apellido materno, nombre de pila, apellido paterno o inicial, así como el nombre de la compañía es el mismo que en otro registro. Tenga cuidado de borrar correctamente todos los datos de asignación dinámica que se han retirado de la lista.

Seguimiento de caso práctico

1. Escriba y aplique un plan de prueba para la clase `Date`.
2. No hemos incluido ni un constructor de copia ni un destructor en la clase `SortedList` revisada. ¿Lo deberíamos haber hecho? Explique.
3. Las clases en este caso práctico se construyeron a lo largo de varios capítulos. ¿Esto demuestra un diseño jerárquico arriba-abajo o un diseño orientado a objetos? Explique.
4. Resuma el siguiente paso en la elaboración de una agenda de citas. ¿Cómo se podrá implementar una lista de días ordenados por fechas?

CAPÍTULO

16

Estructuras ligadas

Objetivos

Objetivos de conocimiento

- *Entender el concepto de una estructura ligada de datos.*

Objetivos de habilidades

Ser capaz de:

- *Declarar los tipos de datos y variables necesarios para una lista ligada dinámica.*
- *Imprimir el contenido de una lista ligada.*
- *Insertar nuevos elementos en una lista ligada.*
- *Borrar elementos de una lista ligada.*

En el capítulo anterior vimos que C++ tiene un mecanismo para crear variables dinámicas. Éstas, que pueden ser de tipo simple o estructurado, se pueden crear o destruir en cualquier momento durante la ejecución del programa, usando los operadores `new` y `delete`. Una variable dinámica no se identifica por medio de un nombre, sino de un apuntador que contiene su ubicación (dirección). Cada variable dinámica tiene un apuntador asociado mediante el cual se puede tener acceso. Hemos usado variables dinámicas para ahorrar espacio y tiempo de máquina. En este capítulo veremos cómo usarlas para construir estructuras de datos que pueden crecer y decrecer mientras se ejecuta el programa.

16.1 Estructuras secuenciales *versus* estructuras ligadas

Como hemos señalado en capítulos anteriores, muchos problemas de computación involucran listas de elementos. Una lista es un tipo de datos abstractos (ADT) con ciertas operaciones permisibles: búsqueda en la lista, ordenamiento, impresión, etc. La estructura que hemos usado como la representación concreta de datos de una lista es el array, esto es, una estructura secuencial. El término *estructura secuencial* significa que los componentes sucesivos del array están ubicados uno junto al otro en la memoria.

Si la lista que creamos está ordenada —una lista cuyos componentes se deberán mantener en orden ascendente o descendente—, se ejecutan ciertas operaciones de manera eficiente usando una representación de array. Por ejemplo, la revisión de una lista ordenada para buscar un valor particular se realiza rápidamente usando una búsqueda binaria. Sin embargo, insertar y borrar elementos en una lista ordenada es ineficiente con una representación de array. Para insertar un nuevo elemento en el lugar apropiado en la lista, tenemos que mover elementos del array hacia abajo a fin de crearle un espacio (véase la figura 16-1). De modo similar, para borrar un elemento de la lista se requiere que movamos hacia arriba todos los elementos del array que le siguen.

Lista ligada Lista en la que el orden de los componentes se determina por un miembro de liga explícito en cada nodo, y no por orden secuencial de los componentes en la memoria.

Cuando las inserciones y eliminaciones son frecuentes, hay una mejor representación de datos para una lista: la **lista ligada**. Una lista ligada es una colección de elementos, llamado *nodos*, que pueden estar dispersos en la memoria, no necesariamente en ubicaciones consecutivas. Cada nodo, en general representado como una estructura (*struct*), consta de dos miembros:

1. Un componente o miembro de un elemento, que contiene uno de los valores de datos en la lista
2. Un miembro de liga que da la ubicación del siguiente nodo en la lista

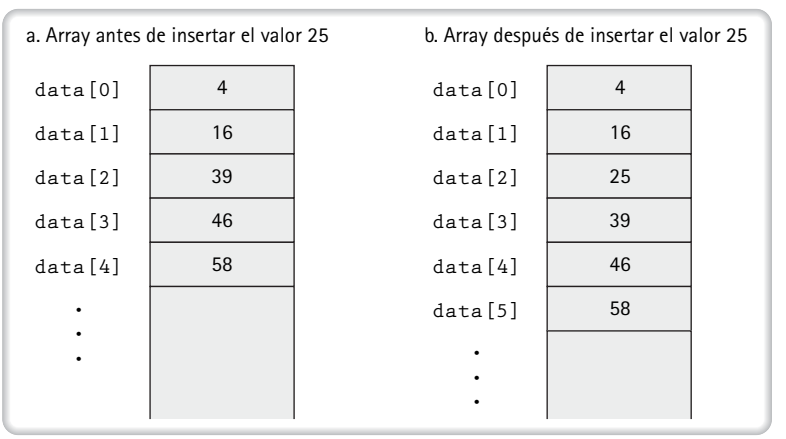
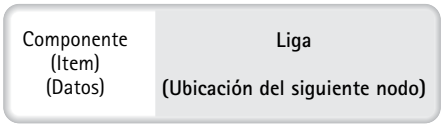


Figura 16-1 *Inserción en una representación secuencial de una lista ordenada*



En la figura 16-2 se muestra un diagrama de una lista ligada. En el miembro de liga de cada nodo se usa una flecha para indicar la ubicación del siguiente nodo. La barra inclinada (/) en el miembro de liga del último nodo significa el final de la lista. La variable separada `head` no es un nodo en la lista ligada; su propósito es dar la ubicación del primer nodo.

Tomar acceso a los elementos en una lista ligada es parecido al juego de la búsqueda del tesoro: cada niño recibe una pista para encontrar dónde se encuentra la siguiente pista, y la cadena de pistas finalmente conduce al tesoro.

Observe la figura 16-2; usted encontrará dos cosas: primero, hemos colocado los nodos deliberadamente en posiciones al azar; hemos hecho esto para destacar que los elementos en una lista ligada no se encuentran necesariamente en ubicaciones de memoria adyacentes (como en la representación de array de la figura 16-1). En segundo lugar, usted ya podrá estar pensando en apuntadores cuando ve las flechas en la gráfica, ya que dibujamos variables de apuntadores de este modo en el capítulo 15. Sin embargo, hasta ahora hemos evitado cuidadosamente el uso de la palabra *apuntador*; sólo dijimos que el miembro de liga de un nodo da la ubicación del siguiente nodo. Como veremos, hay dos formas de organizar una lista ligada. Una es almacenarla en un array de estructuras (*structs*), una técnica que no usa apuntadores. La segunda forma es usar datos dinámicos y apuntadores. Empecemos con la primera de estas dos.

16.2 Representación de array de una lista ligada

Una lista ligada se puede representar como un array de estructuras (*structs*). Para una lista ligada de componentes `int` usamos las siguientes declaraciones:

```
struct NodeType
{
    int component;
    int link;
};

NodeType node[1000];     // Máximo 1000 nodos
int      head;
```

Todos los nodos residen en un array denominado `node`. Cada nodo tiene dos miembros: `component` (en este ejemplo un valor de datos `int`) y `link`, que contiene el *índice de array* del siguiente nodo en la lista. El último nodo en la lista tendrá un miembro `link` de –1. Puesto que –1 no es un índice de array válido en C++, es apropiado como un valor especial de "fin de lista". La figura 16-3 muestra una representación de array de la lista ligada de la figura 16-2.

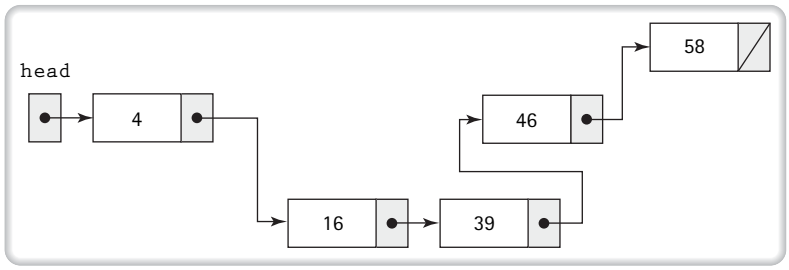


Figura 16–2 *Una lista ligada*

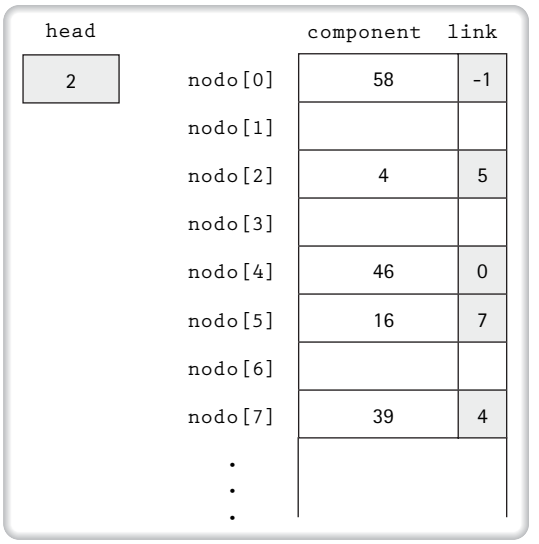


Figura 16-3 *Representación de array de una lista ligada*

Compare las figuras 16-1 y 16-3. La figura 16-1 muestra una lista representada directamente como un array. La figura 16-3 muestra una lista representada como una lista ligada que, a su vez, se representa como un array (de estructuras). Dijimos que cuando ocurren con frecuencia inserciones y eliminaciones, es mejor usar una lista ligada que usar un array de modo directo. Veamos por qué.

En la figura 16-1 se muestra el efecto de insertar 25 en la lista; tuvimos que mover los elementos de array 2, 3, 4,... hacia abajo para insertar el valor 25 en el elemento 2. Si la lista es larga, probablemente tengamos que mover cientos de miles de números. En cambio, la inserción del valor 25 en la lista ligada de la figura 16-3 *no* requiere el movimiento de datos existentes. Sólo buscamos una ranura no usada en el array, guardamos 25 en el miembro `component`, y ajustamos el miembro `link` del nodo que contiene 16 (véase la figura 16-4).

Antes de introducir la segunda técnica para implementar una lista ligada —esto es, el uso de datos dinámicos y apuntadores— demos un paso atrás y observemos el panorama general. Estamos interesados en la lista como un ADT. Puesto que es un ADT, tenemos que organizarla usando alguna

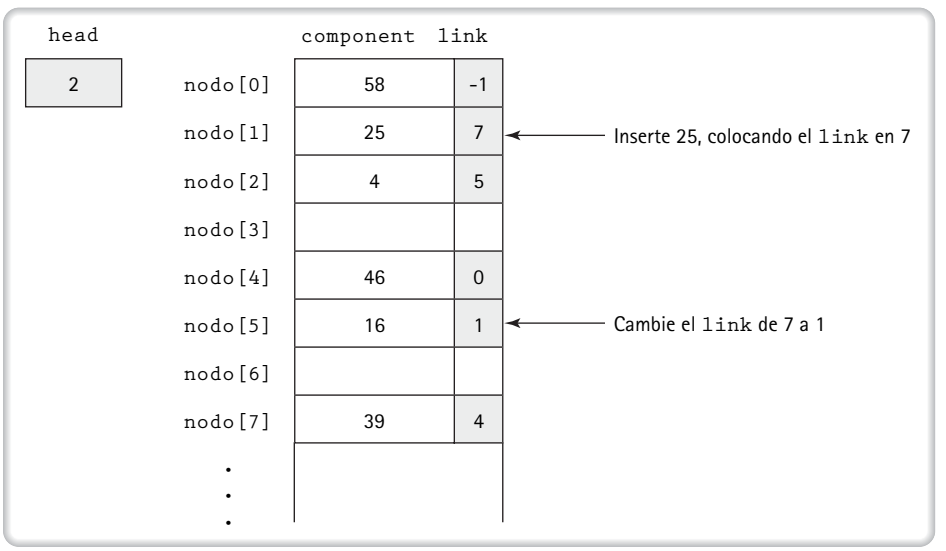


Figura 16-4 *Representación de array de una lista ligada después de que se insertó 25*

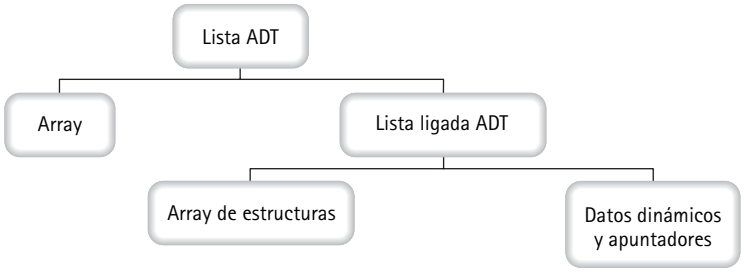


Figura 16-5 *Jerarquía de implementación para una lista ADT*

representación de datos existente. Una representación de datos es el array integrado, o sea una estructura secuencial. Otra representación de datos es la lista ligada, esto es, una estructura ligada. Pero una lista ligada es en sí misma un ADT y requiere una representación de datos concreta, por ejemplo un array de estructuras (*structs*). Para poder visualizar todas estas relaciones usamos un *diagrama de jerarquía para implantación*, como el que se ilustra en la figura 16-5. En dicho diagrama cada tipo de datos se ejecuta usando el tipo (o los tipos) de datos directamente debajo de él en la jerarquía.

16.3 Representación de datos dinámicos de una lista ligada

La representación de una lista como array o como lista ligada guardada en un array de estructuras (*structs*) tiene una desventaja: El tamaño del array es fijo y no puede cambiar durante la ejecución del programa. Pero cuando trabajamos con listas, con frecuencia no tenemos idea de cuántos componentes tenemos. El planteamiento común en esta situación es el de declarar un array que es suficientemente grande para retener la máxima cantidad de datos que, por lógica, podemos esperar. Puesto que en general tenemos menos datos que el máximo, se desperdicia espacio de memoria por los elementos de array no usados.

Hay otra técnica en la cual los componentes de lista son variables dinámicas que sólo se crean cuando se necesitan. Representamos la lista como una lista ligada cuyos nodos son dinámicamente asignados en el área de almacenamiento libre (montículo o *heap*), y el miembro de liga de cada nodo contiene la dirección de memoria del siguiente nodo dinámico. En esta representación de datos, denominada **lista ligada dinámica**, las flechas en el diagrama de la figura 16-2 representan apuntadores (y la barra inclinada en el último nodo es el apuntador nulo). Tomamos acceso a la lista mediante una variable de apuntador que contiene la dirección del primer nodo en la lista. Esta variable de apuntador, nombrada `head` en la figura 16-2, se denomina **apuntador externo** o **apuntador principal**. Se accede a cada nodo después del primero usando el miembro de liga en el nodo inmediatamente anterior.

Lista ligada dinámica Lista ligada compuesta por nodos dinámicamente asignados, ligados entre sí por apuntadores.

Apuntador externo (principal) Variable apuntador que apunta al primer nodo en una lista ligada dinámica.

Estructura de datos dinámica Estructura de datos que puede expandirse y contraerse durante la ejecución.

Una lista de este tipo puede expandirse o contraerse mientras se ejecuta el programa. Para insertar un nuevo elemento en la lista asignamos más espacio en el área de almacenamiento libre. Para borrar un elemento eliminamos la memoria asignada a él. La única limitación es la cantidad de espacio disponible en la memoria. Las estructuras de datos construidas que usan esta técnica se denominan **estructuras de datos dinámicas**.

Para crear una lista ligada dinámica empezamos con la asignación del primer nodo, guardando el apuntador a éste en el apuntador externo. Luego asignamos un segundo nodo y guardamos el apuntador correspondiente en el miembro de liga del nodo anterior, hasta que terminemos agregando nodos a la lista.

Veremos cómo podemos usar variables de apuntador de C++ para crear una lista ligada dinámica de valores `float`. Comenzamos con las declaraciones

```
typedef float ComponentType;

struct NodeType
{
    ComponentType component;
    NodeType*     link;
};
typedef NodeType* NodePtr;
NodePtr head;                 // Puntero externo para la lista
NodePtr currPtr;              // Puntero para el nodo actual
NodePtr newNodePtr;           // Puntero para el nodo más reciente
```

El orden de estas declaraciones es importante. El Typedef para `NodePtr` se refiere al identificador `NodeType`, de modo que la declaración de `NodeType` debe ir primero. (Recuerde que C++ requiere que cada identificador sea declarado antes de que se use.) Dentro de la declaración de `NodeType` quisiéramos declarar que `link` sea del tipo `NodePtr`, pero no podemos hacerlo porque aún no se ha declarado el identificador `NodePtr`. Sin embargo, C++ permite *declaraciones directas* (o *incompletas*) de estructuras (*structs*), clases y uniones:

```
typedef float ComponentType;

struct  NodeType;             // Declaración directa (incompleta)
typedef NodeType* NodePtr;

struct NodeType               // Declaración completa
{
    ComponentType component;
    NodePtr       link;
};
```

La ventaja de usar una declaración directa es que podemos declarar que el tipo de `link` sea `NodePtr`, del mismo modo que declaramos que `head`, `currPtr` y `newNodePtr` sean del tipo `NodePtr`.

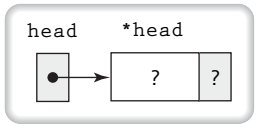
Dadas las declaraciones mencionadas, el siguiente fragmento de código crea una lista ligada dinámica con los valores 12.8, 45.2 y 70.1 como los componentes en la lista.

```
#include <cstddef>   // Para NULL
  ⋮
head = new NodeType;
head->component = 12.8;
newNodePtr = new NodeType;
newNodePtr->component = 45.2;
head->link = newNodePtr;
currPtr = newNodePtr;
newNodePtr = new NodeType;
newNodePtr->component = 70.1;
currPtr->link = newNodePtr;
newNodePtr->link = NULL;
currPtr = newNodePtr;
```

Abordaremos cada uno de estos enunciados describiendo en palabras lo que está pasando, y mostrando la lista ligada como se presenta después de la ejecución de la instrucción.

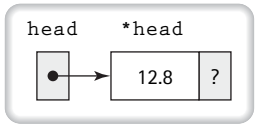
```
head = new NodeType;
```

Se crea una variable dinámica del tipo `NodeType`. El apuntador a este nuevo nodo se guarda en `head`. La variable `head` es el apunador externo que estamos construyendo.



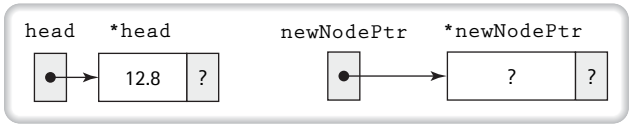

```
head->component = 12.8;
```

El valor 12.8 se guarda en el miembro `component` del primer nodo.



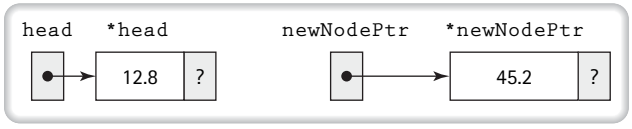

```
newNodePtr = new NodeType;
```

Se crea una variable dinámica del tipo `NodeType`. El apuntador a este nuevo nodo se guarda en `newNodePtr`.



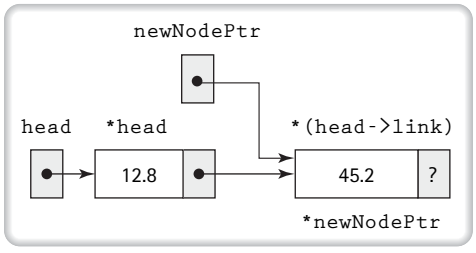

```
newNodePtr->component = 45.2;
```

El valor 45.2 se guarda en el miembro `component` del nuevo nodo.



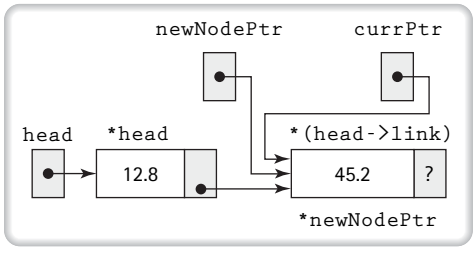

```
head->link = newNodePtr;
```

El apuntador al nuevo nodo que contiene 45.2 en su miembro `component` se copia al miembro `link` de `*head`. La variable `newNodePtr` aún apunta a este nuevo nodo. Se puede tomar acceso al nodo ya sea como `*newNodePtr` o como `*(head->link)`.

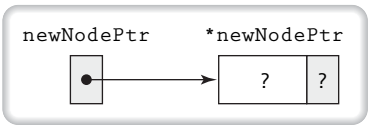



```
currPtr = newNodePtr;
```

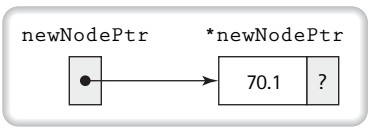
El apuntador al nuevo nodo se copia a `currPtr`. Ahora `currPtr`, `newNodePtr` y `head->link` todos apuntan al nodo que contiene 45.2 como su componente.



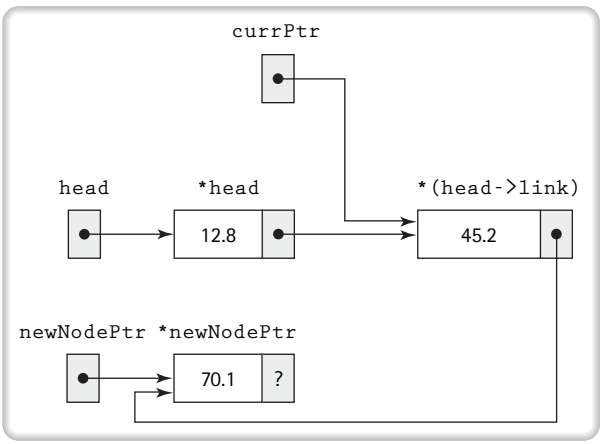
`newNodePtr = new NodeType;` Se crea una variable dinámica del tipo `NodeType`. El apuntador a este nuevo nodo se guarda en `newNodePtr`.



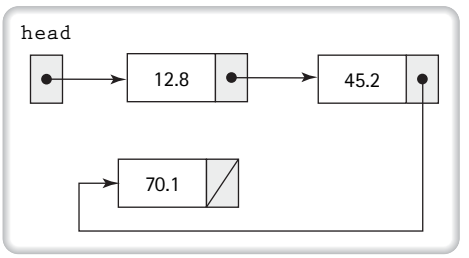
`newNodePtr->component = 70.1;` El valor 70.1 se guarda en el miembro `component` del nuevo nodo.



`currPtr->link = newNodePtr;` El apuntador al nuevo nodo que contiene 70.1 en el miembro `component` se copia al miembro `link` del nodo que contiene 45.2.



`newNodePtr->link = NULL;` La constante apuntador especial `NULL` se guarda en el miembro `link` del último nodo en la lista. Cuando se usa en el miembro `link` de un nodo, `NULL` significa el fin de la lista. `NULL` aparece en el diagrama como / en el miembro `link`.



`currPtr = newNodePtr;` `currPtr` se actualiza

Nos gustaría generalizar este algoritmo de modo que lo podamos usar como un ciclo para crear una lista ligada dinámica de cualquier longitud. En el algoritmo hemos usado tres apuntadores:

1. `head`, que se usó en la creación del primer nodo en la lista y se convirtió en el apuntador externo a la lista.
2. `newNodePtr`, que se usó en la creación de un nuevo nodo cuando fue necesario.
3. `currPtr`, que se actualizó para que siempre apunte al último nodo de la lista ligada.

Cuando elaboramos alguna lista ligada dinámica agregando cada nuevo nodo al final, siempre necesitamos tres apuntadores para realizar estas funciones. A continuación se muestra el algoritmo que hemos usado en forma generalizada para elaborar una lista ligada de números `int` que se lee del dispositivo estándar de entrada. Se supone que el usuario introduce por lo menos un número.

```
Establecer head = new NodeType
Leer head-> componente
Establecer currPtr = head

Leer inputVal
MIENTRAS NO SEA EL FINAL DEL ARCHIVO
    Establecer newNodePtr = new NodeType
    Establecer newNodePtr-> component = inputVal
    Establecer currPtr = newNodePtr
    Leer inputVal
Establecer currPtr-> link = NULL
```

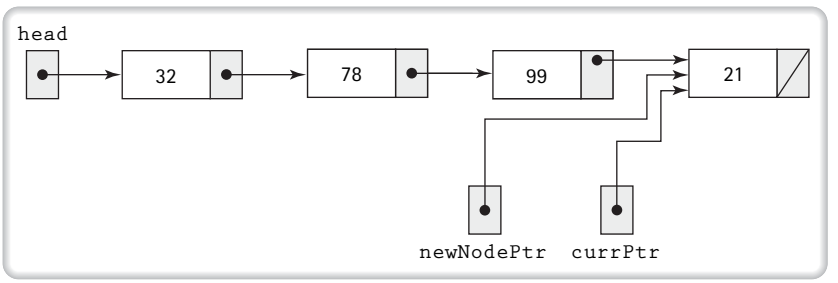
El siguiente fragmento de código implementa este algoritmo. Por motivos de variación, definimos el tipo de componente como `int` en lugar de `float`.

```
typedef int ComponentType;

struct  NodeType;               // Declaración directa
typedef NodeType* NodePtr;

struct NodeType
{
    ComponentType component;
    NodePtr       link;
};

NodePtr       head;             // Puntero externo para la lista
NodePtr       newNodePtr;       // Puntero para el nodo más reciente
NodePtr       currPtr;          // Puntero para el último nodo
ComponentType inputVal;

head = new NodeType;
cin >> head->component;
currPtr = head;

cin >> inputVal;
while (cin)
{
    newNodePtr = new NodeType;          // Create new node
    newNodePtr->component = inputVal;   // Set its component value
    currPtr->link = newNodePtr;         // Link node into list
    currPtr = newNodePtr;               // Set currPtr to last node
    cin >> inputVal;
}
currPtr->link = NULL;                   // Marcar el final de la lista
```

Haremos un repaso para ver cómo funciona este algoritmo en realidad.

`head = new NodeType;`	Se crea una variable del tipo `NodeType`. El apuntador se guarda en `head`. La variable `head` permanecerá sin cambios como apuntador al primer nodo (esto es, `head` es el apuntador externo a la lista).
`cin >> head->component;`	El primer número se extrae al miembro `component` del primer nodo de la lista.
`currPtr = head;`	`currPtr` ahora apunta al último nodo (el único nodo) de la lista.
`cin >> inputVal;`	El siguiente número (si es que existe) se extrae a la variable `inputVal`.
`while (cin)`	Se usa un ciclo controlado por evento para extraer valores de entrada hasta que ocurre `end-of-file`.
`{`	
`newNodePtr = new NodeType;`	Se crea otra variable del tipo `NodeType`, con `new NodePtr` apuntando a él.
`newNodePtr->component = inputVal;`	El valor de entrada actual se guarda en el miembro `component` del nodo de nueva creación.
`currPtr->link = newNodePtr;`	El apuntador al nuevo nodo se guarda en el miembro `link` del último nodo de la lista.
`currPtr = newNodePtr;`	`currPtr` nuevamente apunta al último nodo de la lista.
`cin >> inputVal;`	Se extrae el siguiente valor de entrada (si es que existe).
`}`	El cuerpo del ciclo se repite de nuevo.
`currPtr->link = NULL;`	Al miembro `link` del último nodo se asigna el valor especial `NULL` de end-of-list.

A continuación se muestra la lista ligada que resulta cuando el programa se ejecuta con los números 78, 99 y 21 como datos. Los valores finales se muestran para las variables auxiliares.



Algoritmos en listas ligadas dinámicas

Ahora que hemos visto dos ejemplos de creación de una lista ligada dinámica, vamos a ver algoritmos que procesan nodos en una lista ligada. Tenemos que estar en condiciones para insertar un nodo en una lista, borrar un nodo de una lista, imprimir los valores de datos en una lista, etc. Para cada una de estas operaciones usaremos `NULL`; recordemos que está en el miembro de liga del último nodo. `NULL` se puede asignar a cualquier variable apuntador; esto significa que el apuntador apunta a nada. Su importancia estriba en que podemos comparar el miembro de liga de cada nodo con `NULL` para ver cuándo hemos llegado al final de la lista.

El desarrollo de estos algoritmos lo haremos en el siguiente contexto: queremos escribir una clase C++ para una lista (no de una lista ligada). Como se destaca en la figura 16-5, una lista ADT

se puede organizar de varias maneras. Elegimos una lista ligada dinámica como la representación de datos para una lista, y creamos la clase `HybridList`, cuya especificación se muestra en la figura 16-6. La denominamos "`HybridList`" porque tiene dos operaciones de inserción y dos operaciones delete.

Figura 16-6 *Especificación de la clase* `HybridList`

```
//**************************************************************************
// ARCHIVO DE ESPECIFICACIÓN (hybrid.h)
// Este archivo da la especificación de un tipo de datos abstractos de
// lista híbrida con dos operaciones insertar y dos operaciones borrar.
// Los componentes de la lista se mantienen en orden de valor ascendente
//**************************************************************************

typedef int ComponentType;    // Tipo de cada componente
                              //   (un tipo simple o el tipo de cadena)

struct NodeType;              // Declaración directa
                              //   (La declaración completa está oculta
                              //   en el archivo de ejecución)
class HybridList
{
public:
    bool IsEmpty() const;
        // Poscondición:
        //     Valor de función == true, si la lista está vacía
        //                      == false, en caso contrario

    void Print() const;
        // Poscondición:
        //     Se ha producido todos los componentes de la lista (si existen)

    void InsertAsFirst( /* in */ ComponentType item );
        // Precondición:
        //     elemento < primer componente de la lista
        // Poscondición:
        //     el elemento es el primer componente de la lista
        //  && los componentes de List están en orden ascendente

    void Insert( /* in */ ComponentType item );
        // Poscondición:
        //     el elemento está en la lista
        //  && los componentes de List están en orden ascendente
    void RemoveFirst( /* out */ ComponentType& item );
        // Precondición:
        //     NOT IsEmpty()
        // Poscondición:
        //     elemento == primer componente de la lista en la entrada
        //  && el elemento ya no está en la lista
        //  && los componentes de List están en orden ascendente

    void Delete( /* in */ ComponentType item );
        // Precondición:
        //     el elemento está en alguna parte de la lista
```

```
        // Poscondición:
        //     La primera aparición del elemento ya no está en la lista
        //  && los componentes de List están en orden ascendente

    HybridList();
        // Constructor
        // Poscondición:
        //     Se crea la lista vacía

    HybridList( const SortedList2& otherList );
        // Constructor de copia
        // Poscondición:
        //     La lista se crea como un duplicado de otherList

    ~HybridList();
        // Destructor
        // Poscondición:
        //     Se destruye la lista
private:
    NodeType* head;
};
```

Observe que en la declaración de clase las precondiciones y poscondiciones de las funciones miembro no mencionan nada acerca de listas ligadas. La abstracción es una lista, no una lista ligada. El usuario de la clase está sólo interesado en la manipulación de listas de elementos y no le importa cómo las hemos organizado. Si cambiamos a una implantación diferente —un array, por ejemplo— la interfaz pública sigue siendo válida.

Los datos privados de la clase `HybridList` consisten en un solo elemento: una variable de apuntador `head`. Esta variable es el apuntador externo a una lista ligada dinámica. Como es el caso con todas las clases C++, diferentes objetos de clase tienen sus propias copias de los datos privados. Por ejemplo, suponga que el código del cliente declara y manipula dos objetos de clase como en el siguiente programa.

```
//**************************************************************************
// Programa ListDemo
// Éste es un cliente muy simple de la clase HybridList
//**************************************************************************
#include <iostream>
#include "hybrid.h"   // Para la clase HybridList

using namespace std;

int main()
{
    HybridList    list1;    // Primera lista, inicialmente vacía
    HybridList    list2;    // Segunda lista, vacía al principio
    ComponentType item;     // Un elemento de la lista

    list1.Insert(-35);
    list1.Insert(100);
    list1.Insert(12);
    cout << "First list:" << endl;
```

```
    list1.Print();                      // Imprime -35, 12 y 100
                                        //   en ese orden
    while ( !list1.IsEmpty() )
    {
        list1.RemoveFirst(item);
        if (item > 0)
            list2.Insert(item);
    }
    cout << endl;
    cout << "First list:" << endl;
    list1.Print();                      // Ningún resultado (la lista está
                                           vacía)
    cout << "Second list:" << endl;
    list2.Print();                      // Imprime 12 y 100
                                        //   en ese orden
    return 0;
}
```

Luego, en el programa ListDemo, cada uno de los dos objetos `list1` y `list2` tiene su propia variable privada `head` y mantiene su propia lista ligada dinámica en el área de almacenamiento libre.

En la figura 16-6, el archivo de especificación `hybridList.h` declara un tipo `NodeType`, pero sólo como una declaración directa. La única razón por la que se requiere declarar el identificador `NodeType` en el archivo de especificación es que sea posible especificar el tipo de datos de la variable privada `head`. Con la idea de la ocultación de informaciones colocamos la declaración completa de `NodeType` en el archivo de implementación `hybridList.cpp`. La declaración completa es un detalle que el usuario no necesita saber. `hybridList.cpp` empieza de la siguiente manera:

```
//****************************************************************************
// ARCHIVO DE EJECUCIÓN (hybrid.cpp)
// Este archivo pone en práctica las funciones miembros de clase HybridList
// Representación de lista: una lista vinculada de nodos dinámicos
//****************************************************************************
#include "hybridList.h"
#include <iostream>
#include <cstddef>     // Para NULL

using namespace std;

typedef NodeType* NodePtr;
struct NodeType
{
    ComponentType component;
    NodePtr       link;
};

// Miembros de clase privados:
//     NodePtr head;            Puntero externo para lista vinculada
  ⋮
```

A fin de ilustrar algunos algoritmos de uso común en listas ligadas dinámicas, veamos las aplicaciones de las funciones de miembros `HybridList`. Empecemos con la creación de una lista ligada vacía, que es el más fácil de los algoritmos.

Crear una lista ligada vacía Para crear una lista ligada sin nodos, sólo es necesario asignar el valor `NULL` al apuntador externo. Para la clase `HybridList`, el constructor de clase es el lugar apropiado para hacer esto:

```
HybridList::HybridList()

// Constructor

// Poscondición:
//     head == NULL

{
    head = NULL;
}
```

Como hemos estudiado en el capítulo 13, las afirmaciones de implementación (las precondiciones y poscondiciones que aparecen en el archivo de implementación) con frecuencia se enuncian de manera distinta de las afirmaciones abstractas (las que se ubican en el archivo de especificación). Las afirmaciones abstractas están escritas en términos significativos para el usuario del ADT; no se deberán mencionar detalles de implementación. En contraste, las afirmaciones de implementación se pueden hacer más precisas refiriéndose directamente a variables y algoritmos en el código de implementación. En el caso del constructor de clase `HybridList`, la poscondición abstracta es simplemente que se ha creado una lista vacía (no una lista ligada). Por otra parte, la poscondición de implementación

```
// Poscondición:
//     head == NULL
```

está formulada en términos de nuestros datos privados (`head`) y de nuestra particular implementación de lista (una lista ligada dinámica).

Prueba para una lista ligada vacía La función de miembro `IsEmpty` de `HybridList` devuelve `true` si la lista está vacía, y `false` si la lista no está vacía. Usando una representación de lista ligada dinámica, devolvemos `true` si `head` contiene el valor `NULL` y `false` en caso contrario:

```
bool HybridList::IsEmpty() const

// Poscondición:
//     Valor de retorno == true, si head == NULL
//                      == false, en caso contrario

{
    return (head == NULL);
}
```

Impresión de una lista ligada Para imprimir los componentes de una lista ligada tenemos que tomar acceso a los nodos uno por uno. Este requerimiento implica un ciclo controlado por eventos, donde el evento que detiene el ciclo se encuentra al final de la lista. La variable de control de ciclo es un apuntador que inicializa en el apuntador externo y avanza de nodo a nodo quedando en el mismo nivel que el miembro ligado del nodo actual. Cuando el apuntador de control de ciclo es igual a `NULL`, se ha llegado al último nodo.

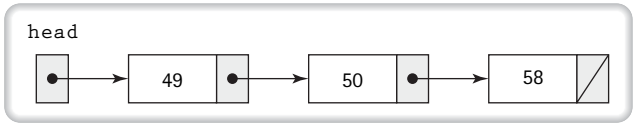
Imprimir ()

Establecer currPtr = head
WHILE currPtr no sea igual a NULL
  Imprimir miembro componente de *currPtr
  Establecer currPtr = miembro de enlace de *currPtr

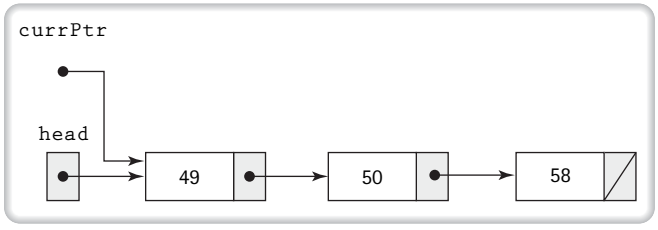
Observe que este algoritmo funciona correctamente aun cuando la lista está vacía (`head` es igual a `NULL`).

```
void HybridList::Print() const

// Poscondición:
//     Se ha producido los miembros componentes de todos los nodos
//     (si existen) en la lista vinculada

{
    NodePtr currPtr = head;    // Puntero de control de bucle

    while (currPtr != NULL)
    {
        cout << currPtr->component << endl;
        currPtr = currPtr->link;
    }
}
```

Haremos un repaso usando la siguiente lista.



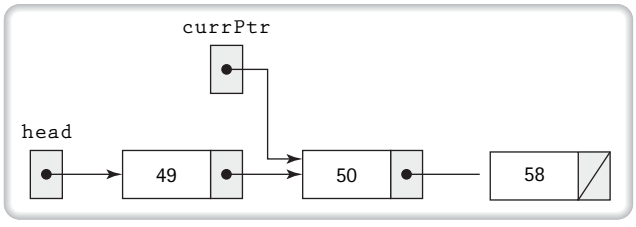
`currPtr = head;` Tanto `currPtr` como `head` apuntan al primer nodo de la lista.



`while (currPtr != NULL)` Se introduce al cuerpo del ciclo porque `currPtr` no es `NULL`.

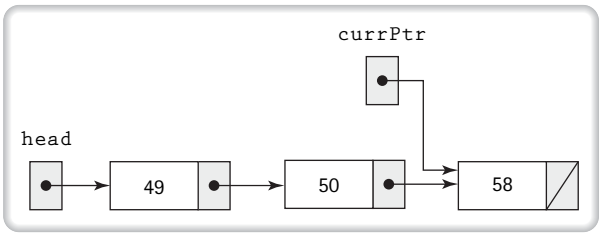
`cout << currPtr->component << endl;` Se imprime el número 49.

`currPtr = currPtr->link;` `currPtr` ahora apunta al segundo nodo de la lista.



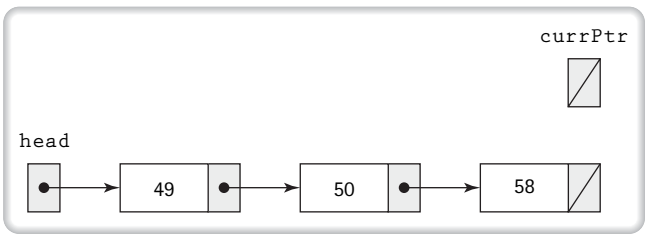
```
while (currPtr != NULL)
cout << currPtr->component << endl;
currPtr = currPtr->link;
```

Se repite el ciclo porque `currPtr` no es `NULL`.
Se imprime el número 50.
`currPtr` ahora apunta al tercer nodo de la lista.




```
while (currPtr != NULL)
cout << currPtr->component << endl;
currPtr = currPtr->link;
```

Se repite el ciclo porque `currPtr` no es `NULL`.
Se imprime el número 58.
`currPtr` ahora es `NULL`.




```
while (currPtr != NULL)
```

No se repite el ciclo porque `currPtr` es `NULL`.

Inserción en una lista ligada Una función para insertar un componente en una lista ligada debe tener un argumento, esto es, el elemento a insertar. La frase *inserción en una lista ligada* puede significar la inserción del componente en la parte superior de la lista (como el primer nodo) o la inserción del componente en su lugar correcto de acuerdo con un ordenamiento (alfabético o numérico). Examinemos cada una de estas situaciones.

Insertar un componente en la parte superior de una lista es fácil porque no tenemos que revisar la lista para buscar a dónde pertenece el elemento.

InsertAsFirst(entrada: elemento)

Establecer newNodePtr = new NodeType
Establecer el miembro componente de *newNodePtr = elemento
Establecer el miembro de enlace de *newNodePtr = head
Establecer head = newNodePtr

Este algoritmo se codifica en la siguiente función.

```
void HybridList::InsertAsFirst( /* in */ ComponentType item )

// Precondición:
//     Los miembros componentes de nodos de lista están en orden ascendente
//  && elemento < miembro componente del primer nodo de la lista
// Poscondición:
//     El nuevo nodo que contiene el elemento está en la parte superior de la
       lista vinculada
//  && los miembros componentes de los nodos de la lista están en orden
    ascendente
```

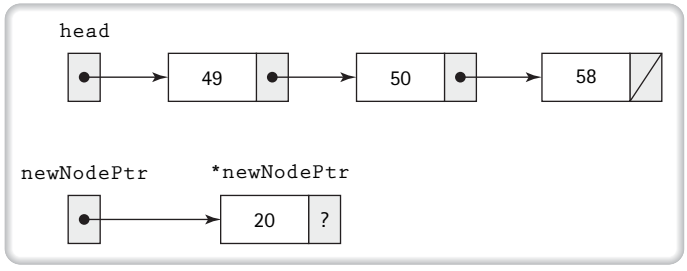
```
{
    NodePtr newNodePtr = new NodeType;     // Puntero temporal

    newNodePtr->component = item;
    newNodePtr->link = head;
    head = newNodePtr;
}
```

La precondición de la función indica que `item` debe ser menor que el valor en el primer nodo. Esta precondición no es un requerimiento de listas ligadas en general. Sin embargo, la abstracción de `HybridList` que estamos realizando es una lista ordenada. El contrato de precondición/poscondición indica que *si* el cliente transmite un valor menor que el primero de la lista, entonces la función garantiza que mantendrá el orden ascendente. Si el cliente viola la precondición, el contrato se anula.

El siguiente repaso de código muestra los pasos para la inserción de un componente con el valor 20 como primer nodo en la lista ligada que se imprimió en la última sección.

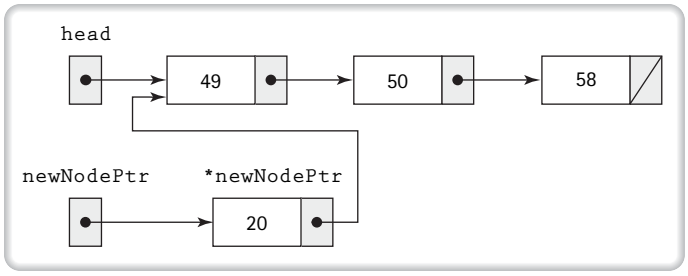
`newNodePtr = new NodeType;`
`newNodePtr->component = item;`

Se crea un nuevo nodo.
El número 20 se guarda en el miembro `componente` del nuevo nodo.



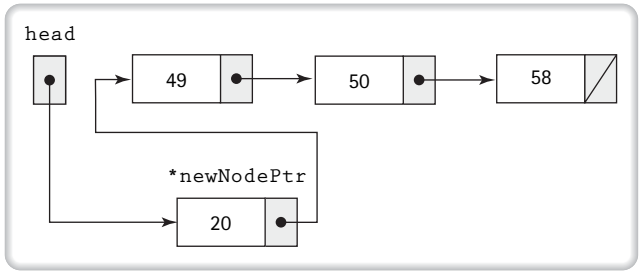
`newNodePtr->link = head;`

El miembro `link` de *`newNodePtr` ahora apunta al primer nodo de la lista.



`head = newNodePtr;`

El apuntador externo a la lista ahora apunta al nodo que contiene el nuevo componente.



Para insertar un componente en el lugar correcto en una lista ordenada es necesario hacer un ciclo a través de los nodos hasta que se encuentre el lugar donde que le corresponde al componente. Puesto que la clase `HybridList` mantiene los componentes en orden ascendente, podemos identificar el lugar de un componente buscando el nodo que contiene un valor mayor al valor que se inserta. Nuestro nuevo nodo se deberá insertar directamente antes del nodo con ese valor; por eso es necesario vigilar el nodo anterior a fin de insertar nuestro nuevo nodo. Usamos un apuntador `prevPtr` para apuntar a este nodo anterior. Este método conduce al siguiente algoritmo:

Insertar (entrada: elemento)

```
Establecer newNodePtr = new NodeType
Establecer el miembro componente de *newNodePtr = elemento
Establecer prevPtr = NULL
Establecer currPtr = head
WHILE elemento > miembro componente de *currPtr
    Establecer prevPtr = currPtr
    Establecer currPtr = miembro de enlace de *currPtr
Insertar *newNodePtr entre *prevPtr y *currPtr
```

Este algoritmo es básicamente sano, pero crea problemas en casos especiales. Si el nuevo componente es más grande que el resto, no ocurre el evento que detiene el ciclo (buscando un nodo cuyo componente es mayor que el componente a insertar). Cuando se llega al final de la lista, la condición While intenta invalidar `currPtr`, que ahora contiene `NULL`. En algunos sistemas el programa se colapsará. Podemos prevenir esta situación usando la siguiente expresión para controlar el ciclo While:

```
currPtr isn't NULL AND item > component member of *currPtr
```

Esta expresión nos protege contra la invalidación del apuntador nulo porque C++ usa la evaluación de cortocircuito de expresiones lógicas. Si la primera parte evalúa `false`, esto es, si `currPtr` es igual a `NULL`, la segunda parte de la expresión, que invalida `currPtr`, no se evalúa.

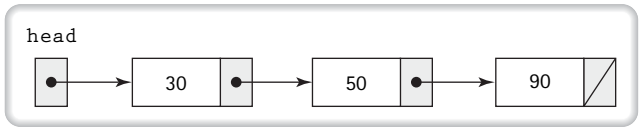
Hay un punto más que considerar en nuestro algoritmo: el caso especial donde la lista está vacía o el nuevo valor es menor que el primer componente de la lista. La variable `prevPtr` permanece `NULL` en este caso, y `*newNodePtr` debe insertarse en la parte superior y no entre `*prevPtr` y `*currPtr`.

La siguiente función reorganiza nuestro algoritmo con estos cambios ya incorporados.

```
void HybridList::Insert( /* in */ ComponentType item )

// Precondición:
//     Los miembros componentes de los nodos de la lista están en orden
       ascendente
// Poscondición:
//     El nuevo nodo que contiene al elemento está en su propio lugar
//       en la lista vinculada
//  && los miembros componentes de los nodos de la lista están en orden
    ascendente

{
    NodePtr currPtr;       // Puntero móvil
    NodePtr prevPtr;       // Puntero para el nodo antes de *currPtr
    NodePtr newNodePtr;    // Puntero para el nuevo nodo

    // Establecer el nodo que se insertará

    newNodePtr = new NodeType;
```

```
    newNodePtr->component = item;

    // Hallar el punto de inserción previo

    prevPtr = NULL;
    currPtr = head;
    while (currPtr != NULL && item > currPtr->component)
    {
        prevPtr = currPtr;
        currPtr = currPtr->link;
    }

    // Insertar el nuevo nodo

    newNodePtr->link = currPtr;
    if (prevPtr == NULL)
        head = newNodePtr;
    else
        prevPtr->link = newNodePtr;
}
```

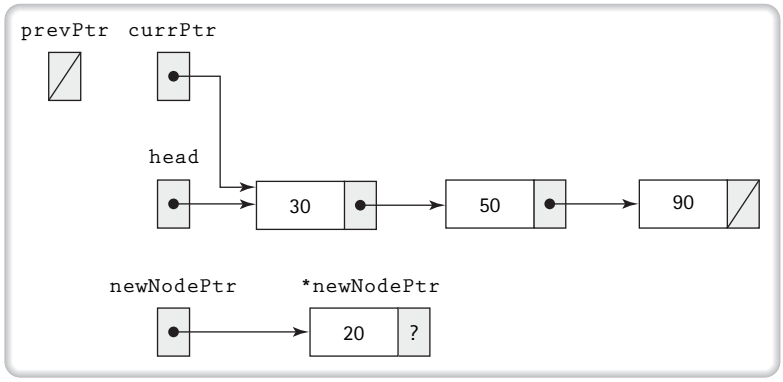
Repasemos este código para cada uno de los tres casos: inserción en la parte superior (`item` es 20), inserción en la parte media (`item` es 60), e inserción al final (`item` es 100). Cada inserción empieza con la lista que sigue a continuación.



Insert(20)

```
newNodePtr = new NodeType;
newNodePtr->component = item;
prevPtr = NULL;
currPtr = head;
```

Estas cuatro declaraciones inicializan las variables que se usan en el proceso de búsqueda. Las variables y su contenido se muestran enseguida.

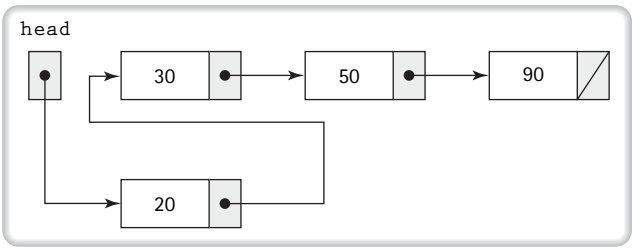



```
while (currPtr != NULL &&
  item > currPtr->component)
```

Puesto que 20 es menor que 30, la expresión es `false` y el ciclo no concluye.

```
newNodePtr->link = currPtr;

if (prevPtr == NULL)
head = newNodePtr;
```

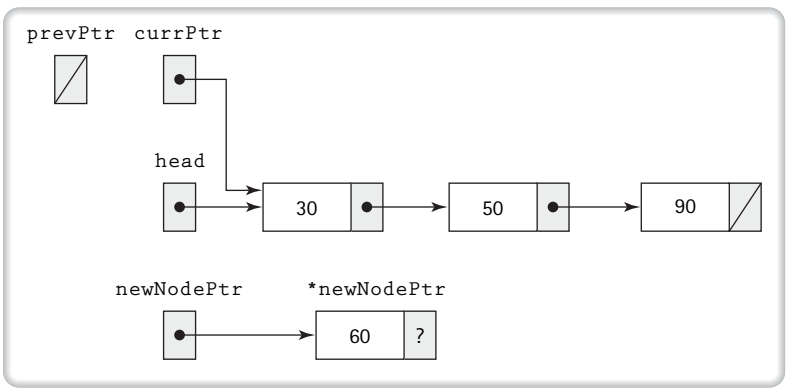
El miembro `link` de `*newNodePtr` ahora apunta a `*currPtr`.
Puesto que `prevPtr` es `NULL`, se ejecuta la cláusula Then y se inserta 20 en la parte superior de la lista.



Insert(60)

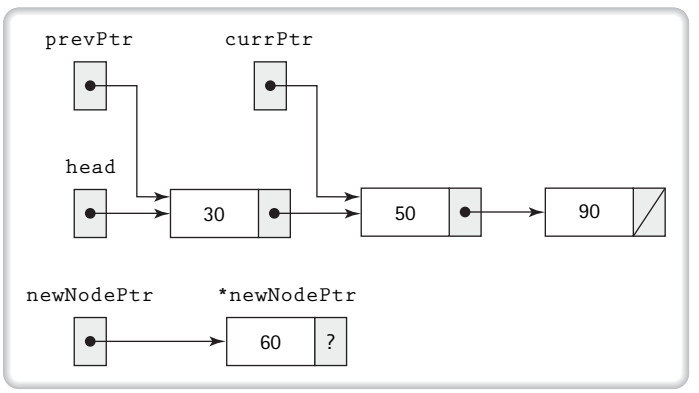
```
newNodePtr = new NodeType;
newNodePtr->component = item;
prevPtr = NULL;
currPtr = head;
```

Estas cuatro sentencias inicializan las variables que se usan en el proceso de búsqueda. Las variables y su contenido se muestran enseguida.




```
while (currPtr != NULL &&
  item > currPtr->component)
prevPtr = currPtr;
currPtr = currPtr->link;
```

Puesto que 60 es mayor que 30, esta expresión es `true` y se introduce el ciclo completo.
Se avanzan las variables de apuntadores.

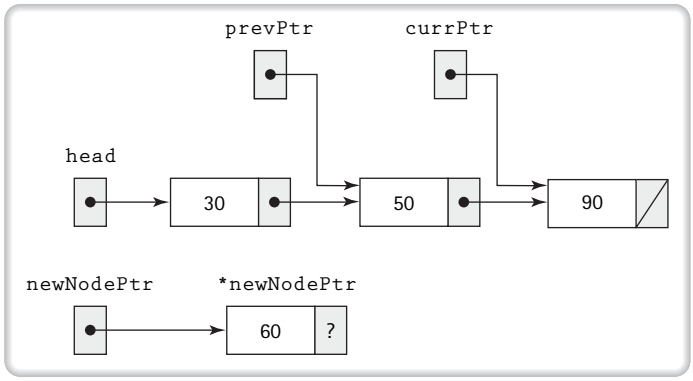



```
while (currPtr != NULL &&
   item > currPtr->component)
```

Puesto que 60 es mayor que 50, esta expresión es `true` y se repite el ciclo.

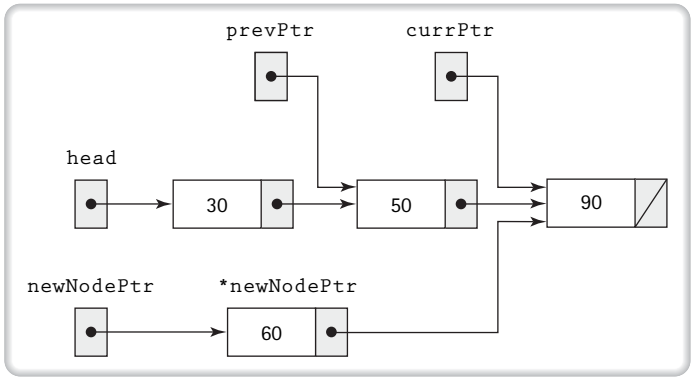
```
prevPtr = currPtr;
currPtr = currPtr->link;
```

Se avanzan las variables de apuntadores.



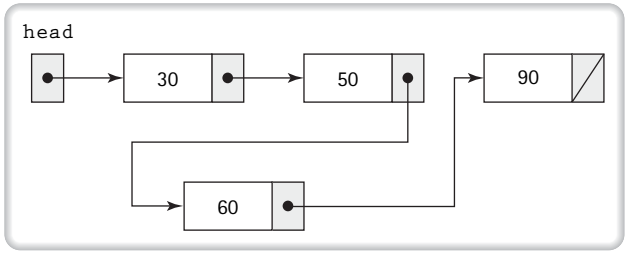

```
while (currPtr != NULL &&
   item > currPtr->component)
newNodePtr->link = currPtr;
```

Puesto que 60 no es mayor que 90, la expresión es `false` y no se repite el ciclo.
El miembro `link` de `*newNodePtr` ahora apunta a `*currPtr`.




```
if (prevPtr == NULL)
prevPtr->link = newNodePtr;
```

Puesto que `prevPtr` no es igual a `NULL`, se ejecuta la cláusula Else. Se muestra la lista completa con las variables auxiliares eliminadas.



Insert(100)

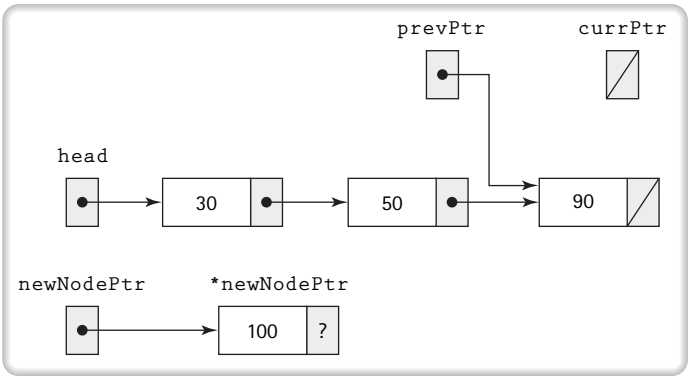
No repetimos la primera parte de la búsqueda, pero seguiremos con el repaso donde `prevPtr` apunta al nodo cuyo componente es 50 y donde `currPtr` está apuntando al nodo cuyo componente es 90.

```
while (currPtr != NULL &&
  item > currPtr->component)
```

Puesto que 100 es mayor que 90, esta expresión es `true` y se repite el ciclo.

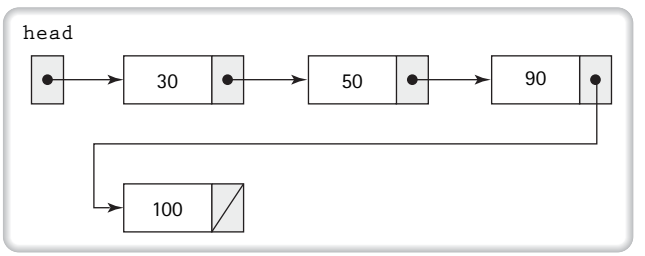
```
prevPtr = currPtr;
currPtr = currPtr->link;
```

Avanzan las variables de apuntadores.




```
while (currPtr != NULL &&
   item > currPtr->component)
newNodePtr->link = currPtr;

if (prevPtr == NULL)
prevPtr->link = newNodePtr;
```

Puesto que `currPtr` es igual a `NULL`, la expresión es `false` y no se repite el ciclo.
`NULL` es copiado al miembro `link` de `*newNodePtr`.
Puesto que `prevPtr` no es igual a `NULL`, se ejecuta la cláusula else. Se inserta el nodo `*newNodePtr` después de `*prevPtr`. Se muestra la lista completa con las variables auxiliares eliminadas.



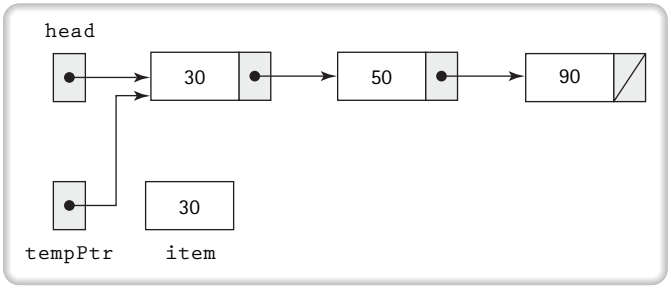
Borrado de una lista ligada Para borrar un nodo de una lista ligada, tenemos que hacer un ciclo a través de los nodos hasta que encontremos qué queremos borrar. Vemos la imagen espejo de nuestras inserciones: borrando del nodo superior y borrando un nodo cuyo componente es igual que un parámetro de entrada.

Para borrar el primer nodo sólo cambiamos el apuntador externo para que apunte al segundo nodo (o para que contenga `NULL` si estamos borrando el único nodo en una lista de un solo nodo). El valor en el nodo a borrar puede ser devuelto como un parámetro de salida. Observe la precondición para la siguiente función: el cliente no deberá llamar la función si la lista está vacía.

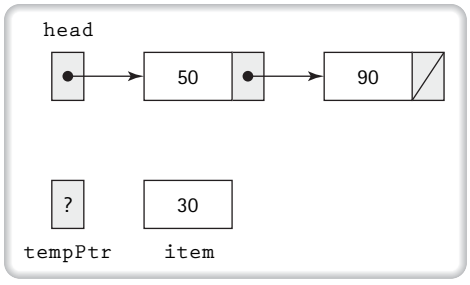
```
void HybridList::RemoveFirst( /* out */ ComponentType& item )

// Precondición:
//     La lista vinculada no está vacía (head ! = NULL)
//  && los miembros componentes de los nodos de la lista están en orden
    ascendente
// Poscondición:
//     elemento == miembro componente del primer nodo de la lista en la
       entrada
//  && el nodo que contiene al elemento ya no está en la lista vinculada
//  && los miembros componentes de los nodos de la lista están en orden
    ascendente
```

```
{
    NodePtr tempPtr = head;     // Puntero temporal

    item = head->component;
    head = head->link;
    delete tempPtr;
}
```

No mostramos un repaso completo porque el código es muy obvio. En su lugar mostramos el estado de la estructura de datos en dos etapas: después de las primeras dos sentencias y al final. Usamos una de nuestras listas anteriores. Lo que sigue es la estructura de datos después de la ejecución de las primeras dos sentencias de la función.



Después de la ejecución de la función, la estructura queda como sigue:



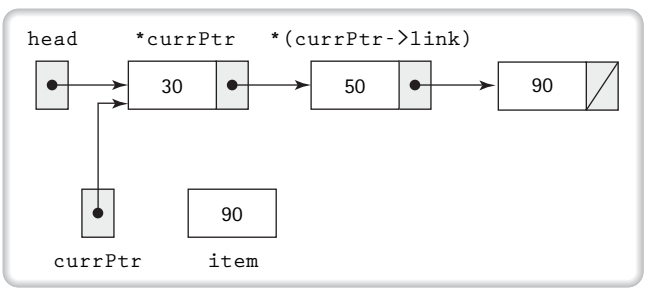
La función para borrar un nodo cuyo componente contiene un determinado valor es similar a la función `Insert`. La diferencia es que estamos buscando una correspondencia, y no un miembro `component` mayor que nuestro `item`. Puesto que la precondición de la función indica que el componente que buscamos definitivamente se encuentra en la lista, nuestro control de ciclo es sencillo. No tenemos que preocuparnos de la invalidación del apuntador nulo.

Como en la función `Insert`, necesitamos el nodo antes del nodo a borrar para que podamos cambiar su miembro `link`. En la siguiente función mostramos otra técnica para mantener la cuenta del nodo anterior. En lugar de comparar `item` con el miembro `component` de `*currPtr`, lo comparamos con el miembro `component` del nodo a que apunta `currPtr->link`; esto es, que comparamos `item` con `currPtr->link->component`. Cuando `currPtr->link->component` es igual a `item`, `*currPtr` es el nodo anterior.

```
void HybridList::Delete( /* in */ ComponentType item )

// Precondición:
//     elemento == miembro componente de algún nodo de la lista
//  && los miembros componentes de los nodos de la lista están en orden
    ascendente
```

```
// Poscondición:
//     El nodo que contiene la primera aparición del elemento ya no está en
       la lista vinculada
//       linked list
//  && los miembros componentes de los nodos de la lista están en orden
    ascendente
{
    NodePtr delPtr;     // Puntero para el nodo que será eliminado
    NodePtr currPtr;    // Puntero de control de bucle

    // Comprobar si el elemento está en el primer nodo

    if (item == head->component)
    {
        // Borrar el primer nodo

        delPtr = head;
        head = head->link;
    }
    else
    {
        // Buscar el nodo en el resto de la lista

        currPtr = head;
        while (currPtr->link->component != item)
            currPtr = currPtr->link;

        // Borrar *(currPtr -> link)

        delPtr = currPtr->link;
        currPtr->link = currPtr->link->link;
    }
    delete delPtr;
}
```

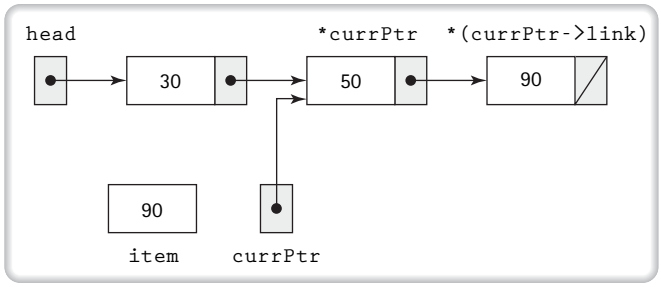
Vamos a borrar el nodo cuyo componente es 90. La estructura se muestra a continuación. Los nodos están etiquetados en el punto en se alcanza la declaración While.



`while (currPtr->link->component != item)` Puesto que 50 no es igual a 90, se introduce el ciclo.

```
currPtr = currPtr->link;
```

Avanza el apuntador.

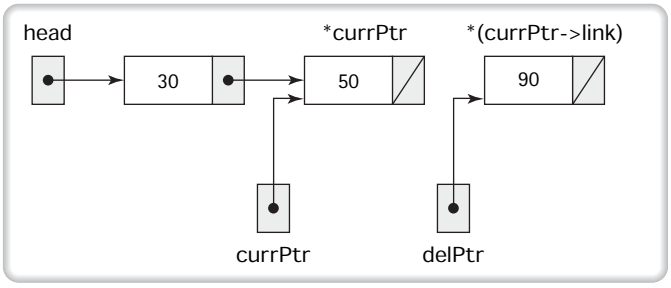



```
while (currPtr->link->component != item)
```

Puesto que 90 es igual a 90, el ciclo se abandona.

```
delPtr = currPtr->link;
currPtr->link = currPtr->link->link;
```

El miembro `link` del nodo cuyo componente es 90 se copia al miembro `link` del nodo cuyo componente es 50. En este caso, el miembro `link` es igual a `NULL` .



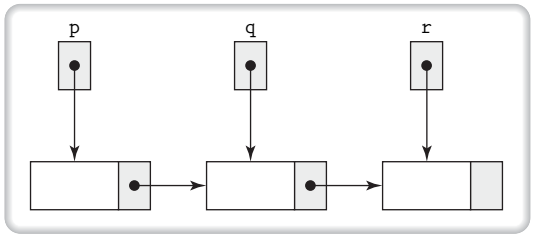

```
delete delPtr;
```

La memoria asignada a `*delPtr` (el apuntador que fue borrado) se devuelve al área de almacenamiento libre. El valor de `delPtr` es indefinido.

Observe que `NULL` fue guardado en `currPtr->link` sólo porque el nodo cuyo componente era 90 fue el último en la lista. Si hubieran existido más nodos después de éste, se hubiera guardado un apuntador al siguiente nodo en `currPtr->link`.

Expresiones con apuntadores

Como se puede ver en la función `HybridList::Delete`, las expresiones con apuntadores pueden ser muy complejas. Veamos algunos ejemplos.



`p`, `q` y `r` apuntan a nodos en una lista ligada dinámica. Los propios nodos son `*p`, `*q` y `*r`. Use el diagrama anterior para convencerse de que lo siguiente es verdadero.

```
p->link == q
*(p->link) es el mismo nodo que *q
p->link->link == r
*(p->link->link) es el mismo nodo que *r
q->link == r
*(q->link) es el mismo nodo que *r
```

Y recuerde la semántica de declaraciones de asignación para apuntadores.

`p = q;` Asigna el contenido del apuntador `q` al apuntador `p`.
`*p = *q;` Asigna el contenido de la variable a la que apunta `q` a la variable a la que apunta `p`.

Clases y listas ligadas dinámicas

En el capítulo 15 dijimos que las clases cuyos objetos manipulan datos dinámicos en el área de almacenamiento libre deberán proporcionar no sólo un constructor de clase sino también un destructor, una operación de copia profunda y un constructor de copia. La clase `HybridList` incluye todos excepto (para mantener el ejemplo más sencillo) una operación de copia profunda. Veamos el destructor de clase.

El propósito del destructor es la omisión de la lista ligada dinámica cuando se destruye un objeto de clase `HybridList`. Sin destructor, la lista ligada se quedaría en el área de almacenamiento libre, todavía asignada, pero inaccesible. El código para el destructor es fácil de escribir. Usando las funciones de miembros `IsEmpty` y `DeleteTop` existentes, simplemente avanzamos a través de la lista y borramos cada nodo:

```
HybridList::~HybridList()

// Destructor

// Poscondición:
//     Todos los nodos de la lista vinculada han sido cancelados del
       almacenamiento libre
{
    ComponentType temp;     // Variable temporal

    while ( !IsEmpty() )
        RemoveFirst(temp);
}
```

El constructor de copia es más difícil de escribir. Antes de contemplarlo, debemos destacar la importancia de proporcionar un constructor de copia cada vez que también proporcionemos un destructor. Consideremos que `HybridList` no tiene constructor de copia, y supongamos que un cliente pasa un objeto de clase a una función, usando un paso por valor. (Recuerde que el paso de un argumento por valor transmite una *copia* del valor del argumento a la función.) Dentro de la función el parámetro se inicializa para que sea una copia del objeto de clase del invocador, incluyendo el valor del invocador de la variable privada `head`. En este momento, tanto el argumento como el parámetro están apuntando a la misma lista ligada dinámica. Cuando vuelve la función del cliente, el destructor de clase es invocado para el parámetro, destruyendo la única copia de la lista ligada. Cuando regresa de la función, ¡la lista ligada del invocador ha desaparecido!

Proporcionar un constructor de copia asegura el copiado profundo de un argumento a un parámetro cada vez que ocurre un paso por valor. La ejecución de un constructor de copia, como se muestra a continuación, emplea un algoritmo de uso común para la creación de una nueva lista ligada en forma de una copia de otra.

```
HybridList::HybridList( const SortedList2& otherList )

// Constructor de copia

// Poscondición:
//     IF otherList.head == NULL (es decir, la otra lista está vacía)
//         head == NULL
//     ELSE
//         head apunta a una nueva lista vinculada que es una copia de
//         la lista enlazada señalada por otherList.head

{
    NodePtr fromPtr;    // Puntero hacia la lista que es copiada
    NodePtr toPtr;      // Puntero hacia la nueva lista que se construye

    if (otherList.head == NULL)
    {
        head = NULL;
        return;
    }
    // Copiar el primer nodo

    fromPtr = otherList.head;
    head = new NodeType;
    head->component = fromPtr->component;

    // Copiar los nodos restantes

    toPtr = head;
    fromPtr = fromPtr->link;
    while (fromPtr != NULL)
    {
        toPtr->link = new NodeType;
        toPtr = toPtr->link;
        toPtr->component = fromPtr->component;
        fromPtr = fromPtr->link;
    }
    toPtr->link = NULL;
}
```

16.4 Elección de la representación de datos

Hemos examinado en detalle dos formas de representar listas de componentes: una forma en que los componentes se encuentran físicamente uno al lado de otro (una representación de array directo, como se muestra en la figura 16-1), y otra en que los componentes están ubicados uno al lado de otro de manera lógica (una lista ligada). Además, una lista ligada es una abstracción que puede implementarse ya sea usando un array de estructuras o usando estructuras y apuntadores dinámicamente asignados (una lista ligada dinámica).

Comparemos la representación de array con la representación de la lista ligada dinámica. (Durante este análisis usaremos *array* con el significado de una representación de array directo, y no un array de estructuras (*structs*) que forman una lista ligada.) Veamos operaciones comunes en listas y examinemos las ventajas y desventajas de cada representación para cada operación.

Operaciones comunes

1. Extraer los componentes a una lista inicialmente vacía.
2. Acceder a todos los componentes de la lista en secuencia.
3. Insertar o borrar el primer componente de la lista.
4. Insertar o borrar el último componente de la lista.
5. Insertar o borrar el componente *n* de una lista.
6. Acceder al componente *n* de una lista.
7. Ordenar los componentes de una lista.
8. Revisar la lista en busca de un componente específico.

Leer componentes dentro de una lista es más rápido mediante una representación de array que mediante una lista ligada dinámica porque la operación `new` no se tiene que ejecutar para cada componente. Acceder a los componentes en secuencia requiere casi el mismo tiempo con ambas estructuras.

Insertar o borrar el primer componente es mucho más rápido usando una representación ligada. Recuerde que, en caso de un array, todos los demás componentes de la lista se tienen que desplazar hacia abajo (para una inserción) o hacia arriba (para una eliminación). A la inversa, insertar o borrar el último componente es mucho más eficiente por medio de un array; hay acceso directo al último componente y no se requiere desplazamiento. En una representación ligada se tiene que revisar toda la lista para encontrar el último componente.

En promedio, el tiempo necesario para insertar o borrar el componente *n* es casi igual para los dos tipos de lista. Una representación ligada sería mejor para valores pequeños de *n*, y una representación de array sería mejor para valores de *n* cerca del final de la lista.

Llegar al elemento *n* es *mucho* más rápido en una representación de array. Podemos acceder a un elemento en forma directa usando $n - 1$ como el índice dentro del array. En una representación ligada tenemos que acceder a los primeros componentes de $n - 1$ en forma secuencial para llegar al componente *n*.

Para muchos algoritmos de ordenación, incluyendo los tipos de ordenamiento, las dos representaciones son casi iguales en eficiencia. Sin embargo, hay algunos algoritmos de ordenamiento más elaborados y muy rápidos que dependen del acceso directo a elementos de array usando índices de arrays. Estos algoritmos no son adecuados para una representación ligada, que requiere el acceso secuencial a los componentes.

En general, revisar una lista ordenada buscando un componente específico es mucho más rápido en una representación de array porque se puede hacer una búsqueda binaria. Cuando los componentes de la lista a revisar no se encuentran ordenados, las dos representaciones son casi iguales.

Cuando quiera decidir entre el uso de una representación de array y una representación ligada, determine cuál de estas operaciones comunes se aplicará quizá con más frecuencia. Analice para determinar qué estructura sería mejor en el contexto de su problema particular.

Un punto adicional a considerar para la decisión entre el uso de un array o una lista ligada dinámica es: ¿con qué grado de precisión puede usted predecir el máximo número de componentes de la lista? ¿Fluctúa el número de componentes de la lista ampliamente? Si usted conoce el máximo, y si este número permanece relativamente constante, lo adecuado en términos del uso de memoria es una representación de array. Si éste no es el caso, será mejor elegir una representación ligada dinámica a fin de usar la memoria de manera más eficiente.

Hay un punto final a considerar para la decisión entre una implementación basada en array o una ligada. Una estructura basada en array se puede escribir en un archivo y extraer posteriormente. Una estructura ligada sólo es válida para la construcción del programa en el cual se forma. Puesto que las ligas son direcciones de memoria, y la memoria se puede asignar de manera diferente cada vez que se ejecuta un programa, no podemos escribir una lista ligada directamente en un archivo para usarla después. Claro que los *componentes* se pueden escribir en un archivo, pero el programa tendría que recrear la lista cada vez que se ejecuta.

Caso práctico de resolución de problemas

El calendario de citas completo

PROBLEMA Complete el calendario de citas mediante la creación de una lista de objetos de la clase `Day`.

ANÁLISIS ¡Finalmente hemos llegado! Tenemos todas las piezas, sólo tenemos que completar la lista de objetos `Day`. Antes de examinar la estructura, determinemos las operaciones que necesitamos. Estamos definiendo el nivel final ahora, así que necesitamos las operaciones que se hacen con un calendario de citas.

Crear el calendario
Insertar un día
Insertar una cita dada a una fecha
Borrar una cita dada una fecha y una hora
Revisar si una hora está libre, dada una fecha
Imprimir las citas de un día, dada una fecha

ESTRUCTURAS DE DATOS Ya sabemos cómo crear una lista de objetos. En cada fase de la construcción de este calendario de citas hemos cambiado y aumentado la estructura de la lista mientras hemos aprendido nuevas técnicas. Dejemos por ahora la clase `Day`, y veamos simplemente la estructura para la lista de días. Podríamos crear una lista con base en arrays como la que hemos usado para la lista de objetos de la clase `Entry`. Podríamos hacer una lista ligada de días ordenada por fechas. También tenemos una opción diferente y no tan obvia.

Siempre hay 12 meses en un año, pero el número de días con citas varía por el número de días en un mes, el número de fines de semana y el número de días festivos. Una estructura eficiente sería un array de 12 registros de meses donde cada celda contiene un apuntador a una lista de días ordenados por días. Es decir, que en el array, el mes sería un índice para las citas del mes correspondiente. La lista de citas sería una lista ligada de objetos `Day` ordenados por los días del mes. Véase la figura 16-7. La estructura ligada a la que apunta `calendar[0]` se repite para cada mes para el cual existe una cita.

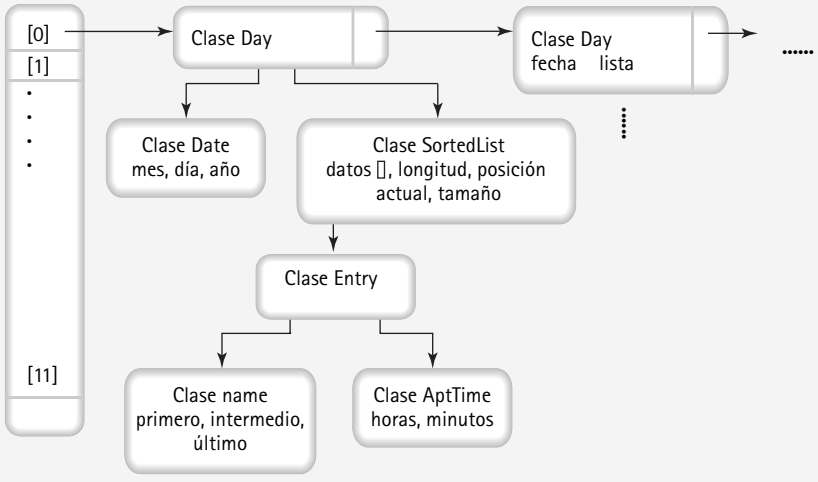


Figura 16-7

¿Qué información adicional necesitamos en nuestro programa final además del array `calendar`? Sería bueno tener el nombre del mes, no sólo el número del mes en la lista de salida. Podemos declarar un array de cadenas para los nombres de los meses, con acceso por medio del número del mes menos uno; el mismo valor que indexa la lista de citas para el mes. Puesto que sólo la operación de impresión necesita esta información, podemos hacer esta estructura local para dicha operación.

¡Es una estructura muy compleja! Primero crearemos una lista de días. Una vez que esta estructura está codificada y verificada, podemos diseñar el programa principal que declara un array de 12 de esta lista.

Lista ligada de días ¿Podemos usar las operaciones en la lista híbrida ADT en este capítulo? Veamos. Tenemos que insertar un día en la lista y encontrar un día a fin de ejecutar todas las operaciones que dicen "dada una fecha". Hay dos operaciones de inserción en la lista híbrida, una de las cuales inserta un objeto en su lugar correcto. Sin embargo, no se ha definido ninguna operación de búsqueda separada, y ninguna de las otras operaciones nos sirve aquí. Entonces, sólo vamos a canibalizar el código, definiendo una inserción y una búsqueda de nuestra lista de días. (Recuerde: nunca reinvente la rueda.)

A continuación se muestra el archivo de especificación para la clase `DayList`, incorporando todas las operaciones que hemos definido en nuestro estudio.

```
//**************************************************************************
// ARCHIVO DE ESPECIFICACIÓNI (DayList.h)
// Este archivo da la especificación de la clase DayList, que
// es una lista de objetos de clase Day. Dos funciones operan
// en la clase DayList: InsertDay y FindDay. Las otras funciones
// operan en los componentes de la lista. Para ahorrar espacio
// se omiten de cada función los comentarios de precondición
// que documentan las suposiciones hechas acerca de los datos
// de parámetros de entrada válidos. Éstos se incluirían en un programa
// propio para uso real.
//**************************************************************************

#include <iostream>
#include <fstream>
#include "Date.h"

using namespace std;

struct NodeType;

class DayList
{
public:
    void InsertDay(Date date);

        // Precondición:
        //     Los miembros componentes de la lista de días están
        //     en orden ascendente
        // Poscondición:
        //     El nuevo nodo que contiene la fecha está en su propio lugar
        //     en la lista de días
        //  && los miembros componentes de la lista de días están en orden
        //     de día ascendente

    void InsertApt(Date date);

        // Precondición:
        //     Los miembros componentes de la lista de días están en orden
        //     de día ascendente
        //  && la lista de entradas para el día están en orden de tiempo
        // Poscondición:
        //     Una entrada introducida por el usuario se inserta en la lista
        //     de elementos para el día dado
        //  && la lista de entrada está en orden de tiempo ascendente
```

```
    bool IsFree(Date date);

        // Precondición:
        //     Los miembros componentes de la lista de días están en orden
        //     de día ascendente
        //  && la lista de entradas para el día están en orden de tiempo
        // Poscondición:
        //     El valor de retorno es verdadero si el tiempo introducido por
        //     el usuario está libre en un determinado día; falso en caso
                contrario

    void PrintDay(ofstream& outFile, Date date);

        // Precondición:
        //     Los miembros componentes de la lista de días están en orden
        //     de día ascendente
        // Poscondición:
        //     La lista de entrada para día ha sido escrita en outFile

    void DeleteApt(Date date);

        // Poscondición:
        //     Los miembros componentes de la lista de días están en orden
        //     de día ascendente
        //  && la lista de entradas para la fecha están en orden de tiempo
        // Poscondición:
        //     El día ha sido borrado de la lista de días

    DayList();

        // Poscondición:
        //     Se crea la lista vacía

private:
    NodeType* dayList;

    NodeType* FindDay(Date date);

        // Precondición:
        //     Los miembros componentes de la lista de días están en orden
        //     de día ascendente
        // Poscondición:
        //     El valor de retorno es un puntero para el día para la fecha
        //     dada o NULL si no se halla el día.
};
```

Lea cuidadosamente la documentación. Observe que las precondiciones indican el orden de los elementos de la lista que están manipulando. Aunque las operaciones originales están en términos de una fecha, nuestra estructura garantiza que la lista de días sólo contiene las fechas del mismo mes.

Crear una `DayList` El único registro de datos en la clase `DayList` es un apuntador; lo tenemos que fijar en NULL para representar la lista vacía. No hemos incluido ni un destructor ni un constructor de copia para este problema. La lista de días no se transmite como un parámetro, no es el valor de devolución de una función, y es declarada en el nivel superior. Sale del alcance cuando el programa está completo.

Insertar un objeto `Day` en la lista Para esta operación, podemos usar el algoritmo que se ha proporcionado en la página 863.

Encontrar un objeto `Day` mediante un día dado Tenemos que fijar un apuntador móvil al principio de la lista y continuar moviendo el apuntador a través de la lista hasta que encuentre la fecha o se llegue al final de la lista.

HallarDía(entrada: fecha)
Salida: valor de función

```
Establecer currPtr en dayList
WHILE currPtr != NULL AND currPtr -> component.DateIs()! = date
    Incrementar currPtr
Devolver currPtr
```

Tenemos que lograr que esta función devuelva un apuntador al nodo que queremos, y no el propio nodo. ¿Por qué? Si devolvemos el propio nodo, habrá un error si el día con la fecha dada no se encuentra.

InsertApt Podemos usar la operación `FindDay` para encontrar el día con la fecha dada, pedir que el usuario introduzca el nombre y la hora, e insertar la entrada en la fecha.

InsertApt(entrada: fecha)

```
Establecer day en FindDay(fecha)
IF day es NULL
    Escribir "El día no ha sido inicializado"
ELSE
    entry.ReadEntry()
    day -> component.InsertEntry(entry)
```

DeleteApt Esta operación es idéntica a la operación `InsertApt`, excepto que se aplica `Delete` en lugar de `InsertEntry`.

IsFree Esta operación sólo aplica la función `TimeFree` al componente devuelto.

Imprimir las citas para un día Usamos `FindDay` para encontrar el día a imprimir. Si se devuelve `NULL`, se deberá imprimir un mensaje de error; de lo contrario, iteramos a través de la lista imprimiendo cada componente.

PrintDay(entrada-salida: outFile; entrada: fecha)

```
Establecer day en FindDay(fecha)
IF day es NULL
    Escribir en outFile "El día no ha sido inicializado"
ELSE
    Escribir el encabezado en outFile
    Establecer el límite en Número de entradas
    Restablecer las entradas
    WHILE límite no es 0
        Establecer la entrada en el siguiente elemento
        Escribir en outFile nombre y tiempo
        Disminuir el límite
```

A continuación se proporciona el archivo de implementación para la clase `DayList`.

```
//**************************************************************************
// ARCHIVO DE EJECUCIÓN (DayList.cpp)
// Este archivo pone en práctica la clase DayList, que es una lista
// vinculada de objetos de clase Day.
//**************************************************************************
#include "DayList.h"
#include <iostream>
#include "Day.h"
#include <cstddef>                  // Para NULL
using namespace std;

typedef NodeType* NodePtr;

struct NodeType
{
    Day component;
    NodePtr link;
};

DayList::DayList()

// Constructor

// Poscondición:
//     dayList is NULL

{
    dayList = NULL;
}

NodePtr DayList::FindDay(Date date)

// Precondición:
//     los miembros componentes de la lista están en orden de fecha
       ascendente
// Poscondición:
//     El valor de retorno es un puntero para el día para la fecha
//     especificada o NULL si no se halla el día.

{
    NodePtr currPtr = dayList;

    while (currPtr != NULL &&
           currPtr->component.DateIs().ComparedTo(date) != SAME)
        currPtr = currPtr->link;
    return currPtr;
}

void DayList::InsertApt(Date date)
```

```
// Precondición:
//     los miembros componentes de los nodos de la lista de días están en
       fecha ascendente
//  && la lista de entradas para fecha está en orden de tiempo
// Poscondición:
//     El nuevo nodo que contiene el elemento se inserta en la lista de
       entradas
//  && los miembros componentes de la lista están en orden de tiempo
    ascendente

{
    Entry entry;
    NodePtr day;

    day = FindDay(date);
    if (day == NULL)
        cout << "El día no ha sido inicializado" << endl;
    else
    { // Insertar la entrada en la lista de entradas para fecha
        entry.ReadEntry();
        day->component.InsertEntry(entry);
    }
}

void DayList::InsertDay(Date date)

// Precondición:
//     los miembros componentes de la lista de días están en orden de fecha
//     ascendente
// Poscondición:
//     El nuevo nodo que contiene al elemento está en su propio lugar
//     en la lista de días
//  && los miembros componentes de la lista de días están en orden de fecha
    ascendente

{
    int numberOfApts;
    cout << "¿Cuántas citas se pueden hacer en esta fecha?"
         << endl;
    cin >> numberOfApts;
    Day day(date, numberOfApts);

    NodePtr currPtr;       // Puntero móvil
    NodePtr prevPtr;       // Puntero para nodo antes de *currPtr
    NodePtr newNodePtr;    // Puntero para nuevo nodo

    // Establecer el nodo que se insertará

    newNodePtr = new NodeType;
    newNodePtr->component = day;

    // Hallar el punto de inserción
    prevPtr = NULL;
    currPtr = dayList;
```

```
    while (currPtr != NULL &&
        date.ComparedTo(currPtr->component.DateIs()) == AFTER)
    {
        prevPtr = currPtr;
        currPtr = currPtr->link;
    }

    // Insertar el nuevo nodo

    newNodePtr->link = currPtr;
    if (prevPtr == NULL)
        dayList = newNodePtr;
    else
        prevPtr->link = newNodePtr;
}

void DayList::DeleteApt(Date date)

// Precondición:
//     Los miembros componentes de la lista de días están en orden de fecha
       ascendente
//  && la lista de elementos para fecha están en orden de tiempo
// Poscondición:
//     El tiempo leído desde el teclado se elimina de la lista de entradas
//     para fecha si está presente; de lo contrario se escribe un mensaje de
       error

{
    AptTime time;
    time.ReadTime();
    Entry entry(" ", " ", " ", time.Hours(), time.Minutes());

    NodePtr  day;

    day = FindDay(date);

    if (day == NULL)
        cout << "El día no ha sido inicializado" << endl;
    else
        day->component.Delete(entry);
}

bool DayList::IsFree(Date date)

// Precondición:
//     Los miembros componentes de la lista de días están en orden de fecha
       ascendente
//  && la lista de entradas para fecha están en orden de tiempo
// Poscondición:
//     El valor de retorno es verdadero si el tiempo definido por el usuario
//     está libre en el día especificado; falso en caso contrario
```

```
{
    AptTime time;
    time.ReadTime();

    NodePtr day;

    day = FindDay(date);

    if (day == NULL)
    {
        cout << "El día no ha sido inicializado" << endl;
        return true;
    }
    else
        return day->component.TimeFree(time);
}

void DayList::PrintDay(ofstream& outFile, Date date)

// Precondición:
//     Los miembros componentes de la lista de días están en orden de fechas
       ascendente
// Poscondición:
//     La lista de entradas para fecha ha sido escrita en outFile si está
//     presente; de lo contrario se ha escrito un mensaje de error

{
    static string  months[12] = {"January", "February", "March",
        "April", "May", "June", "July", "August", "September",
        "October", "November", "December"};

    int limit;
    Entry entry;
    NodePtr day;

    day = FindDay(date);

    if (day == NULL)
       outFile << "El día no ha sido inicializado " << endl;
    else
    {
        outFile << "Citas para " << months[date.Month() -1]
                << " " << date.Day()
                << ", " << date.Year() << endl;

        limit = day->component.NumberOfEntries();
        day->component.ResetEntries();
        while (limit != 0)
        {
            entry = day->component.GetNextItem();
            outFile << entry.NameStr() << " at "
                    << entry.TimeStr() << endl;
            limit--;
```

```
        }
        outFile << endl;
    }
}
```

PRUEBA Se deberá elaborar un controlador que verifique cada operación de la clase.

Programa principal ¡Finalmente podemos escribir el programa principal para nuestro calendario de citas! La estructura de datos se declara de la siguiente manera:

```
DayList calendar[12];
```

El programa principal es simplemente un controlador que pide que el usuario introduzca un comando. Este comando, una cadena, se convierte en un tipo enumerado que se puede usar en un sentencia switch. Los diferentes casos en la sentencia switch obtienen una fecha en caso necesario y llaman a las operaciones `DayList` apropiadas. Todo este procesamiento está encerrado en un ciclo.

Principal

```
Establecer notQuit en verdadero

WHILE notQuit
    Imprimir menú
    Obtener la operación
    SWITCH (Convert(operation))
        caja INSERT_DAY
            GetDate(date)
            calendar[date.Month()-1].InsertDay(date)
        caja INSERT_APT
            GetDate(date)
            calendar[date.Month()-1].InsertApt(date)
        caja DELETE_APT
            GetDate(date)
            calendar[date.Month()-1].DeleteApt(date)
        caja IS_FREE
            GetDate(date)
            IF calendar[date.Month()-1].IsFree(date)
                Escribir "El tiempo está libre"
            ELSE
                Escribir "El tiempo no está libre"
        caja PRINT_DAY
            GetDate(date)
            calendar[date.Month()-1].PrintDay(outFile, date)
        caja QUIT
            Establecer notQuit en falso
        caja ERROR
            Escribir "La cadena de entrada no fue válida. Leer el menú e intentar de nuevo."
```

¿Cuál de estas operaciones necesita una expansión? Print menu debe expandirse para dar instrucciones al usuario respecto a lo que debe capturar. Get the operation(obtener la operación) puede ser una simple lectura de cadena. Convert(operation) debe ser codificado como una función que toma una cadena y devuelve el tipo enumerado apropiado. GetDate(date) debe pedir y leer la fecha.

Imprimir menú

```
Escribir "opciones de menú"
Escribir "InsertDay establece un nuevo día"
Escribir "InsertApt inserta una nueva entrada"
Escribir "DeleteApt borra una entrada"
Escribir "IsFree comprueba si un tiempo está libre"
Escribir "PrintDay imprime todas las citas para un día"
Escribir "Quit termina el proceso"
Escribir "Teclear la cadena apropiada"
```

Convert(entrada: operación)
Salida: valor de función

```
IF la operación es igual a "InsertDay"
    Devolver INSERT_DAY
ELSE IF la operación es igual a "InsertApt"
    Devolver INSERT_APT
ELSE IF la operación es igual a "DeleteApt"
    Devolver DELETE_APT
ELSE IF la operación es igual a "IsFree"
    Devolver IS_FREE
ELSE IF la operación es igual a "PrintDay"
    Devolver PRINT_DAY
ELSE IF la operación es igual a "Quit"
    Devolver QUIT
ELSE
    Devolver ERROR
```

GetDate(salida: fecha)

```
Escribir "Introducir el mes, día, año en ese orden"
Obtener mes, día, año
date.Set(mes, día, año)
```

Finalmente estamos listos para codificar nuestro calendario de citas.

```
//**************************************************************************
// PROGRAMA CALENDARIO DE CITAS
// Este archivo pone en práctica un calendario de citas.
//**************************************************************************
#include "DayList.h"
#include <fstream>
#include <string>
#include <iostream>
#include <cstddef>     // Para NULL
using namespace std;

enum Operations{INSERT_DAY, INSERT_APT, DELETE_APT, IS_FREE,
                PRINT_DAY, QUIT, ERROR};
```

```
// Prototipos de función
Operations Convert(string operation);
void GetDate(Date& date);
void PrintMenu();

int main()
{
    DayList calendar[12];
    bool notQuit = true;
    Date date;

    string operation;
    ofstream outData;
    ofstream outList;
    outData.open("Appointments");

    while (notQuit)
    {
        PrintMenu();
        cin >> operation;
        switch (Convert(operation))
        {
             case INSERT_DAY:
                 GetDate(date);
                 calendar[date.Month()-1].InsertDay(date);
                 break;
             case INSERT_APT:
                 GetDate(date);
                 calendar[date.Month()-1].InsertApt(date);
                 break;
             case DELETE_APT:
                 GetDate(date);
                 calendar[date.Month()-1].DeleteApt(date);
                 break;
             case IS_FREE:
                 GetDate(date);
                 if (calendar[date.Month()-1].IsFree(date))
                     cout << "El tiempo está libre" << endl;
                 else
                     cout << "El tiempo no está libre" << endl;
                 break;
             case PRINT_DAY:
                 GetDate(date);
                 calendar[date.Month()-1].PrintDay(outData, date);
                 break;
             case QUIT:
                 notQuit = false;
                 break;
             case ERROR:
                 cout << "La cadena de entrada no fue válida."
                      << endl
                      << "Leer el menú y probar de nuevo."
                      << endl;
```

```
        }
    }
    outData.close();
    outList.close();
    return 0;

 }

void GetDate(Date& date)
{
    int month;
    int day;
    int year;
    cout << "Introducir el mes, día y año en ese orden" << endl;
    cin >> month >> day >> year;
    date.Set(month, day, year);
}

void PrintMenu()
{
    cout << "Opciones de menú" << endl;
    cout << "      InsertDay establece un nuevo día " << endl;
    cout << "      InsetApt inserta una nueva entrada" << endl;
    cout << "      DeleteApt borra una entrada " << endl;
    cout << "      IsFree comprueba si un tiempo está libre"
         << endl;
    cout << "      PrintDay imprime todas las citas "
         << " para un día" << endl;
    cout << "      Quit termina el proceso" << endl;
    cout << "Teclear la cadena apropiada " << endl << endl;
}

Operations Convert(string operation)
{
    if (operation == "InsertDay")
        return INSERT_DAY  ;
    else if (operation == "InsertApt")
        return INSERT_APT;
    else if (operation == "DeleteApt")
        return DELETE_APT;
    else if (operation == "IsFree")
        return IS_FREE;
    else if (operation == "PrintDay")
        return PRINT_DAY;
    else if (operation == "Quit")
        return QUIT;
    else return ERROR;
}
```

PRUEBA Un plan de prueba mínimo requerirá que cada operación sea verificada por lo menos una vez, y algunas de ellas cuando menos dos veces.

Razón para caso de prueba	Valores de entrada	Salida esperada	Salida observada
Insert day		Ninguno esperado	
one month	InsertDay 1 3 2004 5		
another month	InsertDay 4 5 2004 6		
InsertApt		Ninguno esperado	
one month	InsertApt 1 3 2004 Sarah Jane Jones 10:30		
same month	InsertApt 1 3 2004 Susan Margaret Smith 9:30		
another month	InsertApt 4 5 2004 Judy Dale David 3:00		
PrintDay			
month with one	PrintDay 4 5 2004	Judy Dale David 3:00	
month with more	PrintDay 1 3 2004	Susan Margaret Smith 9:30 Sarah Jane Jones 10:30	
day not there	PrintDay 2 3 2004	Día no ha sido inicializado	
DeleteApt			
appointment there	DeleteApt 1 3 2004 9:30 PrintDay 1 3 2004	Sarah Jane Jones 10:30	
apt. not there	DeleteApt 1 3 2004 12:30 PrintDay 1 3 2004	Sarah Jane Jones 10:30	
day not there	DeleteApt 3 3 2004 12:30	Día no ha sido inicializado	
IsFree			
time is free	IsFree 1 3 2004 4:30	Hora está libre	
time is not free	IsFree 1 3 2004 10:30	Hora no está libre	
day not there	IsFree 1 6 2004 1:30	Día no ha sido inicializado	
Error in input	DeletApt	Cadena de entrada no fue válida. Lea el menú e intente de nuevo.	
Quit	Quit		

Puesto que la salida es de cinco páginas, aquí sólo mostramos extractos; el menú fue eliminado.

```
Opciones de menú
       InsertDay establece un nuevo día
       InsertApt inserta una nueva entrada
       DeleteApt borra una entrada
       IsFree comprueba si un tiempo está libre
       PrintDay imprime todas las citas para un día
       Quit termina el proceso
Teclear la cadena apropiada

InsertDay
Introducir el mes, día y año en ese orden
1 3 2004
¿Cuántas citas se pueden hacer en esta fecha?
5

...
InsertDay
Introducir el mes, día y año en ese orden
4 5 2004
¿Cuántas citas se pueden hacer en esta fecha?
6
```

```
...

InsertApt
Introducir el mes, día y año en ese orden
1 3 2004
Introducir el nombre: Sarah
Introducir el apellido paterno: Jane
Introducir el apellido materno: Jones
Introducir las horas (<= 23):
10
Introducir los minutos (<= 59):
30

...

InsertApt
Introducir el mes, día y año en ese orden
1 3 2004
Introducir el nombre: Susan
Introducir el apellido paterno: Margaret
Introducir el apellido materno: Smith
Introducir las horas (<= 23):
9
Introducir los minutos (<= 59):
30

...

InsertApt
Introducir el mes, día y año en ese orden
4 5 2004
Introducir el nombre: Judy
Introducir el apellido paterno: Dale
Introducir el apellido materno: David
Introducir las horas (<= 23):
3
Introducir los minutos (<= 59):
0

...

PrintDay
Introducir el mes, día y año en ese orden
4 5 2004

...

PrintDay
Introducir el mes, día y año en ese orden
1 3 2004

...
```

```
PrintDay
Introducir el mes, día y año en ese orden
2 3 2004
El día no ha sido inicializado

...

DeleteApt
Introducir el mes, día y año en ese orden
1 3 2004
Introducir las horas (<= 23):
9
Introducir los minutos (<= 59):
30

...

PrintDay
Introducir el mes, día y año en ese orden
1 3 2004

...
DeleteApt
Introducir el mes, día y año en ese orden
1 3 2004
Introducir las horas (<= 23):
12
Introducir los minutos (<= 59):
30

...

PrintDay
Introducir el mes, día y año en ese orden
1 3 2004

...

DeleteApt
Introducir el mes, día y año en ese orden
3 3 2004
Introducir las horas (<= 23):
12
Introducir los minutos (<= 59):
30
El día no ha sido inicializado

...

IsFree
Introducir el mes, día y año en ese orden
1 3 2004
```

```
Introducir las horas (<= 23):
4
Introducir los minutos (<= 59):
30
El tiempo está libre

...

IsFree
Introducir el mes, día y año en ese orden
1 3 2004
Introducir las horas (<= 23):
10
Introducir los minutos (<= 59):
30
El tiempo está libre

...

IsFree
Introducir el mes, día y año en ese orden
1 6 2004
Introducir las horas (<= 23):
1
Introducir los minutos (<= 59):
30
El día no ha sido inicializado
El tiempo está libre

...

DeletApt
La cadena de entrada no fue válida.
Leer el menú y probar de nuevo.

...

Quit
```

Enseguida se muestra el contenido del archivo `Appointments`.

```
Citas para abril 5, 2004
Judy David at 03:00

Citas para enero 3, 2004
Susan Smith at 09:30
Sarah Jones at 10:30

Citas para enero 3, 2004
Sarah Jones at 10:30

Citas para enero 3, 2004
Sarah Jones at 10:30
```

Prueba y depuración

La prueba y depuración de una estructura ligada es complicada porque cada elemento en la estructura contiene no sólo una porción de datos sino también una liga al siguiente elemento. Los algoritmos deben dar cuenta correctamente tanto de los datos como de la liga.

Cuando se ejecutan listas ligadas con datos dinámicos y apuntadores, pueden surgir los errores que hemos analizado en el capítulo 15: fugas de memoria, apuntadores suspendidos (*dangling pointers*) e intentos de desreferenciar un apuntador nulo o un apuntador no inicializado. A continuación se plantean algunas sugerencias para ayudarle a localizar estos errores o evitarlos desde el principio.

Sugerencias para prueba y depuración

1. Revise las sugerencias para la prueba y depuración del capítulo 15. Éstas se aplican a los apuntadores y datos dinámicos que se usan en listas ligadas dinámicas.
2. Asegúrese de que el miembro de liga en el último nodo de una lista ligada dinámica se haya fijado a `NULL`.
3. Cuando se visitan los componentes en una lista ligada dinámica, asegúrese de que se haga la prueba del final de la lista de modo que no se trate de invalidar el apuntador nulo. En muchos sistemas, la invalidación del apuntador nulo causa un error en tiempo de ejecución.
4. Asegúrese de inicializar el apuntador externo para cada estructura de datos dinámicos.
5. No use

   ```
   currPtr++;
   ```

   para hacer que `currPtr` apunte al siguiente nodo en una lista ligada dinámica. Los nodos de la lista no se encuentran necesariamente en ubicaciones consecutivas en el área de almacenamiento libre.
6. Deberá hacer un cuidadoso seguimiento de los apuntadores. Cambiar los valores de apuntadores de manera prematura podrá causar problemas cuando intente regresar a la variable a que se apunta.
7. Si una clase C++ que apunta a datos dinámicos tiene un destructor de clase pero no tiene un constructor de copia, no pase un objeto de clase a una función usando el paso por valor. Ocurrirá un copiado superficial, y tanto el parámetro como el argumento apuntan a los mismos datos dinámicos. Cuando la función regresa se ejecuta el destructor del parámetro, que destruye los datos dinámicos del argumento.

Resumen

Las estructuras de datos dinámicos crecen y se contraen durante el tiempo de ejecución. Estas estructuras consisten en nodos que contienen dos tipos de miembros: el componente y uno o varios apuntadores a nodos del mismo tipo. El apuntador al primer nodo se guarda en una variable denominada apuntador externo a la estructura.

Una lista ligada es una estructura de datos en que los componentes se encuentran lógicamente uno al lado de otro en lugar de físicamente uno al lado de otro, como es el caso en un array. Una lista ligada puede ser representada ya sea como un array de estructuras (*structs*) o como una colección de nodos dinámicos, ligados por apuntadores. El final de una lista ligada dinámica se indica por la constante con apuntador especial `NULL`. Las operaciones comunes en listas ligadas incluyen la inserción de un nodo, el borrado de un nodo y atravesar la lista (visitando cada nodo del primero al último).

En este capítulo hemos usado listas ligadas para ejecutar listas. Sin embargo, las listas ligadas también se usan para ejecutar muchas estructuras de datos. El estudio de las estructuras de datos constituye un tema importante en las ciencias de la computación. Se desarrollan libros y cursos enteros para cubrir este tema. Un entendimiento sólido de los fundamentos de las listas ligadas es requisito previo para la creación de estructuras más complejas.

Comprobación rápida

1. Describa una situación en que podemos usar un array dinámicamente asignado para ejecutar una lista en lugar de usar una lista ligada dinámica. (pp. 723-724)
2. ¿Qué miembro necesitamos en adición a los miembros de datos de una estructura para usar la estructura como nodo en una lista ligada? (pp. 698-699)
3. ¿Cuáles son los pasos generales que debe repasar un programa para imprimir los componentes de una lista ligada? (pp. 710-712)
4. ¿Cuáles son los pasos generales necesarios para insertar un nuevo nodo en una lista ligada siguiendo un nodo específico? (pp. 714-718)
5. ¿Cuáles son los pasos generales necesarios para borrar un nodo de una lista ligada siguiendo un nodo específico? (pp. 718-721)

Respuestas

1. Cuando no sabemos de antemano cuántos datos se almacenarán en la lista. **2.** Un apuntador al siguiente nodo de la lista. **3.** Fijar el apuntador de posición actual al encabezado de la lista, luego imprimir el nodo en forma iterativa y avanzar la posición actual al siguiente nodo, fijándolo igual al apuntador de nodo dentro del nodo actual. **4.** Asignar un nuevo nodo. Asignar el valor desde el apuntador de nodo del nodo actual al apuntador de nodo del nuevo nodo. Asignar todos los miembros de datos al nuevo nodo según sea apropiado. **5.** Asignar el valor del apuntador de nodo del nodo actual a un apuntador temporal. Asignar el valor del apuntador de nodo desde el nodo a que apunta el campo del apuntador de nodo del nodo actual al campo de apuntador de nodo del nodo actual. (Asignar la dirección del nodo siguiendo el nodo a borrar al apuntador de nodo del nodo actual.) Borrar los datos asociados mediante el apuntador temporal.

Ejercicios de preparación para examen

1. Defina los siguientes términos:

   Nodo
   Miembro de componente
   Miembro de liga
   Apuntador actual

2. Sólo necesitamos un apuntador principal para una lista ligada dinámica. Mediante una lista ligada basada en array podemos encontrar el encabezado de modo automático. ¿Verdadero o falso?
3. Usar una declaración directa del tipo de apuntador de nodo nos permite evitar el uso de un tipo anónimo para declarar el campo de ligas dentro de la estructura que representan el nodo. ¿Verdadero o falso?
4. ¿A qué valor accede cada una de las siguientes expresiones? Los nombres de variables tienen los mismos significados que hemos usado en otras partes de este capítulo.
   a) `currPtr->link`
   b) `currPtr->component`
   c) `currPtr->link->component`
   d) `currPtr->link->link`
   e) `currPtr->link->link->component`
   f) `head->link->link->link->component`
5. *a*) En una aplicación de array de una lista ligada, ¿almacena el campo de ligas la dirección del siguiente nodo o el índice del siguiente nodo?
   b) En una construcción de datos dinámicos de una lista ligada, ¿almacena el campo de ligas la dirección del siguiente nodo o el índice del siguiente nodo?
6. *a*) ¿Qué estado especial de una lista verifica la condición `head == NULL?`
   b) ¿Qué condición especial verifica la expresión `currPtr == NULL?`
7. ¿Cuáles son los pasos algorítmicos para insertar un nuevo nodo en una lista ligada ordenada?
8. Suponiendo que el nodo actual no es el último nodo de la lista, ¿cuáles son los pasos algorítmicos para borrar el sucesor del nodo actual?

9. Para cada una de las siguientes operaciones, decida cuál es más rápida: una representación directa de array o una representación ligada.
   a) Insertar cerca del encabezado de la lista.
   b) Borrar el último elemento de la lista.
   c) Acceder al elemento *n*.
   d) Buscar un elemento en una lista ordenada.
10. Explique la diferencia entre una aplicación directa de array de una lista y una aplicación de array de una lista ligada.
11. ¿Por qué produce el siguiente segmento de código un error de compilación? ¿Qué es lo que falta en las declaraciones?

```
typedef NodeType* NodePtr;
struct NodeType
{
    ComponentType component;
    NodePtr link;
};
```

12. Para cada uno de los siguientes puntos, decida si una aplicación directa de array o una representación de lista ligada dinámica sería la mejor opción. Suponga que la capacidad de memoria es limitada y una buena velocidad es deseable.
   a) Una lista de títulos de CD en una discoteca personal, con no más de 500 discos. Rara vez se borran CD; ocasionalmente se agregan, y la operación más frecuente en la lista es la búsqueda de un título especifico.
   b) Una lista de pedidos de envío a procesar. Los pedidos se insertan como entran, y se borran cuando se envían. La lista puede ser muy grande antes de las vacaciones de diciembre, y puede estar casi vacía en enero. En ocasiones se revisa la lista buscando un término específico para verificar su situación.
   c) Un itinerario de visitas de clientes para un vendedor. Es posible que se agreguen visitas en cualquier parte dentro del itinerario, y se borran del encabezado después de haberse realizado. El itinerario puede tener cualquier longitud.

Ejercicios de calentamiento para programación

1. Dada una aplicación de array de una lista ligada que se muestra a continuación, escriba expresiones que hagan lo siguiente, suponiendo que `currPtr` se encuentra en alguna parte en medio de la lista:
   a) Acceda al miembro de componente del primer elemento de la lista.
   b) Avance `currPtr` para que apunte al siguiente elemento.
   c) Tome acceso al miembro de componente del siguiente elemento (el que sigue después del elemento actual).
   d) Acceda al miembro de componente del elemento que sigue al siguiente elemento.

```
struct NodeType
{
    int component;
    int link;
};
NodeType node[100];
int head;
int currPtr;
```

2. Dada la implementación ligada dinámica de una lista ligada que se muestra a continuación, escriba expresiones que hagan lo siguiente, suponiendo que `currPtr` se encuentra en alguna parte en medio de la lista:

a) Tome acceso al miembro componente del primer elemento de la lista.
b) Avance `currPtr` para que apunte al siguiente elemento.
c) Acceda al miembro de componente del siguiente elemento (el que sigue después del elemento actual).
d) Tome acceso al miembro de componente del elemento que sigue al siguiente elemento.

```
typedef int ComponentType;

struct NodeType;
typedef NodeType* NodePtr;

struct NodeType
{
    ComponentType component;
    NodePtr link;
}

NodePtr head;
NodePtr currPtr;
NodePtr newNodePtr;
```

3. Dadas las declaraciones en el ejercicio 2, escriba un segmento de código que cree un nuevo nodo, asigne el valor 100 al miembro de componente, ligue `head` al nuevo nodo y fije `currPtr` a que también apunte al nodo.
4. Dadas las declaraciones en el ejercicio 2, y que el primer nodo haya sido insertado en la lista, como en el ejercicio 3, escriba un segmento de código que determine un nuevo nodo, con el valor de componente igual a 212, y que lo inserte al final de la lista, actualizando los apuntadores conforme sea necesario.
5. Dadas las declaraciones en el ejercicio 2, suponga que la lista tenga una colección grande de miembros y que `currPtr` se encuentre en alguna parte en medio de la lista. Escriba un segmento de código para insertar un nuevo nodo con el valor de componente 32 siguiendo al nodo a que apunta `currPtr`, y actualice `currPtr` para que apunte al nuevo nodo.
6. Dadas las declaraciones en el ejercicio 2, suponga que la lista tiene una colección grande de miembros y que `currPtr` se encuentra en alguna parte en medio de la lista. Escriba un segmento de código para eliminar el nodo que sigue al nodo a que apunta `currPtr`, y lo reinserte al inicio de la lista.
7. Dadas las declaraciones en el ejercicio 2, suponga que la lista tiene una colección grande de miembros y que `currPtr` se encuentra en alguna parte en medio de la lista. Escriba un segmento de código que declare un nuevo apuntador, `auxPtr`, y que revise desde `currPtr` hasta el final de la lista, y luego borre el último elemento de la lista.
8. Reescriba el segmento de código en el ejercicio 7 para que trabaje sin ninguna suposición acerca del número de elementos en la lista. Esto quiere decir que la lista podría estar vacía o tener uno o varios elementos.
9. Dadas las declaraciones en el ejercicio 2, escriba una función vacía que ordene los elementos de la lista en orden ascendente. El algoritmo de ordenamiento escaneará la lista, manteniendo un apuntador al valor más bajo que se ha visto hasta ese momento. Cuando se llega al final de la lista, el valor más bajo se retira de ese punto para insertarse al final de una nueva lista. Cuando se hayan movido todos los elementos de la lista original a la lista ordenada, cambie `head` y `currPtr` para que apunten al primer elemento de la nueva lista. Cuando vuelva a la función, el código de cliente simplemente verá la lista ordenada.
10. Dadas las declaraciones en el ejercicio 2, escriba un segmento de código que declare las variables para manejar una segunda lista y luego copie sus elementos a la nueva lista en orden inverso.

Problemas de programación

1. Este problema significa que usted debe reescribir el programa del problema 1 del capítulo 15, usando una lista ligada dinámica. Puesto que el tamaño de la lista puede variar, ya no necesitamos conocer la longitud del archivo de datos de antemano. He aquí el problema, reformulado en este nuevo contexto.

   Usted trabaja para la oficina de registro de automóviles de su estado, y se ha detectado que algunas personas tienen registros múltiples en la base de datos del departamento de licencias de conducir. Los registros de licencias están guardados en orden alfabético en un conjunto de archivos, uno por letra del alfabeto. El primer archivo es `licensesa.dat`, y el último es `licensesz.dat`. Para este problema, sólo nos vamos a enfocar a que el programa funcione para el archivo `licensesa.dat`. Algunos de los duplicados de registros se deben a ligeras diferencias en la ortografía de nombres, así que cada archivo se tiene que ordenar por números de licencia (un entero de 8 dígitos) para encontrar los duplicados. Cada registro consiste en un número de licencia, un nombre y una dirección, todo en una sola línea. Para los fines de este problema, el nombre y la dirección se pueden guardar en una sola cadena porque no están procesados por separado. El número de licencia y la cadena correspondiente se deberán mantener juntos en una estructura (*struct*).

   Use la clase `HybridList` de este capítulo, donde el array de la lista se crea dinámicamente para tener la longitud necesaria para los datos en el archivo particular. Cada componente de la lista será un tipo de estructura (*struct*) de licencia. Además, agregue a la clase `HybridList` una función que devuelva valores de la lista en orden sucesivo (llamado `GetNext`), y una función compañera (denominada `Restart`) que reinicia la otra función para que vuelva a empezar en el primer componente de la lista.

   Una vez que se hayan extraído los datos desde un archivo a `HybridList`, use las nuevas funciones para repasar la lista, comparando cada número de licencia con el número que antecede para verificar si son idénticos. Conforme se encuentra cada registro duplicado, se deberá escribir en un archivo de salida nombrado `duplicensesa.dat`. Mantenga un conteo del número de registros duplicados detectados y escríbalo en `cout`. Recuerde que debe cerrar el archivo al final del programa.
2. Extienda el programa del problema 1 en dos formas. La primera extensión debe lograr que el programa procese automáticamente los 26 archivos de datos. La segunda extensión debe lograr que los registros duplicados se extraigan en orden alfabético, de modo que el archivo de salida tenga la misma organización que el archivo de entrada. Para realizar esto, usted deberá crear, después del primer paso a través de la lista, una segunda `HybridList` que tenga la longitud adecuada para retener todos los registros duplicados, y esta lista se debe ordenar en el campo de cadenas `struct` de licencia. Una vez que se hayan copiado todos los registros duplicados en esta lista, usted podrá borrar la primera lista y luego extraer la segunda lista al archivo.
3. Este problema le obliga a reescribir el programa del problema 3 del capítulo 15 usando una lista ligada dinámica. Puesto que el tamaño de la lista puede variar, ya no es necesario limitar el número máximo de tarjetas de presentación. Tampoco hay razón para extraer el número de entradas al archivo de salida. He aquí el problema, reformulado en este nuevo contexto.

   Usted trabaja para una empresa que quiere tomar las informaciones de contacto de clientes tal como las introduce un empleado de ventas desde una pila de tarjetas de presentación que coleccionó, para luego extraer las informaciones a un archivo (`contacts.dat`) en orden alfabético. Se supone que nunca hay más de 100 tarjetas que introducir. El programa deberá pedir al empleado de ventas cada uno de los siguientes valores para cada tarjeta:

   Apellido materno
   Nombre
   Apellido paterno
   Título
   Nombre de la compañía
   Domicilio
   Ciudad

Estado
Código postal
Número de teléfono
Número de fax
Dirección electrónica

Después de capturar los datos para cada tarjeta, se deberá preguntar al usuario si hay otra tarjeta que introducir. Estos datos se deberán ordenar por apellido materno, usando una versión de la clase `HybridList` de este capítulo que debe retener valores de estructuras, cada uno con un miembro correspondiente a un elemento en la lista mencionada. Será necesario modificar la función `Print` de la clase para extraer la lista al archivo `contacts.dat` en lugar de `cout`. Cada miembro de la estructura se deberá escribir en una línea separada.

4. Este problema le obliga a reescribir el programa del problema 4 del capítulo 15 usando una lista ligada dinámica. Puesto que la lista no tiene un tamaño fijo, ya no es necesario determinar el número total de registros en los archivos antes de asignar el espacio de almacenamiento. He aquí el problema, reformulado en este nuevo contexto.

   Usted trabaja para una empresa que tiene una colección de archivos, cada uno de los cuales contiene informaciones de hasta 100 tarjetas de presentación. Los archivos fueron creados por otro programa (véase el problema 3 para una descripción de lo que emite este programa). La empresa desea unir los archivos para formar uno solo, ordenado de manera alfabética por apellido paterno. Al usuario se le deberá pedir que introduzca los nombres de los archivos hasta que resulte un archivo con el nombre "done".

   Modifique la `HybridList` de este capítulo de modo que cada nodo de la lista contenga un tipo de estructura, tal como se describe en el problema 3. Al capturar cada nombre de archivo, el archivo se abre y extrae los datos, y crea e inserta nodos en la lista ordenada. Una vez que todos los datos se hayan capturado, reescriba todos los datos ordenados en un archivo nominado `mergedcontacts.dat`. Será necesario modificar la función `Print` de la clase `HybridList` para extraer la lista al archivo `mergedcontacts.dat` en lugar de `cout`. Cada miembro de la estructura se deberá escribir en una línea separada.

5. Modifique el programa en el problema 4 de forma que después de la introducción de los datos el programa escanee la lista y borre todos los registros duplicados antes de extraer la lista. Un registro es un duplicado si el apellido paterno, nombre, segundo nombre o inicial, así como el nombre de la compañía corresponden a los del otro registro. Tenga cuidado de borrar correctamente todos los datos de asignación dinámica retirados de la lista.

Seguimiento de caso práctico

1. ¿Podríamos haber aplicado la lista de días usando una clase de lista que ya hemos aplicado? Explique su respuesta. (Piense cuidadosamente esta pregunta; no es tan simple como parece.)
2. Se le pregunta al usuario cuántas citas se podrán hacer en un día cuando el día se inicializa, pero nunca se saca ventaja de este valor. Agregue una prueba a la función `InsertApt` que escribe un mensaje de error en la pantalla si el usuario pretende insertar más citas que el número permitido.
3. Se hizo muy tediosa la solicitud y captura del nombre de pila, apellido materno o inicial y apellido paterno. Derive una clase que herede de `Name` y que tenga una operación `ReadName` que pida que el usuario capture el nombre de pila y apellido paterno con un espacio en blanco entre los dos.
4. La solicitud y captura de horas y minutos también se volvió muy tediosa. Altere la clase `AptTime` de modo que la función `ReadTime` pida que el usuario capture las horas y minutos con un espacio en blanco entre los dos.
5. Vuelva a ejecutar el programa de citas usando los datos de muestra que se emplearon en el estudio práctico con la clase derivada `Name` y la clase alterada `AptTime`. ¿La captura fue más fácil?
6. Hemos comparado fechas en lugar de días, aunque sabemos que los meses son los mismos. ¿Sería más eficiente acceder a los campos de días de la fecha y componente y compararlos usando el operador de igualdad? Explique.

CAPÍTULO

17

Plantillas y excepciones

Objetivos

Objetivos de conocimiento

- *Comprender el concepto de plantilla.*
- *Entender el concepto de excepción.*

Objetivos de habilidades

Ser capaz de:

- *Escribir una plantilla de función de C++.*
- *Escribir un código que establece una plantilla de función.*
- *Escribir una especialización definida por el usuario de una plantilla de función.*
- *Escribir una plantilla de clase de C++.*
- *Escribir un código que produce una plantilla de clase.*
- *Escribir definiciones de función para miembros de una clase de plantillas.*
- *Definir una clase de excepción y escribir código que lanza una excepción.*
- *Escribir un manejador de excepción.*

En este capítulo se introducen dos características del lenguaje C++ que pueden tener un impacto poderoso en nuestra forma de diseñar y ejecutar software: *plantillas* y *excepciones*. Plantillas y excepciones básicamente son conceptos no relacionados, y se podría dedicar un capítulo separado a cada uno. Sin embargo, este libro pretende ser sólo una introducción a la ciencia de la computación y el diseño de software, así que presentaremos los dos temas en un solo capítulo, explorando cada uno con menos detalle del que daría un libro de texto para un curso más avanzado.

Como sugiere su nombre, una plantilla es un patrón a partir del cual podemos crear múltiples instancias de algo. En capítulos anteriores hemos usado la frase "múltiples instancias" cuando hablamos sobre tipos de datos. Hemos dicho que un tipo de datos es un patrón a partir del cual creamos múltiples instancias (variables u objetos de clase) de este tipo. El mecanismo de plantillas en C++ lleva el concepto de "instancia" a un nivel superior. En lugar de ser variables u objetos de clase, las instancias son funciones completas de tipo de clases de C++. En este capítulo usted aprenderá cómo definir una *plantilla de función* a partir de la cual el compilador crea múltiples versiones de una función. De modo similar usted verá cómo definir una *plantilla de clase* a partir de la cual el compilador crea múltiples versiones de un tipo de clase.

Excepciones son sucesos poco usuales, a menudo errores, que pueden ocurrir durante la ejecución de un programa. El mecanismo de manejo de excepciones en C++ permite que una parte del programa informe a otra parte del programa que ha ocurrido una excepción en caso de que el problema no se pueda arreglar de manera local. Como descubrirá, el tema del manejo de excepciones usa terminología muy vívida. De la parte del programa que detecta un error se dice que *lanza* una excepción, esperando que otra parte del programa (un manejador de excepción) *capture* la excepción.

Si usted saltó a este capítulo desde otra parte del libro, aquí están los temas de requisitos previos. La sección 17.1 (Plantillas de funciones) supone que ya leyó el capítulo 10 del libro; la sección 17.2 (Plantilla de clase) supone que ya leyó el capítulo 13, y la sección 17.3 (Excepciones) supone que ya leyó el capítulo 11.

17.1 Plantilla de funciones

Cuando diseñamos o probamos software, en ocasiones necesitamos un solo algoritmo que se pueda aplicar a objetos de diferentes tipos de datos en diferentes momentos. Queremos estar en condiciones de describir el algoritmo sin tener que especificar los tipos de datos de los elementos que se están manipulando. Con frecuencia se refiere a este tipo de algoritmo como **algoritmo genérico**. C++ soporta algoritmos genéricos proporcionando dos mecanismos: *sobrecarga de función* y *plantilla de funciones*.

Algoritmo genérico Algoritmo en el cual las acciones o pasos están definidos, pero los tipos de datos de los elementos que se manipulan no lo está.

Sobrecarga de función

Sobrecarga de función es el uso del mismo nombre para diferentes funciones, siempre que sus tipos de parámetros sean suficientemente diferentes para que el compilador los pueda distinguir.

Sobrecarga de función Uso del mismo nombre para diferentes funciones de C++, distintas una de la otra por sus listas de parámetros.

Veamos un ejemplo. Estamos depurando un programa y queremos rastrear su ejecución mediante la impresión de los valores de determinadas variables mientras que el programa se ejecuta. Las variables que queremos rastrear son de los siguientes seis tipos de datos: `char`, `short`, `int`, `long`, `float` y `double`. Podríamos crear seis funciones con diferentes nombres a fin de extraer valores de diferentes tipos:

```
void PrintInt( int n )
{
    cout << "***Debug" << endl;
    cout << "Value is " << n << endl;
}
void PrintChar( char ch )
```

```
{
    cout << "***Debug" << endl;
    cout << "Value is " << ch << endl;
}
void PrintFloat( float x )
{
 ⋮
}
void PrintDouble( double d )
{
 ⋮
}
⋮
```

En los lugares del programa donde queremos extraer los valores trazados, insertaríamos llamadas a las diferentes funciones de la siguiente manera:

```
 ⋮
sum = alpha + beta + gamma;
PrintInt(sum);
 ⋮
PrintChar(initial);
 ⋮
PrintFloat(angle);
```

En lugar de inventar diferentes nombres para todas estas funciones similares, podemos usar la sobrecarga de función dando a todos el mismo nombre: `Print`.

```
void Print( int n )
{
    cout << "***Debug" << endl;
    cout << "Value is " << n << endl;
}
void Print( char ch )
{
    cout << "***Debug" << endl;
    cout << "Value is " << ch << endl;
}
void Print( float x )
{
 ⋮
}
⋮
```

El código que llama a estas funciones ahora se ve de la siguiente manera:

```
Print(someInt);
Print(someChar);
Print(someFloat);
```

Podemos considerar a `Print` como un algoritmo genérico en el sentido de que el propio algoritmo —imprimiendo la cadena `"***Debug"` y luego el valor de una variable— es independiente del tipo de datos de la variable que se está imprimiendo. Durante la programación sólo tenemos que usar un nombre para este algoritmo (`Print`), aunque en realidad haya seis funciones diferentes.

¿Cómo funciona la sobrecarga de función? Cuando nuestro programa es compilado, quien lo hace encuentra seis funciones diferentes con el nombre `Print`, pero internamente les asigna seis nombres distintos. No sabemos qué son estos nombres internos, pero para el propósito de este análisis suponemos que los nombres sean `Print_int`, `Print_char`, etcétera. Cuando el compilador encuentra la llamada de función

```
Print(someVar);
```

tendrá que determinar a cuál de nuestras seis funciones debe invocar. Para dicho objetivo, el compilador compara el tipo del argumento actual con los tipos de parámetros formales de las seis funciones. Antes, si `someVar` es del tipo `int`, el compilador genera un código para llamar a la función `Print` que tiene un parámetro `int` (el que tiene el nombre interno de `Print_int`). Si `someVar` es del tipo `float`, el compilador genera un código para llamar a la función `Print` que tiene un parámetro `float` (el que tiene el nombre interno de `Print_float`), etcétera.

Como se puede ver, la sobrecarga de función beneficia al programador eliminando la necesidad de inventar muchos nombres diferentes para funciones que realizan tareas idénticas (excepto para operar en variables de diferentes tipos de datos). La sobrecarga también reduce la posibilidad de resultados inesperados, causados por el uso del nombre de función equivocado; por ejemplo, la llamada de `PrintInt` cuando se pasa una variable `float` como argumento.

A pesar de los beneficios de sobrecarga de función en nuestro ejemplo de la función `Print`, fue necesario proporcionar seis diferentes definiciones de funciones. Esto implica una cantidad tediosa de captura o uso de copiar y pegar en un editor, y el código fuente que resulta es una confusión de un grupo de definiciones de funciones casi idénticas. Ahora veremos una manera mucho más limpia de ejecutar algoritmos genéricos en plantilla de funciones en C++.

Definición de una plantilla de función

Plantilla de función Construcción del lenguaje C++ que permite que el compilador genere múltiples versiones de una función permitiendo tipos de datos parametrizados.

En C++ una **plantilla de función** le permite escribir una definición de función con espacios en blanco que se dejan en la definición para ser llenados con el código de llamada. Las más de las veces las áreas en blanco a ser llenadas son los nombres de tipos de datos. A continuación se muestra una plantilla de función para nuestra función `Print`:

```
template<class SomeType>
void Print( SomeType val )
{
    cout << "***Debug" << endl;
    cout << "Value is " << val << endl;
}
```

Esta plantilla de función empieza con `template<class SomeType>`, y `SomeType` se conoce como el *parámetro de plantilla*. Se puede usar cualquier identificador para el parámetro de plantilla; en nuestro ejemplo usamos `SomeType`.

A continuación se presenta la sintaxis para una plantilla de función:

FunctionTemplate

```
template < TemplateParamList >
FunctionDefinition
```

donde FunctionDefinition es una definición de función ordinaria. La descripción de sintaxis completa de TemplateParamList en C++ es muy complicada, y la simplificamos para nuestros fines de la

manera siguiente. TemplateParamList es una secuencia de una o más declaraciones de parámetros separada por comas, donde cada una es definida como sigue:

TemplateParamDeclaration

Observe en la plantilla de sintaxis para FunctionTemplate que los paréntesis angulares son requeridos, pero la lista de parámetros es opcional. Más adelante estudiaremos por qué se querrá omitir la lista de parámetros.

Creación de una plantilla de función

Dado que hemos escrito nuestra plantilla de función `Print`, podemos hacer llamadas de función como la que sigue:

```
Print<int>(sum);
Print<char>(initial);
Print<float>(angle);
```

En este código, el nombre del tipo de datos entre paréntesis angulares se denomina *argumento de plantilla*. Al tiempo de compilar, el compilador genera (*crea*) tres diferentes funciones y asigna su propio nombre interno a cada una de ellas. Las tres funciones nuevas se denominan *funciones de plantilla* o *funciones generadas* (a diferencia de la *plantilla de función* a partir de la cual fueron creadas). Asimismo, una versión de una plantilla para un argumento de plantilla particular se denomina *especialización*.

Cuando el compilador crea una plantilla, literalmente sustituye el argumento de plantilla para el parámetro de plantilla a través de toda la plantilla de función, igual que se haría una operación de buscar y remplazar en un procesador o editor de texto. Por ejemplo, la primera vez que el compilador encuentra `Print<float>` en el código de llamada, genera una nueva función, sustituyendo `float` por cada ocurrencia de `SomeType` en la plantilla de función:

```
                          float
void Print( SomeType val )
{
    cout << "***Debug" << endl;
    cout << "Value is " << val << endl;
}
```

Hay dos cosas qué observar en cuanto a los parámetros de plantilla. Primero, la plantilla de función usa la palabra reservada `class` en su lista de parámetros: `template<class SomeType>`. Sin embargo, la palabra `class` en este contexto sólo es sintaxis requerida y no significa que el argumento de plantilla del invocador debe ser el nombre de una clase de C++. (De hecho, usted puede usar la palabra reservada *typename* en lugar de `class`, si así lo desea.) El argumento de plantilla puede ser el nombre de cualquier tipo de datos, integrado o definido por el usuario. En nuestro ejemplo de código de llamada hemos usado `int`, `char` y `float` como argumentos de plantilla. Segundo: observe que estos argumentos de plantilla son nombres de tipos de datos, no son nombres de variables. Esto parece extraño al principio porque cuando se dan argumentos a funciones, siempre surgen nombres de variables o expresiones, pero no nombres de tipos de datos. Además, observe que el paso de un argumento a una plantilla tiene un efecto al *tiempo de compilar* (el compilador genera una nueva definición de función a partir de la plantilla), mientras que el paso de un argumento a una función tiene un efecto al *tiempo de ejecución*.

A continuación se encuentra la plantilla de sintaxis para una llamada a una función de plantilla:

TemplateFunctionCall

```
FunctionName < TemplateArgList > ( FunctionArgList )
```

Como puede ver, la lista de argumentos de plantilla entre paréntesis angulares es opcional. De hecho, los programadores la omiten en general. En este caso, se dice que el compilador *deduce* el (los) argumento(s) examinando la lista de *argumentos de función*. Esto es, nuestro ejemplo anterior de código de llamada usando argumentos de plantilla explícitos se escribiría más bien de la siguiente manera:

```
Print(sum);        // Implicit: Print<int>(sum)
Print(initial);    // Implicit: Print<char>(initial)
Print(angle);      // Implicit: Print<float>(angle)
```

Cuando el compilador encuentra este código `Print(sum)`, verá a los tipos de datos del argumento de función `sum` (que es `int`) y deduce que el argumento de plantilla debe ser `int`. Por ende, la llamada de función es a la especialización `Print<int>` de la plantilla.

Mejora de la plantilla de impresión `Print`

En nuestra versión actual de `Print`, si la variable *sum* contiene el valor 38 y la variable `angle` contiene el valor 64.5, las llamadas de función

```
Print(sum);
Print(angle);
```

producen la siguiente salida:

```
***Debug
Value is 38
***Debug
Value is 64.5
```

Esta salida no es tan útil como podría ser; no indica el nombre de la variable cuyo valor se está extrayendo. Reescribamos la plantilla `Print` de modo que extraiga tanto el nombre de una variable como su valor actual:

```
template<class SomeType>
void Print( string   vName,    // Nombre de la variable
            SomeType val   )   // Valor de la variable
{
    cout << "***Debug" << endl;
    cout << "Value of " << vName << " = " << val << endl;
}
```

Aquí le estamos diciendo al compilador que el primer parámetro de función siempre es de un tipo específico (`string`), mientras que el segundo es de un tipo parametrizado. Por tanto, cuando se crea la plantilla en el código de llamada, el compilador deberá ver al segundo argumento de función a fin de deducir el argumento de plantilla. En otras palabras, en el código de llamada

```
Print("sum", sum);
Print("angle", angle);
```

la primera llamada de función implícitamente llama a la especialización `Print<int>` porque el argumento *sum* es del tipo `int`, y la segunda llama a la especialización `Print<float>`. La salida de estas dos llamadas de función es

```
***Debug
Value of sum = 38
***Debug
Value of angle = 64.5
```

Especializaciones definidas por el usuario

Hemos empezado este capítulo suponiendo que tuviésemos que extraer —por motivos de depuración— los valores de variables de seis tipos de datos: `char`, `short`, `int`, `long`, `float` y `double`. Hemos revisado tres formas de realizar esta tarea. La primera fue escribir seis diferentes definiciones de función con diferentes nombres de funciones. La segunda fue escribir seis diferentes definiciones de función, todas con el mismo nombre de función (sobrecarga de función). La tercera fue escribir sólo una definición de función (una plantilla de función) y dejar que el compilador genere las funciones individuales a partir de la plantilla. Este último método —usar funciones de plantilla— es el más compacto y conveniente para el programador. Además, las funciones de plantilla soportan el concepto de algoritmos genéricos porque nos permiten focalizar más en los algoritmos y menos en lo específico de los tipos de datos que manipulan.

Pensamos en nuestra función de plantilla `Print` como en una función genérica porque puede extraer un valor de cualquier tipo de datos. Sin embargo, la parte de "cualquier tipo de datos" no es del todo cierto. El cuerpo de la función `Print` usa el operador `<<` para extraer un valor al flujo `cout`. Desgraciadamente el operador `<<` sólo está definido para tipos integrados y determinadas clases de biblioteca, como `string`. Si nuestro programa ha definido un tipo de enumeración (capítulo 10) y una variable de este tipo como sigue:

```
enum StatusType {OK, OUT_OF_STOCK, BACK_ORDERED};
StatusType currentStatus;
```

entonces nuestra función de plantilla `Print` no puede ser pasada como un argumento del tipo `StatusType`. (Recuerde que una variable de un tipo de enumeración no puede ser directamente extraída por el operador `<<`.) Para obligar a la función `Print` a que acomode un argumento del tipo `StatusType` usamos el siguiente código:

```
template<>
void Print( string     vName,    // Nombre de la variable
            StatusType val   )   // Valor de la variable
{
    cout << "***Debug" << endl;
    cout << "Valor de " << vName << " = ";
    switch (val)
    {
        case OK           : cout << "OK";
                            break;
        case OUT_OF_STOCK : cout << "OUT_OF_STOCK";
                            break;
        case BACK_ORDERED : cout << "BACK_ORDERED";
                            break;
        default           : cout << "Valor no válido";
    }
    cout << endl;
}
```

El prefijo `template<>` dice que esto es una definición alternativa de la plantilla `Print` que no necesita ningún parámetro de plantilla y se debería usar cada vez que el segundo argumento en una llamada a `Print` es del tipo `StatusType`. Este tipo de plantilla se denomina *especialización definida por el usuario* o simplemente *especialización*. Dadas nuestras dos definiciones de plantilla para `Print` —la general y la especializada para variables de `StatusType`— el compilador considera el siguiente código de llamada como se muestra en los comentarios:

```
Print("sum", sum);                          // Llamar a la especialización
                                            //   Print<int>
Print("currentStatus", currentStatus);      // Llamar a la especialización
                                            //   definida por el usuario
```

Organización de códigos de programa

Dadas nuestras dos plantillas de función `Print`, ¿dónde las colocamos físicamente en un programa? Hay tres posibilidades, que se describen aquí en orden creciente de conveniencia (según nuestra opinión). La primera es colocar las definiciones de plantilla cerca del inicio del archivo de programa, antes de la función `main`.

```
// myprog1.cpp
#include <iostream>
#include <string>
using namespace std;

enum StatusType {OK, OUT_OF_STOCK, BACK_ORDERED};
template<class SomeType>
void Print( string   vName,
            SomeType val   )
{
 ⋮
}
template<>
void Print( string     vName,
            StatusType val   )
{
 ⋮
}

int main()
{
    int intVar;
    StatusType status;
      ⋮
    Print("intVar", intVar);
    Print("status", status);
      ⋮
}
```

Esta organización no sería usada por programadores que prefieren colocar la función `main` en primer término, seguido por otras funciones. La segunda opción es, entonces, colocar primero prototipos de función (declaraciones directas) de las plantillas, luego la función `main` y finalmente las definiciones de plantillas.

```
// myprog2.cpp
#include <iostream>
#include <string>
using namespace std;

enum StatusType {OK, OUT_OF_STOCK, BACK_ORDERED};

template<class SomeType>                  // Primero los prototipos
void Print( string vName, SomeType val );

template<>
void Print( string vName, StatusType val );

int main()                                // Después main()
{
    int intVar;
    StatusType status;
      ⋮
    Print("intVar", intVar);
    Print("status", status);
      ⋮
}
template<class SomeType>                  // Después las definiciones
void Print( string   vName,               //   de plantilla
            SomeType val   )
{
 ⋮
}
template<>
void Print( string     vName,
            StatusType val   )
{
 ⋮
}
```

La tercera y, en nuestra opinión, mejor opción es esconder las definiciones de plantillas en un archivo de encabezado (`.h`) y luego simplemente usar `#include` para insertar este archivo en nuestro programa. Si colocamos la directiva `#include` antes de la función `main`, no necesitamos prototipos de plantilla porque el compilador habrá visto las definiciones de plantillas antes de encontrar las llamadas a las funciones de plantillas. A continuación se muestra el archivo de encabezado:

```
// templs.h - Archivo de encabezado que contiene las definiciones de
   plantilla
#include <iostream>
#include <string>
using namespace std;

template<class SomeType>
void Print( string   vName,
            SomeType val   )
{
 ⋮
}
template<>
```

```
void Print( string     vName,
            StatusType val   )
{
 ⋮
}
```

Dado este archivo de encabezado, nuestro archivo de programa principal sólo puede incluirlo (`#include`) como sigue.

```
// myprog3.cpp - Archivo de programa principal
#include <iostream>
#include <string>
using namespace std;

enum StatusType {OK, OUT_OF_STOCK, BACK_ORDERED};

#include "templs.h"     // Insertar las definiciones de plantilla DESPUÉS
                        //   que se definió el StatusType
int main()
{
    int intVar;
    StatusType status;
     ⋮
    Print("intVar", intVar);
    Print("status", status);
     ⋮
}
```

Dos importantes ventajas de usar esta opción del archivo de encabezado son: (1) el archivo de programa principal está menos revuelto de código, y (2) sólo podemos incluir (`#include`) el archivo `templs.h` en cualquiera de nuestros otros archivos de programa cuando necesitamos una salida de depuración.

17.2 Plantilla de clase

En el capítulo 13 hemos definido una lista de tipos de datos abstractos (TDA) que representa una lista de componentes no ordenados, cada uno del tipo ItemType. Hemos codificado este TDA como una clase de C++ denominada `List`, y el archivo de encabezado `List.h` se escribió como sigue (aquí abreviado por omitir las precondiciones y poscondiciones de función):

```
const int MAX_LENGTH = 50;      // Número posible máximo de
                                //   componentes necesarios
typedef int ItemType;           // Tipo de cada componente
                                //   (un tipo simple o clase de cadena)
class List
{
public:
    bool IsEmpty() const;
    bool IsFull() const;
    int  Length() const;
    void Insert( /* in */ ItemType item );
    void Delete( /* in */ ItemType item );
    bool IsPresent( /* in */ ItemType item ) const;
    void Reset();
    ItemType GetNextItem();
```

```
    void SelSort();
    List();                    // Constructor
private:
    int      length;
    int      currentPos;
    ItemType data[MAX_LENGTH];
};
```

La sentencia `typedef` nos permite elegir un tipo particular para `ItemType`, siempre que sea un tipo simple o de la clase `string`. (La razón de la restricción se debe a que las funciones de miembros `List` usan operadores relacionales para comparar elementos de lista, y usan el operador `<<` para extraer elementos de lista. `ItemType` no puede ser un tipo de arreglo, por ejemplo, porque las operaciones relacionales y la operación `<<` no pueden ser aplicadas a un arreglo entero como agregado.)

La clase `List` es similar a un **tipo de datos genéricos** en el sentido de que los elementos de lista pueden ser de (casi) todos los tipos. Si cambiamos `ItemType` cambiando la sentencia `typedef`, podemos recompilar la clase sin cambiar ninguno de los algoritmos en las funciones de miembro.

Tipo de datos genéricos Tipo para el cual las operaciones están definidas, pero el tipo de datos de los elementos, siendo manipulados, no lo es.

Plantilla de clase Constructor del lenguaje C++ que permite que el compilador genere múltiples versiones de una clase permitiendo tipos de datos parametrizados.

Sin embargo, la clase `List`, tal como está escrita, tiene dos limitaciones serias. Primero, una vez que la clase se haya compilado usando, por decir así, `int` como `ItemType`, los objetos `List` de un programa de cliente sólo pueden ser listas de `int`. No hay manera de que el cliente mantenga listas de `int`, listas de `float` y listas de `chart`, todas dentro del mismo programa. Segundo, un programa de cliente no puede ni especificar ni cambiar `ItemType`; un ser humano tiene que entrar en el archivo `list.h` con un editor para cambiarlo de manera manual.

Para hacer que una clase como `List` sea un tipo verdaderamente genérico, quisiéramos un constructor de lenguaje que permite que `ItemType` sea un parámetro para la declaración de clase. Por fortuna, C++ prevé un constructor de esta índole: la **plantilla de clase**.

A continuación se presenta un ejemplo de la plantilla de clase (el nombre `GList` significa lista genérica):

```
template<class ItemType>
class GList
{
public:
    bool IsEmpty() const;
    bool IsFull() const;
    int  Length() const;
    void Insert( /* in */ ItemType item );
    void Delete( /* in */ ItemType item );
    bool IsPresent( /* in */ ItemType item ) const;
    void SelSort();
    void Reset();
    ItemType GetNextItem();
    GList();                   // Constructor
private:
    int      length;
    int      currentPos;
    ItemType data[MAX_LENGTH];
};
```

Como puede observar, esta plantilla se ve igual a la declaración de clase para el tipo `List`, excepto que le precede `template<class ItemType>`. Aquí `ItemType` (o cualquier identificador que se quiera usar a través de la plantilla) es el parámetro de plantilla. Como en el caso de las plantillas de

función, es posible usar opcionalmente la palabra `class` o la palabra `typename` para declarar un parámetro de plantilla.

Creación de una plantilla de clase

Dada la plantilla de clase `GList`, el programa de cliente puede usar un código como el siguiente para crear varias listas cuyos componentes son de diferentes tipos de datos:

```
// Client code

GList<int> list1;
GList<float> list2;
GList<string> list3;

list1.Insert(356);
list2.Insert(84.375);
list3.Insert("Muffler bolt");
```

En las declaraciones de `list1`, `list2` y `list3` el nombre del tipo de datos entre paréntesis angulares es el *argumento de plantilla*. Los argumentos de plantillas de clase *tienen* que ser explícitos; no pueden ser implícitos como en el caso de las funciones de plantilla. (Recuerde que un argumento de plantilla de función usualmente es omitido, y el compilador deduce el argumento viendo la lista de argumentos de la función.)

Cuando el compilador encuentra las declaraciones `list1`, `list2` y `list3` que se mencionaron antes, genera (crea) tres distintos tipos de clase y da a cada uno de estos tres tipos su propio nombre interno. Se podrá imaginar que las declaraciones son transformadas internamente en algo parecido a lo siguiente:

```
GList_int list1;
GList_float list2;
GList_string list3;
```

En la terminología de C++, los tres nuevos tipos de clase se llaman *clases de plantilla* o *clases generadas* (a diferencia de la *plantilla de clase* de la que fueron creadas). Como en el caso de las plantillas de función, una versión de una plantilla para un particular argumento de plantilla se denomina *especialización*.

Cuando el compilador crea una plantilla, literalmente sustituye el argumento de plantilla por el parámetro de plantilla a través de toda la plantilla de clase. Por ejemplo, la primera vez que el compilador encuentra `GList<int>` en el código del cliente, genera una nueva clase, sustituyendo `int` para cada ocurrencia de `ItemType` en la plantilla de clase:

```
class GList_int
{
public:
      ⋮
    void Insert( /* in */ ItemType item );
    void Delete( /* in */ ItemType item );
    bool IsPresent( /* in */ ItemType item ) const;
      ⋮
private:
    int      length;
    ItemType data[MAX_LENGTH];
};
```

Una perspectiva útil para plantillas de clase es la siguiente: Mientras que una declaración de clase ordinaria es un patrón para estampar variables u objetos individuales, una plantilla de clase es un patrón para estampar tipos de datos individuales.

Ahora que hemos establecido la definición de una plantilla de clase, ¿qué vamos a hacer acerca de las definiciones de las funciones de miembros de clases? Tenemos que escribirlas como *plantillas de función* para que el compilador pueda asociar cada una con la plantilla de clase correspondiente. Por ejemplo, codificamos la función `Insert` como la siguiente plantilla de función:

```
template<class ItemType>
void GList<ItemType>::Insert( /* in */ ItemType item )
{
    data[length] = item;
    length++;
}
```

Dentro de la plantilla de función, cada ocurrencia de `GList` como nombre de clase debe ser acompañado por `<ItemType>`. Si el cliente ha declarado un tipo `GList<float>`, el compilador genera una función parecida a la siguiente:

```
void GList<float>::Insert( /* in */ float item )
{
    data[length] = item;
    length++;
}
```

Organización de código de programa

Al trabajar con plantillas de clase, tenemos que cambiar las reglas básicas respecto del (de los) archivo(s) donde colocamos el código fuente. Anteriormente hemos colocado la declaración de clase en un archivo de encabezado `list.h` y las definiciones de funciones de miembros en un archivo de implementación `list.cpp`. Como resultado `list.cpp` podía ser compilado en código de objetos, independientemente de algún código de cliente. Esta estrategia no funcionará con plantillas. El compilador no puede crear una plantilla de función sino hasta que conozca el argumento para la plantilla, y este argumento está ubicado en el código de cliente. Diferentes compiladores tienen diferentes mecanismos para resolver este problema.

Una solución general que funciona con todos los compiladores es la de compilar el código de cliente y las funciones miembros de clase al mismo tiempo. Con nuestra plantilla `GList`, una técnica es prescindir de un archivo de implementación `glist.cpp` y colocar la plantilla de clase y las definiciones de funciones miembros en el mismo archivo, `glist.h`. Otra técnica es retener dos archivos distintos, `glist.h` y `glist.cpp` como antes, pero colocar la directiva `#include "glist.cpp"` al final del archivo `glist.h`. De cualquier modo, cuando el código de cliente especifica `#include "glist.h"`, el compilador tendrá el código fuente completo —tanto el código de cliente y las definiciones de funciones miembro— inmediatamente disponible.

En el siguiente código para la plantilla `GList` usamos la segunda técnica, esto es, la de mantener dos archivos separados y hacer que el archivo `.h` use `#include` al final para insertar el contenido del archivo `.cpp`. A continuación se muestra el archivo de encabezado, con omisión de precondiciones y poscondiciones para ahorrar espacio:

```
//****************************************************************************
// ARCHIVO DE ESPECIFICACIÓN (glist.h)
// Este archivo da la especificación de un TDA de lista genérica.
// No se supone que los componentes de la lista están en orden por valor
//****************************************************************************
```

```
const int MAX_LENGTH = 50;      // Número posible máximo de
                                //   componentes necesarios

// A continuación, ItemType debe ser un tipo simple de clase de cadena

template<class ItemType>
class GList
{
public:
    bool IsEmpty() const;
    bool IsFull() const;
    int  Length() const;
    void Insert( /* in */ ItemType item );
    void Delete( /* in */ ItemType item );
    bool IsPresent( /* in */ ItemType item ) const;
    void SelSort();
    void Reset();
    ItemType GetNextItem();
    GList();                    // Constructor
private:
    int      length;
    int      currentPos;
    ItemType data[MAX_LENGTH];
};

#include "glist.cpp"   // Inserta las definiciones de función miembro
```

Luego sigue el archivo de implementación `glist.cpp`. Gran parte de la documentación interna fue omitida para ahorrar espacio. La versión completamente documentada se puede encontrar en el disco de programa para este libro en el sitio web de la editorial. En lo siguiente, ponga mucha atención a la sintaxis requerida de las definiciones de funciones miembro.

```
//****************************************************************************
// ARCHIVO DE EJECUCIÓN (glist.cpp)
// Este archivo pone en práctica las funciones miembros de clase GList
// Representación de lista: un arreglo unidimensional y una variable
//                       de longitud
//****************************************************************************

// Advertencia: este archivo es #include por glist.h, así que
// NO use #include "glist.h" en este archivo, y
// NO compile este archivo por separado del archivo de código cliente

#include <iostream>

using namespace std;

template<class ItemType>
GList<ItemType>::GList()    // Constructor
{
    length = 0;
}

template<class ItemType>
```

```
bool GList<ItemType>::IsEmpty() const
{
    return (length == 0);
}

template<class ItemType>
bool GList<ItemType>::IsFull() const
{
    return (length == MAX_LENGTH);
}

template<class ItemType>
int GList<ItemType>::Length() const
{
    return length;
}

template<class ItemType>
void GList<ItemType>::Insert( /* in */ ItemType item )
{
    data[length] = item;
    length++;
}

template<class ItemType>
void GList<ItemType>::Delete( /* in */ ItemType item )
{
    int index = 0;

    while (index < length && item != data[index])
        index++;
    if (index < length)
    {                                   // Eliminar el elemento
        data[index] = data[length-1];
        length--;
    }
}

template<class ItemType>
bool GList<ItemType>::IsPresent( /* in */ ItemType item ) const
{
    int index = 0;

    while (index < length && item != data[index])
        index++;
    return (index < length);
}

template<class ItemType>
void GList<ItemType>::Reset()
{
    currentPos = 0;
}
```

```
template<class ItemType>
ItemType GList<ItemType>::GetNextItem()
{
    ItemType item;
    item = data[currentPos];
    if (currentPos == length - 1)
        currentPos = 0;
    else
        currentPos++;
    return item;
}

template<class ItemType>
void GList<ItemType>::SelSort()
{
    ItemType temp;
    int      passCount;
    int      searchIndx;
    int      minIndx;

    for (passCount = 0; passCount < length - 1; passCount++)
    {
        minIndx = passCount;
        for (searchIndx = passCount + 1; searchIndx < length;
             searchIndx++)
            if (data[searchIndx] < data[minIndx])
                minIndx = searchIndx;

        temp = data[minIndx];
        data[minIndx] = data[passCount];
        data[passCount] = temp;
    }
}
```

Advertencia

Si usted desarrolla sus programas usando un IDE (ambiente de desarrollo integrado, por sus siglas en inglés), donde el editor, compilador y ligador están integrados en una sola aplicación, deberá tener cuidado cuando usa plantillas. En caso de un IDE, normalmente se le pide que defina un "proyecto", esto es, una lista de archivos individuales que constituyen un programa. Mediante la clase `List` del capítulo 13 que incluye dos archivos (`list.h` y `list.cpp`) y su código de cliente ubicados en un archivo `myprog.cpp`, usted especificará `myprog.cpp` y `list.cpp` en su proyecto. La idea es que estos dos archivos sean compilados *por separado*, y luego se ligarán (por parte del ligador) sus archivos de código de objetos para que formen un archivo ejecutable.

Con plantillas, la situación podrá cambiar. Dados los archivos `glist.h` y `glist.cpp` que acabamos de mostrar, *no* queremos compilar `myprog.cpp` y `list.cpp` por separado. Como se ha dicho, no podemos compilar `glist.cpp` por sí solo porque los argumentos de plantilla están ubicados en un archivo distinto (`myprog.cpp`). Por tanto, si especificamos tanto `myprog.cpp` y `glist.cpp` en un proyecto, recibimos mensajes de error del compilador y/o ligador. La solución en este caso es especificar *sólo* `myprog.cpp` en el proyecto. (Puesto que `myprog.cpp` dice que tenemos que incluir `glist.h`, y `glist.h` a su vez dice que tenemos que incluir `glist.cpp`, queremos que el compilador reciba sólo un fracción grande de código fuente para la compilación.)

Ahora cambiamos nuestra atención de las plantillas y nos dirigimos a otra característica útil de leguaje C++: las *excepciones*.

17.3 Excepciones

Suponga que estamos escribiendo un programa en el cual con frecuencia es necesario dividir dos enteros y obtener el cociente. En cada caso se necesita revisar si no hay ninguna división entre 0, así que escribimos una función denominada `Quotient` que devuelve el cociente de cualquier par de enteros, a menos que el denominador sea 0:

```
int Quotient( /* in */ int numer,     // El numerador
              /* in */ int denom )    // El denominador
{
    if (denom != 0)
        return numer / denom;
    else
        // What to do??
}
```

Estamos frente a un problema: ¿qué deberá hacer esta función si el denominador es 0? Ésta no es una pregunta fácil de contestar. A continuación se indican algunas posibilidades, pero ninguna de ellas es completamente satisfactoria:

1. No realice la división y regrese silenciosamente al código de llamada.
2. Imprima un mensaje de error y devuelva un valor entero arbitrario.
3. Devuelva un valor especial, por ejemplo –9999, al invocador como señal de que algo salió mal.
4. Rescriba la función con un tercer parámetro, una bandera booleana indicando éxito o fracaso.
5. Imprima un mensaje de error y detenga el programa.

La opción 1 es claramente irresponsable. La opción 2 podrá ayudar al ser humano que ejecuta el programa, visualizando un mensaje de error, pero no notificará al código de llamada de que algo salió mal. La opción 3 podría ser buena en algunas circunstancias, pero en general no es una buena solución. Si se permite que el numerador y el denominador sean cualquier entero, entonces –9999 es un cociente perfectamente válido y no se podrá distinguir de un valor especial de señal. La opción 4 es un planteamiento razonable, y los programadores la usan con frecuencia. La desventaja en nuestro caso es que tenemos que cambiar nuestra función de devolución de valor (que sólo devuelve un valor) a una función void, de modo que sea posible devolver *dos* valores por medio de la lista de parámetros como parámetros de referencia: el cociente y la bandera booleana. La opción 5 casi nunca es satisfactoria. Se deberá permitir que el código de llamada, mas no la función llamada, decida qué hacer en caso de error. Tal vez el invocador quisiera tomar medidas para recuperarse del error y seguir ejecutando en lugar de terminar el programa.

¡Una salida de nuestro dilema es eliminar el dilema! Use una precondición de función como la que sigue:

```
int Quotient( /* in */ int numer,     // El numerador
              /* in */ int denom )    // El denominador

// Precondición:
//     denom != 0
// Poscondición:
//     Valor de retorno == numer / denom

{
    return numer / denom;
}
```

Aquí el invocador debe asegurar que la precondición es verdadera antes de llamar la función y, por tanto, la función no tendrá que hacer ninguna detección de errores. Mediante este planteamiento, el invocador es responsable tanto de la detección de errores como del manejo de errores. A través de todo este libro hemos usado esta estrategia de usar precondiciones para eliminar la detección de errores de las funciones de llamada. Sin embargo, hay situaciones donde los errores se pueden detectar sólo *después* de haber intentado una acción, no antes. Por ejemplo, una función `ReadInt` que supuestamente debe leer un valor entero del teclado puede descubrir que el usuario ha tecleado erróneamente algunas letras (caso en que la variable de entrada no cambia, y la flujo `cin` entra en estado de falla). En este supuesto no podemos declarar una precondición para `ReadInt` para predecir qué es lo que el ser humano tecleará. A fin de manejar errores de este tipo (y para programadores que prefieren no usar la opción de precondiciones), el lenguaje C++ proporciona lo que se denomina un mecanismo de *manejo de excepciones*.

Excepción Suceso inusual, con frecuencia impredecible, detectable por software o hardware, que requiere un procesamiento especial; también, en C++, una variable u objeto de clase que representa un suceso excepcional.

Manejador de excepción Sección del código de programa que es ejecutada cuando ocurre una excepción particular.

Lanzar Señalar el hecho de que una excepción ha ocurrido; también se llama *levantar*.

Capturar Procesar una excepción lanzada. (La captura es realizada por un manejador de excepción.)

En el mundo del software, una **excepción** es un suceso inusual, a menudo un error, que requiere un procesamiento especial. Una sección del código de programa que proporciona este procesamiento especial se denomina **manejador de excepción**. Cuando una sección de código anuncia que una excepción ha ocurrido, se dice que **lanza** (o **levanta**) una excepción, esperando que otra sección del código (el manejador de excepción) **capture** la excepción y la procese. Si no existe ningún manejador de excepción para esta excepción particular, el programa completo terminará con un mensaje de error. Ahora veremos el mecanismo que C++ proporciona para lanzar y capturar excepciones.

La sentencia `throw`

En C++ la palabra *excepción* no sólo tiene el significado general de un suceso inusual, sino también tiene un significado más específico: una variable u objeto de clase que representa este tipo de suceso. Para lanzar (levantar) una excepción, el programador usa una sentencia `throw`, cuya sintaxis es la siguiente:

ThrowStatement

```
throw  Expression ;
```

En la sentencia `throw`, Expresión puede ser un valor o una variable de cualquier tipo de datos, ya sea integrada o definida por el usuario. Veamos tres ejemplos de sentencias `throw`. A continuación el ejemplo 1:

```
throw 5;
```

En este ejemplo lanzamos una excepción del tipo `int` con la expectativa de que uno o varios manejadores de excepción quieran capturar una excepción del tipo `int` (en la siguiente sección veremos cómo se realiza esto). En el ejemplo 2, a continuación, lanzamos una excepción del tipo `string`, o sea la clase estándar de biblioteca:

```
string str = "Edad del cliente no válida";
throw str;
```

Por último, el ejemplo 3 lanza una excepción de tipo de clase que definimos nosotros mismos:

```
class SalaryError
{};              // La lista de miembros está vacía
```

```
 ⋮
SalaryError sal;
throw sal;
```

En el ejemplo 3 observamos cómo declaramos primero un objeto de clase denominado `sal` y luego lanzamos este objeto. Más comúnmente los programadores de C++ hacen lo siguiente:

```
class SalaryError
{};              // La lista de miembros está vacía
 ⋮
throw SalaryError();>
```

En este código la sentencia `throw` crea un objeto anónimo (no nombrado) del tipo `SalaryError`, explícitamente llamando a su constructor de default (significado por la lista vacía de argumentos), y luego pasa este objeto anónimo a un manejador de excepción. Sería un error de sintaxis omitir los paréntesis:

```
throw SalaryError;   // No válida
```

porque la plantilla de sintaxis para la sentencia `throw` muestra que necesitamos una expresión, y no el nombre de un tipo de datos.

Por último, observe en la plantilla de sintaxis que Expresión es opcional. Más adelante analizaremos qué pasa si la expresión `throw` no se encuentra.

La sentencia `try-catch`

Si una parte de un programa lanza una excepción, ¿cómo capturará otra parte del programa esta excepción para procesarla? La respuesta es: usando una sentencia `try-catch` con la siguiente forma:

TryCatchStatement

```
try
  Block
catch ( FormalParameter )
  Block
catch ( FormalParameter )
  Block
    ⋮
```

La sintaxis de FormalParameter es la siguiente:

FormalParameter

```
{ DataType VariableName
{ ...
```

(En esta última plantilla de sintaxis, los tres puntos suspensivos son, literalmente, tres periodos que están capturados en el programa.)

Como lo muestra la primera plantilla de sintaxis, la sentencia `try-catch` consiste en una *cláusula try* seguida por una o varias *cláusulas catch*. La cláusula *try* consiste en la palabra reservada `try` y un bloque [un par de ( ) conteniendo un número de sentencias]. Cada cláusula `catch` consiste en la palabra reservada `catch`, una declaración de parámetro simple entre paréntesis, y un bloque. Cada cláusula `catch` es, de hecho, un manejador de excepción.

Cuando una sentencia o un grupo de sentencias en un programa pueden resultar en una excepción, los encerramos en una cláusula *try*. Para cada tipo de excepción que puede ser producido por

las sentencias escribimos una cláusula *catch* (manejador de excepción). A continuación se presenta un ejemplo de una sentencia `try-catch` que involucra los tres tipos de excepciones (`int`, `string` y `SalaryError`) que hemos analizado con la sentencia `throw`:

```
try
{
    ⋮       // Sentencias que procesan los datos del personal y que podrían
            // lanzar excepciones del tipo int, string y SalaryError
}
catch ( int )
{
    ⋮       // Sentencias para manejar una excepción int
}
catch ( string s )
{
    cout << s << endl;  // Imprime "Edad del cliente no válida"
    ⋮       // Más sentencias para manejar un error de edad
}
catch ( SalaryError )
{
    ⋮       // Sentencias para manejar un error de salario
}
```

La sentencia `try-catch` suena como el entrenador que dice al gimnasta: "Ve e intenta ese salto mortal, y yo te agarro si te caes." Le estamos diciendo a la computadora que intente la ejecución de algunas operaciones que podrían fracasar, y luego le proporcionamos un código para atrapar las excepciones potenciales.

Ejecución de `try-catch` La ejecución de la sentencia `try-catch` que antecede funciona de la siguiente manera. Si ninguna de las sentencias en la cláusula *try* lanza una excepción, entonces el control se transfiere a la sentencia siguiente a la sentencia completa de `try-catch`. Esto es, que probamos unas sentencias, y si todo marcha de acuerdo con el plan, simplemente continuamos con las sentencias siguientes. Sin embargo, si una excepción es lanzada por una sentencia en la cláusula *try*, el control inmediatamente se transfiere al manejador de excepción correspondiente (cláusula c*atch*). Es importante entender que el control salta directamente de la sentencia causante de la excepción al manejador de excepción. Si existen sentencias en la cláusula *try* que siguen a la que causó la excepción, entonces se brincan. Cuando el control llega al manejador de excepción, se ejecutan las sentencias que se ocupan de la excepción, y si estas sentencias no causan ninguna nueva excepción y no transfieren el control a otra parte (como en el caso de una sentencia `return`), entonces el control pasará a la siguiente sentencia siguiendo la estructura completa de `try-catch`.

Parámetros formales en manejadores de excepción Cuando se lanza una excepción, ¿cómo sabe la computadora cuál de los manejadores de excepción es el apropiado? La computadora revisa el tipo de datos del parámetro formal declarado en cada manejador y selecciona el primero cuyo tipo de datos es igual al que lanzó la excepción. Un parámetro formal que sólo consiste en una elipsis (tres puntos o periodos), como se mostró en la plantilla de sintaxis para FormalParameter, es un "comodín": se iguala a cualquier tipo de excepción. Puesto que la computadora busca entre los manejadores de excepción para encontrar un parámetro igual en orden secuencial o "de norte a sur", un manejador de excepción final con un parámetro de elipsis sirve como manejador de "atrapa todo" para cualquier excepción cuyo tipo no está enumerado:

```
try
{
    ⋮       // Sentencias que pueden lanzar una excepción
}
```

```
catch ( Type1 )
{
    ⋮       // Maneja una excepción de tipo 1
}
catch ( Type2 )
{
    ⋮       // Maneja una excepción de tipo 2
}
catch ( . . .  )                  // Manipulador de atrapar todo
{
    cout << "¡Pánico! Excepción inesperada." << endl;
    ⋮       // Sentencias para tratar con esta situación
}
```

Observe que la búsqueda "de norte a sur" para encontrar un tipo de parámetro igual nos obliga a colocar el manejador que atrapa todo hasta el final. Si lo colocáramos primero, atraparía a *cada* excepción, y los demás manejadores serían ignorados.

Otro problema respecto a los parámetros formales es: ¿debería la declaración de parámetro incluir un nombre para el parámetro o no? (Si regresamos y vemos la plantilla de sintaxis para FormalParameter, vemos que el tipo de datos del parámetro es requerido, pero su nombre es opcional.) La respuesta es que el nombre del parámetro es necesario sólo si existen sentencias en el cuerpo del manejador de excepción que usan esta variable. En nuestro ejemplo anterior de capturar excepciones de `int`, `string` y `SalaryError`, las listas de parámetros para el primer y tercer manejador de excepción contienen sólo nombres de tipos de datos, mientras que la lista de parámetros para el segundo manejador especifica `string s`. La razón es que el cuerpo de ese manejador usa la sentencia

```
cout << s << endl;  // Imprime "edad del cliente no válida"
```

para imprimir el mensaje contenido en `s`.

Por último, abordaremos un tema muy importante en la programación de excepciones: los tipos de datos de las propias excepciones. En nuestro ejemplo de `int`, `string` y `SalaryError` hemos usado los tres tipos de excepción sólo para demostrar las posibilidades para el programador y para ilustrar cómo se escriben manejadores de excepción a fin de capturar excepciones de estos tipos. En la práctica, las excepciones de tipos integrados (`int`, `float`, etcétera), e incluso de la clase `string`, son de utilidad limitada. Si una cláusula *try* lanza varias excepciones del tipo `int` con diferentes valores de enteros, entonces un manejador de excepción que recibe un valor `int` se vuelve complicado. Debe incluir una lógica que verifique el entero para determinar exactamente qué tipo de error ocurrió. Además, sentencias como

```
throw 23;
```

son mucho menos legibles y de autodocumentación que sentencias como:

```
throw SalaryError();
```

En consecuencia, es mejor usar sólo clases (y *structs*) definidas por el usuario como tipos de excepciones, definiendo un tipo para cada tipo de excepción y usando identificadores descriptivos:

```
class SalaryError    // Excepción de clase
{};
class BadRange       // Excepción de clase
{};
 ⋮
if ( condition )
```

```
    throw SalaryError();
 ⋮
if ( condition )
    throw BadRange();
```

Manejadores de excepción no locales

Hasta ahora, en nuestro análisis hemos supuesto que una sentencia `throw` está físicamente ubicada dentro de la sentencia `try-catch` diseñada para capturar la excepción. En este caso, si la excepción es lanzada, el control transfiere a la cláusula *catch* con el correspondiente tipo de datos.

Sin embargo, es más común en programas de C++ que el `throw` ocurra dentro de una función que se *llama* desde dentro de una cláusula *try*, como se muestra con las funciones `Func3` y `Func4` en la figura 17-1. Al tiempo de ejecución la computadora primero busca si existe un `catch` dentro de `Func4`. Cuando no encuentra ninguno, causa que `Func4` regrese inmediatamente y devuelve la excepción a su invocador `Func3`. Luego la computadora busca alrededor del punto donde `Func4` fue llamada, busca una cláusula `catch` apropiada y ejecuta el `catch`. Como se puede observar, la excepción fue lanzada en `Func4`, pero el manejador de excepción no es local (está en la función de llamada, `Func3`).

Suponga que en la figura 17-1 no hubo una cláusula catch en `Func3`. Entonces `Func3` regresa inmediatamente y devuelve la excepción a *su* invocador, tal vez `Func2`. Este proceso es como el juego de niños de la "papa caliente". Cada función que no sabe cómo manejar el problema devuelve la papa (la excepción) a la función precedente. Esta secuencia continua de devoluciones hacia atrás a través de la cadena de llamadas de función hasta que se encuentre un manejador de excepción correspondiente o hasta que el control alcance `main`. Si `main` falla en la captura de la excepción (situación denominada *excepción no capturada*), el sistema termina el programa y muestra un mensaje relativamente inútil, como "ABNORMAL PROGRAM TERMINATION". La figura 17-2*a* y *b* ilustra este proceso.

Observe que aunque una sentencia `throw` *esté* presente en la cláusula try de un `try-catch`, pero no haya ningún manejador de excepción correspondiente en esa sentencia `try-catch`, el resultado es el mismo que acabamos de describir, es decir, que la función que contiene regresa inmediatamente y vuelve a pasar la excepción de nuevo a la cadena de llamadas.

Quizás usted se está preguntando por qué hemos dicho que es lo más común que programas de C++ usen manejadores de excepción no locales. La razón se encuentra en la médula del manejo

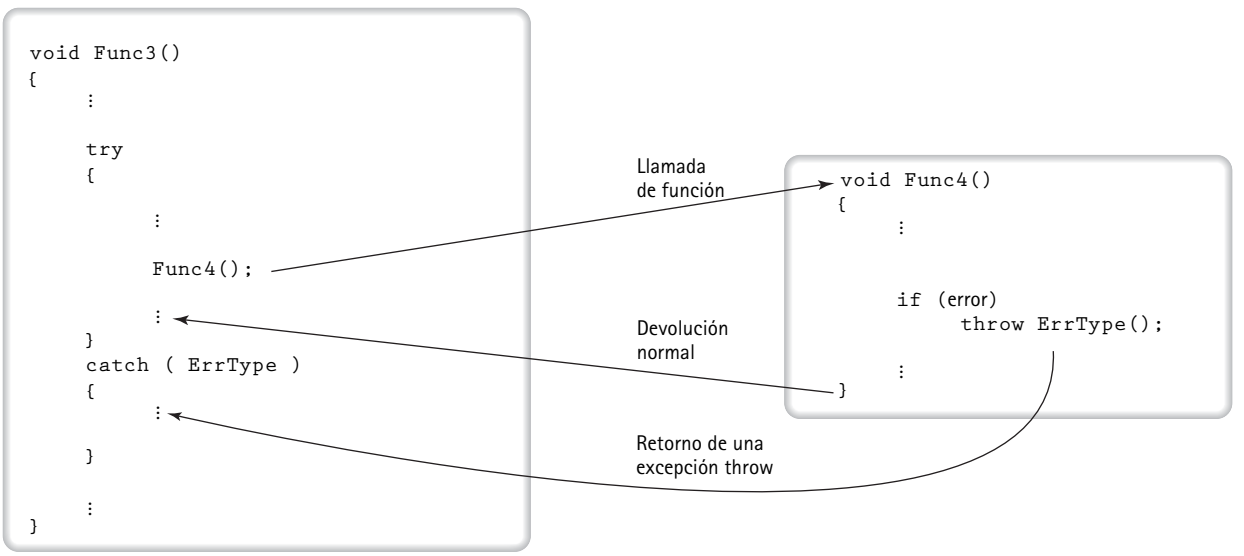


Figura 17-1 *Lanzamiento de una excepción para ser capturada por el código de llamada*

de excepciones, independientemente del lenguaje de programación que se use. El propósito fundamental de las excepciones es permitir que una parte de un programa reporte un error a otra parte del programa si el error no se puede manejar localmente. Cuando una función llama a otra función, en general las dos muestran una relación de maestro/esclavo. El maestro (función que llama) le dice al esclavo (función llamada) que haga algo por él. Si el esclavo encuentra una situación excepcional, tal vez no tenga la información suficiente acerca del "mundo exterior" para saber cómo recuperarse del error. Así que el esclavo se rinde y devuelve el error al maestro, lanzando una excepción. Si el maestro sabe cómo recuperarse del error, captura la excepción por medio de un manejador de excep-

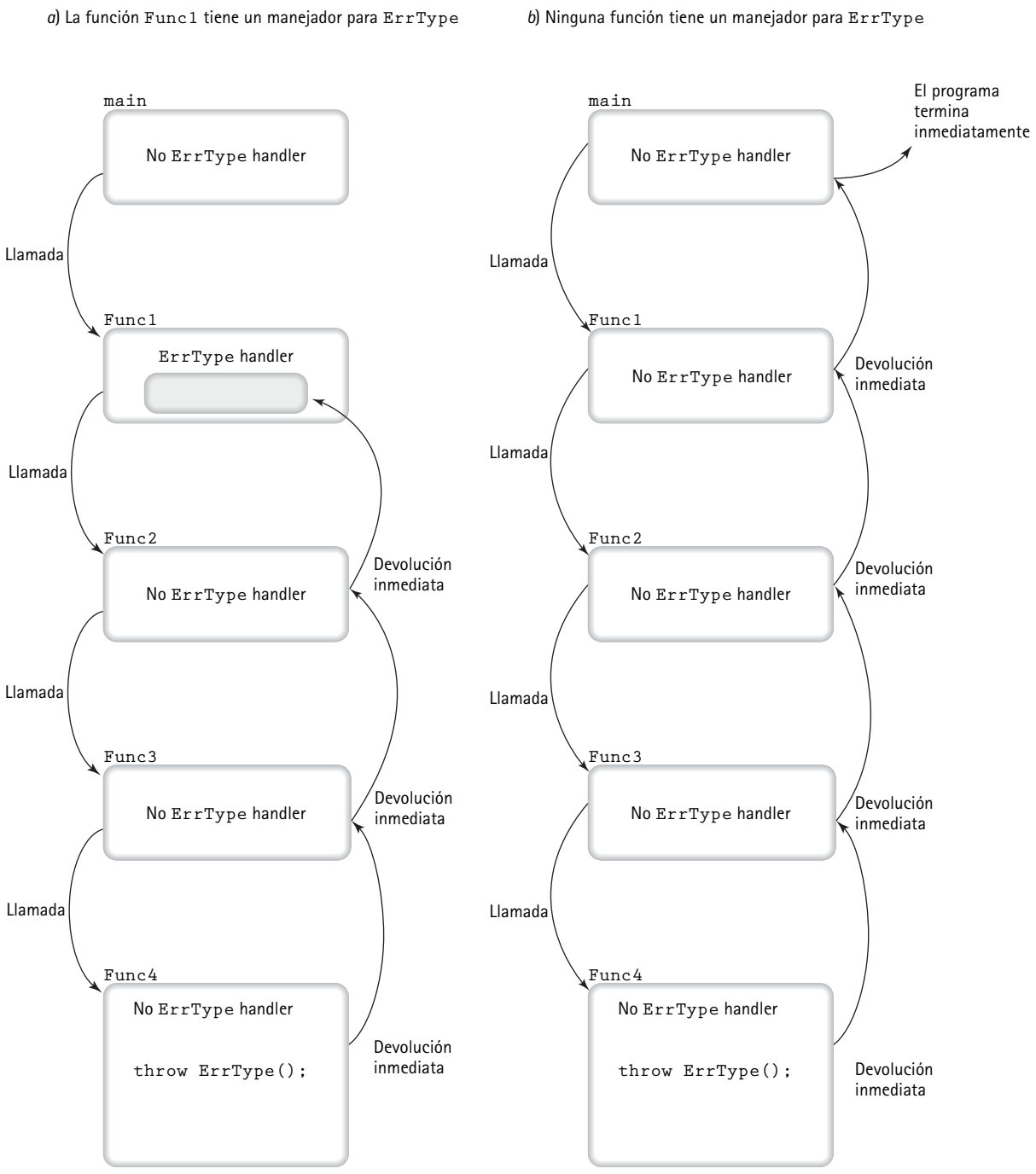


Figura 17-2 *Pasar una excepción a través de la cadena de llamadas de función*

ción; de otro modo devuelve la excepción a *su* maestro, etcétera. Con frecuencia la excepción regresa todo el camino hasta `main` antes de que sea posible tomar una decisión, ya sea de terminar el programa o recuperarse del error y seguir ejecutando.

Relanzamiento de una excepción

Hemos usado la sentencia `throw` para la siguiente forma:

```
throw SalaryError();
```

La plantilla de sintaxis para la sentencia `throw` que presentamos al inicio del capítulo muestra que usamos la palabra `throw`, seguida *opcionalmente* por una excepción. Por tanto, la expresión `throw` puede ser omitida, como en la sentencia

```
throw;
```

Esta sentencia se usa dentro de un manejador de excepción que ha capturado una excepción y ha ejecutado algunas acciones, pero luego desea pasar la excepción a su manejador (*relanzar* la excepción) para el procesamiento subsiguiente. A continuación se ofrece un ejemplo:

```
void WriteToFile( parámetros )
{
    ⋮  // Abrir un archivo para resultado
    try
    {
        while ( condición )
        {
            DoSomething( argumentos );  // Podría lanzar una excepción de
                                        // BadData
     ⋮  // Escribir para el archivo de salida
        }
    }
    catch ( BadData )
    {
        ⋮       // Escribir un mensaje para archivo de salida y cerrarlo
        throw;  // Volver a lanzar la excepción
    }
    ⋮  // Continuar el proceso
       //  y cerrar el archivo de salida
}
```

El relanzamiento de una excepción es la forma en que C++ permite un *manejo parcial de excepciones*. En el código anterior, la cláusula *catch* manipula la excepción `BadData` de manera parcial (escribiendo un mensaje de error al archivo y cerrándolo), y luego regresa, relanzando la excepción a través de la cadena de llamadas de función hasta que se capture.

¿Qué pasaría en el código anterior si borráramos la sentencia `throw`? ¿Qué sucedería si cambiáramos `throw` por `return`? Un ejercicio de preparación para examen, al final de este capítulo, le pide que considere estas preguntas.

Excepciones estándar

No todas las excepciones son definidas por nosotros como programadores de C++. Varias clases de excepciones están predefinidas en la biblioteca estándar de C++ y son lanzadas ya sea por operaciones de C++ o por un código que se proporciona en rutinas de la biblioteca estándar.

Excepciones lanzadas por el lenguaje Algunas operaciones de C++ lanzan excepciones si ocurren errores durante su ejecución. Estas operaciones son `new`, `dynamic_cast`, `typeid` y algo que se denomina especificación de excepción. Las últimas tres no se abordarán en este libro, así que aquí sólo examinamos el operador `new`. Si usted no ha leído acerca de la asignación dinámica en el capítulo 15, tal vez querrá omitir este análisis.

En el capítulo 15 dijimos que el operador `new` obtiene una fracción de memoria del área de almacenamiento libre (*heap*) y regresa un apuntador a (la dirección de) dicha fracción. Por ejemplo, el código

```
int* intPtr;
intPtr = new int[1000];
```

crea un arreglo de 1 000 elementos tipo `int` en el área de almacenamiento libre y asigna la dirección base del arreglo a `intPtr`. También dijimos que si al sistema se le acaba el espacio en el área de almacenamiento libre, la ejecución `new` detiene el programa con un mensaje de error. Ahora que sabemos acerca de las excepciones, es posible describir con más detalle lo que sucede. Si `new` se da cuenta que el almacén libre está agotado, lanza una excepción del tipo `bad_alloc`, una clase de C++ que está predefinida en la biblioteca estándar. Si esta excepción no es capturada por algún manejador de excepción, entonces (como en caso de cualquier excepción no capturada) el programa se detiene con un mensaje genérico, como "ABNORMAL PROGRAM TERMINATION". Por otra parte, nuestro programa puede capturar la excepción y tomar alguna acción correctiva (o por lo menos visualizar un mensaje de error más específico). Por ejemplo, podríamos escribir, en nuestra función `main`, un código como el que sigue:

```
float* arr;
try
{
    arr = new float[50000];
}
catch ( bad_alloc )
{
    cout << "*** Memoria agotada. No se puede asignar el arreglo." << endl;
    return 1;    // Terminar el programa
}
⋮ // Continuar. Asignación exitosa
```

Observe la sentencia `return` en el manejador de excepción. Esta sentencia terminará el programa sólo si se encuentra en la función `main`. Si deseamos que el código anterior sea ubicado en alguna otra función, será necesario cambiar la sentencia `return` a otra cosa. Podríamos relanzar la excepción `bad_alloc` a una función de nivel superior o lanzar una nueva excepción (denominándola, por ejemplo, `OutOfMem`) para ser capturada por una función de nivel superior.

Nota: Al momento de escribir esto, no todos los compiladores C++ ejecutan completamente la norma de lenguaje C++ ISO/ANSI respecto al operador `new`. Algunos compiladores aún usan la vieja definición (pre-estándar) de `new` en que una asignación de memoria fracasada devuelve el apuntador nulo (*0*) en lugar de lanzar una excepción. Si usted usa uno de estos compiladores, escribiría el código precedente de la manera siguiente:

```
float* arr;

arr = new float[50000];
if (arr == 0)
{
    cout << "*** Memoria agotada. No se puede asignar el arreglo." << endl;
    return 1;    // Terminar el programa
}
⋮ // Continuar. Asignación exitosa
```

Excepciones lanzadas por rutinas de bibliotecas estándar La biblioteca estándar de C++ consiste en dos bibliotecas: facilidades heredadas del lenguaje C (denominada biblioteca C) y facilidades diseñadas específicamente para C++. En general, los archivos de encabezado empiezan con la letra *c*: `cmath`, `cstddef` y `cctype`, por ejemplo; son parte de la biblioteca C, y todos los demás archivos de encabezado son específicos de C++. (Un archivo de encabezado de nombre `complex` no encaja en este patrón. Empieza con *c*, pero es específico de C++.) Las rutinas de biblioteca en la biblioteca C no lanzan excepciones porque las excepciones no forman parte del lenguaje C. En su lugar, estas rutinas utilizan una variable de entero global denominada `errno`, que está fijada a un valor distinto de cero si un error ocurre en una rutina de biblioteca. El valor exacto asignado a `errno` depende de la rutina de biblioteca particular que se está ejecutando, y el programador lo podrá buscar en la documentación para la biblioteca C. Sin embargo, a menudo sólo nos interesa si un error ocurrió (`errno` es distinto de cero) o si no ocurrió. Aquí hay una manera de simular el lanzamiento de excepciones a partir de rutinas de biblioteca C, aunque la biblioteca C no lo haga. Empiece por fijar `errno` a 0. Llame la rutina de biblioteca y luego verifique `errno`. Si `errno` sigue como 0, no hubo error. Si no es cero, lance su propia excepción. En el siguiente código, *C_lib_routine* sería remplazado con el nombre de una verdadera rutina de biblioteca C, como `sqrt` o `abs`:

```
class CLibErr    // Nuestra propia clase de excepción
{};
 ⋮
void SomeFunc( parámetros )
{
    ⋮
    errno = 0;
    C_lib_routine( argumentos );
    if (errno != 0)
        throw CLibErr();
    ⋮
}
```

La otra parte de la biblioteca estándar de C++ (la parte con facilidades que son específicas de C++) define varias clases de excepciones, así como diversos objetos de lanzamiento de rutinas de biblioteca de estas clases cuando ocurren errores. Las clases de excepción son `bad_alloc`, `out_of_range`, `length_error`, `domain_error` y otras. Aparte de `bad_alloc` que hemos estudiado en la sección anterior, los únicos tipos de excepciones que tienen relación con el material que se aborda en este libro son `out_of_range` y `length_error`. Estas excepciones se pueden lanzar cuando se trabaja con objetos de la clase `string`, disponibles mediante el archivo de encabezado denominado `string`. Veamos ahora estas excepciones.

Casi al final del capítulo 3 examinamos una función de miembro de clase `string` denominada `substr`. Dijimos que si `str` es un objeto `string`, entonces la expresión `str.substr(pos, len)` devuelve un nuevo objeto `string` que es la subcadena de `str` comenzando en la posición `pos` y con la longitud `len`. Dijimos que si `pos` es demasiado grande, el programa termina con un mensaje de error. En más detalle, lo que pasa es que si `pos` es demasiado grande, se lanza una excepción `out_of_range`. Si esta excepción no se captura, entonces sucede —como en caso de toda excepción no capturada— que es cierto que el programa termina con un mensaje de error vago. Pero podemos capturar la excepción si así lo deseamos, para luego manipularla de algún modo.

Otra excepción que puede ser lanzada por la clase `string` es `length_error` y esto puede pasar de la siguiente manera. Recuerde que las posiciones de enteros de caracteres individuales dentro de un objeto `string` están representadas por un tipo de entero sin signo `string::size_type`. El intervalo de valores en este tipo es 0 a través de `string::pos`-1, donde `string::npos` es una constante cuyo valor en muchas máquinas es aproximadamente 4 mil millones. En el (improbable) supuesto de que su programa intente construir una cadena cuya longitud sea mayor a `string::npos`, se lanza una excepción `length_error`. Una manera en que esto podría suceder sería que su programa concatene repetidamente grandes cadenas (usando expresiones como `str1 + str2`) de manera

no contenida. Entonces, para protegerse contra errores de intervalo y errores de longitud cuando se trabaja con objetos `string`, podríamos escribir un código como:

```
void SomeFunc( parámetros )
{
    string s1, s2;
    try
    {
        ⋮
        s2 = s1.substr(pos, len);  // Podría lanzar out_of_range()
        s1 = s1 + s1 + s2;         // Podría lanzar length_error()
        ⋮
    }
    catch ( out_of_range )
    {
        cout << "Excepción: out_of_range en SomeFunc" << endl;
        throw;    // Lanzar de nuevo la excepción para un llamador
    }
    catch ( length_error )
    {
        cout << "Excepción: length_error en SomeFunc" << endl;
        throw;    // Volver a lanzar la excepción para un llamador
    }
    ⋮   // Continuar si no hay errores
}
```

Aunque hemos mencionado sólo dos clases de excepciones como las usa la clase `string`, hay varias otras clases de excepción estándar que son usadas por otras facilidades de biblioteca estándar. Trabajando más con la biblioteca estándar de C++ en el futuro, usted conocerá estos tipos de excepciones.

Regresando al problema de división entre cero

Hemos empezado el tema de las excepciones proponiendo una función `Quotient` que devuelve el cociente de dos enteros, a menos que el divisor sea 0. Dijimos que si el divisor es 0, no es claro cómo la función deberá manejar el error. Dijimos que una solución era desaparecer el problema por medio de una precondición (que el invocador no debe transmitir un argumento de 0 como divisor). Ahora presentaremos otra solución que combina lo que hemos aprendido sobre las excepciones. Definimos nuestra propia clase de excepción `DivByZero`, y la función lanza una excepción de este tipo si descubre un divisor 0. La idea es que la función `Quotient` no puede saber si se deberá terminar el programa o no, así que lanza una excepción y deja que una función de nivel superior decida qué hacer. El siguiente es un programa muy sencillo que pide al usuario un par de enteros y extraiga su cociente repetidamente hasta que el usuario teclee la secuencia de fin del archivo. En este programa es decisión de `main` determinar que *no* se termina el programa si captura una excepción. En su lugar se visualiza un mensaje de error para el usuario, y el ciclo continúa.

```
// quotient.cpp - Programa Cociente

#include <iostream>
#include <string>

using namespace std;

int Quotient( int, int );
```

```
class DivByZero    // Clase de excepción
{};

int main()
{
    int numer;   // Numerador
    int denom;   // Denominador

    cout << "Introducir el numerador y el denominador: ";
    cin >> numer >> denom;
    while (cin)
    {
        try
        {
            cout << "Su cociente: "
                 << Quotient(numer, denom) << endl;
        }
        catch ( DivByZero )
        {
            cout << "*** El denominador no puede ser 0" << endl;
        }
        cout << "Introducir el numerador y el denominador: ";
        cin >> numer >> denom;
    }
    return 0;
}

int Quotient( /* in */ int numer,    // El numerador
              /* in */ int denom )   // El denominador
{
    if (denom == 0)
        throw DivByZero();
    return numer / denom;
}
```

Caso práctico de resolución de problemas

Reimplementación de la especificación SortedList y mejora del calendario de citas

PROBLEMA Este caso práctico consta de dos partes. Primero, para comprobar que un cambio de la implementación subyacente de una clase no afecta al código de cliente que usa la clase, tome el archivo de especificación de clase `SortedList` que hemos usado en el caso práctico del capítulo 14, cambie la representación de datos y ejecute la especificación usando una lista ligada. Verifique la implementación utilizando el mismo controlador en la nueva implementación.

Después de verificar la `SortedList` regrese al calendario de citas y haga los cambios de la especificación que considere necesarios para hacer el TDA más robusto. Implemente y verifique los cambios.

Implementación ligada de SortedList

ANÁLISIS Lo primero que es necesario hacer es revisar de nuevo las especificaciones y tomar apuntes de los cambios necesarios. Haremos los comentarios dentro del texto de la especificación.

```
//****************************************************************************
// ARCHIVO DE ESPECIFICACIÓN (SortedList.h)
// Este archivo da la especificación de un tipo de datos abstractos
// SortedList. Se supone que los elementos de SortedList están en orden
// por valor
//****************************************************************************

#include "Entry.h"

const int MAX_LENGTH =100;       // Número posible máximo de
                                 //   elementos necesarios
```

En una lista basada en arreglo el número máximo de celdas debe ser conocido cuando se crea el arreglo: ya sea al tiempo de compilación o cuando el arreglo es creado, usando `new` durante el tiempo de ejecución. En una lista ligada no es necesario conocer el número de elementos; sin embargo, para algunas aplicaciones un número máximo podrá ser útil.

```
typedef Entry ItemType;          // Tipo de cada elemento
                                 //   (un tipo simple de clase de cadena)
```

Ahora que sabemos cómo usar plantillas y cómo hacer un tipo parámetro de la clase, vamos a hacerlo aquí. El único cambio que requerirá el programa es incluir el tipo como parámetro cuando se declare la estructura. La restricción del tipo debe ser simple, como el que se dio para que se definan los operadores relacionales. Esto es un tema de implementación que abordaremos más adelante.

```
class SortedList
{
public:
    bool IsEmpty() const;
    // Poscondición:
    //     El valor de retorno es verdadero, si SortedList está vacía;
    //     falso en caso contrario

    bool IsFull() const;
    // Poscondición:
    //     El valor de retorno es verdadero, si SortedList está llena;
    //     falso en caso contrario
```

Si no tenemos un número máximo de elementos de la lista, no necesitamos una operación `IsFull`. Si queremos que todas las operaciones estén disponibles por motivos de consistencia, podríamos hacer que `IsFull` devuelva `false`. Si decidimos que deberíamos dar al usuario la opción de fijar un máximo, entonces `IsFull` verificaría contra ese valor. Por ahora, simplemente devolvamos `false`.

```
 int Length() const;
        // Poscondición:
        //     El valor de retorno es la longitud de SortedList

    void Insert( /* in */ ItemType& item );
        // Inserta el elemento en SortedList
        // Precondición:
        //     longitud < MAX_LENGTH
        //  && data[0..length - 1] están en orden ascendente
        // Poscondición:
        //     el elemento está en SortedList
```

```
        //  && longitud == length@entry + 1
        //  && data[0..length - 1] están en orden ascendente
```

Naturalmente, la documentación para la lista basada en arreglo ya no aplicará.

```
    void Delete( /* in */ ItemType item );
        // Poscondición:
        //     NOT IsEmpty()
        // Poscondición:
        //     IF el elemento está en SortedList en la entrada
        //           La primera aparición del elemento ya no está en
        //           SortedList
        //        && Length() == Length()@entry - 1
        //     ELSE
        //           SortedList permanece sin cambio
```

`Delete` es "borrar si está allí". El algoritmo asignado a la función `Delete` en la versión ligada que se dio en el capítulo 17 es un "borrar, está allí".

```
    bool IsPresent( /* in */ ItemType item ) const;
        // Precondición:
        //     se asigna el elemento
        // Poscondición:
        //     Valor de retorno == true, si el elemento está en SortedList
        //                      == false, en caso contrario

    void Reset();
        // Poscondición:
        //     Se inicializa la iteración

    ItemType GetNextItem();
        // Precondición:
        //     Ningún transformador ha sido invocado desde la última llamada
        // Poscondición:
        //     Devuelve el elemento en la posición actual en SortedList

    SortedList();
        // Constructor
        // Poscondición:
        //     Se crea la SortedList vacía
private:
    int      length;
    int      currentPos;
    ItemType data[MAX_LENGTH];
    void     BinSearch( ItemType, bool&, int& ) const;
};
```

En una lista basada en arreglo, la variable de longitud debe estar allí para distinguir la lista del arreglo en la que están guardados los elementos. En una lista ligada tenemos dos opciones: podemos mantener una variable de longitud, la cual incrementamos y disminuimos con cada inserción y eliminación; por otro lado, podemos contar el número de elementos cada vez que se llame la función `Length`. ¿Cuál es la mejor opción? Depende de cuántas veces se llame la función `Length`. Si hay muchas inserciones y eliminaciones, y la función `Length` se llama sólo unas cuantas veces, entonces el conteo es mejor. Usemos aquí una variable `Length`.

Aún necesitamos un modo de rastrear la posición actual en la lista durante una corrida. Sin embargo, esta variable contendrá un apuntador en lugar de un índice. En vez de una variable de arreglo para contener los elementos tenemos un apuntador al encabezado de la lista ligada. Con esto es evidente que el algoritmo de búsqueda binaria no será necesario. De hecho, ni se puede usar. No es posible aplicar una búsqueda binaria a una lista ligada.

Ya que la implementación es ligada, deberíamos incluir un constructor de copia y un destructor. A continuación se muestra el archivo de especificación para `SortedList2`.

```
//****************************************************************************
// ARCHIVO DE ESPECIFICACIÓN (SortedList2.h)
// Este archivo da la especificación de un tipo de datos abastractos
// de lista clasificada. La lista de elementos se mantiene en orden
// de valor ascendente
// Suposición: ItemType debe ejecutar LessThan y Equal.
//****************************************************************************

#include "Entry.h"

template<class ItemType>
struct NodeType;              // Declaración directa

template<class ItemType>
class SortedList2
{
public:
        // Suposición: ItemType debe poner en práctica LessThan y Equal.
    bool IsEmpty() const;
        // Poscondición:
        //     El valor de retorno es verdadero, si la lista está vacía;
        //     falso, en caso contrario

    bool IsFull() const;
        // Poscondición:
        //     El valor de retorno es verdadero, si la lista está vacía
        //     falso, en caso contrario

    void Insert( /* in */ ItemType item );
        // Precondición:
        //     se asigna el elemento
        // Poscondición:
        //     el elemento está en la lista
        //  && los componentes de la lista están en orden ascendente

    void Delete( /* in */ ItemType item );
        // Precondición:
        //     el elemento está en alguna parte de la lista
        // Poscondición:
        //     La primera aparición del elemento ya no está en la lista
        //  && los componentes de la lista están en orden ascendente

    void Reset();
        // Poscondición:
        //     Se inicializa la iteración
```

```
    ItemType GetNextItem();
        // Precondición:
        //     Ningún transformador ha sido invocado desde la última llamada
        // Poscondición:
        //     Devuelve el componente en la posición actual
        //     de SortedList

    int Length() const;
        // Poscondición:
        //     El valor de retorno es la longitud de la lista

    bool IsPresent(ItemType item) const;

    // Busca en la lista al elemento e informa si fue hallado
    // Poscondición:
    //     El valor de retorno es verdadero, si el elemento está en la lista;
    //     falso, en caso contrario

    SortedList2();
        // Constructor
        // Poscondición:
        //     Se crea la lista vacía

    SortedList2( const SortedList2& otherList );
        // Constructor de copia
        // Poscondición:
        //     Se crea la lista como un duplicado de otherList

    ~SortedList2();
        // Destructor
        // Poscondición:
        //     Se destruye la lista
private:
    NodeType<ItemType>* head;
    NodeType<ItemType>* currentPos;
    int length;
};

#include "SortedList2.cpp"
```

En el capítulo 17 se proporcionaron todos los algoritmos, excepto Reset y GetNextItem. ¿Qué es análogo a fijar el índice a cero en la versión basada en arreglo? Fijar el apuntador a `NULL`. ¿Qué es análogo a incrementar un índice en una versión basada en arreglo? Fijar el apuntador a la siguiente liga.

Reset

Establecer currentPos en NULL

GetNextItem
Salida: valor de función

IF currentPos es NULL   currentPos es el encabezado Establecer el elemento en currentPos->item Establecer currentPos en currentPost-> link

¿Altera el cambio en la precondición el algoritmo `Delete`? Sí, lo altera. La técnica que se usa para buscar el componente para borrar depende del ítem en la lista. Si esta función se aplica y el ítem no está en la lista, resultará un apuntador NULL. Es necesario tener un apuntador de fin para usar en la eliminación, como se hizo en la inserción. También es necesario cambiar el operador relacional a `Equal` porque la lista de entradas está ordenada por tiempo. Se debe asegurar de agregar la suposición de que `ItemType` ejecuta `LessThan` y `Equal` a la documentación del encabezado del archivo de especificación.

Delete(entrada: elemento)

```
Establecer currentPtr en encabezado
Establecer prevPtr en NULL
WHILE currentPtr != NULL AND !(currentPtr -> item.Equal(item))
    Establecer prevPtr en currentPtr
    Establecer currentPtr en currentPtr->link
IF currentPtr!= NULL
    Establecer prevPtr-> link en currentPtr-> link
    Disminuir la longitud
```

El algoritmo dado para Insert también usaba un operador relacional. Es necesario entrar en el algoritmo de inserción, remplazar el operador relacional < con `LessThan` y recodificar el algoritmo. Esto es tan sencillo (¿podemos atrevernos a decir algorítmico?), que saltamos directamente al código.

```
//**************************************************************************
// ARCHIVO DE EJECUCIÓN (SortedList2.cpp)
// Este archivo pone en práctica las funciones miembros de clase de
   SortedList2
// Representación de lista: una lista vinculada de nodos dinámicos con
   plantilla
//**************************************************************************
#include <iostream>
#include <cstddef>     // Para NULL

using namespace std;

template<class ItemType>
struct NodeType
{
    ItemType item;
    NodeType<ItemType>* link;
};

// Miembros de clase privados:
//     NodeType* head; Puntero externo para lista vinculada

//**************************************************************************

template<class ItemType>
SortedList2<ItemType>::SortedList2()

// Constructor

// Poscondición:
//     head == NULL
```

```
{
    head = NULL;
    length = 0;
}

//**************************************************************************

template<class ItemType>
SortedList2<ItemType>::SortedList2
    ( const SortedList2<ItemType>& otherList )

// Constructor de copia

// Poscondición:
//     IF otherList.head == NULL (es decir, la otra lista
//         está vacía)
//     ELSE
//         el encabezado apunta a una nueva lista vinculada que es una copia
//         de la lista vinculada señalada por otherList.head

{
    NodeType<ItemType>* fromPtr;  // Lista original de puntero
    NodeType<ItemType>* toPtr;    // Se construye el puntero hacia
                                  //   una nueva lista

    if (otherList.head == NULL)
    {
        head = NULL;
        return;
    }

    // Copiar el primer nodo

    fromPtr = otherList.head;
    head = new NodeType<ItemType>;
    head->item = fromPtr->item;

    // Copiar los nodos restantes

    toPtr = head;
    fromPtr = fromPtr->link;
    while (fromPtr != NULL)
    {
        toPtr->link = new NodeType<NodeType>;
        toPtr = toPtr->link;
        toPtr->item = fromPtr->item;
        fromPtr = fromPtr->link;
    }
    toPtr->link = NULL;
    length = otherList.length;
    currentPos = otherList.currentPos;
}
```

```
//**************************************************************************

template<class ItemType>
SortedList2<ItemType>::~SortedList2()

// Destructor

// Poscondición:
//     Los nodos de lista vinculada han sido cancelados en el almacenamiento
       libre

{
    NodeType<ItemType>* temp;    // Variable temporal

    while ( !IsEmpty() )
    {
        temp = head;
        head = head->link;
        delete temp;
    }
}
//**************************************************************************

template<class ItemType>
bool SortedList2<ItemType>::IsEmpty() const

// Poscondición:
//     El valor de retorno es verdadero, si head == NULL; falso en caso
       contrario

{
    return (head == NULL);
}

//**************************************************************************

template<class ItemType>
bool SortedList2<ItemType>::IsFull() const

// Poscondición:
//     El valor de retorno es falso

{
    return false;
}

//**************************************************************************

template<class ItemType>
void SortedList2<ItemType>::Insert( /* in */ ItemType item )

// Precondición:
//     Los miembros de elemento de nodos de lista están en orden ascendente
//  && se asigna el elemento
```

```
// Poscondición:
//     El nuevo nodo que contiene al elemento está en su propio
//       lugar en la lista vinculada
//  && los miembros de elemento de nodos de lista están en orden ascendente

{
    NodeType<ItemType>* currentPtr;   // Puntero móvil
    NodeType<ItemType>* prevPtr;      // Puntero para el nodo previo
    NodeType<ItemType>* newNodePtr;   // Puntero para el nuevo nodo

    // Establecer el nodo que se insertará

    newNodePtr = new NodeType<ItemType>;
    newNodePtr->item = item;

    // Hallar el punto de inserción previo
    prevPtr = NULL;
    currentPtr = head;
    while (currentPtr != NULL && currentPtr->item.LessThan(item))
    {
        prevPtr = currentPtr;
        currentPtr = currentPtr->link;
    }

    // Insertar el nuevo nodo

    newNodePtr->link = currentPtr;
    if (prevPtr == NULL)
        head = newNodePtr;
    else
        prevPtr->link = newNodePtr;
    length++;
}

//**************************************************************************

template<class ItemType>
void SortedList2<ItemType>::Delete( /* in */ ItemType item )

// Precondición:
//     elemento == miembro de elemento de algún nodo de lista
//  && los miembros de elemento de nodos de lista están en orden ascendente
// Poscondición:
//     El nodo que contiene la primera aparición del elemento ya no está
//       en la lista vinculada
//  && los miembros de elemento de nodos de lista están en orden ascendente

{
    NodeType<ItemType>* currentPtr;
    NodeType<ItemType>* prevPtr;

    currentPtr = head;
    prevPtr = NULL;
```

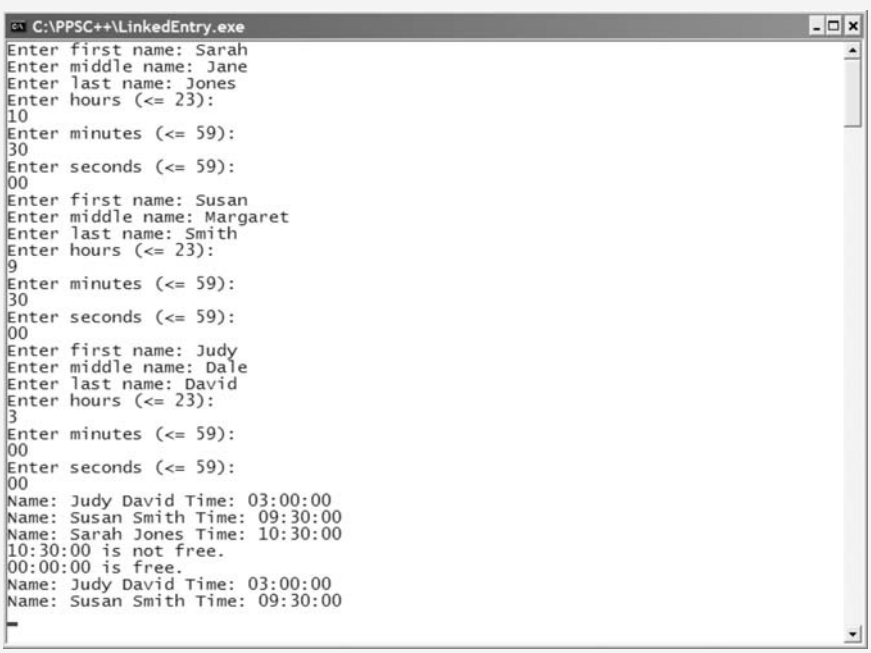
```
    while (currentPtr != NULL && !(currentPtr->item.Equal(item)))
    {
        prevPtr = currentPtr;
        currentPtr = currentPtr->link;
    }

    if (currentPtr != NULL)
    {
        prevPtr->link = currentPtr->link;
        length--;
    }

}

//****************************************************************************

template<class ItemType>
void SortedList2<ItemType>::Reset()

// Poscondición:
//     Se inicializa la iteración
{
    currentPos = NULL;
}

template<class ItemType>
ItemType SortedList2<ItemType>::GetNextItem()

//****************************************************************************
// Precondición:
//     Ningún transformador ha sido invocado desde la última llamada
// Poscondición:
//     Devuelve el elemento en la posición actual en SortedList y restablece
//     actual a la posición siguiente o primera posición si es devuelto
//     el último elemento

{
    ItemType item;
    if (currentPos == NULL)
        currentPos = head;
    item = currentPos->item;
    currentPos = currentPos->link;
    return item;
}

//****************************************************************************
template<class ItemType>
int SortedList2<ItemType>::Length() const

// Poscondición:
//     El valor de retorno es la longitud

{
```

```
    return length;
}

//****************************************************************************
template<class ItemType>
bool SortedList2<ItemType>::IsPresent(ItemType item) const

// Busca en la lista al elemento e informa si fue hallado
// Poscondición:
//     El valor de retorno es verdadero si el elemento está en la lista;
//     falso en caso contrario

{
    NodeType<ItemType>* currentPtr;       // Puntero móvil

    // Buscar el elemento

    currentPtr = head;
    while (currentPtr != NULL && currentPtr->item.LessThan(item))
        currentPtr = currentPtr->link;

    if (currentPtr == NULL)
        return false;
    return currentPtr->item.Equal(item);
}
```

Regrese y observe el controlador para la clase Entry en el capítulo 14. Se han cambiado dos sentencias.

```
#include "SortedList2.h"
SortedList2<Entry> list;
```

Enseguida se presenta la pantalla de entrada/salida. Este ejercicio demuestra el valor de usar clases para encapsular la implementación de una estructura.



La imagen mostrada corresponde a la salida producida por el programa original, escrito en idioma inglés.

Mejorar el calendario de citas

ANÁLISIS Vamos a pensar en formas de hacer que nuestro calendario de citas sea más robusto, agregando excepciones. Por ejemplo, las listas que hemos usado y que permiten duplicados. Cuando hablamos de horarios de citas, esto no tiene sentido. Hay una precondición respeto a la función `InsertEntry` en la clase `Day` que indica que la hora es libre. Sería más seguro quitar esta precondición y lanzar una excepción si la hora no es libre. ¿No podríamos dejar la precondición adentro y revisar de todos modos? No. Una precondición es un contrato que no se revisa, a menos que las consecuencias sean mortales.

Hay ocasiones en las cuales escribimos mensajes en la pantalla cuando un día no ha sido definido. De cada uno de estos casos deberemos hacer una excepción. ¿Qué sucede en caso de que un día se ha duplicado? Sí, también debe ser una excepción. El reconocimiento de una cadena de entrada ilegal también deberá ser una excepción. Por último, es necesario verificar para estar seguros que hay espacio para insertar una entrada antes de realizarla en la clase `Day`. Enseguida presentamos nuestras clases de excepción.

```
class DuplicateTime
{};

class UndefinedDay
{};

class DuplicateDay
{};

class UndefinedString
{};

class EntryListIsFull
{};>
```

¿Dónde se deberán lanzar estos errores? Una hora duplicada y una lista llena se deberán detectar en la función `InsertEntry` de la clase `Day`. Un día indefinido o duplicado se deberá detectar en las funciones apropiadas de la clase `DayList`. La cadena indefinida es detectada en el programa principal. Una lista llena se deberá detectar en la clase `Day`.

¿Dónde se deberán capturar estos errores? El mensaje de error se deberá presentar en la pantalla para que lo pueda ver el usuario. Puesto que todas las funciones tienen acceso a `cout`, las excepciones se podrán capturar en las funciones donde se detectan. Sin embargo, como hemos estudiado en este capítulo, es mejor que una excepción detectada en una operación de bajo nivel en una estructura compleja simplemente reporte que el error ha ocurrido, dejando el control a una autoridad superior. En este caso captamos las excepciones en el programa principal, donde el mensaje apropiado se presenta en la pantalla.

`DayList`: Agregue los controles por días duplicados e indefinidos en la clase `DayList`.
`Day`:
- Controle por lista llena en `InsertEntry` y lance la excepción `EntryListIsFull` si esto es el caso.
- Controle por hora duplicada en `InsertFunction` y lance una excepción `DuplicateTime` si esto es el caso.

`Appointments(main)`:
- Cambie el programa principal para que lance una excepción si una cadena no puede ser decodificada.
- Encierre el ciclo principal entero en una sentencia `try-catch`.
- Inserte las cláusulas *catch*.

Examinemos cada una de éstas en orden. Es necesario capturar tanto los días indefinidos como los días duplicados en la clase `DayList` y lanzar las excepciones apropiadas. Estos casos se presentan en todo, menos los constructores. Cada una de estas funciones usa la función auxiliar `FindDay`. Si el día se encuentra en `InsertDay`, en-

tonces ocurrió una excepción `DuplicateDay`. Si un día *no* se encuentra en `InsertApt`, `DeleteApt`, `IsFree` y `PrintDay`, se lanza una excepción `UndefinedDay`. No mostraremos el código aquí, pero se encuentra en la Web.

Los controles para una lista llena y una hora duplicada son muy evidentes.

```
void Day::InsertEntry( /* in */ Entry newEntry )

// Poscondición:
//     Si la lista está llena, se lanza la excepción EntryListIsFull; si el
//     tiempo ya está lleno, se lanza la excepción DuplicateTime;
//     de otro modo se inserta newEntry en la lista

{
    if (list.IsFull())
        throw EntryListIsFull();
    else
        if (list.IsPresent(newEntry))
            throw DuplicateTime();
        else
          list.Insert(newEntry);
}
```

El lanzamiento de una excepción, cuando una cadena de entrada no es un comando, se puede lograr mediante el lanzamiento de una excepción en la sentencia Switch cuando se devuelve el tipo enumerado `ERROR`. La colocación del ciclo dentro de una sentencia Try-Catch no requiere de explicación. ¿Qué pasa con las cláusulas Catch? Simplemente colocamos la cadena de salida original dentro de las cláusulas Catch para los casos que se han controlado previamente, y se insertan cadenas apropiadas para las nuevas. Enseguida se presenta el programa revisado del calendario de citas. Repase muy cuidadosamente el código de este programa. Este programa contiene varias nuevas construcciones sintácticas.

```
//**************************************************************************
// PROGRAMA CALENDARIO DE CITAS
// Este archivo pone en práctica un calendario de citas. Se lanzan
// las excepciones si se definen días o tiempos duplicados, si una lista
// de entradas está llena al tratar de insertar una entrada y no se puede
// decodificar un código de operación
//**************************************************************************

#include "DayList.h"
#include <fstream>
#include <string>
#include <iostream>
#include <cstddef>     // Para NULL
using namespace std;

enum Operations{INSERT_DAY, INSERT_APT, DELETE_APT, IS_FREE,
                PRINT_DAY, QUIT, ERROR};

// Prototipos de función
Operations Convert(string operation);
void GetDate(Date& date);
void PrintMenu();
```

```
int main()
{
    DayList calendar[12];
    bool notQuit = true;
    Date date;

    string operation;
    ofstream outData;
    ofstream outList;
    outData.open("Appointments");

    while (notQuit)
    {
        try
        {
            PrintMenu();
            cin >> operation;
            switch (Convert(operation))
            {
                 case   INSERT_DAY   :
                     GetDate(date);
                     calendar[date.Month()-1].InsertDay(date);
                     break;
                 case   INSERT_APT :
                     GetDate(date);
                     calendar[date.Month()-1].InsertApt(date);
                     break;
                 case DELETE_APT :
                     GetDate(date);
                     calendar[date.Month()-1].DeleteApt(date);
                     break;
                 case IS_FREE    :
                     GetDate(date);
                     if (calendar[date.Month()-1].IsFree(date))
                         cout << "Time is free" << endl;
                     else
                         cout << "Time is not free" << endl;
                     break;
                 case PRINT_DAY  :
                     GetDate(date);
                     calendar[date.Month()-1].PrintDay(outData,
                                                       date);
                     break;
                 case QUIT   :
                     notQuit = false;
                     break;
                 case ERROR  :
                    throw UndefinedString();
            }
        }
        catch ( UndefinedDay )
        {
            cout << "Se ha definido una operación en un "
                 << "día indefinido" << endl;
```

```
            }

            catch ( DuplicateTime )
            {
                cout <<        "La entrada con el mismo tiempo ya está en la
                               lista"
                     << endl;
            }

            catch ( DuplicateDay )
            {
                cout << "El día con la misma fecha ya se definió."
                     << endl;
            }

            catch ( UndefinedString )
            {
                cout << "La cadena de entrada no fue válida."
                     << endl
                     << "Leer el menú e intentar de nuevo."
                     << endl;
            }

            catch ( EntryListIsFull )
            {
                cout << "La lista de entrada ya está llena para esta fecha" <<
                        endl;
        }
        outData.close();
        outList.close();
        return 0;

    }

    void GetDate(Date& date)
    {
        int month;
        int day;
        int year;
        cout << "Introducir el mes, día y año en ese orden" << endl;
        cin >> month >> day >> year;
        date.Set(month, day, year);
    }

    void PrintMenu()
    {
        cout << endl;
        cout << "Opciones de menú" << endl;
        cout << "      InsertDay establece un nuevo día " << endl;
        cout << "      InsertApt inserta una nueva entrada" << endl;
        cout << "      DeleteApt borra una entrada " << endl;
        cout << "      IsFree comprueba si un tiempo está libre" << endl;
        cout << "      PrintDay imprime todas las citas "
```

```
         << " para un día" << endl;
    cout << "      Quit termina el proceso" << endl;
    cout << "Teclear la cadena apropiada " << endl << endl;
}

Operations Convert(string operation)
{
    if (operation == "InsertDay")
        return INSERT_DAY  ;
    else if (operation == "InsertApt")
        return INSERT_APT;
    else if (operation == "DeleteApt")
        return DELETE_APT;
    else if (operation == "IsFree")
        return IS_FREE;
    else if (operation == "PrintDay")
        return PRINT_DAY;
    else if (operation == "Quit")
        return QUIT;
    else return ERROR;
}
```

PRUEBA Es posible usar el mismo plan de prueba que se ha usado en el capítulo 16, pero es necesario tratar de insertar una entrada duplicada e insertar en un horario lleno. ¿Se debe usar el nuevo TDA ligado y con plantilla SortedList o se debe regresar al basado en un arreglo? En teoría no importa, pero en la práctica sí. Se ha definido `NodeType` en `DayList` y en la lista ordenada ligada; por tanto, queda definido dos veces. Aunque haya maneras de resolver este problema, vamos a dejar esto a los ejercicios de Seguimiento de caso práctico, usando la versión basada en arreglo.

A continuación se muestra la salida de la corrida que verifica las nuevas características.

```
Opciones de menú
      InsertDay establece un nuevo día
      InsertApt inserta una nueva entrada
      DeleteApt borra una entrada
      IsFree comprueba si un tiempo está libre
      PrintDay imprime todas las citas para un día
      Quit termina el proceso
Teclear la cadena apropiada

InsertDay
Introducir el mes, día y año en ese orden
1 3 2004
¿Cuántas citas se pueden hacer en esta fecha?
2

...

InsertApt
Introducir el mes, día y año en ese orden
1 3 2004
Introducir el nombre: Susy
Introducir el segundo nombre: Sunshine
Introducir el apellido: Smith
```

```
Introduir las horas (<= 23):
10
Introducir los minutos (<= 59):
30

...
InsertApt
Introducir el mes, día y año en ese orden
1 3 2004
Introducir el nombre: Clara
Introducir el segundo nombre: Cloudy
Introducir el apellido: Clear
Introducir las horas (<= 23):
10
Introducir los minutos (<= 59):
30
La entrada con el mismo tiempo ya está en la lista

...

InsertApt
Introducir el mes, día y año en ese orden
1 3 2004
Introducir el nombre: Clara
Introducir el segundo nombre: Cloudy
Introducir el apellido: Clear
Introducir las horas (<= 23):
11
Introducir los minutos (<= 59):
30

...

InsertApt
Introducir el mes, día y año en ese orden
1 3 2004
Introducir el nombre: Nora
Introducir el segundo nombre: Night
Introducir el apellido: Knight
Introducir las horas (<= 23):
12
Introducir los minutos (<= 59):
30
La lista de entrada está llena para esta fecha

...

Quit
```

Prueba y depuración

Cuando trabajamos con plantillas es importante recordar que una plantilla de función generada o una plantilla de clase se convierten en una función o clase ordinaria y está sujeto a todas las reglas usuales de sintaxis y semántica.

Para verificar una plantilla de función o una plantilla de clase, deberá empezar con una versión de *no plantilla*, usando un tipo de datos específico, como `int` o algún tipo que sea el más apropiado para sus necesidades. Aplique las estrategias de prueba usuales que hemos indicado en capítulos previos para algoritmos y clases. Después que se hayan identificado y corregido todos los errores, convierta la función o clase en una forma de plantilla y siga verificando por medio de diferentes tipos de datos como argumentos de plantilla.

Un programa que maneja excepciones debe ser verificado para asegurar que las excepciones son generadas de manera apropiada y que después se manipulen de modo correcto. Se deberán incluir casos de prueba que causen que se lancen excepciones y para especificar los resultados esperados de su manejo.

Sugerencias para la prueba y depuración

1. Cuando declara un parámetro de plantilla, por ejemplo en

   ```
   template<class AType>
   ```

   recuerde que debe usar la palabra `class` (o `typename`). Sin embargo, el argumento de plantilla que se usa cuando se genera una plantilla no tiene que ser el nombre de una clase de C++. Se permite cualquier tipo de datos, ya sea integrado o definido por el usuario.
2. En la forma restringida que se analiza en este libro, los argumentos de plantilla son *nombres de tipos de datos*, y no nombres de variables o expresiones.
3. Las plantillas de función se llaman usualmente por medio de argumentos de plantilla, pero las plantillas de clase *deben* usar argumentos de plantilla explícitos.
4. Igual que en el caso de las plantillas de no función, si las definiciones de plantillas de función se colocan físicamente después del código que llama las funciones, entonces los prototipos de funciones (declaraciones directas) deben preceder el código de llamada.
5. En el caso de las plantillas de clase, las funciones de miembro *tienen* que ser compiladas junto con el código del cliente, y no de manera independiente. Una estrategia es agrupar la función de plantilla de clase y las definiciones de miembro de funciones en un solo archivo (el archivo `.h`). Otra estrategia es colocar la plantilla de clase en un archivo `.h`, y las definiciones de miembro de función en un archivo `.cpp`, donde el archivo `.h` indica que, por medio de `#include` se debe incluir el archivo `.cpp`. En este último planteamiento, si usted usa un IDE (ambiente de desarrollo integrado) *no* deberá listar este archivo `.cpp` en su "proyecto". Incluya *sólo* el archivo del código del cliente en el proyecto.
6. Es buena idea organizar de modo que la función `main` capture todas las excepciones, aun si otras funciones realizan el manejo parcial de excepciones. De esta manera, la función `main` puede decidir si detiene el programa o si continúa su ejecución.
7. Evite el uso de tipos integrados como tipos de excepción. Lance objetos de una clase o struct definidos por el usuario, cuyo nombre sugiere la naturaleza del error o suceso excepcional.
8. Si `BadData` es el nombre de una clase de excepción, asegúrese de lanzar un objeto de esta clase, escribiendo

   ```
   throw BadData();    // Throw a constructed object
   ```

   en lugar de

   ```
   throw BadData;      // Syntax error
   ```

9. Asegúrese de que se capturen todas las excepciones. Una excepción no capturada resultará en la detención del programa con un mensaje de error vago.

Resumen

Las plantillas C++ son un mecanismo poderoso y conveniente para implementar algoritmos genéricos y tipos de datos genéricos. Por medio de una plantilla de función es posible definir una familia completa de funciones que son iguales, con excepción de los tipos de datos que manipulan. Por medio de una plantilla de clase es posible definir una familia completa de tipos de clase que sólo se distinguen en los tipos de datos de sus representaciones internas. Las plantillas de clase nos permiten definir estructuras (por ejemplo, listas de `int`, de `float` y de `string`, todas en el mismo programa).

Una plantilla se crea colocando el argumento de plantilla entre paréntesis angulares junto al nombre de la función o clase. En los ejemplos que se proporcionan en este capítulo, un argumento de plantilla es el nombre de un *tipo de datos*, no el nombre de una variable o expresión. El compilador genera enseguida una nueva función o clase, sustituyendo el argumento de plantilla por el parámetro de plantilla en todas partes en que el parámetro aparezca en la plantilla. En el caso de las plantillas de clase, el argumento de plantilla debe ser explícito para que el compilador pueda generar la plantilla. En el caso de las plantillas de función, el argumento de plantilla es normalmente omitido, y el compilador deduce el argumento de plantilla mediante el análisis de la lista de argumentos de la función.

Una excepción es un suceso inusual, con frecuencia impredecible, que requiere un procesamiento especial. El propósito principal del manejo de excepciones es permitir que una parte de un programa reporte un error a otra parte del programa si el error no se puede controlar de manera local. C++ soporta el manejo de excepciones por medio de la sentencia `throw` y la sentencia `try-catch`. Una sección de código que detecta un error usa la sentencia `throw` para lanzar una excepción, lo que en C++ es un objeto de algún tipo de datos; a menudo una clase o *struct*. Se dice que la excepción lanzada fue capturada si la sentencia `try-catch` tiene una cláusula *catch* (manejador de excepción) con un parámetro formal cuyo tipo se apareja al tipo de la excepción. Si no existe ningún manejador de excepción correspondiente, la función que encierra regresa inmediatamente y pasa la excepción a su función invocadora. Este proceso continúa hacia arriba por la cadena de llamada hasta que se encuentre un manejador de excepción correspondiente o el control llegue a la función `main`. Si `main` no captura la excepción, entonces el programa se detiene con un mensaje de error vago.

Comprobación rápida

1. ¿Para la escritura de qué tipo de algoritmo nos habilita una plantilla? (pp. 748-750)
2. ¿Qué es una excepción? (pp. 763-764)
3. Escriba una plantilla de función para una función de devolución de valores `Thrice` que recibe un parámetro de algún tipo simple y devuelve este valor tres veces. (pp. 750-751)
4. Escriba un código de llamada que llame `Thrice` dos veces, una vez mediante un argumento de entero y una vez mediante un argumento de punto flotante. (pp. 751-752)
5. Escriba una especialización definida por el usuario de la plantilla `Thrice` que extraiga para un parámetro `char "Can't triple a char value"` y devuelva el carácter `'x'`. (pp. 753-754)
6. En matemática, un par ordenado es un par de números escritos en la forma (*a*, *b*). Reescriba la siguiente declaración `OrdPair` como una plantilla de clase, de modo que los clientes puedan manipular pares ordenados de cualquier tipo simple, y no sólo pares ordenados de `int`. (pp. 756-758)

```
class OrdPair
{
public:
    int  First() const;       // Devuelve el primer componente
                              //   del par
    int  Second() const;      // Devuelve el segundo componente
                              //   del par
    void Print() const;       // Produce el par
    OrdPair( int m, int n ); // Constructor. Crea el
                              //   par ordenado (m, n)
```

```
private:
    int first;
    int second;
};
```

7. Dada la plantilla de clase `OrdPair`, escriba el código de cliente que declare tres objetos de clase denominados `pair1`, `pair2` y `pair3` que representen los pares ordenados (5, 6), (2.95, 6.34) y `('+', '#')`, respectivamente, e imprima los pares ordenados representados por `pair1`, `pair2` y `pair3`. (pp. 758-759)
8. Dada la plantilla de clase `OrdPair`, escriba definiciones de funciones para las funciones de miembro. (pp. 759-762)
   a) Escriba la declaración de un tipo de excepción definido por el usuario denominado `BadData`.
   b) Escriba una función nula `GetAge` que pide e ingresa la edad del usuario (tipo `int`) desde el teclado. La función devuelve la edad del usuario por medio de la lista de parámetros, a menos que el valor de entrada sea un número negativo; en este caso se lanza una excepción `BadData`. (pp. 763-765)
10. `Escriba` una sentencia `try-catch` que llame la función `GetAge` de la pregunta 7. Si se lanza una excepción `BadData`, imprima un mensaje de error y relance la excepción a un invocador; de no ser así, la ejecución deberá continuar. (pp. 765-770)

Respuestas

1. Algoritmos genéricos. **2.** Un suceso inusual, a menudo impredecible, que requiere de procesamiento especial.
3.
```
template<class SimpleType>
SimpleType Thrice( SimpleType val )
{
    return 3*val;
}
```
4.
```
cout << Thrice(75) << ' ' << Thrice(64.35);
```
   o
```
cout << Thrice<int>(75) << ' ' << Thrice<float>(64.35);
```
5.
```
template<>
char Thrice( char val )
{
    cout << "No se puede triplicar un valor char" << endl;
    return 'x';
}
```
6.
```
template<class SomeType>
class OrdPair
{
public:
    SomeType First() const;
    SomeType Second() const;
    void Print() const;
    OrdPair( SomeType m, SomeType n );
private:
    SomeType first;
 SomeType second;
};
```
7.
```
OrdPair<int>   pair1(5, 6);
OrdPair<float> pair2(2.95, 6.34);
OrdPair<char>  pair3('+', '#');

pair1.Print();
pair2.Print();
pair3.Print();
```

8.
```
template<class SomeType>
OrdPair<SomeType>::OrdPair( SomeType m, SomeType n )
// Constructor
{
    first = m;
    second = n;
}

template<class SomeType>
SomeType OrdPair<SomeType>::First() const
{
    return first;
}

template<class SomeType>
SomeType OrdPair<SomeType>::Second() const
{
    return second;
}

template<class SomeType>
void OrdPair<SomeType>::Print() const
{
    cout << '(' << first << ", " << second << ')' << endl;
}
```

9. ***a)***
```
class BadData
{};           // No olvide el punto y coma
```
b)
```
void GetAge( int& age )
{
    cout << "Enter your age: ";
    cin >> age;
    if (age < 0)
        throw BadData();
}
```

10.
```
try
{
    GetAge(age);
}
catch ( BadData )
{
    cout << "La edad no debe ser negativa." << endl;
    throw;
}
```

Ejercicios de preparación para examen

1. En C++ dos funciones (sin plantilla) pueden tener el mismo nombre. ¿Verdadero o falso?
2. En C++ dos clases (sin plantilla) pueden tener el mismo nombre. ¿Verdadero o falso?
3. Lo que sigue, ¿es una plantilla de función, una función de plantilla o ninguna de las dos?

```
template<class T>
T OneLess( T var )
{
    return var-1;
}
```

4. En la sentencia

```
Group<char> oneGroup;
```

a) `Group<char>`, ¿es una pantilla de clase, una clase de plantilla o ninguna de las dos?
b) `oneGroup`, ¿es una plantilla de clase, una clase de plantilla o ninguna de las dos?

5. Defina los siguientes términos:
generar (una plantilla) especialización (de una plantilla)
parámetro de plantilla especialización definida por el usuario (de una plantilla)
argumento de plantilla
6. Considere la siguiente llamada de función:

```
DoThis<float>(3.85)
```

a) ¿Qué ítem es el argumento de plantilla?
b) ¿Qué ítem es el argumento de función?
c) ¿Se podría escribir como `DoThis(3.85)`?

7. ¿Cuál es la estructura de control de C++ que se debe usar si usted piensa que una operación podría lanzar una excepción?
8. ¿Cuál es la sentencia de C++ que levanta una excepción?
9. ¿Qué parte de la sentencia `try-catch` se debe escribir mediante un parámetro formal?
10. Marque si las siguientes sentencias son verdaderas o falsas. Si considera que una sentencia es falsa, explique por qué.
a) Sólo puede haber una cláusula `catch` para cada sentencia `try-catch`.
b) Una cláusula *catch* es un manejador de excepción.
c) Una sentencia `throw` debe estar ubicada dentro de una cláusula try.
11. En la función `WriteToFile` de la sección 17.3,
a) ¿Qué pasaría si borráramos la sentencia `throw` en el manejador de excepción?
b) ¿Qué pasaría si cambiáramos `throw` a `return`?
12. Considere el programa de división entre cero al final de la sección 17.3. ¿Cómo cambiaría usted la función `main` de modo que el programa se detenga si un denominador resulta 0?
13. ¿Qué sucede si usted llama una función que lanza una excepción, sin colocar la llamada en una *cláusula try*?
14. ¿Qué hará la siguiente sentencia si se escribe dentro de un manejador de excepción?

```
throw;
```

15. ¿Qué tipo de excepción se escribiría como el parámetro de una *cláusula catch* si la cláusula try está llamando `new` y si quisiéramos revisar si el área de almacenamiento libre se quedó sin espacio?

Ejercicios de calentamiento para programación

1. Escriba una plantilla de función para una función `void` que eleve al cuadrado y extraiga el valor que se pasó a su parámetro. El tipo de parámetro es del tipo de plantilla.
2. Escriba sentencias que llamen la función del ejercicio 1 para valores del tipo `int`, `long` y `float`.
3. Escriba una especialización de usuario de la función en el ejercicio 1 que acepte un argumento de cadena y lo "cuadre", imprimiéndolo dos veces sin espacios en medio.
4. Quisiéramos tener una función de devolución de valor que devuelva dos veces el valor de su parámetro entrante.
a) Usando la sobrecarga de función, escriba dos definiciones de función en C++ para una de estas funciones, una con un parámetro `int`, y otra con un parámetro `float`.
b) Escriba un código de llamada que haga llamadas a ambas funciones.
5. Escriba una plantilla de función para una función que devuelva la suma de todos los elementos en un arreglo unidimensional. Los elementos del arreglo pueden ser de cualquier tipo numérico simple, y la función tiene dos parámetros: la dirección base del arreglo y el número de elementos en el arreglo.

6. Escriba una plantilla de función para una función nula, `GetData`, que reciba una cadena mediante la lista de parámetros, que use esta cadena para pedir que el usuario introduzca, que lea un valor desde el teclado y que devuelva este valor (como parámetro de referencia) por medio de la lista de parámetros. El tipo de datos del valor de entrada puede ser cualquier tipo para el cual esté definido el operador >>.
7. Suponga que tiene un tipo de enumeración

```
enum AutoType {SEDAN, COUPE, OTHER};
```

   Escriba una especialización definida por el usuario de la plantilla `GetData` (véase el ejercicio 6) que acomode el tipo `AutoType`. Como entrada, el usuario deberá teclear el carácter `'s'` para un sedán, `'c'` para coupé y `'o'` (o todo lo demás) para el resto.
8. Considere la plantilla de clase `GList` de la sección 17.2.
   a) Escriba un código de cliente para generar la plantilla dos veces, creando listas de `int` y `float`.
   b) Suponga que en alguna parte del código del cliente la lista de `int` ya contiene valores de los cuales se sabe que se encuentran en el intervalo entre 10 y 80, y que la lista de `float` está vacía. Escriba un código de cliente que vacíe la lista de `ints` como sigue: a la vez que cada ítem es removido de la lista de `int`, multiplíquelo por 0.5 e inserte el resultado en la lista de `float`.
9. Escriba una plantilla de clase `MixedPair` similar a la plantilla `OrdPair` de la pregunta 6 de la comprobación rápida, excepto que el par de elementos puede ser de dos diferentes tipos de datos. *Consejo*: Empiece la plantilla con

```
template<class Type1, class Type2>
```

10. Dada la plantilla de clase `MixedPair` del ejercicio 9, escriba un código de cliente que establezca dos objetos de clase representando los pares (5, 29.48) y ("Book", 36).
11. Dada la plantilla de clase `MixedPair` del ejercicio 9, escriba definiciones de funciones para las funciones de miembro de clase.
12. *a*) Declare una excepción definida por el usuario denominada `MathError`.
    b) Escriba una sentencia que lanza una excepción del tipo `MathError`.
13. Escriba una sentencia `try-catch` —que supuestamente se encuentra en la función `main`— que intente concatenar dos objetos `string` e imprima un mensaje de error y detenga el programa si se lanza una excepción. Tal vez usted quiera consultar de nuevo la sección 17.3 para ver qué excepciones lanza la clase `string`.
14. Escriba una función `Sum` que devuelva la suma de sus dos parámetros no negativos `int`, a menos que la suma exceda `INT_MAX`. En este caso lance una excepción. (*Atención*: No puede revisar si existe desbordamiento *después* de sumar los números. En la mayoría de las máquinas, el desbordamiento de enteros resulta en un cambio de signo, pero no se deberá escribir un código que dependa de este hecho.)
15. Escriba una sentencia `try-catch` que llame la función `Sum` del ejercicio 14 y, si se lanza una excepción, imprima un mensaje de error y relance la excepción a un invocador.

Problemas de programación

1. El problema de programación 1 en el capítulo 9 le pide que extraiga la palabra ICAO correspondiente a cada letra en una palabra. Cambie la función que convierte una letra a la palabra correspondiente de modo que lance una excepción si la letra no está en el alfabeto, y agregue un manejador de excepción al programa que extraiga "Invalid Letter" y la letra ofensiva cada vez que la excepción es lanzada.
2. El problema de programación 2 en el capítulo 9 le había pedido que calcule el peso de una persona en un planeta específico. Reescriba ese programa de modo que un nombre de planeta erróneo resulte en el lanzamiento de una excepción por parte de la rutina de entrada, y que es capturada por el invocador.

3. El problema de programación 2 en el capítulo 12 le había pedido que escribiera un programa de base de datos para una galería de arte. El problema requería que un usuario pudiera buscar la base de datos usando cualquiera de los valores en un archivo para una obra de arte. Reescriba el programa usando una plantilla de función de prueba de igualdad, de manera que sea posible usar una sola función de búsqueda para los diferentes tipos de valores en la base de datos.
4. Usted está trabajando para un observatorio que mantiene una base de datos para estrellas. Las estrellas más destacadas están indexadas por nombre (una cadena), las estrellas menos destacadas están indexadas por el nombre de la constelación donde se localizan, así como un número de asignación secuencial (un `int`), y las estrellas de importancia mínima están indexadas por coordenadas que consisten en dos números de punto flotante. Cada archivo de estrella también contiene una medición del brillo de la estrella que señala su magnitud (un `float`) y una letra que indica su color (un `char`).

   Escriba un programa en C++ que lea los datos de estrella desde tres archivos (`named.dat`, `numbered.dat` y `coord.dat`) y que las coloque a todas en una sola lista. El programa deberá permitir que el usuario recupere los datos para cualquier estrella usando los diferentes tipos de listas de los índices. Use plantillas para simplificar la codificación de esta aplicación.
5. El problema de programación 5 en el capítulo 9 le pidió que escribiera un programa para introducir una hora en forma numérica e imprimirlo en inglés. Reescriba el programa para revisar si existe una introducción de horas erróneas en la función de entrada, y lance una excepción controlada por el invocador.
6. Desarrolle una clase de C++ para una fecha que incluya funciones para construir una fecha, imprimir una fecha y comparar dos fechas en cuanto a igualdad. La función que construya una fecha deberá revisar diferentes errores (mes inválido, día inválido, combinación de mes/día inválida; por ejemplo, 31 de abril y 29 de febrero en un año no bisiesto). Para cada tipo de error de fecha, la función deberá lanzar una excepción diferente al invocador. Escriba un programa de controlador que lea dos fechas, las imprima e indique si son iguales. El controlador deberá capturar todas las excepciones y extraer mensajes de error apropiados conforme sean necesarios.

Seguimiento de caso práctico

1. ¿Por qué debería existir un problema si NodeType estuviera definido en dos lugares?
2. ¿Cómo podría usted resolver el problema de identificadores duplicados?
3. Escriba un plan de prueba para la versión mejorada del calendario de citas.
4. En el calendario de citas, las clases `Entry` y `Time` tienen las operaciones `Equal` y `LessThan`. Las clases `Name` y `Date` tienen la operación `ComparedTo`. Estas clases deberán ser consistentes en su uso de funciones de comparación. Reescriba `Entry` y `Time` para usar una función `ComparedTo`.
5. Ejecute el programa del controlador de Entry del capítulo usando las clases modificadas en el ejercicio 4.

CAPÍTULO

18

Recursión

Objetivos

Objetivos de conocimiento

- *Entender la distinción entre el (los) caso(s) base y el caso general en una definición recursiva.*

Objetivos de habilidades

Ser capaz de:

- *Escribir un algoritmo recursivo para un problema que involucra únicamente variables simples.*
- *Escribir un algoritmo recursivo para un problema que involucra variables estructuradas.*
- *Escribir un algoritmo recursivo para un problema que involucra listas ligadas.*

Llamada recursiva Una llamada de función en la que la función que se llama es la misma que la función que hace la llamada.

En C++ toda función puede llamar a otra función. ¡Una función puede incluso llamarse a sí misma! Cuando una función se llama a sí misma está haciendo una **llamada recursiva**. La palabra *recursivo* significa "tener la característica de volverse a presentar o repetirse". En este caso una llamada de función está siendo repetida por la propia función. La recursión es una técnica poderosa que se puede usar en lugar de la iteración (ciclo).

Por lo general, las soluciones recursivas son menos eficientes que las soluciones iterativas para el mismo problema. Sin embargo, algunos problemas se prestan para soluciones recursivas simples y elegantes y son sumamente difíciles de resolver en forma iterativa. Algunos lenguajes de programación, tales como las versiones tempranas de FORTRAN, BASIC y COBOL, no permiten la recursión. Otros lenguajes están especialmente orientados a algoritmos recursivos; LISP es uno de ellos. C++ nos permite elegir: podemos implementar algoritmos tanto iterativos como recursivos.

Nuestros ejemplos se dividen en dos grupos: problemas que sólo usan variables simples y problemas que usan variables estructuradas. Si usted está estudiando la recursión antes de leer los capítulos 11 a 16 sobre tipos de datos estructurados, entonces lea sólo el primer conjunto de ejemplos y deje el resto hasta que haya completado los capítulos sobre tipos de datos estructurados.

18.1 ¿Qué es recursión?

Probablemente usted ha visto un conjunto de muñecas rusas, alegremente pintadas que caben una dentro de la otra. Dentro de la primera muñeca está una muñeca más pequeña, dentro de la cual se encuentra una muñeca todavía más pequeña, etc. Un algoritmo recursivo es como uno de estos conjuntos de muñecas rusas. Se reproduce a sí mismo con ejemplos cada vez más pequeños de sí mismo hasta que se haya encontrado una solución: es decir hasta que ya no hay más muñecas. El algoritmo recursivo es implementado mediante el uso de una función que hace llamadas recursivas a sí misma.

En el capítulo 8 escribimos una función llamada `Power` que calcula el resultado de elevar un entero a una potencia positiva. Si X es un entero y N es un entero positivo, entonces la fórmula para X^N es la siguiente:

$$x^N = \underbrace{X \times X \times X \times X \times \ldots \times X}_{N \text{ times}}$$

También podríamos escribir esta fórmula como sigue:

$$x^N = X \times \underbrace{(X \times X \times \quad \times X)}_{(N\text{-}1) \text{ times}}$$

o incluso así:

$$x^N = X \times X \times \underbrace{(X \times X \times \quad \times X)}_{(N\text{-}2) \text{ times}}$$

De hecho podemos escribir la fórmula de la manera más concisa así:

$$X^N = X \times X^{N-1}$$

Definición recursiva Una definición en la que algo está definido en términos de una versión más pequeña de sí mismo.

La definición de X^N es una clásica **definición recursiva**, es decir una definición dada en términos de una versión más pequeña de sí misma.

X^N está definido en términos de multiplicar X por X^{N-1}. ¿Cómo está definido X^{N-1}? ¡Claro, como $X \times X^{N-2}$, desde luego! Y X^{N-2}

es $X \times X^{N-3}$; X^{N-3} es $X \times X^{N-4}$, etc. En este ejemplo, "en términos de versiones más pequeñas de sí mismo" significa que el exponente es disminuido cada vez.

¿Cuándo se detiene este proceso? Cuando hemos llegado a un caso para el cual conocemos la respuesta sin recurrir a una definición recursiva. En este ejemplo, esto es el caso donde N iguala a 1: X^1 es X. El caso (o los casos) para el cual (los cuales) una respuesta es explícitamente conocida se llama el **caso base**. El caso para cada solución se expresa en términos de versiones más pequeñas de las mismas llamadas **recursivas** o **caso general**. Un **algoritmo recursivo** es un algoritmo que expresa la solución en términos de una llamada a sí misma, o sea una llamada recursiva. Un algoritmo recursivo debe terminar; esto quiere decir que debe tener un caso base.

Caso base El caso para el cual la solución se puede declarar en forma no recursiva.

Caso general El caso para el cual la solución es expresada en términos de una versión más pequeña de sí mismo.

Algoritmo recursivo Una solución que es expresada en términos de *a*) instancias más pequeñas de sí misma, y *b*) un caso base.

La figura 18-1 muestra una versión recursiva de la función `Power` donde el caso base y la llamada recursiva están marcados. La función está incrustada en un programa que lee un número y un exponente e imprime el resultado.

Figura 18-1 *La función* `Power`

```
//*********************************************************************
// Programa Exponenciación
//*********************************************************************
#include <iostream>

using namespace std;

int Power( int, int );

int main()
{
    int number;              // Número que es elevado a una potencia
    int exponent;            // Elevar el número a una potencia

    cin >> number >> exponent;
    cout << Power(number, exponent); <------- // Llamada no recursiva
    return 0;
}

//*********************************************************************

int Power( /* in */ int x,    // Número que es elevado a una potencia
           /* in */ int n )   // Elevar el número a una potencia

// Calcula x a la potencia n al multiplicar x por el resultado de
// calcular x a la potencia n - 1

// Precondición:
//     Se asigna x && n > 0
// Poscondición:
//     Valor de función == x elevada a la potencia n
// Nota:
//     Los exponentes grandes podrían dar como resultado desbordamiento de
//     enteros
```

```
{
    if (n == 1)
        return x; <-------------------------------------------- // Caso base
    else
        return x * Power(x, n - 1); <-------------- // Llamada recursiva
}
```

Cada llamada recursiva a `Power` se podrá considerar como creando una copia completamente nueva de la función, cada una con sus propias copias de los parámetros `x` y `n`. El valor de `x` queda igual para cada versión de `Power`, pero el valor de `n` disminuye en 1 para cada llamada hasta que llega a 1.

Vamos a trazar la ejecución de esta función recursiva, con `number` igual a 2 y `exponent` igual a 3. Usamos un nuevo formato para trazar rutinas recursivas: Numeramos las llamadas y luego discutimos lo que pasa en forma de párrafos.

Llamada 1: `Power` es llamado por `main`, con `number` igual a 2 y `exponent` igual a 3. Dentro de `Power` los parámetros `x` y `n` son inicializados a 2 y 3, respectivamente. Puesto que `n` no es igual a 1, `Power` es llamado en forma recursiva mediante `x` y `n` − 1 como argumentos. La ejecución de Llamada 1 se detiene hasta que se envíe una respuesta de regreso desde esta llamada recursiva.

Llamada 2: `x` es igual a 2 y `n` es igual a 2. Puesto que `n` no es igual a 1, la función `Power` es llamada nuevamente, esta vez con `x` y `n` − 1 como argumentos. La ejecución de Llamada 2 se detiene hasta que se envíe una respuesta de regreso desde esta llamada recursiva.

Llamada 3: `x` es igual a 2 y `n` es igual a 1. Puesto que `n` es igual a 1, el valor de `x` debe ser devuelto. Esta llamada a la función ha terminado su ejecución, y el valor de devolución de la función (que es 2) es pasado de vuelta al lugar en la declaración de donde se había hecho la llamada.

Llamada 2: Esta llamada a la función puede ahora completar la declaración que contuvo la llamada recursiva porque la llamada recursiva ha regresado. El valor de devolución de Llamada 3 (que es 2) es multiplicado por `x`. Esta llamada a la función ha terminado su ejecución, y el valor de devolución de la función (que es 4) es pasado de vuelta al lugar en la declaración de donde se había hecho la llamada.

Llamada 1: Esta llamada a la función puede ahora completar la declaración que contuvo la llamada recursiva porque la llamada recursiva ha regresado. El valor de devolución de Llamada 2 (que es 4) es multiplicado por `x`. Esta llamada a la función ha terminado su ejecución, y el valor de devolución de la función (que es 8) es pasado de vuelta al lugar en la declaración de donde se había hecho la llamada. Puesto que la primera llamada (la llamada no recursiva en `main`) se ha completado ahora; esto es el valor final de la función `Power`.

Este seguimiento está resumido en la figura 18-2. Cada casillero representa una llamada a la función `Power`. Los valores para los parámetros para esta llamada se muestran en cada casillero.

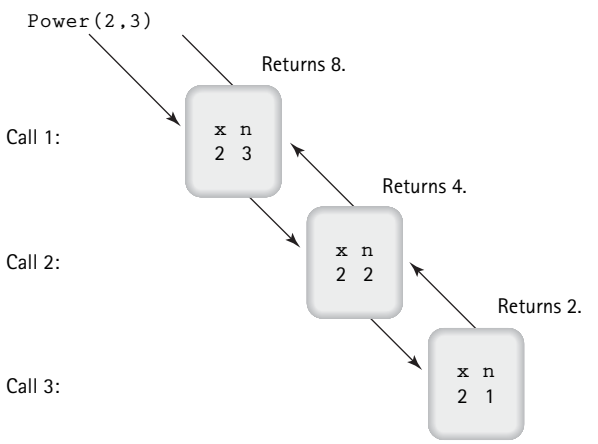


Figura 18-2 *Ejecución de* `Power(2, 3)`

¿Qué pasa si no hay ningún caso base? Entonces tenemos una **recursión infinita**, el equivalente recursivo de un ciclo infinito. Por ejemplo, si la condición

Recursión infinita La situación donde una función se llama a sí misma una y otra vez sin fin.

```
if (n == 1)
```

fuese omitida, se le llamaría a `Power` una y otra vez. La recursión infinita también ocurre si `Power` es llamado con un `n` menor o igual a 0.

En realidad, las llamadas recursivas no pueden seguir para siempre. La razón es ésta: Cuando se llama a una función, ya sea en forma recursiva o no recursiva, el sistema de cómputo crea un almacenamiento temporal para los parámetros y las variables locales (automáticos) de la función. Este almacenamiento temporal es una región de memoria que se llama *pila en tiempo de ejecución*. Cuando la función regresa, sus parámetros y variables locales son liberadas de la pila en tiempo de ejecución. En el caso de la recursión infinita, las llamadas recursivas de función nunca regresan. Cada vez que la función se llama a sí misma, se usa un poco más de la pila en tiempo de ejecución para guardar las nuevas copias de las variables. En este punto el programa se bloquea y muestra un mensaje de error, como por ejemplo "RUN-TIME STACK OVERFLOW" (desbordamiento de pila en tiempo de ejecución), o la computadora simplemente se bloquea.

18.2 Algoritmos recursivos con variables simples

Vamos a revisar otro ejemplo: el cálculo de un factorial. El factorial de un número N (que se escribe $N!$) es multiplicado por $N - 1$, $N - 2$, $N - 3$, etc. Otra forma de expresar un factorial es

$$N! = N \times (N - 1)!$$

Esta expresión se ve como una definición recursiva. $(N - 1)!$ es una instancia menor de $N!$, o sea que se necesita una multiplicación menos para calcular $(N - 1)!$ que para calcular $N!$ Si podemos encontrar un caso base, podemos escribir un algoritmo recursivo. Afortunadamente no tenemos que ver muy lejos: 0! está definido en la matemática como 1.

Factorial(entrada:n)

```
IF n es 0
    Devolver 1
ELSE
    Devolver n * Factorial(n – 1)
```

Este algoritmo se puede codificar en forma directa como sigue:

```
int Factorial ( /* in */ int n )

// Precondición:
//     n >= 0
// Poscondición:
//     Valor de función == n!
// Nota:
//     Los valores grandes de n pueden causar desbordamiento de enteros

{
    if (n == 0)
        return 1;                          // Caso base
    else
        return n * Factorial(n - 1);       // Caso general
}
```

Vamos a trazar esta función con un `n` original de 4.

Llamada 1: `n` es 4. Puesto que `n` no es 0, la bifurcación `else` está ocupada. La declaración Return no puede ser completada hasta que la llamada recursiva a `Factorial` con `n – 1` como argumento ha sido completada.

Llamada 2: `n` es 3. Puesto que `n` no es 0, la bifurcación `else` está ocupada. La declaración Return no puede ser completada hasta que la llamada recursiva a `Factorial` con `n – 1` como argumento ha sido completada.

Llamada 3: `n` es 2. Puesto que `n` no es 0, la bifurcación `else` está ocupada. La declaración Return no puede ser completada hasta que la llamada recursiva a `Factorial` con `n – 1` como argumento ha sido completada.

Llamada 4: `n` es 1. Puesto que `n` no es 0, la bifurcación `else` está ocupada. La declaración Return no puede ser completada hasta que la llamada recursiva a `Factorial` con `n – 1` como argumento ha sido completada.

Llamada 5: `n` es 0. Puesto que `n` es igual a 0, esta llamada a la función regresa, devolviendo 1 como resultado.

Llamada 4: La declaración Return en esta copia puede ahora ser completada. El valor a ser devuelto es `n` (que es 1) por 1. Esta llamada a la función regresa, devolviendo 1 como resultado.

Llamada 3: La declaración Return en esta copia puede ahora ser completada. El valor a ser devuelto es `n` (que es 2) por 1. Esta llamada a la función regresa, devolviendo 2 como resultado.

Llamada 2: La declaración Return en esta copia puede ahora ser completada. El valor a ser devuelto es `n` (que es 3) por 2. Esta llamada a la función regresa, devolviendo 6 como resultado.

Llamada 1: La declaración Return en esta copia puede ahora ser completada. El valor a ser devuelto es `n` (que es 4) por 6. Esta llamada a la función regresa, devolviendo 24 como resultado. Puesto que ésta es la última de las llamadas a `Factorial`, el proceso recursivo ha terminado. El valor 24 es devuelto como valor final de la llamada a `Factorial` con un argumento de 4. La figura 18-3 resume la ejecución de la función `Factorial` con un argumento de 4.

Vamos a organizar lo que hemos hecho en estas dos soluciones en un boceto para escribir algoritmos recursivos.

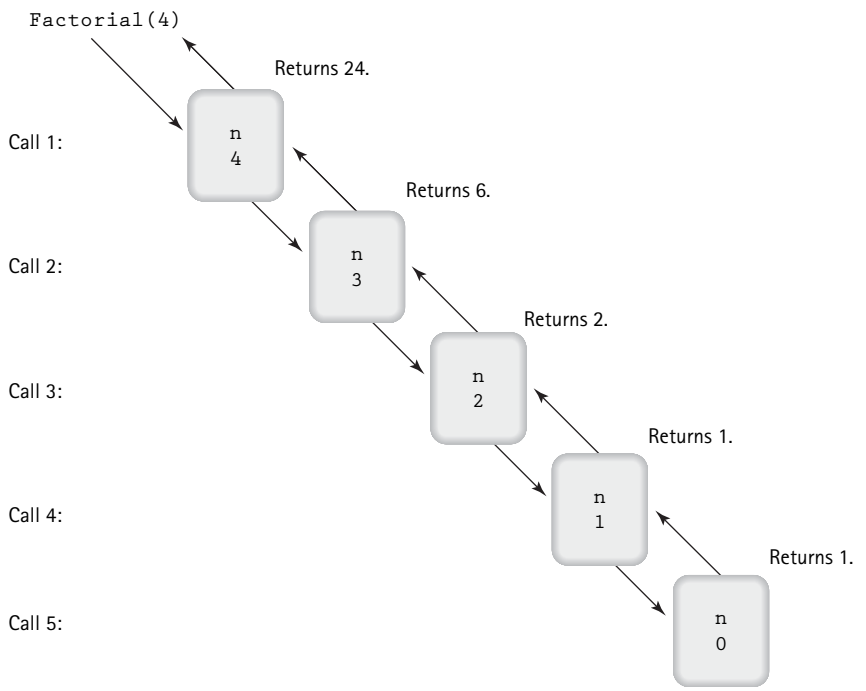


Figura 18–3 *Ejecución de* `Factorial(4)`

1. Entienda el problema. (Esto lo hemos dicho por añadidura; siempre es el primer paso.)
2. Determine el (los) caso(s) base.
3. Determine el (los) caso(s) recursivo(s).

Hemos usado los algoritmos de factorial y power para demostrar la recursión porque son fáciles de visualizar. En la práctica, nunca se quiere calcular una de estas funciones usando la solución recursiva. En ambos casos, las soluciones iterativas son más simples y mucho más eficientes, porque iniciar una nueva iteración de un ciclo es una operación más rápida que llamar una función. Vamos a comparar el código para las versiones iterativa y recursiva de un problema factorial.

Solución iterativa

```
int Factorial ( /* in */ int n )
{
    int factor;
    int count;

    factor = 1;
    for (count = 2; count <= n; count++)
        factor = factor * count;
    return factor;
}
```

Solución recursiva

```
int Factorial ( /* in */ int n )
{
    if (n == 0)
        return 1;
    else
        return n * Factorial(n - 1);
}
```

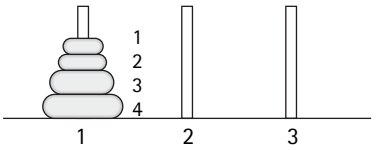
La versión iterativa tiene dos variables locales, mientras que la versión recursiva no tiene ninguna. Por lo regular existen menos variables locales en una rutina recursiva que en una rutina iterativa. La versión iterativa siempre tiene también un solo ciclo, mientras que la versión recursiva siempre tiene una declaración de selección, ya sea un If o un Switch. Una estructura de bifurcación es la principal estructura de control en una rutina recursiva. Una estructura de ciclo es la estructura principal de control en una rutina iterativa.

En la siguiente sección revisaremos un problema más complicado: uno donde la solución recursiva no es inmediatamente aparente.

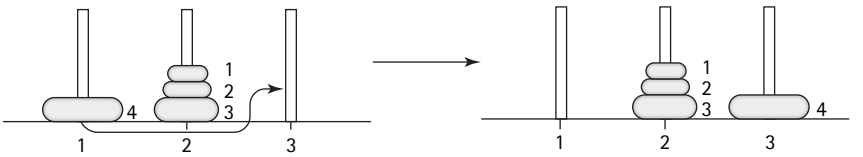
18.3 Las torres de Hanoi

Uno de sus primeros juguetes fueron posiblemente tres palos con círculos de colores de diferentes diámetros. Si esto fue así, usted probablemente pasó horas incontables moviendo los círculos de un palo a otro. Si ponemos algunas restricciones en el movimiento de estos círculos o discos, tendremos un juego de adultos llamado Torres de Hanoi. Al inicio del juego todos los círculos están en el primer palo en el orden de su tamaño y con el más pequeño arriba. El objeto del juego es mover los círculos, uno a la vez, al tercer palo. La trampa es que un círculo no se deberá colocar sobre otro que tiene un diámetro menor. El palo central se puede usar como palo auxiliar, pero debe estar vacío al principio y al final del juego.

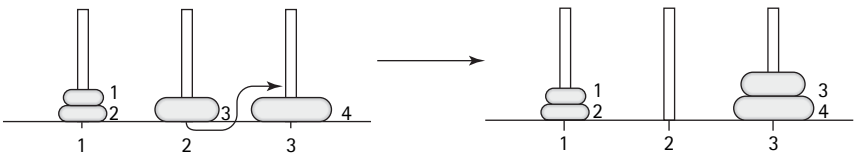
Para entender cómo se puede hacer esto, vamos a ver algunos dibujos de cómo debe ser la configuración en ciertos puntos, si una solución es posible. Usamos cuatro círculos o discos. La configuración inicial es la siguiente:



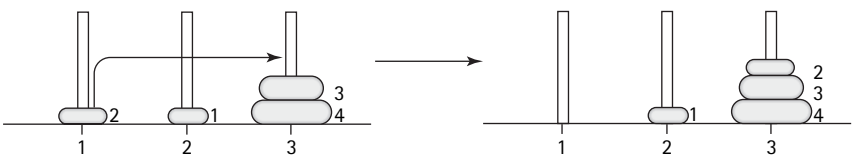
Para mover el círculo más grande (círculo 4) al palo 3 tenemos que mover los tres círculos más pequeños al palo 2. Luego el círculo 4 se puede mover a su lugar final:



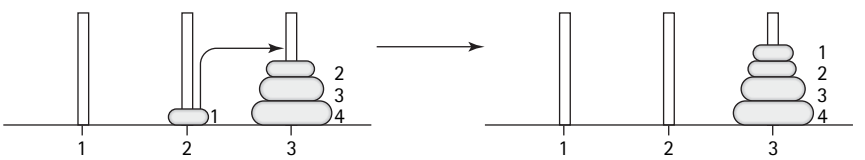
Vamos a suponer que podemos hacer esto. Ahora, para mover el próximo círculo más grande (círculo 3) a su lugar, tenemos que mover los dos círculos arriba de él a otro palo auxiliar (palo 1, en este caso):



A fin de poner el círculo 2 en su lugar, tenemos que mover el círculo 1 a otro palo, liberando al círculo 2 para ser movido a su lugar en el palo 3:



El último círculo (círculo 1) se puede ahora mover a su lugar final, y así hemos terminado:



Observe que para liberar el círculo 4 tuvimos que mover tres círculos a un palo distinto. Para liberar el círculo 3, tuvimos que mover dos círculos a otro palo. Esto suena como un algoritmo recursivo: A fin de liberar el círculo n, tuvimos que mover n – 1 círculos. Cada etapa se puede considerar como un nuevo inicio con tres palos, pero cada vez con un círculo menos. Vamos a ver si podemos resumir este proceso, usando n en lugar de un número real.

GetN Circles Moved from Peg 1 to Peg 3

```
Get n 2 1 circles moved from peg 1 to peg 2
Move nth circle from peg 1 to peg 3
Get n 2 1 circles moved from peg 2 to peg 3
```

Este algoritmo suena bastante simple; seguramente debe haber más. Pero esto en realidad es todo lo que tiene.

Vamos a escribir una función recursiva que ejecute este algoritmo. Claro que no podemos realmente mover discos, pero podemos imprimir un mensaje para que se haga. Observe que el palo inicial, el palo final y el palo auxiliar siguen cambiando durante el algoritmo. Para hacer que el algoritmo sea más fácil de seguir, llamaremos a los palos `beginPeg`, `endPeg` y `auxPeg`. Estos tres palos, junto con el número de círculos en el palo inicial, son los parámetros de la función.

Tenemos ahora el caso recursivo o general, pero ¿qué tal un caso base? ¿Cómo sabemos cuándo parar el proceso recursivo? La pista está en la expresión "Mueve `n` círculos". Si no tenemos ningún círculo que mover, no tenemos nada que hacer. Hemos terminado con esta etapa. Por ende, cuando el número de círculos es igual a 0, no hacemos nada (es decir, simplemente regresamos).

```
void DoTowers(
   /* in */ int circleCount,     // Número de círculos por mover
   /* in */ int beginPeg,        // Palo que contiene los círculos a mover
   /* in */ int auxPeg,          // Palo que contiene los círculos
                                    temporalmente
   /* in */ int endPeg      )    // Palo que recibe los círculos que son
                                    movidos
{
    if (circleCount > 0)
    {
        // Mover n - 1 círculos del palo inicial al palo auxiliar

        DoTowers(circleCount - 1, beginPeg, endPeg, auxPeg);
        cout << "Mover el círculo del palo " << beginPeg
             << " al palo " << endPeg << endl;

        // Mover n - 1 círculos del palo auxiliar al palo final

        DoTowers(circleCount - 1, auxPeg, beginPeg, endPeg);
    }
}
```

Es difícil de creer que un algoritmo tan simple realmente funcione, pero se lo vamos a probar. A continuación sigue un programa manejador que llama a la función `DoTowers`. Declaraciones de salida fueron añadidas para que usted pueda ver los valores de los argumentos en cada llamada recursiva. Puesto que hay dos llamadas recursivas dentro de la función, hemos indicado cuál de las declaraciones recursivas inició la llamada.

```
//*************************************************************************
// Programa TestTowers
// Este programa, un manejador de prueba para la función DoTowers,
// lee un valor de la entrada estándar y lo pasa a DoTowers
//*************************************************************************
#include <iostream>
#include <iomanip>    // Para setw()

using namespace std;

void DoTowers( int, int, int, int );

int main()
{
```

```
    int circleCount;    // Número de círculos en el palo inicial

    cout << "Introducir el número de círculos: ";
    cin >> circleCount;
    cout << "RESULTADO CON " << circleCount << " CÍRCULOS" << endl
         << endl;
    cout << "LLAMADOS DE #CÍRCULOS" << setw(8) << "COMENZAR"
         << setw(8) << "AUXIL." << setw(5) << "TERMINAR"
         << "    INSTRUCCIONES" << endl
         << endl;
    cout << "Original   :";
    DoTowers(circleCount, 1, 2, 3);
    return 0;
}

//*************************************************************************

void DoTowers(
   /* in */ int circleCount,    // Número de círculos a mover
   /* in */ int beginPeg,       // Palo que contiene los círculos a mover
   /* in */ int auxPeg,         // Palo que contiene los círculos
                                   temporalmente
   /* in */ int endPeg      )   // Palo que recibe los círculos que son
                                   movidos

// Esta función recursiva mueve los círculos de circleCount desde
// beginPeg a endPeg. Todos excepto uno de los círculos se mueven
// de beginPeg a auxPeg, luego el último círculo se mueve de beginPeg
// a endPeg, y después los círculos se mueven de auxPeg a endPeg.
// Los subobjetivos de mover los círculos hacia y desde auxPeg
// son los que tienen que ver con la recursión

// Precondición:
//     Se asignan todos los parámetros && circleCount >= 0
// Poscondición:
//     Se han impreso los valores de todos los parámetros
//  && IF circleCount > 0
//         los círculos de circleCount han sido movidos de beginPeg
//         a endPeg en la manera detallada antes
//     ELSE
//         No han tenido lugar más acciones

{
    cout << setw(6) << circleCount << setw(9) << beginPeg
         << setw(7) << auxPeg << setw(7) << endPeg << endl;
    if (circleCount > 0)
    {
        cout << "De la primera:";
        DoTowers(circleCount - 1, beginPeg, endPeg, auxPeg);
        cout << setw(58) << "Mover el circulo " << circleCount
             << " de " << beginPeg << " a " << endPeg << endl;
        cout << "De la segunda:";
        DoTowers(circleCount - 1, auxPeg, beginPeg, endPeg);
    }
}
```

La salida de una ejecución con tres círculos sigue a continuación. "Original" significa que los parámetros listados a su lado son de la llamada no recursiva, que es la primera llamada a `DoTowers`. "From first" significa que los parámetros listados son para una llamada iniciada desde la primera declaración recursiva. "From second" significa que los parámetros listados son para una llamada iniciada desde la segunda declaración recursiva. Observe que una llamada no puede ser iniciada desde la segunda declaración recursiva sino hasta que la llamada precedente desde la primera declaración recursiva haya completada su ejecución.

```
RESULTADO CON 3 CÍRCULOS

LLAMADOS DE    #CÍRCULOS  COMENZAR   AUXIL.  TERMINAR   INSTRUCCIONES

Original   :       3          1         2        3
De la primera:     2          1         3        2
De la primera:     1          1         2        3
De la primera:     0          1         3        2
                                                        Mover el círculo 1 de 1 a 3
De la segunda:     0          2         1        3
                                                        Mover el círculo 2 de 1 a 2
De la segunda:     1          3         1        2
De la primera:     0          3         2        1
                                                        Mover el círculo 1 de 3 a 2
De la segunda:     0          1         3        2
                                                        Mover el círculo 3 de 1 a 3
De la segunda:     2          2         1        3
De la primera:     1          2         3        1
De la primera:     0          2         1        3
                                                        Mover el círculo 1 de 2 a 1
De la segunda:     0          3         2        1
                                                        Mover el círculo 2 de 2 a 3
De la segunda:     1          1         2        3
De la primera:     0          1         3        2
                                                        Mover el círculo 1 de 1 a 3
De la segunda:     0          2         1        3
```

18.4 Algoritmos recursivos con variables estructuradas

En nuestra definición de un algoritmo recursivo hemos dicho que había dos casos: el caso recursivo o general y el caso base para el cual una respuesta puede ser expresada en forma no recursiva. En el caso general para todos nuestros algoritmos hasta ahora se había expresado un argumento en términos de un valor cada vez más pequeño. Cuando se usan variables estructuradas, el caso recursivo frecuentemente es en términos de una estructura más pequeña en lugar de un valor más pequeño; el caso base ocurre cuando ya no hay valores a procesar en la estructura.

Examinaremos un algoritmo recursivo para imprimir el contenido de un arreglo unidimensional de `n` elementos para mostrar a qué nos referimos.

Print Array

```
IF más elementos
    Imprimir el valor del primer elemento
    Imprimir array de n – 1 elementos
```

El caso recursivo es imprimir los valores en un arreglo que es un elemento "más pequeño"; es decir que el tamaño del arreglo disminuye en 1 con cada llamada recursiva. El caso base es cuando el tamaño del arreglo se vuelve 0, o sea cuando ya no hay más elementos que imprimir.

Nuestros argumentos deben incluir el índice del primer elemento (el elemento a imprimir). ¿Cómo sabemos cuando ya no hay más elementos qué imprimir (o sea cuando el tamaño del arreglo a imprimir es 0)? Sabemos que hemos impreso el último elemento en el arreglo cuando el índice del siguiente elemento a imprimir está más allá del índice del último elemento en el arreglo. Por lo tanto el índice del último elemento del arreglo debe ser pasado como argumento. Llamaremos a los índices `first` y `last`. Cuando `first` es mayor que `last`, hemos terminado. El nombre del arreglo es `data`.

```
void Print( /* in */ const int data[],   // Array que será impreso
            /* in */       int first,    // Índice del primer elemento
            /* in */       int last  )   // Índice del último elemento
{
    if (first <= last)
    {                                                  // Caso recursivo
        cout << data[first] << endl;
        Print(data, first + 1, last);
    }
    // La cláusula else vacía es el caso base
```

Aquí sigue un repaso de la llamada de función

```
Print(data, 0, 4);
```

usando el arreglo de la ilustración.

Llamada 1: `first` es 0 y `last` es 4. Puesto que `first` es menor que `last`, el valor en `data[first]` (que es 23) se imprime. La ejecución de esta llamada se detiene mientras que se imprime el arreglo desde `first` + 1 hasta `last`.

	data
[0]	23
[1]	44
[2]	52
[3]	61
[4]	77

Llamada 2: `first` es 1 y `last` es 4. Puesto que `first` es menor que `last`, el valor en `data[first]` (que es 44) se imprime. La ejecución de esta llamada se detiene mientras que se imprime el arreglo desde `first` + 1 hasta `last`.

Llamada 3: `first` es 2 y `last` es 4. Puesto que `first` es menor que `last`, el valor en `data[first]` (que es 52) se imprime. La ejecución de esta llamada se detiene mientras que se imprime el arreglo desde `first` + 1 hasta `last`.

Llamada 4: `first` es 3 y `last` es 4. Puesto que `first` es menor que `last`, el valor en `data[first]` (que es 61) se imprime. La ejecución de esta llamada se detiene mientras que se imprime el arreglo desde `first` + 1 hasta `last`.

Llamada 5: `first` es 4 y `last` es 4. Puesto que `first` es igual a `last`, el valor en `data[first]` (que es 77) se imprime. La ejecución de esta llamada se detiene mientras que se imprime el arreglo desde `first` + 1 hasta `last`.

Llamada 6: `first` es 5 y `last` es 4. Puesto que `first` es mayor que `last`, la ejecución de la llamada está completada. El control regresa a la llamada precedente.

Llamada 5: La ejecución de esta llamada está completa. El control regresa a la llamada precedente.

Llamadas 4, 3, 2 y 1: Cada ejecución se completa en su turno y el control regresa a la llamada precedente.

Observe que una vez que la llamada más profunda (la llamada con el número más alto) fue alcanzada, cada una de las llamadas precedentes regresó sin hacer nada. Cuando no se ejecutan declaraciones después del retorno de la llamada recursiva a la función, la recursión se llama **recursión de cola**. La recursión de cola a menudo indica que el problema se podría resolver más fácilmente usando la iteración. Hemos usado el ejemplo del arreglo porque esto hizo que el proceso recursivo fuera más fácil de visualizar; en la práctica, un arreglo se deberá imprimir en forma iterativa.

Recursión de cola Un algoritmo recursivo en el cual no se ejecutan declaraciones después del retorno de la llamada recursiva.

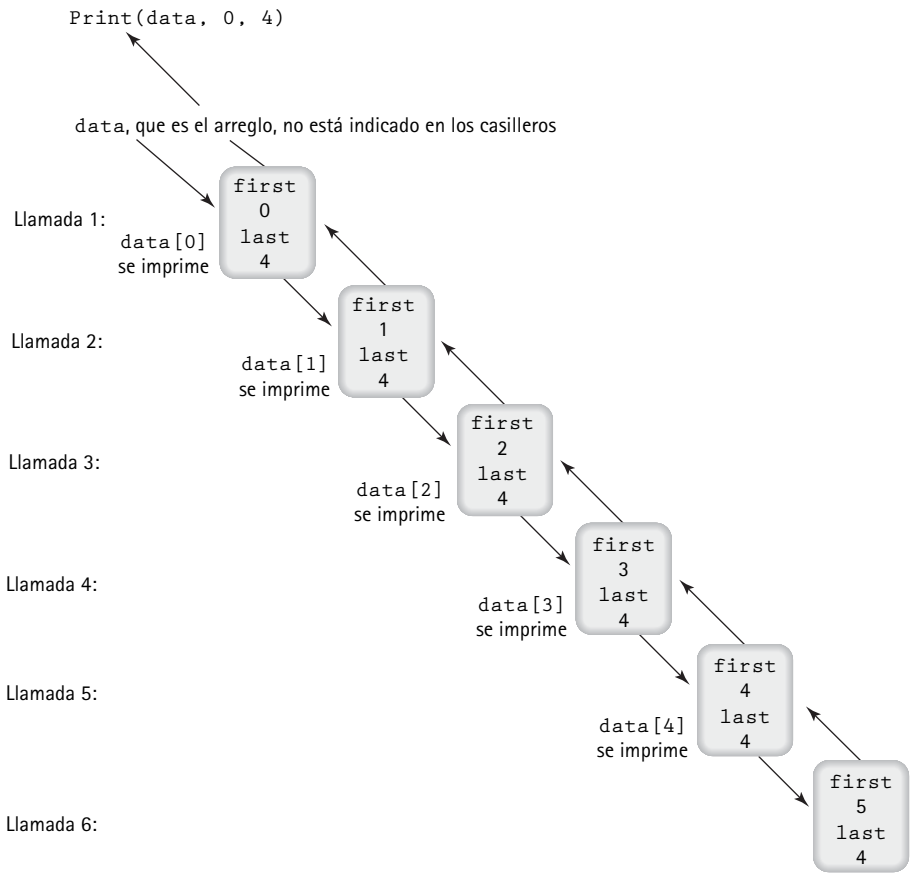


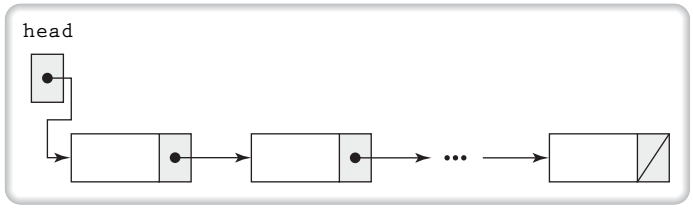
Figura 18-4 *Ejecución de* `Print(data, 0, 4)`

La figura 18-4 muestra la ejecución de la función `Print` con los valores de los parámetros para cada llamada. Observe que el arreglo se hace más pequeño con cada llamada recursiva (desde `data[first]` hasta `data[last]`. Si deseamos imprimir los elementos del array en forma recursiva en orden inverso, todo lo que tenemos que hacer es intercambiar las dos declaraciones dentro de la declaración If.

18.5 Recursión usando variables apuntador

El previo algoritmo recursivo usando un arreglo unidimensional se podría haber hecho de manera más fácil usando la iteración. Ahora veremos dos algoritmos que no se pueden realizar más fácilmente mediante la iteración: impresión de una lista ligada en orden inverso y creación de una copia duplicada de una lista ligada.

Impresión de una lista ligada dinámica en orden inverso



Imprimir una lista en orden de principio a fin es fácil. Fijamos un apuntador circulante (`ptr`) igual a `head` y ciclamos a través de la lista hasta que `ptr` se vuelve `NULL`.

Print List (entrada: head)

```
Establecer ptr = head
WHILE ptr no sea NULL
    Imprimir ptr -> component
    Establecer ptr = ptr -> link
```

A fin de imprimir la lista en orden inverso tenemos que imprimir el valor en el último nodo en primer lugar, luego el valor en el penúltimo nodo, etc. Otro modo de expresar esto es decir que no imprimimos ningún valor hasta que se hayan impreso los valores en todos los nodos que le siguen. Podríamos visualizar el proceso como que el primer nodo se dirige a su vecino diciendo: "Dime cuando hayas impreso tu valor. Luego imprimiré el mío." Aquel nodo a su vez le dice lo mismo a su vecino, y esto continúa hasta que ya no haya más que imprimir.

Puesto que el número de vecinos es cada vez menor, parece que tenemos lo necesario para una solución recursiva. El final de la lista se alcanza cuando el apuntador circulante es `NULL`. Cuando esto sucede, el último nodo puede imprimir su valor y devolver el mensaje al nodo anterior, etcétera.

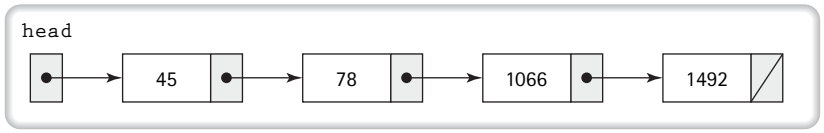
RevPrint (entrada: head)

```
IF head no es NULL
    RevPrint el resto de los nodos en lista
    Imprimir el nodo actual en la lista
```

Este algoritmo se puede codificar en forma directa como la siguiente función:

```
void RevPrint( /* in */ NodePtr head )

// Precondición:
//     puntos principales para un nodo de lista (o == NULL)
// Poscondición:
//     IF head != NULL
//         Se han producido todos los nodos que siguen a *head,
//         Entonces se ha producido *head
//     ELSE
//         Ninguna acción ha tenido lugar

{
    if (head != NULL)
    {
        RevPrint(head->link);                    // Llamada recursiva
        cout << head->component << endl;
    }
    // La cláusula else vacía es el caso base
}
```

Este algoritmo parece lo suficientemente complejo para merecer un recorrido de código. Usaremos la siguiente lista:



Llamada 1: `head` apunta al nodo que contiene 45 y no es `NULL`. La ejecución de esta llamada se detiene hasta que la llamada recursiva con el argumento `head->link` haya sido completada.

Llamada 2: `head` apunta al nodo que contiene 78 y no es `NULL`. La ejecución de esta llamada se detiene hasta que la llamada recursiva con el argumento `head->link` haya sido completada.

Llamada 3: `head` apunta al nodo que contiene 1066 y no es `NULL`. La ejecución de esta llamada se detiene hasta que la llamada recursiva con el argumento `head->link` haya sido completada.

Llamada 4: `head` apunta al nodo que contiene 1492 y no es `NULL`. La ejecución de esta llamada se detiene hasta que la llamada recursiva con el argumento `head->link` haya sido completada.

Llamada 5: `head` es `NULL`. La ejecución de esta llamada está completa. El control regresa a la llamada precedente.

Llamada 4: `head->component` (que es 1492) ha sido impreso. La ejecución de esta llamada está completa. El control regresa a la llamada precedente.

Llamada 3: `head->component` (que es 1066) ha sido impreso. La ejecución de esta llamada está completa. El control regresa a la llamada precedente.

Llamada 2: `head->component` (que es 78) ha sido impreso. La ejecución de esta llamada está completa. El control regresa a la llamada precedente.

Llamada 1: Se imprime `head->component` (que es 45). La ejecución de esta llamada está completa. Puesto que ésta es la llamada no recursiva, la ejecución continúa con la declaración inmediatamente siguiente a `RevPrint(head)`.

La figura 18-5 muestra la ejecución de la función `RevPrint`. Los parámetros son apuntadores (direcciones de memoria), así que usamos → 45 para indicar el apuntador al nodo cuyo componente es 45.

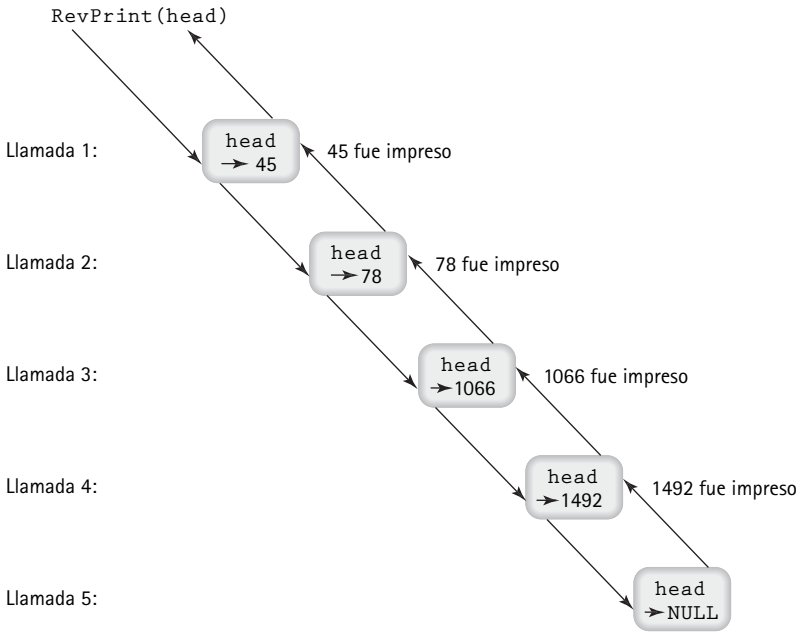


Figura 18-5 *Ejecución de* `RevPrint(head)`

Copiar una lista ligada dinámica

Cuando trabajamos con listas ligadas, a veces tenemos que crear una copia duplicada (un *clon*) de una de ellas. Por ejemplo, en el capítulo 16 escribimos un constructor de copia para la clase `SortedList2`. Este constructor de copia crea un nuevo objeto de clase para que sea un clon de otro objeto de clase, incluyendo su lista ligada dinámica.

Supongamos que quisiéramos escribir una función de devolución de valor que reciba el apuntador externo a una lista ligada (`head`), haga un clon de ella y devuelva el apuntador externo a la nueva lista como valor de función. Una típica llamada a la función sería la siguiente:

```
NodePtr head;
NodePtr newListHead;
  ⋮
newListHead = PtrToClone(head);
```

Usar una iteración para copiar una lista ligada es bastante complicado. El siguiente algoritmo es esencialmente igual al algoritmo que se usó en el constructor de copia `SortedList2`.

PtrToClone (entrada: head) //Algoritmo iterativo
Salida: valor de función

```
IF head es NULL
    Devolver NULL

// Copiar el primer nodo
Establecer fromPtr = head
Establecer cloneHead = nuevo NodeType
Establecer cloneHead -> component = fromPtr -> component

// Copiar los nodos restantes
Establecer toPtr = cloneHead
Establecer fromPtr = fromPtr -> link
WHILE fromPtr no sea NULL
    Establecer toPtr -> link = nuevo NodeType
    Establecer toPtr = toPtr -> link
    Establecer toPtr -> component = fromPtr -> component
    Establecer fromPtr = fromPtr -> link
Establecer toPtr -> link = NULL
Devolver cloneHead
```

Una solución recursiva para este problema es mucho más simple, pero requiere que pensemos en forma recursiva. A fin de clonar el primer nodo de la lista original podemos asignar un nuevo nodo dinámico y copiar el valor `component` desde el nodo original al nodo nuevo. Sin embargo, aún no podemos llenar el miembro `link` del nuevo nodo. Tenemos que esperar hasta que hayamos clonado el segundo nodo para que podamos guardar su dirección en el miembro `link` del primer nodo. De igual manera la clonación del segundo nodo no puede completarse hasta que hayamos terminado la clonación del tercer nodo. Finalmente clonamos el último nodo de la lista original y fijamos el miembro `link` del nodo clonado a `NULL`. En este punto el último nodo devuelve su propia dirección al penúltimo nodo, que a su vez guarda la dirección en su miembro `link`. El penúltimo nodo devuelve su propia dirección al nodo que le precede, etc. El proceso está completo cuando el primer nodo devuelve su dirección a la primera llamada (no recursiva), cediendo un apuntador externo a la nueva lista ligada.

PtrToClone (entrada: fromPtr) // Algoritmo recursivo
Salida: valor de función

```
IF fromPtr es NULL
    Devolver NULL
ELSE
    Establecer toPtr = nuevo NodeType
    Establecer toPtr -> component = fromPtr -> component
    Establecer toPtr -> link = PtrToClone(fromPtr-> link)
    Devolver toPtr
```

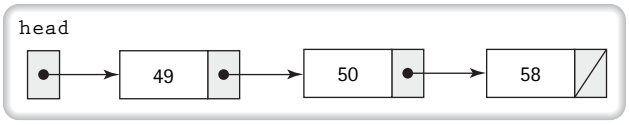
Igual que la solución del problema de las torres de Hanoi, esto se ve demasiado simple, no obstante, es el algoritmo. Puesto que el argumento que es pasado a cada llamada recursiva es `fromPtr->link`, el número de nodos que se quedan en la lista original se hace más pequeño con cada llamada. El caso base ocurre cuando el apuntador a la lista original se vuelve `NULL`. A continuación se muestra la función C++ que ejecuta el algoritmo.

```
NodePtr PtrToClone( /* in */ NodePtr fromPtr )

// Precondición:
//     fromPtr apunta a un nodo de lista (o == NULL)
// Poscondición:
//     IF fromPtr != NULL
//           Un clon de la sublista completa que empieza con *fromPtr
//           está en el almacenamiento libre
//        && Valor de función == puntero para el frente de esta sublista
//     ELSE
//           Valor de retorno == NULL

{
    NodePtr toPtr;      // Puntero para el nodo recién creado

    if (fromPtr == NULL)
        return NULL;                    // Caso base
    else
    {                                   // Caso recursivo
        toPtr = new NodeType;
        toPtr->component = fromPtr->component;
        toPtr->link = PtrToClone(fromPtr->link);
        return toPtr;
    }
}
```

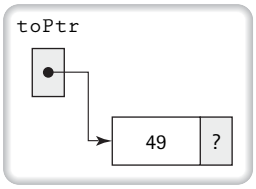
Vamos a realizar un recorrido a través de la llamada de función

```
newListHead = PtrToClone(head);
```

usando la siguiente lista:

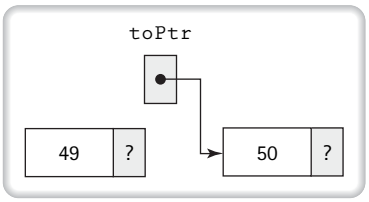


Llamada 1: `fromPtr` apunta al nodo que contiene 49 y no es `NULL`. Un nuevo nodo es asignado y su valor `component` es fijado en 49.



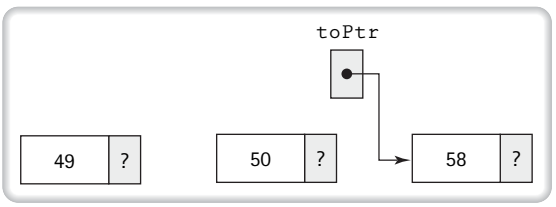
La ejecución de esta llamada se detiene hasta que la llamada recursiva con el argumento `fromPtr->link` haya sido completada.

Llamada 2: `fromPtr` apunta al nodo que contiene 50 y no es `NULL`. Un nuevo nodo es asignado y su valor `component` es fijado en 50.



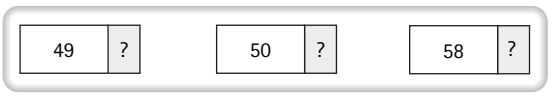
La ejecución de esta llamada se detiene hasta que la llamada recursiva con el argumento `fromPtr->link` haya sido completada.

Llamada 3: `fromPtr` apunta al nodo que contiene 58 y no es `NULL`. Un nuevo nodo es asignado y su valor `component` es fijado en 58.



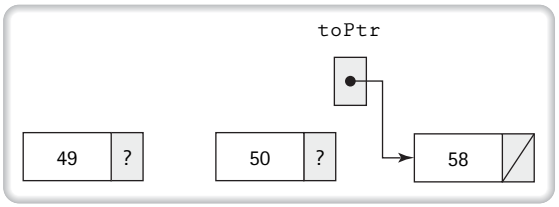
La ejecución de esta llamada se detiene hasta que la llamada recursiva con el argumento `fromPtr->link` haya sido completada.

Llamada 4: `fromPtr` es `NULL`. La lista está sin cambios.



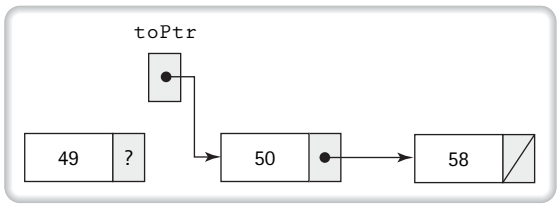
La ejecución de esta llamada está completa. `NULL` es devuelto como valor de función a la llamada precedente.

Llamada 3: La ejecución de esta llamada resume mediante la asignación del valor de función devuelto (`NULL`) a `toPtr->link`.



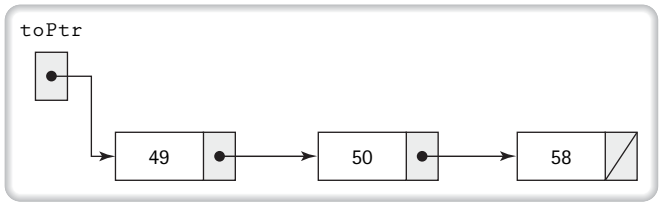
La ejecución de esta llamada está completa. El valor de `toPtr` es devuelto a la llamada precedente.

Llamada 2: La ejecución de esta llamada resume mediante la asignación del valor de función devuelto (la dirección del tercer nodo) a `toPtr->link`.

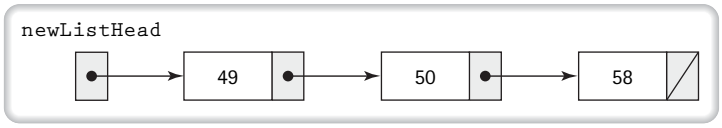


La ejecución de esta llamada está completa. El valor de `toPtr` es devuelto a la llamada precedente.

Llamada 1: La ejecución de esta llamada resume mediante la asignación del valor de función devuelto (la dirección del segundo nodo) a `toPtr->link`.



La ejecución de esta llamada está completa. Puesto que ésta es la llamada no recursiva, el valor de `toPtr` es devuelta la declaración de asignación que contiene la llamada original. La variable `newListHead` apunta ahora a un clon de la lista original.



18.6 ¿Recursión o iteración?

Recursión e iteración son formas alternativas de expresar la repetición en un programa. Cuando se usan estructuras de control iterativas, los procesos son obligados a repetirse mediante el empotramiento de código en una estructura de ciclo como por ejemplo While, For o Do-While. En la recursión, un proceso es obligado a repetirse haciendo que una función se llame a sí misma.

¿Cuál sería mejor usar, recursión o iteración? No hay ninguna respuesta simple a esta pregunta. La elección depende normalmente de dos puntos: la eficiencia y la naturaleza del problema a resolver.

Históricamente la búsqueda de la eficiencia, tanto en términos de velocidad de ejecución como de uso de memoria, ha favorecido la iteración sobre la recursión. Cada vez que se hace una llamada recursiva, el sistema debe asignar un área de memoria para todos los parámetros y variables locales (automáticas). El lastre de programación involucrado en cualquier llamada de función lleva mucho tiempo. En las primeras computadoras lentas, con capacidad de memoria limitada, los algoritmos recursivos fueron visiblemente (y a veces exasperantemente) más lentos que las versiones iterativas. Sin embargo, existen estudios que han mostrado que en las computadoras rápidas modernas el lastre de la recursión es tan frecuente que el usuario casi no se da cuenta del aumento del tiempo de cómputo. Excepto en casos donde la eficiencia es absolutamente crítica, pues la elección entre recursión e iteración más que nada depende del segundo punto: la naturaleza del problema a resolver.

Considere los algoritmos de factorial y potencias que hemos discutido al principio de este capítulo. En ambos casos las soluciones iterativas fueron obvias y fáciles de diseñar. Impusimos solucio-

nes recursivas sobre estos problemas sólo para demostrar cómo funciona la recursión. Una regla general es: si una solución iterativa es más obvia o más fácil de entender, úsela; será más eficiente. Sin embargo, hay problemas para los cuales la solución recursiva es más obvia o más fácil de diseñar; tal fue el caso del problema de las torres de Hanoi. (Resulta que el problema de las torres de Hanoi es sorprendentemente difícil de resolver por medio de la iteración.) Los estudiantes de ciencias de la computación deberán estar conscientes del poder de la recursión, pues la definición de varios problemas es inherentemente recursiva.

Caso práctico de resolución de problemas

QuickSort

PROBLEMA A través de toda la segunda mitad de este libro hemos trabajado con listas de elementos, tanto ordenadas como no ordenadas. Al nivel de lógica, los algoritmos de ordenamiento toman una lista no ordenada y la convierten en una lista ordenada. Al nivel de ejecución, los algoritmos de ordenamiento toman un arreglo y reorganizan los datos siguiendo algún tipo de orden. Hemos usado el algoritmo de selección directo para ordenar una lista de números, y hemos insertado entradas en una lista ordenada por horarios. En este caso práctico vamos a crear una plantilla de función que ejecuta el algoritmo Quicksort.

DISCUSIÓN El algoritmo Quicksort, desarrollado por C.A.R. Hoare, está basado en la idea de que es más fácil y rápido ordenar dos listas pequeñas que una grande. El nombre viene del hecho de que por lo general un ordenamiento rápido puede ordenar una lista de elementos de datos con bastante rapidez. La estrategia básica de este ordenamiento es la de "divide y vencerás".

Si usted tuviera la tarea de ordenar una gran pila de exámenes finales por nombre, podría usar el siguiente planteamiento: Elige un valor fraccionador, por decir L, y divide la pila de pruebas en dos pilas, A-l y M-Z. (Observe que las dos pilas no necesariamente contienen el mismo número de pruebas.) Luego tome la primera pila y subdivídala en dos pilas, A-F y G-L. La pila de A-F se puede subdividir aún más en A-C y D-F. Este proceso de división continúa hasta que las pilas sean lo suficientemente pequeñas para ordenarlas en forma manual. El mismo proceso se aplica a la pila M-Z.

Finalmente todas las pilas pequeñas ordenadas se pueden amontonar una encima de la otra para producir un conjunto ordenado de pruebas. (Véase la figura 18-6.)

Esta estrategia está basada en la recursión: la pila de pruebas se divide en cada intento de ordenarlas, y luego se usa el mismo planteamiento para ordenar cada una de las pilas más pequeñas (un caso más pequeño). Este proceso continúa hasta que las pilas pequeñas ya no necesitan más división (el caso base). La lista de parámetros del algoritmo Quicksort especifica la parte de la lista que se está procesando actualmente. Tome nota de que estamos ordenando los elementos de lista *guardados en el arreglo*, no una lista abstracta de cuya ejecución no sabemos nada. Para poner esta distinción muy clara, llamamos al array `data` en lugar de `list`.

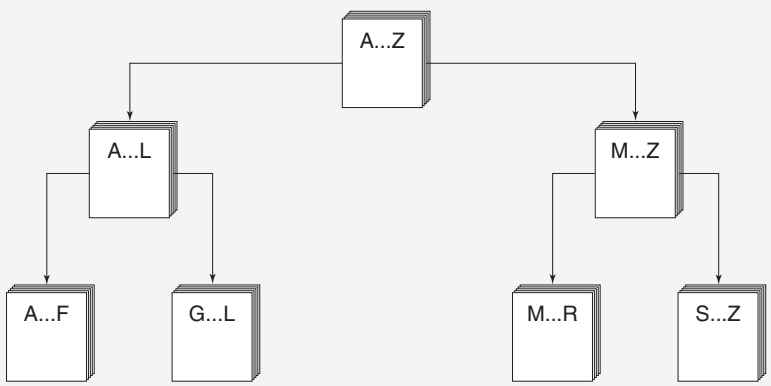
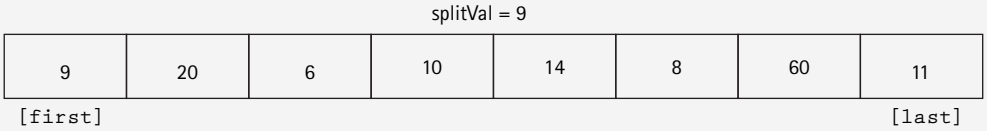


Figura 18-6 *Ordenamiento de lista mediante el algoritmo QuickSort.*

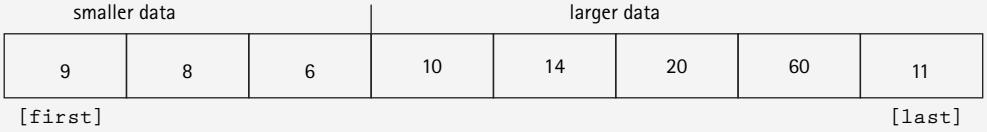
QuickSort(entrada: first, last)

```
IF hay más de un elemento en data[first]..data[last]
Seleccionar splitVal
Dividir los datos de modo que
    data[first]..data[splitPoint-1] <= splitVal
    data[splitPoint] = splitVal
    data[splitPoint+1]..data[last] > splitVal
Quicksort la mitad izquierda
Quicksort la mitad derecha
```

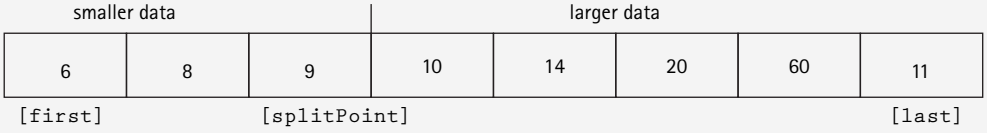
¿Cómo seleccionamos splitVal? Una solución simple es usar sea lo que fuere el valor se encuentre en data[first] como valor fraccionador. Vamos a ver un ejemplo usando data[first] como splitVal.



Después de la llamada a Split, todos los elementos menores que o iguales a splitVal se encuentran en el lado izquierdo de los datos, y todos que son mayores que splitVal están en el lado derecho de los datos.



Las dos "mitades" se unen en splitPoint, el índice del último elemento menor que o igual a splitVal. Observe que no conocemos el valor de splitPoint hasta que el proceso de fraccionamiento esté completo. Luego podemos intercambiar splitVal (data[first]) con el valor en data[split].



Nuestras llamadas recursivas a Quicksort usan este índice (splitPoint) para reducir el tamaño del problema en el caso general.

Quicksort(first, splitPoint-1) ordena la "mitad" izquierda de los datos. Quicksort(first, splitPoint+1,last) ordena la mitad "derecha" de los datos. (Las "mitades" no tienen necesariamente el mismo tamaño.) splitVal ya está en su posición correcta en data[splitPoint].

¿Qué es el caso base? Cuando el segmento que se está examinando sólo tiene un elemento, no tenemos que continuar. De modo que "hay más que un elemento en `data[first]..data[last]`" se traduce "`si (first < last)`". Ahora podemos codificar la función Quicksort.

```
template<class ItemType>
void QuickSort(ItemType data[], int first, int last)

// Precondición:
//     ComparedTo ha sido definida en ItemType
// Poscondición:
//     se clasifican los datos

{
```

```
    if (first < last)
    {
        int splitPoint;

        Split(data, first, last, splitPoint);
        // data[first]..data[splitPoint-1] <= splitVal
        // data[splitPoint] = splitVal
        // data[splitPoint+1]..data[last] > splitVal

        QuickSort(data, first, splitPoint-1);
        QuickSort(data, splitPoint+1, last);
   }
}
```

Ahora tenemos que buscar una manera de meter todos los elementos iguales a o menores que `splitVal` al otro lado. Lo hacemos moviendo un par de los índices hacia la parte media de los datos, buscando elementos que se encuentren en el lado equivocado del punto de fraccionamiento. Cuando encontramos pares que están del lado equivocado, los intercambiamos y continuamos trabajando hacia el centro de los datos.

La figura 18-7*a* muestra el estado inicial del arreglo a ordenar. Empezamos moviendo `first` a la derecha, hacia el centro, comparando `data[first]` con `splitVal`. Si `data[first]` es menor que o igual a `splitVal`, seguimos incrementando `first`; en caso contrario dejamos `first` donde se encuentre y empezamos a mover `last` hacia el centro. (Véase la figura 18-7*b*.)

Ahora se compara `data[last]` con `splitVal`. Si es más grande, continuamos disminuyendo `last`; si no, dejamos `last` en su lugar (véase la figura 18-7*c*). En este punto queda claro que tanto `data[last]` como `data[first]` están del lado equivocado del arreglo. Observe que los elementos a la izquierda de `data[first]` y a la derecha de `data[last]` no están necesariamente ordenados; simplemente están en el lado correcto *con respecto a* `splitVal`. Para colocar `data[first]` y `data[last]` en su respectivo lado correcto, simplemente los intercambiamos, y luego incrementamos `first` y disminuimos `last` (véase la figura 18-7*d*).

Ahora repetimos el ciclo completo, incrementando `first` hasta que encontremos un valor que sea más grande que `splitVal`, y luego disminuyendo `last` hasta que encontremos un valor que sea menor que o igual a `splitVal` (véase la figura 18-7*e*).

¿Cuándo se detiene el proceso? Cuando `first` y `last` se encuentran, ya no se necesitan más intercambios. Se encuentran en el `splitPoint`. Éste es el lugar al que pertenece `splitVal`, así que intercambiamos `data[saveFirst]`, que contiene `splitVal`, con el elemento en `data[splitPoint]` (figura 18-7f). El índice `splitPoint` es devuelto desde la función para ser usado por `QuickSort` a fin de preparar la siguiente llamada recursiva.

A fin de hacer que la plantilla de función Quicksort sea verdaderamente genérica, vamos a suponer que los elementos a ordenar se pueden comparar con la función `ComparedTo`.

```
template<class ItemType>
void Swap(ItemType& item1, ItemType& item2)

// Poscondición:
//     item1 es igual a item2@entry E item2 es igual a item@entry

{
    ItemType tempItem;
    tempItem = item1;
    item1 = item2;
    item2 = tempItem;
}


template<class ItemType>
```

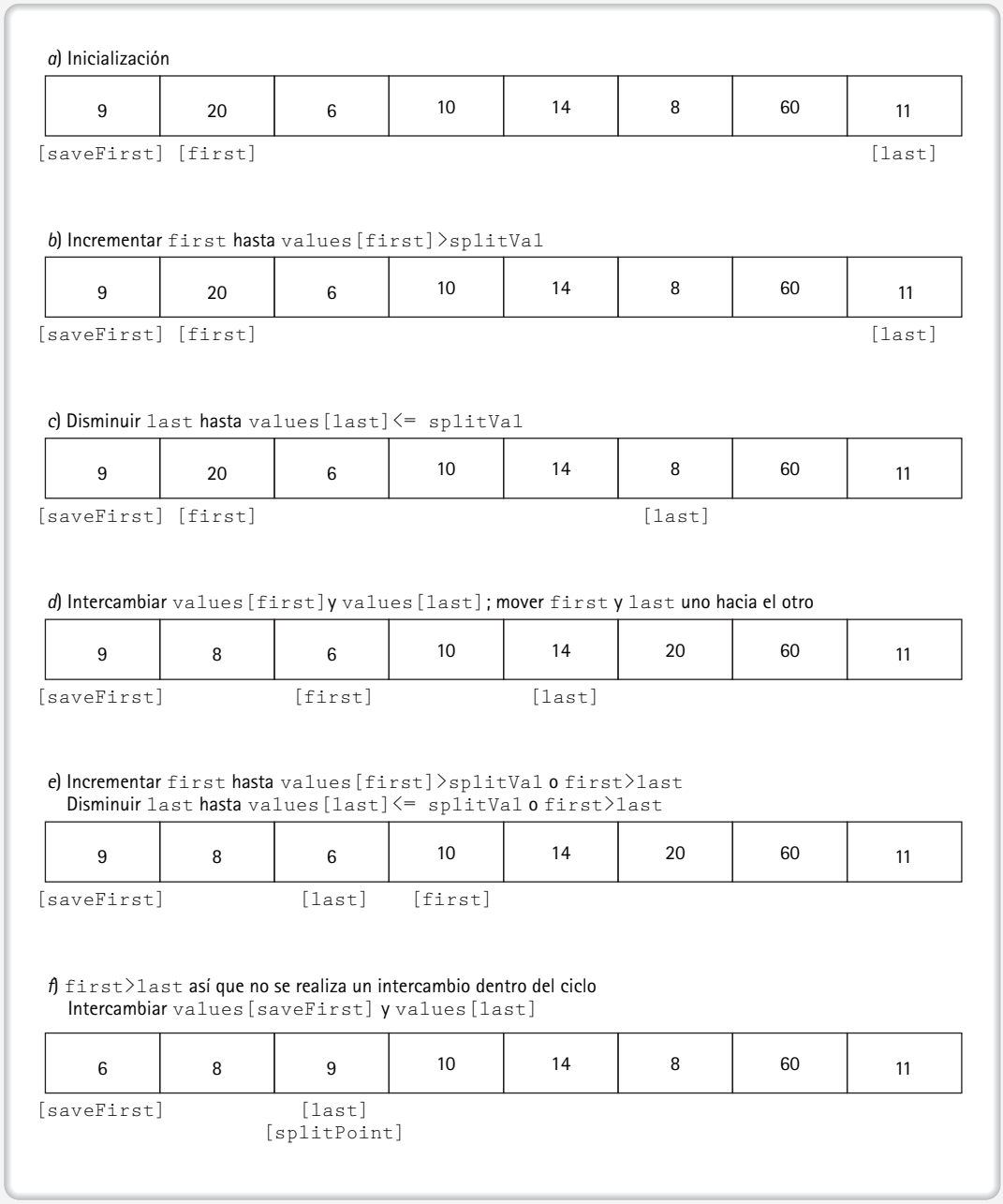


Figura 18-7 *División de función*

```
void Split(ItemType data[], int first, int last, int& splitPoint)
// Precondición:
//     ComparedTo se ha definido en ItemType
// Poscondición:
//     Todos los elementos mayores que el valor de división están a la
//     izquierda Y todos los elementos menores o iguales que un valor de
//     división están a la derecha
{
    ItemType splitVal = data[first];
    int saveFirst = first;
```

```
    bool onCorrectSide;

    first++;
    do
    {
        onCorrectSide = true;
        while (onCorrectSide)          // Mover el primero al último
            switch (data[first].ComparedTo(splitVal))
            {
                case AFTER   : onCorrectSide = false;
                               break;
                case SAME    :
                case BEFORE  : first++;
                               onCorrectSide = (first <= last);
                               break;
            }

        onCorrectSide = (first <= last);
        while (onCorrectSide)           // Mover el último hacia el primero.
            switch (data[last].ComparedTo(splitVal))
            {
                case SAME     :
                case BEFORE   : onCorrectSide = false;
                                break;
                case AFTER    : last--;
                                onCorrectSide = (first <= last);
                                break;
            }

        if (first < last)
        {
            Swap(data[first], data[last]);
            first++;
            last--;
        }
    } while (first <= last);

    splitPoint = last;
    Swap(data[saveFirst], data[splitPoint]);
}
```

PRUEBA Estas tres funciones se colocan en un solo archivo y se incluyen en un manejador para ser verificadas. La clase `Name` define a `ComparedTo`, así que vamos a capturar nombres, ordenarlos e imprimirlos. A continuación se muestra el controlador y la pantalla desde la ejecución.

```
//**************************************************************************
//  Driver (manejador) para el algoritmo Quicksort.
//**************************************************************************

#include <iostream>
#include <string>
#include "Name.h"
#include "Quicksort.h"
using namespace std;
```

```
int main()
{
    Name data[15];
    Name name;

    // Leer en 15 nombres
    for (int index = 0; index < 15; index++)
    {
        name.ReadName();
        data[index] = name;
    }

    // Clasificar 15 nombres
    Quicksort(data, 0, 14);

    // Imprimir el nombre y el apellido materno en orden clasificado
    for (int index = 0; index < 15; index++)
        cout << data[index].FirstName() << " "
             << data[index].LastName() << endl;

    return 0;
}
```

```
Enter first name: Sam
Enter middle name: S
Enter last name: Smith
Enter first name: Joe
Enter middle name: J
Enter last name: Jones
Enter first name: Bill
Enter middle name: B
Enter last name: Black
Enter first name: Jane
Enter middle name: J
Enter last name: Jones
Enter first name: Gray
Enter middle name: G
Enter last name: Green
Enter first name: Pansy
Enter middle name: P
Enter last name: Potter
Enter first name: Rose
Enter middle name: R
Enter last name: Red
Enter first name: Yellow
Enter middle name: Y
Enter last name: Yarn
Enter first name: Calvin
Enter middle name: C
Enter last name: Carson
Enter first name: Alfred
Enter middle name: A
Enter last name: Alred
Enter first name: Nell
```

```
Enter middle name: N
Enter last name: Night
Enter first name: Carol
Enter middle name: Carter
Enter last name: Carton
Enter first name: Zoe
Enter middle name: Z
Enter last name: Zebra
Enter first name: June
Enter middle name: May
Enter last name: July
Enter first name: Son
Enter middle name: S
Enter last name: Sunlight
Alfred Alred
Bill Black
Calvin Carson
Carol Carton
Gray Green
Jane Jones
Joe Jones
June July
Nell Night
Pansy Potter
Rose Red
Sam Smith
Son Sunlight
Yellow Yarn
Zoe Zebra
```

Prueba y depuración

La recursión es una técnica poderosa, pero cuando se usa en forma incorrecta puede causar errores que son difíciles de diagnosticar. La mejor manera de depurar un algoritmo recursivo es construirlo correctamente desde el principio. Para ser realistas, sin embargo, daremos unos cuantos consejos sobre los puntos donde hay que buscar posibles errores.

Sugerencias para la prueba y depuración

1. Asegúrese de que haya un caso base. De no haberlo, el algoritmo seguirá expidiendo llamadas hasta que se haya usado toda la memoria. Cada vez que se llama la función, ya sea en forma recursiva o iterativa, se asigna un espacio de pila para los parámetros y variables locales automáticas. Si no existe un caso base para terminar las llamadas recursivas, la pila en tiempo de ejecución desbordará finalmente. Un mensaje de error como "STACK OVERFLOW" (desbordamiento de pila en tiempo de ejecución) indica que falta el caso base.
2. Asegúrese que no haya usado una estructura While. La estructura básica en un algoritmo recursivo es la declaración If. Deberá haber por lo menos dos casos: el caso recursivo y el caso base. Si el caso base no hace nada, se omite la cláusula `else`. La estructura de selección, sin embargo, debe estar en su lugar. Si se usa una declaración While en un algoritmo recursivo, la declaración While usualmente no deberá contener una llamada recursiva.

3. Como en el caso de las funciones no recursivas, no se deben referenciar variables globales directamente dentro de una función recursiva, a menos que exista una justificación para hacerlo.
4. Los parámetros relacionados con el tamaño del problema deben ser parámetros de valor, no de referencia. Los argumentos relacionados con el tamaño del problema son normalmente expresiones. Las expresiones arbitrarias sólo se podrán pasar a parámetros de valor.
5. Use el programa de depuración de su sistema (o use declaraciones de salida de depuración) para trazar una serie de llamadas recursivas. La inspección de los valores de parámetros y variables locales frecuentemente ayuda a localizar errores en un algoritmo recursivo.

Resumen

Un algoritmo recursivo es expresado en términos de una instancia menor de sí mismo. Debe incluir un caso recursivo para el cual el algoritmo es expresado en términos de sí mismo, así como un caso base para el cual el algoritmo es expresado en términos no recursivos.

En muchos problemas recursivos la instancia menor remite a un argumento numérico que se reduce con cada llamada. En otros problemas, remite al tamaño de la estructura de datos que se está manipulando. El caso base es el caso en el cual el tamaño del problema (valor o estructura) llega a un punto para el cual se conoce una respuesta explícita.

En el ejemplo para encontrar la recursión mínima, el tamaño del problema fue el tamaño del arreglo en el cual se buscaba, Cuando el tamaño del arreglo llegó a 1, la solución estaba conocida. Si sólo hay un elemento de arreglo, será claramente el mínimo (así como también el máximo).

En el juego de las torres de Hanoi, el tamaño del problema fue el número de discos a mover. Cuando ya sólo había uno en el palo de inicio, se podía mover a su destino final.

Comprobación rápida

1. ¿Qué caso causa que un algoritmo recursivo termine su recursión: el caso base o el caso general? (pp. 800-801)
2. Para escribir un algoritmo recursivo que calcule el factorial de N, ¿qué usaría usted como caso base, y qué sería el caso general? (pp. 803-805)
3. Para escribir un algoritmo recursivo que extraiga los valores a un arreglo, ¿qué usaría usted como caso base, y qué sería el caso general? (pp. 809-811)
4. Para escribir un algoritmo recursivo que extraiga los valores en una lista ligada en orden inverso, ¿qué usaría usted como caso base, y qué sería el caso general? (pp. 811-817)

Respuestas

1. El caso base. **2.** El caso base sería: cuando N es cero, el resultado es uno. El caso general sería: cuando N es mayor que cero, lo multiplicamos por el producto de los números de 1 a N-1, como devuelto por una llamada recursiva. **3.** El caso base sería: si ya no quedan elementos, retornamos. El caso general sería: si hay elementos remanentes, imprimimos el primero, y luego recursión para imprimir el resto. **4.** El caso base sería que la liga del nodo actual es `NULL`, así que lo imprimimos. El caso general sería: si la liga del nodo actual no es `NULL`, imprimimos el resto de la lista antes de imprimir el nodo actual.

Ejercicios de preparación para examen

1. La recursión es una alternativa para:
   a) bifurcación
   b) ciclo
   c) invocación de función
   d) manejo de sucesos
2. Una función recursiva puede ser vacía o de devolución de valores. ¿Verdadero o falso?
3. Cuando una función llama a sí misma en forma recursiva, sus parámetros y variables locales son guardados en la pila en tiempo de ejecución. ¿Verdadero o falso?

4. La recursión de cola ocurre cuando todo el procesamiento se realiza al final de la función, después del retorno de la llamada recursiva. ¿Verdadero o falso?
5. Dada la fórmula recursiva F(N) = F(N-2), con el caso base F(0) = 0, ¿cuáles son los valores de F(4), F(5) y F(6)? Indique si algunos de estos valores son indefinidos.
6. Dada la fórmula recursiva F(N) = F(N – 1) * 2, con el caso base F(0) = 1, ¿cuáles son los valores de F(3), F(4) y F(5)? Indique si algunos de estos valores son indefinidos.
7. ¿Qué pasa cuando una función recursiva nunca encuentra un caso base?
8. ¿Qué limitación práctica impide que una función se llame a sí misma en forma recursiva sin fin?
9. Una función recursiva de cola sería más eficientemente implementada con un ciclo en la mayoría de los casos. ¿Verdadero o falso?
10. Cuando usted desarrolla un algoritmo recursivo para operar sobre una variable simple, ¿qué es lo que el caso general normalmente hace más pequeño con cada llamada recursiva?
    a) El tipo de datos de la variable
    b) El número de veces que la variable es referenciada
    c) El valor de la variable
11. Cuando usted desarrolla un algoritmo recursivo para operar sobre una estructura de datos, ¿qué es lo que el caso general normalmente hace más pequeño con cada llamada recursiva?
    a) El número de elementos de la estructura
    b) El número de veces que la variable es referenciada
    c) La distancia hasta el final de la estructura
12. Dados los siguientes datos de entrada:

```
10
20
30
```

¿Qué es lo que el siguiente programa extraerá?

```
#include <iostream>

using namespace std;

void Rev();

int main()
{
    Rev();
    return 0;
}

//********************

void Rev()
{
    int number;
    cin >> number;
    if (cin)
    {
        Rev();
        cout << number << endl;
    }
}
```

13. Repite el ejercicio 12, remplazando la función Rev por la siguiente versión:

```
void Rev()
{
    int number;
    cin >> number;
    if (cin)
    {
        cout << number << endl;
        Rev();
        cout << number << endl;
    }
}
```

14. ¿Qué es lo que el siguiente programa extraerá?

```
#include <iostream>

using namespace std;

void Rec(string word);

int main()
{
    Rec("abcde");
    return 0;
}

//********************

void Rec(string word)
{
    if (word.length() > 0)
    {
        cout << word.substr(0, 1);
        Rec(word.substr(1, word.length()-2));
        cout << word.substr(word.length()-1, 1) << endl;
    }
}
```

15. ¿Qué es lo que el programa del ejercicio 14 extrae si la llamada inicial a `Rec` desde `main` usa `"abcdef"` como argumento?

Ejercicios de calentamiento de programación

1. Escriba una función recursiva de devolución de valor que calcule la suma de los dígitos en un argumento positivo `int` dado. Por ejemplo, si el argumento es 12345, entonces la función devuelve 1 + 2 + 3 + 4 + 5 = 15.
2. Escriba una función recursiva de devolución de valor que use la función `DigitSum` del ejercicio 1 para calcular el solo dígito a que se suman en última instancia los dígitos del argumento. Por ejemplo, dado el argumento 999, la suma `DigitSum` sería 9 + 9 + 9 = 27, pero la suma recursiva de dígitos sería entonces 2 + 7 = 9.
3. Escriba una versión recursiva de una búsqueda binaria de un arreglo ordenado de valores `int` que estén en orden ascendente. Los argumentos de la función deberán ser el arreglo, el valor de la búsqueda y los índices máximo y mínimo para el arreglo. La función deberá devolver el índice donde se encuentre la coincidencia, o bien −1.

4. Escriba una función recursiva que pida al usuario que ingrese un número entero positivo cada vez que se llame, hasta que se obtenga cero o un número negativo. La función luego extrae los números ingresados en orden inverso. Si el diálogo de E/S es el siguiente:

```
Enter positive number, 0 to end: 10
Enter positive number, 0 to end: 20
Enter positive number, 0 to end: 30
Enter positive number, 0 to end: 0
```

la función extrae:

```
30
20
10
```

5. Extienda la función del ejercicio 4 de tal manera que también extraiga un total actualizado de los números conforme se estén introduciendo. Por ejemplo, el diálogo de E/S podría ser el siguiente:

```
Enter positive number, 0 to end: 10
Total: 10
Enter positive number, 0 to end: 20
Total: 30
Enter positive number, 0 to end: 30
Total: 60
Enter positive number, 0 to end: 0
```

y la función entonces extraerá:

```
30
20
10
```

6. Extienda la función del ejercicio 5 de tal forma que también extraiga un total actualizado conforme los números se estén imprimiendo en orden inverso. Por ejemplo, el diálogo de E/S podría ser el siguiente:

```
Enter positive number, 0 to end: 10
Total: 10
Enter positive number, 0 to end: 20
Total: 30
Enter positive number, 0 to end: 30
Total: 60
Enter positive number, 0 to end: 0
```

y la función entonces extraerá:

```
30 Total: 30
20 Total: 50
10 Total: 60
```

7. Extienda la función del ejercicio 4 de tal manera que al final de su salida reporte el mayor valor introducido. Si el diálogo de E/S es el siguiente:

```
Enter positive number, 0 to end: 10
Enter positive number, 0 to end: 20
Enter positive number, 0 to end: 30
Enter positive number, 0 to end: 0
```

la función extraerá:

```
30
20
10
The greatest is 30
```

8. Cambie la función del ejercicio 7 de tal modo que extraiga el número más grande que se haya introducido hasta ahora, conforme el usuario esté introduciendo los datos. Por ejemplo, el diálogo de E/S podría ser el siguiente:

```
Enter positive number, 0 to end: 10
Greatest: 10
Enter positive number, 0 to end: 30
Greatest: 30
Enter positive number, 0 to end: 20
Greatest: 30
Enter positive number, 0 to end: 0
```

y la función entonces extraerá:

```
20
30
10
The greatest is 30
```

9. Cambie la función del ejercicio 8 de tal modo que extraiga el número más grande que se haya introducido hasta ahora, conforme extrae los números en orden inverso. Por ejemplo, el diálogo de E/S podría ser el siguiente:

```
Enter positive number, 0 to end: 10
Greatest: 10
Enter positive number, 0 to end: 30
Greatest: 30
Enter positive number, 0 to end: 20
Greatest: 30
Enter positive number, 0 to end: 0
```

y la función entonces extrae:

```
20 Greatest: 20
30 Greatest: 30
10 Greatest: 30
The greatest is 30
```

10. Dadas las siguientes declaraciones:

```
struct NodeType;
typedef NodeType* PtrType;
struct NodeType
{
  int info;
  PtrType link;
};

PtrType head1;
PtrType head2;
```

Suponga que la lista a la que apunta `head1` contiene un número arbitrario de nodos. Escriba una función recursiva que haga una copia de esta lista en orden inverso, a la que apunte `head2`.

11. Dadas las declaraciones en el ejercicio 10, suponga que la lista a la que apunta `head1` contiene un número arbitrario de nodos. Escriba una función recursiva que haga una copia de esta lista en el mismo orden, a la que apunte `head2`.
12. Dadas las declaraciones en el ejercicio 10, suponga que la lista a la que apunta `head1` contiene un número arbitrario de nodos. Escriba una función recursiva que haga una sola lista que contenga dos copias de la lista `head1`. La primera copia estará en el mismo orden, y la segunda estará en orden inverso. La nueva lista estará siendo apuntada por `head2`.

Problemas de programación

1. El divisor común más grande (DCG) de dos enteros positivos es el entero más grande que divide los números en forma exacta. Por ejemplo, el factor común más grande de 14 y 21 es 7; para 13 y 22 es 1, y para 45 y 27 es 9. Podemos escribir una fórmula recursiva para encontrar el DCG, dado que los dos números se llaman a y b:

   DCG(a, b) = a, si b = 0
   DCG(a, b) = LCF(b, a%b), si b > 0

   Ejecute esta fórmula como una función recursiva de C++, y escriba un programa de manejador que le permitirá verificarlo en forma interactiva.
2. Escriba un programa de C++ para extraer la representación binaria (base-2) de un entero decimal. El algoritmo para esta conversión deberá repetidamente dividir el número decimal entre 2 hasta que sea cero. Cada división producirá un residuo de 0 o 1, que se convertirá en un dígito en el número binario. Por ejemplo, si queremos la representación binaria del decimal 13, la encontraremos por medio de la siguiente serie de divisiones:

   ```
   13/2 = 6   remainder 1
   6/2 = 3    remainder 0
   3/2 = 1    remainder 1
   1/2 = 0    remainder 1
   ```

   La representación binaria de 13 es, por tanto, 1101. El único problema con este algoritmo es que la primera división genera el dígito binario de orden inferior, la siguiente división genera el dígito de segundo orden. etc., hasta que la última división produce el dígito de orden superior. Por ende, si extraemos los conforme se generan, se encuentran en orden inverso. Usted deberá usar la recursión para invertir el orden de salida.
3. Cambie el programa para el Problema 2 de tal forma que funcione para cualquier base hasta 10. El usuario deberá introducir el número decimal y la base, y el programa extraerá el número en la base dada.
4. Escriba un programa de C++ usando la recursión para convertir un número binario (de 1 a 10) a un número decimal. El algoritmo para esto es que cada dígito sucesivo en el número es multiplicado por la base (dos) elevado a la potencia correspondiente a su posición en el número. El dígito de orden inferior es la posición 0. Sumaremos todos estos productos para obtener el valor decimal. Por ejemplo, si tenemos el número binario 111001, lo convertimos a decimal de la siguiente manera:

   $1 * 2^0 = 1$
   $0 * 2^1 = 0$
   $0 * 2^2 = 0$
   $1 * 2^3 = 8$
   $1 * 2^4 = 16$
   $1 * 2^5 = 32$

   Valor decimal = 1 + 0 + 0 + 8 + 16 + 32 = 57

Una formulación recursiva de esto puede extraer cada dígito y calcular la potencia correspondiente de la base por la que se debe multiplicar. Una vez que se haya extraído el caso base (el último dígito), la función puede sumar los productos conforme retorna.

En C++ podemos representar un número binario usando enteros. Sin embargo, existe un peligro: el usuario puede introducir un número inválido, tecleando un dígito que no es representable en la base. Por ejemplo, el número 1011201 es un número binario inválido porque 2 no está permitido en el sistema de números binarios. Su programa deberá revisar si existen dígitos inválidos conforme se están extrayendo, y manejar el error en una forma que sea apropiada (es decir, no deberá extraer un resultado numérico), y proporcionar un mensaje de error informativo para el usuario.

5. Modifique el programa del Problema 4 de tal modo que funcione para cualquier base de números en el rango de 1 a 10. El usuario deberá introducir un número y una base, y el programa extraerá el equivalente decimal. Si el usuario introduce un dígito que es inválido para la base, el programa deberá extraer un mensaje de error y no deberá visualizar un resultado numérico.
6. Un laberinto debe ser representado por un arreglo de 12 $\times$ 12, compuesto por tres valores: Abierto, Pared o Salida. Hay una salida del laberinto. Escriba un programa para determinar si es posible salir del laberinto desde cualquier punto de partida (cualquier cuadrado abierto puede ser un punto de partida). Usted se podrá mover vertical u horizontalmente hacia cualquier cuadrado contiguo. Usted no se podrá mover hacia un cuadrado que contenga una pared. La entrada consiste en una serie de 12 líneas de 12 caracteres cada uno, representando el contenido de cada cuadrado en el laberinto. Los caracteres son A, P o E. Su programa deberá verificar que sólo exista una salida. Mientras que se introducen los datos, el programa deberá hacer una lista de todas las coordenadas de puntos de partida. Luego podrá recorrer esta lista, resolviendo el laberinto para cada punto de partida.

Seguimiento de caso práctico

1. ¿Qué pasa si el valor fraccionador en el `QuickSort` es el valor más grande o más pequeño en el segmento? Explique.
2. ¿Qué podrá ser un mejor valor fraccionador?
3. ¿Cómo podrá usted usar otro valor fraccionador sin cambiar el algoritmo?

APÉNDICES

Apéndice A Palabras reservadas

Los siguientes identificadores son *palabras reservadas*, o sea identificadores con significados predefinidos en el lenguaje C++. El programador no las puede declarar para otros usos (por ejemplo para nombres de variables) en un programa de C++.

```
and             double          not             this
and_eq          dynamic_cast    not_eq          throw
asm             else            operator        true
auto            enum            or              try
bitand          explicit        or_eq           typedef
bitor           export          private         typeid
bool            extern          protected       typename
break           false           public          union
case            float           register        unsigned
catch           for             reinterpret_cast using
char            friend          return          virtual
class           goto            short           void
compl           if              signed          volatile
const           inline          sizeof          wchar_t
const_cast      int             static          while
continue        long            static_cast     xor
default         mutable         struct          xor_eq
delete          namespace       switch
do              new             template
```

Apéndice B Precedencia de operador

La siguiente tabla resume la precedencia de operador en C++. Varios operadores no se analizan en este libro (`typeid`, el operador de coma, `->*` y `.*`, por ejemplo). Para información acerca de estos operadores, véase la obra de Soustrup, *The C++ Programming Language*, 3ª edición (Addison-Wesley, 1997).

En esta tabla, los operadores están agrupados por nivel de precedencia (del más alto al más bajo) y una línea horizontal separa cada nivel de precedencia del siguiente nivel más bajo.

En general, los operadores binarios están agrupados de izquierda a derecha: los operadores unarios de derecha a izquierda, y el operador `?:` de derecha a izquierda. Excepción: Los operadores de asignación están agrupados de derecha a izquierda.

Precedencia (del más alto al más bajo)

Operador	Asociatividad	Observaciones
`::`	Izquierda a derecha	Resolución de alcance (binaria))
`::`	Derecha a izquierda	Acceso global (unario)
`()`	Izquierda a derecha	Llamada de función y moldeo de estilo función
`[] -> .`	Izquierda a derecha	
`++ --`	Derecha a izquierda	++ y - - como operadores de sufijo
`typeid dynamic_cast`	Derecha a izquierda	
`static_cast const_cast`	Derecha a izquierda	
`reinterpret_cast`	Derecha a izquierda	
`++ -- ! Unary + Unary –`	Derecha a izquierda	++ y - - como operadores de prefijo
`~ Unary * Unary &`	Derecha a izquierda	
`(cast) sizeof new delete`	Derecha a izquierda	
`->* .*`	Izquierda a derecha	
`* / %`	Izquierda a derecha	
`+ –`	Izquierda a derecha	
`<< >>`	Izquierda a derecha	
`< <= > >=`	Izquierda a derecha	
`= = !=`	Izquierda a derecha	
`&`	Izquierda a derecha	
`^`	Izquierda a derecha	
`|`	Izquierda a derecha	
`&&`	Izquierda a derecha	
`||`	Izquierda a derecha	
`?:`	Derecha a izquierda	
`= += –= *= /= %=`	Derecha a izquierda	
`<<= >>= &= |= ^=`	Derecha a izquierda	
`throw`	Derecha a izquierda	
`,`	Izquierda a derecha	El operador secuenciador, no el separador

Apéndice C Selección de rutinas de biblioteca estándares

La biblioteca estándar de C++ proporciona un amplio caudal de tipos de datos, funciones y constantes nombradas. Este apéndice detalla sólo algunos de los recursos de biblioteca de uso más general. Le recomendamos consultar el manual de su sistema particular para determinar qué otros tipos, funciones y constantes de la biblioteca estándar proporciona.

Este apéndice está organizado de manera alfabética, de acuerdo con el archivo de encabezado que su programa debe incluir (`#include`) antes de tener acceso a los elementos listados. Por ejemplo, para usar una rutina matemática como `sqrt`, se incluirá (`#include`) el archivo de encabezado `cmath` de la siguiente manera:

```
#include <cmath>
using namespace std;
 A
y = sqrt(x);
```

Observe que cada identificador de la biblioteca estándar está definido para estar en el espacio de nombre `std`. Sin la directiva `using` se escribirá:

```
y = std::sqrt(x);
```

C.1 Archivo de encabezado `cassert`

`assert (booleanExpr)`

Argumento:	Una expresión lógica (booleana)
Efecto:	Si el valor de `booleanExpr` es `true`, la ejecución del programa simplemente continúa. Si el valor de `booleanExpr` es `false`, la ejecución termina inmediatamente con un mensaje que indica la expresión booleana, el nombre del archivo que contiene el código fuente y el número de línea en el código fuente.
Function return value:	Ninguno (función void)
Nota:	Si la directiva de preprocesador `#define NDEBUG` se coloca antes de la directiva `#include <cassert>`, todas las sentencias `assert` se ignoran.

C.2 Archivo de encabezado `cctype`

`isalnum(ch)`

Argumento:	Un valor `char`, `ch`
Valor de devolución de función:	Un valor `int` que es
•	no cero (`true`), si `ch` es una letra o un carácter de dígito ('A'-'Z', 'a'-'z', '0'-'9')
•	0 (`false`), en caso contrario

`isalpha(ch)`

Argumento:	Un valor `char`, `ch`
Valor de devolución de función:	Un valor `int` que es
	• no cero (`true`), si `ch` es una letra ('A'-'Z', 'a'-'z')
	• 0 (`false`), en caso contrario

`iscntrl(ch)`

Argumento:	Un valor `char`, `ch`
Valor de devolución de función:	Un valor `int` que es
	• no cero (`true`), si `ch` es un carácter de control (en ASCII, un carácter con valor 0-31 o 127)
	• 0 (`false`), en caso contrario

`isdigit(ch)`

Argumento:	Un valor `char`,
Valor de devolución de función:	Un valor `int` que es
	• no cero (`true`), si `ch` es un carácter dígito ('0'-'9')
	• 0 (`false`), en caso contrario

`isgraph(ch)`

Argumento:	Un valor `char`, `ch`
Valor de devolución de función:	Un valor `int` que es
	• no cero (`true`), si `ch` es un carácter no blanco e imprimible (en ASCII, '!' hasta '~')
	• 0 (`false`), en caso contrario

`islower(ch)`

Argumento: Un valor `char`, `ch`

Valor de devolución de función: Un valor `int` que es

- no cero (`true`), si `ch` es una letra minúscula ('a'-'z')
- 0 (`false`), en caso contrario

`isprint(ch)`

Argumento: Un valor `char`, `ch`

Valor de devolución de función: Un valor `int` que es

- no cero (`true`), si `ch` es un carácter imprimible, incluyendo el espacio vacío (en ASCII, ' ' hasta '~')
- 0 (`false`), en caso contrario

`ispunct(ch)`

Argumento: Un valor `char`, `ch`

Valor de devolución de función: Un valor `int` que es

- no cero (`true`), si `ch` es un carácter de puntuación (equivalente a `isgraph(ch) && !isalnum(ch))`
- 0 (`false`), en caso contrario

`isspace(ch)`

Argumento: Un valor `char`, `ch`

Valor de devolución de función: Un valor `int` que es

- no cero (`true`), si `ch` es un carácter de espacio blanco (blanco, nueva línea, tabulador, retorno de carro, alimentación de forma)
- 0 (`false`), en caso contrario

`isupper(ch)`

Argumento: Un valor `char`, `ch`

Valor de devolución de función: Un valor `int` que es

- no cero (`true`), si `ch` es una letra mayúscula ('A' -'Z')
- 0 (`false`), en caso contrario

`isxdigit(ch)`

Argumento: Un valor `char`, `ch`

Valor de devolución de función: Un valor `int` que es

- no cero (`true`), si `ch` es un dígito hexadecimal ('0'-'9', 'A'-'F', 'a'-'f')
- 0 (`false`), en caso contrario

`tolower(ch)`

Argumento: Un valor `char`, `ch`

Valor de devolución de función: Un valor que es

- el equivalente en minúsculas de `ch`, si `ch` es una letra mayúscula
- `ch`, en caso contrario

`toupper(ch)`

Argumento: Un valor `char`, `ch`

Valor de devolución de función: Un carácter que es

- el equivalente en mayúsculas de `ch`, si `ch` es una letra minúscula
- `ch`, en caso contrario

C.3 Archivo de encabezado `cfloat`

Este archivo de encabezado proporciona constantes nombradas que definen las características de números de punto flotante en su máquina particular. Entre estas constantes se encuentran las siguientes:

`FLT_DIG`	Número aproximado de dígitos significativos en un valor `float` en su máquina
`FLT_MAX`	Valor máximo positivo `float` en su máquina
`FLT_MIN`	Valor mínimo positivo `float` en su máquina
`DBL_DIG`	Número aproximado de dígitos significativos en un valor `double` en su máquina
`DBL_MAX`	Valor máximo positivo `double` en su máquina
`DBL_MIN`	Valor mínimo positivo `double` en su máquina
`LDBL_DIG`	Número aproximado de dígitos significativos en un valor `long double` en su máquina
`LDBL_MAX`	Valor máximo positivo `long double` en su máquina
`LDBL_MIN`	Valor mínimo positivo `long double` en su máquina

C.4 Archivo de encabezado `climits`

Este archivo de encabezado proporciona constantes nombradas que definen los límites de valores enteros en su máquina particular. Entre dichas constantes se encuentran las siguientes:

`CHAR_BITS`	Número de bits en un byte en su máquina (8, por ejemplo)
`CHAR_MAX`	Valor `char` máximo en su máquina
`CHAR_MIN`	Valor `char` mínimo en su máquina
`SHRT_MAX`	Valor `short` máximo en su máquina
`SHRT_MIN`	Valor `short` mínimo en su máquina
`INT_MAX`	Valor `int` máximo en su máquina
`INT_MIN`	Valor `int` mínimo en su máquina
`LONG_MAX`	Valor `long` máximo en su máquina
`LONG_MIN`	Valor `long` mínimo en su máquina
`UCHAR_MAX`	Valor `unsigned char` máximo en su máquina
`USHRT_MAX`	Valor `unsigned short` máximo en su máquina
`UINT_MAX`	Valor `unsigned int` máximo en su máquina
`ULONG_MAX`	Valor `unsigned long` máximo en su máquina

C.5 Archivo de encabezado `cmath`

En las rutinas `math` de la lista que se muestra a continuación, se aplican las siguientes notas:

1. El manejo de errores para resultados incalculables o fuera de intervalo depende del sistema.
2. Todos los argumentos y valores de devolución de funciones son técnicamente del tipo `double` (punto flotante de doble precisión). Sin embargo se podrán pasar valores de precisión sencilla (`float`) a las funciones.

`acos(x)`

Argumento:	Una expresión de punto flotante `x`, donde $-1.0 \leq x \leq 1.0$
Valor de devolución de función:	Arco coseno de `x`, en el intervalo de 0.0 hasta π

`asin(x)`

Argumento:	Una expresión de punto flotante `x`, donde $-1.0 \leq x \leq 1.0$
Valor de devolución de función:	Arco seno de `x`, en el intervalo de $-\pi/2$ hasta $\pi/2$

`atan(x)`
Argumento: Una expresión de punto flotante `x`
Valor de devolución de función: Arco tangente de `x`, en el intervalo de $-\pi/2$ hasta $\pi/2$

`ceil(x)`
Argumento: Una expresión de punto flotante `x`
Valor de devolución de función: "Techo" de `x` (el número entero más pequeño $\geq$ x)

`cos(angle)`
Argumento: Una expresión de punto flotante `angle`, medida en radianes
Valor de devolución de función: Coseno trigonométrico de `angle`

`cosh(x)`
Argumento: Una expresión de punto flotante `x`
Valor de devolución de función: Coseno hiperbólico de `x`

`exp(x)`
Argumento: Una expresión de punto flotante `x`
Valor de devolución de función: El valor *e*(2.718...) elevado a la potencia de `x`

`fabs(x)`
Argumento: Una expresión de punto flotante `x`
Valor de devolución de función: Valor absoluto de `x`

`floor(x)`
Argumento: Una expresión de punto flotante `x`
Valor de devolución de función: "Piso" de `x` (el número entero más grande $\leq$ `x`)

`log(x)`
Argumento: Una expresión de punto flotante `x`, donde `x` > 0.0
Valor de devolución de función: Logaritmo natural (base e) de `x`

`log10(x)`
Argumento: Una expresión de punto flotante `x`, donde `x` $\geq$ 0.0
Valor de devolución de función: Logaritmo común (base 10) de `x`

`pow(x, y)`
Argumento: Expresiones de punto flotante `x` y `y`. Si `x` = 0.0, `y` debe ser positivo; si `x` $\leq$ 0.0, `y` debe ser un número entero
Valor de devolución de función: `x` elevado a la potencia de `y`

`sin(angle)`
Argumento: Una expresión de punto flotante `angle`, medida en radianes
Valor de devolución de función: Seno trigonométrico de `angle`

`sinh(x)`
Argumento: Una expresión de punto flotante `x`
Valor de devolución de función: Seno hiperbólico de `x`

`sqrt(x)`
Argumento: Una expresión de punto flotante `x`, donde `x` $\geq$ 0.0
Valor de devolución de función: Raíz cuadrada de `x`

`tan(angle)`
Argumento: Una expresión de punto flotante `angle`, medida en radianes
Valor de devolución de función: Tangente trigonométrica de `angle`

`tanh(x)`

Argumento:	Una expresión de punto flotante `x`
Valor de devolución de función:	Tangente hiperbólica de `x`

C.6 Archivo de encabezado `cstddef`

Este archivo de encabezado define unas cuantas constantes y tipos de datos dependientes del sistema. El único elemento de este archivo de encabezado que usamos en este libro es la siguiente constante simbólica:

`NULL`	Constante del apuntador cero 0

C.7 Archivo de encabezado `cstdlib`

`abs(i)`

Argumento:	Una expresión `int`, `i`
Valor de devolución de función:	Un valor `int` que es el valor absoluto de `i`

`atof(str)`

Argumento:	Un str de cadena C (arreglo `char` de terminación nula) representando un número de punto flotante, posiblemente precedido por caracteres de espacio blanco y un '+' o '–'
Valor de devolución de función:	Un valor `double` que es el equivalente del punto flotante de los caracteres en `str`
Nota:	La conversión para en el primer carácter en `str` que es inapropiado para un número de punto flotante. Si no se encontraron caracteres apropiados, el valor de devolución depende del sistema.

`atoi(str)`

Argumento:	Un `str` de cadena C (arreglo `char` de terminación nula) representando un número entero, tal vez precedido por caracteres de espacio blanco y un '+' o '–'
Valor de devolución de función:	Un valor `int` que es el equivalente entero de los caracteres en `str`
Nota:	La conversión se detiene en el primer carácter en `str` que es inapropiado para un número entero. Si no se encontraron caracteres apropiados, el valor de devolución depende del sistema.

`atol(str)`

Argumento:	Un `str` de cadena C (arreglo `char` de terminación nula) representando un entero largo, tal vez precedido por caracteres de espacio blanco y un '+' o '–'
Valor de devolución de función:	Un valor `long` que es el equivalente del entero largo de los caracteres en `str`
Nota:	La conversión se detiene en el primer carácter en `str` que es inapropiado para un número entero `long`. Si no se encontraron caracteres apropiados, el valor de devolución depende del sistema.

`exit(exitStatus)`

Argumento:	Una expresión `int`, `exitStatus`
Efecto:	La ejecución del programa termina inmediatamente con todos los archivos correctamente cerrados
Valor de devolución de función:	Ninguno (función void)
Nota:	Por convención, `exitStatus` es 0 para indicar la terminación normal del programa, y es no cero para indicar una terminación anormal.

`labs(i)`

Argumento:	Una expresión `long`, `i`
Valor de devolución de función:	Un valor `long` que es el valor absoluto de `i`

`rand( )`

Argumento:	Ninguno
Valor de devolución de función:	Un valor aleatorio `int` en el intervalo de 0 hasta `RAND_MAX`, una constante definida en `cstdlib` (`RAND_MAX` por lo común es lo mismo que `INT_MAX`)
Nota:	Véase `srand` después

`srand(seed)`

Argumento:	Una expresión `int`, `seed`, donde `seed` ≥ 0
Efecto:	Con el uso de `seed`, el generador de números aleatorios es inicializado en preparación para llamadas subsecuentes a la función `rand`
Valor de devolución de función:	Ninguno (función void)
Nota:	Si `srand` no es llamado antes de la primera llamada a `rand`, se supone un valor `seed` de 1.

`system(str)`

Argumento:	Un `str` de cadena C (arreglo `char` de terminación nula), representando un comando del sistema operativo, exactamente como lo teclearía un usuario en la línea de comandos del sistema operativo
Efecto:	El comando del sistema operativo representado por `str` es ejecutado
Valor de devolución de función:	Un valor `int` dependiente del sistema
Nota:	Con frecuencia los programadores ignoran el valor de devolución de función, usando la sintaxis de una llamada de función void en lugar de una llamada de función de devolución de valor

C.8 Archivo de encabezado `cstring`

El archivo de encabezado `cstring` (que no se debe confundir con el archivo de encabezado denominado `string`) admite manipulaciones de cadenas C (arreglos `char` de terminación nula).

`strcat(toStr, fromStr)`

Argumentos:	Cadenas C (arreglos `char` de terminación nula) `toStr` y `fromStr`, donde `toStr` debe ser lo suficientemente grande para retener el resultado
Efecto:	`fromStr`, incluyendo el carácter nulo '\0', es concatenado (juntado) con el final de `toStr`.
Valor de devolución de función:	La dirección base de `toStr`
Nota:	Con frecuencia los programadores ignoran el valor de devolución de función, usando la sintaxis de una llamada de función void en lugar de una llamada de función de devolución de valor.

`strcmp(str1, str2)`

Argumento:	Cadenas C (arreglos `char` de terminación nula) `str1` y `str2`
Valor de devolución de función:	Un valor `int` < 0, si `str1` < `str2`, lexicográficamente El valor `int` 0, si `str1` = `str2`, lexicográficamente Un valor `int` > 0, si `str1` > `str2`, lexicográficamente

`strcopy(toStr, fromStr)`

Argumentos:	`toStr` es un arreglo `char` y `fromStr` es una cadena C (arreglo `char` de terminación nula), y `toStr` debe ser lo suficientemente grande para retener el resultado

Efecto: `fromStr`, incluyendo el carácter nulo '\0', es copiado a `toStr` sobreescribiendo lo que había ahí
Valor de devolución de función: La dirección base de `toStr`
Nota: Con frecuencia los programadores ignoran el valor de devolución de función, usando la sintaxis de una llamada de función void en lugar de una llamada de función de devolución de valor

`strlen(str)`
Argumento: Un `str` de cadena C (arreglo `char` de terminación nula)
Valor de devolución de función: Un valor `int` ≥ 0 que es la longitud de `str` (excluyendo el '\0')

C.9 Archivo de encabezado `string`

Este archivo de encabezado proporciona un tipo de datos definido por el usuario (específicamente, una clase) denominado `string`. Asociados con el tipo `string` son un tipo de datos `string::size_type` y una constante nombrada `string::npos`, definidos de la siguiente manera:

`string::size_type`	Un tipo de entero sin signo relacionado con el número de caracteres en una cadena
`string::npos`	El valor máximo del tipo `string::size_type`

Hay docenas de funciones asociadas con el tipo `string`. A continuación se muestran varias de las más importantes. En las descripciones se supone que `s` es una variable (un *objeto*) del tipo `string`.

`s.c_str()`
Argumento: Ninguno
Valor de devolución de función: La dirección base de una cadena C (arreglo `char` de terminación nula) correspondiente a los caracteres guardados en `s`

`s.find(arg)`
Argumento: Una expresión del tipo `string` o `char`, o una cadena C (como una cadena literal)
Valor de devolución de función: Un valor del tipo `string::size_type` que indica la posición inicial en `s` donde se encontró `arg`. Si arg no fue encontrado, el valor de devolución es `string::npos`.
Nota: Las posiciones de caracteres dentro de una cadena están numeradas empezando con 0

`getline(inStream, s)`
Argumentos: Una flujo de entrada `inStream` (del tipo `istream` o `ifstream`) y un objeto `string`, `s`
Efecto: Los caracteres son introducidos desde `inStream` y guardados en `s` hasta que se encuentre el nuevo carácter línea nueva (el carácter línea nueva es consumido, pero no guardado en `s`)
Valor de devolución de función: Aunque la función técnicamente devuelve un valor (lo que no analizaremos aquí), los programadores en general invocan la función como si fuera una función void

`s.length ()`
Argumentos: Ninguno
Valor de devolución de función: Un valor del tipo `string`::size_type que indica el número de caracteres en la cadena

`s.size ()`

Argumentos:	Ninguno
Valor de devolución de función:	Idéntico con `s.length ()`

`s.substr(pos, len)`

Argumentos:	Dos enteros no signados, pos y `len`, que representan una posición y una longitud. El valor de pos debe ser menor que `s.length()`.
Valor de devolución de función:	Un objeto temporal `string` que retiene una subcadena de un máximo de `len` caracteres, empezando en la posición `pos` de `s`. Si `len` es demasiado grande, significa "hasta el final" de la cadena en `s`.
Nota:	Las posiciones de caracteres dentro de una cadena están numeradas empezando con 0.

C.10 Archivo de encabezado `sstream`

Este encabezado proporciona varias clases derivadas de `iostream` que permiten al usuario la aplicación de operaciones parecidas a flujos en lugar de archivos. En este texto usamos sólo la clase `ostringstream` que permite convertir tipos numéricos en valores formateados de cadena. Es posible aplicar el operador de inserción (<<) a un objeto `ostringstream` para adjuntar valores a él, y luego releer su contenido en forma de cadena, usando la función de miembros `str`. En la parte que sigue, `oss` es un objeto `ostringstream`.

`ostringstream()`

Argumento:	Ninguno
Valor de devolución de función:	Un objeto `ostringstream` vacío

`ostringstream(arg)`

Argumento:	Una cadena
Valor de devolución de función:	Un objeto `ostringstream` inicializado a la cadena de argumentos

`<<`

Argumento izquierdo:	Un objeto `ostringstream`
Argumento derecho:	Una expresión que se puede convertir en una cadena, incluyendo manipuladores de salida
Valor de devolución:	Un objeto `ostringstream`
Evaluado:	De izquierda a derecha

`oss.str()`

Argumento:	Ninguno
Valor de devolución:	Una cadena que contiene lo que se haya adjuntado a `oss`

`oss.str(arg)`

Argumento:	Una cadena
Efecto:	Asigna el valor del argumento a `oss`. Usando la cadena vacía como argumento, efectivamente fija el contenido del objeto `ostringstream` a ser vacío.

Apéndice D Uso de este libro con una versión pre-estándar de C++

D.1 El tipo `string`

Antes del lenguaje estándar ISO/ANSI C++, la biblioteca estándar no preveía un tipo de datos `string`. Los vendedores de compiladores a menudo proporcionaban sus propios tipos definidos por el programador, con nombres como `String`, `StringType`, etcétera. La sintaxis y semántica de operaciones de cadena con frecuencia variaban de vendedor a vendedor.

Para lectores con compiladores pre-estándar, los autores de este libro han creado un tipo de datos llamado `StrType` que imita un subconjunto del tipo `string` estándar. El subconjunto es suficiente para emparejar las operaciones de cadena que se mencionan en el presente libro.

En la directiva `#include` no se debe poner el nombre del archivo entre paréntesis angulares (< >), lo que indica al preprocesador que busque el archivo en el directorio estándar de `include`. En vez de esto, se debe colocar el nombre del archivo entre comillas (`" "`). Las comillas dobles indican al preprocesador que busque el archivo en el directorio actual del programador. Por tanto, para usar `StrType` en su programa, usted debe (a) verificar que el archivo `strtype.h` se encuentra en el directorio donde actualmente usted está trabajando en su programa, y (b) asegurarse que su programa use la directiva

```
#include "strtype.h"
```

En este libro usted puede usar `StrType` en lugar del tipo de datos `string` de la siguiente manera. Primero, sustituya en su declaración de variable la palabra `string` con `StrType` como nombre del tipo de datos. En segundo término, cambie la directiva `#include <string>` a `#include "strtype.h"`. Por ejemplo, en lugar de

```
#include <string>
  ⋮
string lastName;
```

usted deberá escribir

```
#include "strtype.h"
  ⋮
StrType lastName;
```

Por último, hay una restricción acerca de la realización de capturas en variables `StrType`. En el capítulo 4 se analiza el uso del operador >> y la función `getline` para introducir caracteres en una variable `string`. Si se usa >> con variables `StrType`, es posible extraer y guardar un máximo de 1 023 caracteres en una variable. En la práctica, sin embargo, esto no es una restricción importante. Sería muy raro que una cadena de entrada consistiese en tantos caracteres. La entrada que usa `getline` también está restringida a 1 023 caracteres. En la llamada de función

```
getline(cin, myString);
```

en la que `myString` es una variable `StrType`, la función `getline` no salta a los caracteres de espacio blanco y continúa hasta que haya leído 1 023 caracteres o que llege al carácter línea nueva `'\n'`, lo que ocurra primero. Esto quiere decir que `getline` lee y guarda una línea de entrada completa (hasta un máximo de 1 023 caracteres), con blancos incuidos y todo. Observe que para una línea de entrada de 1 023 caracteres o menos, el carácter línea nueva *sí* es consumido (pero no se guarda en `myString`).

D.2 Archivos de encabezado estándares y espacios de nombre

Históricamente los archivos de encabezado estándar, tanto en C como en C++ tenían nombres que terminaban en `.h` (lo que significa "archivo de encabezado"). Algunos archivos de encabezado —por ejemplo `iostream.h`, `iomanip.h` y `fstream.h`— se realizaban específicamente con C++. Otros, como `math.h`, `stddef.h`, `stdlib.h` y `string.h`, fueron trasladados desde la biblioteca estándar de C y quedaron disponibles tanto para programas C como C++. Cuando usted usaba una directiva `#include` como

```
#include <math.h>
```

cerca del inicio de su programa, todos los identificadores declarados en `math.h` se introdujeron en su programa en alcance global (como se estudió en el capítulo 8). Con la llegada del mecanismo *espacio de nombre* en

C++ de la norma ISO/ANSI (véase el capítulo 2 y, de manera más detallada, el capítulo 8), todos los archivos de encabezado estándar se modificaron de modo que se declaran identificadores dentro de un espacio de nombre denominado `std`. En C++ estándar, cuando incluye (`#include`) un archivo estándar de encabezado, los identificadores que contienen se colocan automáticamente en el alcance global.

Para conservar la compatibilidad con versiones más viejas de C++ que aún necesitan los archivos originales `iostream.h`, `math.h`, etcétera, los nuevos archivos de encabezado estándar se renombraron de la siguiente manera: los archivos de encabezado relacionados con C++ ya no llevan la extensión `.h`, y a los archivos de encabezado de la biblioteca se les removió la `.h` *y* se les insertó la letra *c* al principio. Enseguida se muestra una lista de los nombres anteriores y nuevos para algunos de los archivos de encabezado de mayor uso común:

Nombre anterior	Nombre nuevo
iostream.h	iostream
iomanip.h	iomanip
fstream.h	fstream
assert.h	cassert
ctype.h	cctype
float.h	cfloat
limits.h	climits
math.h	cmath
stddef.h	cstddef
stdlib.h	cstdlib
string.h	cstring

Tenga cuidado: la última entrada en la lista anterior se refiere al concepto del lenguaje C de una cadena y no tiene relación alguna con el tipo `string`, como se define en la biblioteca estándar.

Si usted trabaja con un compilador pre-estándar que no reconoce los nuevos nombres de archivos de encabezado o espacios de nombre, sólo sustituya los viejos nombres de archivos de encabezado por los nuevos conforme los encuentre en el libro. Por ejemplo, donde hemos escrito

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

usted deberá escribir

```
#include <iostream.h>
```

Por motivos de compatibilidad, los sistemas C++ tal vez retendrán ambas versiones de archivos de encabezado por algún tiempo.

D.3 Manipuladores `fixed` y `showpoint`

El capítulo 3 introduce cinco manipuladores para formateo y salida: `end1`, `setw`, `fixed`, `showpoint` y `setprecision`. Si usted usa un compilador pre-estándar con el archivo de encabezado `iostream.h`, los manipuladores `fixed` y `showpoint` tal vez no estarán disponibles.

En lugar del siguiente código, que se mostró en el capítulo 3,

```
#include <iostream>
using namespace std;
 ⋮
cout << fixed << showpoint;         // Set up floating-pt.
                                    //   salida formato
```

usted puede sustituir el siguiente código:

```
#include <iostream.h>
 ⋮
cout.setf(ios::fixed, ios::floatfield);
cout.setf(ios::showpoint);
```

Estas dos sentencias utilizan una notación C++ avanzada. Le aconsejamos que sólo use las sentencias como las vea y no se preocupe de los detalles. Ésta es la idea general. `setf` es una función void asociada con el flujo `cout`. (Observe que se requieren el punto, o periodo, entre `cout` y `setf`.) La primera llamada de función asegura que los números de punto flotante siempre se impriman en forma decimal en vez de la notación científica. La segunda llamada de función especifica que el punto decimal siempre se deberá imprimir, incluso para números enteros. En otras palabras, estas dos llamadas de función surten el mismo efecto que los manipuladores `fixed` y `showpoint`.

Nota: Si su compilador se queja de la sintaxis `ios::fixed`, `ios::floatfield` o `ios::showpoint`, usted tendrá que remplazar `ios` por `ios_base` de la siguiente manera:

```
cout.setf(ios_base::fixed, ios_base::floatfield);
cout.setf(ios_base::showpoint);
```

D.4 El tipo `bool`

Antes del estándar de ISO/ANSI C++, C++ no tenía un tipo de datos `bool`. Algunos compiladores pre-estándar implantaron el tipo `bool` antes de que se aprobara el estándar, pero otros no lo hicieron. Si su compilador no reconoce el tipo `bool`, el análisis siguiente le asistirá en la escritura de programas compatibles con los del presente libro.

En versiones de C++ sin el tipo `bool`, el valor 0 representa *falso*, y todo valor no cero representa *verdadero*. Es costumbre, en C++ pre-estándar, usar el tipo `int` para representar datos booleanos:

```
int dataOK;
  ⋮
dataOK = 1;   // Store "true" into dataOK
  ⋮
dataOK = 0;   // Store "false" into dataOK
```

Para que el código quede de mayor autodocumentación, muchos programadores de C++ pre-estándar definen su propio tipo de datos booleanos mediante el uso de una *sentencia Typedef*. Dicha sentencia le permite introducir un nombre nuevo para un tipo de datos existente:

```
typedef int bool;
```

Lo único que hace esta sentencia es indicar al compilador que sustituya la palabra `int` cada vez que aparezca la palabra `bool` en el resto del programa. Por tanto, cuando el compilador encuentra una sentencia como:

```
bool dataOK;
```

la traduce como

```
int dataOK;
```

Por medio de la sentencia Typedef y las declaraciones de dos constantes nombradas, `true` y `false`, el código al principio de este análisis se convierte en:

```
typedef int bool;
const int true = 1;
const int false = 0;
  ⋮
bool dataOK;
  ⋮
dataOK = true;
  ⋮
dataOK = false;
```

A través de todo el libro, nuestros programas usan las palabras `bool`, `true` y `false` cuando se manipulan datos booleanos. Si su compilador reconoce `bool` como un tipo integrado, no es necesario hacer nada. En caso contrario, hay tres pasos que se pueden seguir.

1. Use el editor de su sistema para crear un archivo que contenga las siguientes líneas:

   ```
   #ifndef BOOL_H
   #define BOOL_H
   typedef int bool;
   const int true = 1;
   const int false = 0;
   #endif
   ```

   No se preocupe del significado de la primera, segunda y última líneas. Éstas se explican en el capítulo 14. Sólo teclee las líneas como las ve arriba.

2. Guarde el archivo que ha creado en el paso 1, y dele el nombre `bool.h`. Guarde este archivo en el mismo directorio en el cual trabaja en su programa C++.
3. Cerca de la cabeza de cada programa en que necesite variables `bool`, teclee la línea

   ```
   #include "bool.h"
   ```

   Asegúrese de poner el archivo entre comillas dobles, no paréntesis angulares (< >). Las comillas indican al preprocesador que deberá buscar `bool.h` en su directorio actual, y no en el directorio de sistema C++.

Con `bool`, `true` y `false` definidos de este modo, los programas de este libro funcionarán correctamente, y usted podrá usar `bool` en sus propios programas, aunque no sea un tipo integrado.

Apéndice E Conjuntos de caracteres

Las siguientes tablas muestran el orden de caracteres en dos conjuntos de caracteres de uso muy común: ASCII (American Standard Code for Information Interchange) y EBCDIC (Extended Binary Coded Decimal Interchange Code). La representación interna para cada carácter se muestra en decimal. Por ejemplo, la letra *A* se representa internamente como el entero 65 en ASCII y como 193 en EBCDIC. El carácter de espacio (blanco) se denota como un "□".

Dígito(s) izquierdo(s) / *Dígito derecho*	*ASCII*									
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
1	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
2	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
3	RS	US	□	!	“	#	$	%	&t	’
4	(	)	*	+	,	–	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	DEL		

Los códigos 00-31 y 127 son los siguientes caracteres de control no imprimibles:

NUL	Carácter nulo	VT	Tabulador vertical	SYN	Libre síncrono
SOH	Inicio de encabezado	FF	Alimentación de forma	ETB	Fin del bloque transmitido
STX	Inicio del texto	CR	Retorno de carro	CAN	Cancelar
ETX	Fin del texto	SO	Elim. por desplazam.	EM	Fin del medio
EOT	Fin de transmisión	SI	Intro. por desplazam.	SUB	Sustituir
ENQ	Consulta	DLE	Escape de transmisión	ESC	Escape
ACK	Confirmar	DC1	Dispositivo de control 1	FS	Separador de archivos
BEL	Carácter de llamada	DC2	Dispositivo de control 2	GS	Separador de grupos (bip)
BS	Retroceso	DC3	Dispositivo de control 3	RS	Separador de registros
HT	Tabulador horizontal	DC4	Dispositivo de control 4	US	Separador de unidades
LF	Avance de línea	NAK	Confirmación negativa	DEL	Borrar

Dígito(s) izquierdo(s)	*Dígito derecho*	*EBCDIC*									
		0	1	2	3	4	5	6	7	8	9
6						□					
7						¢	.	<	(	+	|
8		&									
9		!	$	*	)	;	¬	–	/		
10								^	,	%	_
11		>	?								
12			`	:	#	@		=	“		a
13		b	c	d	e	f	g	h	i		
14							j	k	l	m	n
15		o	p	q	r						
16			~	s	t	u	v	w	x	y	z
17									\	{	}
18		[	]								
19					A	B	C	D	E	F	G
20		H	I								J
21		K	L	M	N	O	P	Q	R		
22								S	T	U	V
23		W	X	Y	Z						
24		0	1	2	3	4	5	6	7	8	9

En la tabla EBCDIC no se muestran los caracteres de control no imprimibles, o sea los códigos 00-63, 250-255, ni tampoco los caracteres para los cuales aparecen espacios vacíos en la tabla.

Apéndice F Estilo de programa, formateo y documentación

A lo largo de todo este texto hemos alentado el uso de un buen estilo de programación y documentación. Aunque tal vez nadie verá los programas que usted escribe como tareas para la clase, excepto la persona que califica su trabajo, fuera de la clase usted escribirá programas que serán usados por otros.

Los programas útiles tienen una vida útil muy larga, durante la cual se deberán modificar y actualizar. Cuando sea necesario hacer trabajos de mantenimiento, usted u otro programador tendrá que realizarlos. Son esenciales un buen estilo y documentación si otro programador debe entender y trabajar con su programa. También descubrirá que después de no haber trabajado con su propio programa durante algunos meses, usted se sorprenderá de cuántos detalles se le han olvidado.

F.1 Normas generales

El estilo que se usa en los programas y fragmentos a lo largo de este texto ofrecen un buen punto de partida para desarrollar su propio estilo. Nuestras metas en la creación de este estilo fueron: hacerlo simple, consistente y fácil de leer.

El estilo sólo beneficia al lector humano de su programa; las diferencias de estilo no le importan a la computadora. El buen estilo incluye el uso de nombres significativos de variables, comentarios y el sangrado de estructuras de control. Todo esto le ayuda a otros para entender y trabajar con su programa. Tal vez el aspecto más importante del estilo de programa es la consistencia. Si el estilo dentro de un programa no es consistente, entonces se torna engañoso y confuso.

A veces su instructor o la empresa para la que trabaja le especifica un estilo particular. Cuando usted modifica el código de otra persona, deberá usar el estilo de esta persona a fin de mantener la consistencia dentro del programa. Sin embargo, usted también desarrollará su propio estilo de programación, basado en lo que le han enseñado, su propia experiencia y su gusto personal.

F.2 Comentarios

Los comentarios son informaciones extra que se incluyen para que el programa sea más fácil de entender. Se deberá insertar un comentario en todas las partes donde el código es difícil de entender. Sin embargo, no exagere esto. Demasiados comentarios en un programa pueden oscurecer el código y serán una fuente de distracción.

En nuestro estilo existen cuatro tipos básicos de comentarios: encabezados, declaraciones, entrelineados y barra lateral.

Los *comentarios de encabezado* aparecen en la parte superior del programa y deberán incluir su nombre, la fecha de escritura del programa y su propósito. También es útil incluir secciones que describen entradas, salidas y supuestos. Piense que los comentarios de encabezado son la introducción del lector a su programa. A continuación se presenta un ejemplo:

```
// Éste programa el tiempo sideral para una fecha y
// día dados.
//
// Escrito por: Su nombre
//
// Fecha de finalización: 4/8/5
//
// Input: Una fecha y día en el formato MM DD AAAA HH MM SS
//
// Output: El tiempo solar se especifica para una longitud de 0 grados (GMT, UT o Z
   zona solar)
//
// Assumptions: Solar time is specified for a longitude of 0
//     degrees (GMT, UT, or Z time zone)
```

Los comentarios de encabezado también se deberán incluir para todas las funciones definidas por el usuario (véanse los capítulos 7 y 8).

Los *comentarios de declaraciones* acompañan a las declaraciones constantes y variables en el programa. Dondequiera que se declare un identificador, es útil incluir un comentario que explique su propósito. En programas en el texto, los comentarios de declaración aparecen a la derecha del identificador que se declara. Por ejemplo:

```
const float E = 2.71828;     // la base de logaritmos naturales

float deltaX;                // La diferencia en la dirección x
float deltaY;                // La diferencia en la dirección y
```

Observe que la alineación de los comentarios da al código una apariencia más nítida y ocasiona menos distracción.

Los *comentarios entrelineados* se usan para fraccionar secciones largas de código en fragmentos más cortos y más comprensibles. A menudo se trata de los nombres de módulos en el diseño de algoritmos, aunque en ocasiones usted podrá decidirse a incluir otras informaciones. En general, es buena idea rodear los comentarios entrelineados con líneas en blanco para destacarlos. Por ejemplo:

```
// Preparar archivo para lectura

scoreFile.open("scores.dat");

// Obtener datos

scoreFile >> test1 >> weight1;
scoreFile >> test2 >> weight2;
scoreFile >> test3 >> weight3;

// Imprimir encabezado

cout << "Test Score   Weight" << endl;
```

Aun cuando no se usan comentarios, se pueden insertar líneas en blanco cada vez que haya una interrupción lógica en el código que usted querrá destacar.

Los *comentarios de barra lateral* aparecen a la derecha de sentencias ejecutables y se usan para esclarecer el propósito de la sentencia. Los comentarios de barra lateral se usan frecuentemente como sentencia de seudocódigo de los niveles más bajos de su diseño. Si alguna sentencia C++ complicada requiere una explicación, se deberá escribir la sentencia de seudocódigo a la derecha de la sentencia C++. Por ejemplo:

```
while (file1 && file2)     // Mientras ninguno de los archivos esté vacío…
{
    ⋮
```

Además de los cuatro tipos principales de comentarios que hemos mencionado, existen algunos comentarios diversos que deberíamos citar. Después de la función `main` recomendamos el uso de una hilera de asteriscos (o guiones o signos iguales o ...) en un comentario antes y después de cada función para que destaquen. Por ejemplo:

```
//*****************************************************************************

void PrintSecondHeading()
{
    ⋮
}

//*****************************************************************************
```

En este texto usamos la forma alternativa de comentario en C++

```
/* Some comment */
```

para documentar el flujo de información para cada parámetro de una función:

```
void GetData( /* out */ int age,         // Patient's age
              /* out */ int weight )     // Patient's weight
{
    ⋮
}

void Print( /* in */    float val,       // Value to be printed
            /* inout */ int&  count )    // Number of lines printed
                                         //   so far
{
    ⋮
}
```

(En el capítulo 7 se describe el propósito de etiquetar cada parámetro como `/* in */`, `/* out */`, o `/* inout */`.)

Los programadores a veces colocan un comentario después de la llave derecha de un bloque (sentencia compuesta) para indicar a qué estructura de control pertenece el bloque:

```
while (num >= 0)
{
     A

    if (num = = 25)
    {
        A
    } // si
} // while
```

Adjuntar comentarios de este modo puede ayudar a clarificar el código y ser útil en la depuración de llaves desiguales.

F.3 Identificadores

La consideración más importante en la elección de un nombre para un elemento de datos o función en un programa es que el nombre debe transmitir la máxima información posible acerca de lo que es el elemento de datos o respecto a lo que hace la función. El nombre también deberá ser legible en el contexto en que se usa. Por ejemplo, los siguientes nombres transmiten la misma información, pero uno es más legible que el otro:

```
datOfInvc          invoiceDate
```

Los identificadores para tipos, constantes y variables deberán ser sustantivos, mientras que los nombres de funciones void (funciones que no devuelven valores) deberán ser verbos imperativos o frases que contengan verbos imperativos. Debido a la manera en que se invocan funciones de devolución de valores, sus nombres deberán ser sustantivos u ocasionalmente adjetivos. Enseguida se muestran algunos ejemplos:

Variables	`address, price, radius, monthNumber`
Constantes	`PI, TAX_RATE, SRING_LENGTH, ARRAY_SIZE`
Tipos de datos	`NameType, CarMakes, RoomLists, Hours`
Funciones void	`GetData, ClearTable, PrintBarChart`
Funciones de devolución de valor	`CubeRoot, Greatest, Color, AreaOf, IsEmpty`

Aunque un identificador pueda ser una serie de palabras, identificadores muy largos pueden llegar a ser muy fastidiosos y causan que el programa sea difícil de leer.

El mejor enfoque para el diseño de un identificador es intentar escribir diferentes nombres hasta llegar a un acuerdo aceptable, y luego escribir un comentario de declaración especialmente informativo junto a la declaración.

El uso de mayúsculas es otra consideración cuando se elige un identificador. C++ es un lenguaje que distingue entre mayúsculas y minúsculas; esto es, que mayúsculas y minúsculas son distintas. Diferentes programadores usan distintas convenciones para el uso de mayúsculas en identificadores. En este texto empezamos cada nombre de variable con una letra minúscula y usamos una mayúscula para cada palabra sucesiva de inglés. Empezamos cada nombre de función y nombre de tipo de datos con una mayúscula y, de nuevo, usamos una mayúscula para el comienzo de cada palabra sucesiva de inglés. Para las constantes nombradas usamos mayúsculas para el identificador completo, separando las palabras sucesivas de inglés con caracteres de raya de subrayado (_). Tenga presente, sin embargo, que las palabras reservadas de C++, como `main`, `if` y `while` siempre son palabras en minúsculas, y el compilador no las reconocerá si se usan mayúsculas.

F.4 Formateo de líneas y expresiones

C++ le permite fraccionar una sentencia larga en medio y continuar a la siguiente línea. (Sin embargo, no se puede partir una línea en medio de un identificador, una constante literal o una cadena.) Cuando tiene que partir una línea, es importante elegir un punto de ruptura lógico y legible. Compare la legibilidad de los siguientes fragmentos de código.

```
cout << "For a radius of " << radius << " the diameter of the cir"
     << "cle is " << diameter << endl;
cout << "For a radius of " << radius
     << " the diameter of the circle is " << diameter << endl;
```

Cuando tenga que partir una expresión mediante múltiples líneas, trate de terminar cada una con un operador. También intente tomar ventaja de cualquier patrón repetitivo en la expresión. Por ejemplo,

```
meanOfMaxima = (Maximum(set1Value1, set1Value2, set1Value3) +
                Maximum(set2Value1, set2Value2, set2Value3) +
                Maximum(set3Value1, set3Value2, set3Value3)) / 3.0;
```

Cuando escriba expresiones, también recuerde que los espacios mejoran la legibilidad. Normalmente usted deberá incluir un espacio en cada lado del operador = y la mayoría de los demás operadores. En ocasiones se prescinde de los espacios para hacer hincapié en el orden en que se realizan las operaciones. A continuación se presentan algunos ejemplos:

```
if (x+y > y+z)
    maximum = x + y;
else
    maximum = y + z;
hypotenuse = sqrt(a*a + b*b);
```

F.5 Sangrado

El propósito del sangrado de sentencia en un programa es proporcionar al lector una ayuda visual y de hacer que el programa sea más fácil de depurar. Cuando un programa está correctamente sangrado, la agrupación de las sentencias queda inmediatamente obvia. Compare los siguientes dos fragmentos de programa:

```
while (count <= 10)
{
cin >> num;
if (num = = 0)
{
count++;
num = 1;
}
cout << num << endl;
cout << count << endl;
}
```

```
while (count <= 10)
   {
    cin >> num;
    if (num = = 0)
       {
        count++;
        num = 1;
       }
    cout << num << endl;
    cout << count << endl;
   }
```

Como regla básica en este texto, cada elemento anidado o de nivel inferior lleva un sangrado de cuatro espacios. Excepciones de esta regla son declaraciones de parámetros y sentencias que están separados por medio de dos o más líneas. El sangrado con cuatro espacios es cuestión de preferencia personal. Algunas personas prefieren el sangrado con tres, cinco o incluso más de cinco espacios.

En este libro sangramos el cuerpo entero de una función. También, en general, se sangra cada sentencia que forma parte de otra. Por ejemplo, If-Then-Else contiene dos partes: la cláusula then y la cláusula else. Las sentencias dentro de ambas cláusulas están sangradas con cuatro espacios más allá del inicio de la sentencia If-Then-Else y el If-Then:

```
if (sex = = MALE)
{
    maleSalary = maleSalary + salary;
    maleCount++;
}
else
    femaleSalary = femaleSalary + salary;

if (count > 0)
    average = total / count;
```

Para sentencias If-Then-Else que forman una bifurcación de trayectoria múltiple generalizada (el If-Then-Else-If que se describe en el capítulo 5), se usa un estilo especial de sangrado en el texto. Enseguida se muestra un ejemplo:

```
if (month = = JANUARY)
    monthNumber = 1;
else if (month = = FEBRUARY)
    monthNumber = 2;
else if (month = = MARCH)
    monthNumber = 3;
else if (month = = APRIL)
    A
else
    monthNumber = 12;
```

Las restantes sentencias de C++ siguen todas a la norma básica de sangrado que se mencionó anteriormente. Para fines de referencia, enseguida se presentan algunos ejemplos de cada una:

```
while (count <= 10)
{
    cin >> value;
    sum = sum + value;
    count++;
}

do
{
    GetAnswer(letter);
    PutAnswer(letter);
} while (letter != 'N');

for (count = 1; count <= numSales; count++)
    cout << '*';

for (count = 10; count >= 1; count--)
{
    inFile >> dataItem;
    outFile << dataItem << ' ' << count << endl;
```

```
}

switch (color)
{
    RED    : cout << "rojo";
             break;
    ORANGE : cout << "naranja";
             break;
    YELLOW : cout << "amarillo;
             break;
    GREEN  :
    BLUE   :
    INDIGO :
    VIOLET : cout << "ondas visibles cortas";
             break;
    WHITE  :
    BLACK  : cout << "no son colores válidos";
             color = NONE;
}
```

GLOSARIO

Abstracción de control Separación de propiedades lógicas de una acción de su ejecución.

Abstracción de datos Separación de las propiedades lógicas de un tipo de datos de su implementación.

Alcance La región del código de programa donde se permite hacer referencia (uso) a un identificador.

Algoritmo Procedimiento paso a paso para resolver un problema en una cantidad de tiempo finita.

Algoritmo genérico Algoritmo en el cual las acciones o pasos están definidos, pero los tipos de datos de los elementos que se manipulan no lo están.

Algoritmo recursivo Una solución que se expresa en términos de *a*) instancias más pequeñas de sí misma, y *b*) un caso base.

Almacenamiento libre (montículo) Agrupación de ubicaciones de memoria reservada para la asignación y omisión de datos dinámicos.

Apuntador externo (principal) Variable apuntador que apunta al primer nodo en una lista ligada dinámica.

Apuntador suspendido (dangling pointer) Apuntador que apunta a una variable que ha sido desasignada.

Archivo Área nombrada en almacenamiento secundario que se usa para almacenar una colección de datos; la propia colección de datos.

Argumento Variable o expresión listada en una llamada a una función; también se denomina *argumento actual* o *parámetro actual.*

Array Colección de componentes, todos del mismo tipo, ordenados en N dimensiones ($N \geq 1$). Cada componente es accedido por N índices, de los que cada uno representa la posición del componente dentro de esta dimensión.

Array bidimensional Colección de componentes, todos del mismo tipo, estructurados en dos dimensiones. Se accede a cada componente por medio de un par de índices que representan la posición del componente en cada dimensión.

Array unidimensional Colección estructurada de componentes, todos del mismo tipo, que recibe un solo nombre. Cada componente (elemento de array) es accesado por medio de un índice que ubica la posición del componente dentro de la colección.

Barrera de abstracción Muro invisible alrededor de un objeto de clase que encapsula detalles de implementación. Dicho muro sólo puede ser atravesado por medio de la interfaz pública.

Cadena C En C y en C++, secuencia de caracteres con terminación nula almacenada en un array *char.*

Caja negra Dispositivo eléctrico o mecánico cuyo funcionamiento interior está oculto.

Campo (miembro, en C++) Componente de un registro.

Caso base Caso para el cual la solución puede ser declarada de modo no recursivo.

Caso general Caso para el cual la solución es expresada en términos de una versión más pequeña de sí misma; también se denomina *caso recursivo*.

Catch Procesar una excepción lanzada. (La captura es realizada por un manipulador de excepción.)

Ciclo Estructura de control que ocasiona que una sentencia o grupo de sentencias sea ejecutado repetidamente.

Ciclo controlado por conteo Ciclo que se ejecuta un determinado número de veces.

Ciclo controlado por sucesos Ciclo que termina cuando algo sucede dentro del cuerpo del ciclo para señalar que se deberá salir del mismo.

Clase Tipo estructurado en un lenguaje de programación que se usa para representar un tipo de datos abstractos.

Clase base (superclase) Clase de la que se hereda.

Clase derivada (subclase) Clase que hereda.

Cliente Software que declara y manipula objetos de una determinada clase.

Código de autodocumentación Código de programa que contiene tanto identificadores significativos como comentarios de aclaración juiciosamente usados.

Coerción de tipo Conversión implícita (automática) de un valor de un tipo de datos a otro.

Cohesión funcional Propiedad de un módulo en el cual todos los pasos concretos están dirigidos hacia la solución de un solo problema, y donde todos los subproblemas significativos se escriben como pasos abstractos. También el principio de que un módulo deberá realizar exactamente una acción abstracta.

Compilador Programa que traduce un lenguaje de nivel alto a código de máquina.

Complejidad Medida del esfuerzo dedicado por la computadora en la realización de un proceso, relativa al tamaño de la computación.

Complejidad de comunicación Medida de la cantidad de datos que pasan a través de la interfaz de un módulo.

Composición (contención) Mecanismo mediante el cual los datos internos (el estado) de una clase incluyen un objeto de otra clase.

Computadora Dispositivo programable que puede almacenar, recuperar y procesar datos.

Condición de terminación Condición que ocasiona la salida de un ciclo.

Constante nombrada (constante simbólica) Lugar de la memoria al que se hace referencia mediante un identificador, que contiene un valor de datos que no se puede cambiar.

Constructor Operación que crea una nueva instancia (variable) de un ADT.

Contador de iteración Variable de contador que es incrementada con cada iteración de un ciclo.

Contador de sucesos Variable que es incrementada cada vez que ocurre un suceso determinado.

Conversión de tipo Conversión explícita de un valor de un tipo de datos a otro.

Copia profunda Operación que no sólo copia un objeto de clase a otro, sino también hace copias de todos los datos a los que se apunta.

Copia superficial Operación que copia un objeto de clase a otro sin copiar algún dato a que se apunta.

Datos Informaciones en forma que la computadora puede usar.

Datos dinámicos Variables creadas durante la ejecución de un programa por medio de operaciones especiales. En C++, estas operaciones son *new* y *delete*.

Declaración Enunciado que relaciona un identificador con un objeto de datos, una función o tipo de datos para que el programador pueda hacer referencia a ese elemento por nombre.

Definición de función Enunciado de función que incluye el cuerpo de la función.

Definición recursiva Definición que enuncia algo en términos de versiones más pequeñas de sí mismo.

Degradación (reducción) Conversión de un valor de un tipo "superior" a un tipo "inferior", de acuerdo con la precedencia de tipos de datos de un lenguaje de programación. La degradación puede causar la pérdida de información.

Descomposición funcional Técnica para desarrollar un software donde el problema se divide en subproblemas de manejo más fácil, cuyas soluciones crean una solución para el problema en general.

Dígitos significativos Dígitos desde el primer dígito no cero a la izquierda hasta el último dígito a la derecha (más todos los dígitos 0 que sean exactos).

Dirección base Dirección de memoria del primer elemento de un arreglo.

Direccionamiento directo Acceder a una variable en un paso usando el nombre de la variable.

Direccionamiento indirecto Tomar acceso a una variable en dos pasos, primero usando un apuntador que proporciona la ubicación de la variable.

Diseño orientado a objetos (OOD) Técnica para desarrollar software en la cual la solución es expresada en términos de objetos; esto es, entidades independientes compuestas por datos y operaciones en dichos datos.

Dispositivo de almacenamiento auxiliar Dispositivo que almacena datos en forma codificada fuera de la memoria principal de la computadora.

Dispositivo periférico Dispositivo de entrada, salida o almacenamiento auxiliar conectado a la computadora.

Dispositivos de entrada/salida (I/O) Partes de la computadora que aceptan los datos a ser procesados (entrada) y presentan los resultados de este procesamiento (salida).

Dispositivos periféricos Dispositivos de entrada, salida o almacenamiento auxiliar que están conectados a la computatora.

Documentación Texto y comentarios escritos que hacen que un programa sea más fácil de entender, usar y modificar para otros.

Editor Programa interactivo que se usa para crear y modificar programas fuente o datos.

Efecto secundario Cualquier efecto de una función sobre otra que no sea parte de la explícitamente definida interfaz entre ellas.

Encapsulación Ocultación de una implementación de módulo en un bloque separado con una interfaz formalmente especificada.

Enlace dinámico Determinación, en tiempo de ejecución, de qué función llamar para un determinado objeto.

Enlace estático Determinación, en tiempo de compilación, de qué función se deberá llamar para un objeto determinado.

Ensamblador Programa que traduce un programa de lenguaje ensamblador en código de máquina.

Entrada de ciclo Punto donde el flujo de control alcanza la primera sentencia dentro de un ciclo.

Enumerador Uno de los valores en el dominio de un tipo de enumeración.

Equivalencia funcional Propiedad de un módulo que realiza exactamente la misma operación que el paso abstracto que define. Un par de módulos también están funcionalmente equivalentes entre sí cuando realizan exactamente la misma función.

Error figurativo Error aritmético que ocurre cuando la precisión del resultado verdadero de una operación aritmética es mayor que la precisión de la máquina.

Estructura de control Declaración que se usa para alterar el flujo de control que normalmente es secuencial.

Estructura de datos dinámica Estructura de datos que puede expandirse y contraerse durante la ejecución.

Evaluación de cortocircuito (condicional) Evaluación de una expresión lógica en orden de izquierda a derecha con detención de la evaluación tan pronto como se determina el valor verdadero final.

Evaluar Calcular un nuevo valor realizando un conjunto específico de operaciones sobre valores dados.

Excepción Evento poco usual, a menudo impredecible, detectable por software o hardware, que requiere un procesamiento especial; en C++ también una variable u objeto de clase que representa un suceso excepcional.

Expresión Disposición de identificadores, literales y operadores que pueden ser evaluados para calcular un valor de un tipo dado.

Expresión de asignación Expresión de C++ con (1) un valor, y (2) el efecto secundario de almacenar el valor de la expresión en una ubicación de memoria.

Expresión de tipo mixto Expresión que contiene operandos de diferentes tipos de datos; también se denomina *expresión de modo mixto*.

Expresión switch Expresión cuyo valor determina qué etiqueta de cambio se selecciona. No puede ser una expresión de punto flotante o de cadena.

Flujo de control Orden de ejecución de sentencias por la computadora en un programa.

Flujo de datos Flujo de información desde el código de llamada hasta una función, y desde la función de regreso a la llamada de código.

Fuga de memoria Pérdida de espacio de memoria disponible que ocurre cuando se asignan datos dinámicos, pero nunca se desasignan.

Función (procedimiento) void Función que no devuelve un valor de función a su invocador y es invocada como una sentencia separada.

Función de devolución de valor Función que devuelve un solo valor a su invocador y es invocada desde el interior de una expresión.

Función Subprograma en C++.

Hardware Componentes físicos de una computadora.

Herencia Mecanismo mediante el cual una clase adquiere las propiedades, o sea los datos y operaciones, de otra clase.

Identificador Nombre asociado con una función u objeto de datos y usado para referir a esta función u objeto de datos.

Identificador no local Respecto a algún bloque dado, cualquier identificador declarado fuera de dicho bloque.

Implementación del plan de prueba Uso de los casos de prueba especificados en un plan de prueba para verificar que un programa produce los resultados pronosticados.

Índice de array fuera de límite Valor de índice que, en C++, es menor que 0 o mayor que el tamaño de array menos 1.

Información Cualquier conocimiento que se puede comunicar.

Ingeniería de software Aplicación de la metodología y las técnicas tradicionales de ingeniería al desarrollo de software.

Interfaz Enlace de conexión en una frontera compartida que permite que se encuentren y actúen sistemas independientes, o donde pueden comunicarse entre sí. También la descripción formal del propósito de un subprograma y el mecanismo para comunicar con él.

Intervalo de valores Intervalo en que deben caer valores de un tipo numérico, especificado en términos de los valores máximos y mínimos admisibles.

Iteración Paso individual a través de —o la repetición de— un cuerpo de un ciclo.

Iterador Operación que permite procesar, uno a la vez, todos los componentes en una instancia de ADT.

Lanzar Señalar el hecho de que ha ocurrido una excepción; también se dice *levantar*.

Lenguaje de máquina Lenguaje conformado por instrucciones en código binario, usado directamente por la computadora.

Lenguaje de programación Conjunto de reglas, símbolos y palabras especiales que se usan para elaborar un programa de computación.

Lenguaje ensamblador Lenguaje de programación de bajo nivel en el que se emplea una ayuda nemotécnica para representar cada una de las instrucciones del lenguaje de máquina para una computadora particular.

Ligadura dinámica Determinación en tiempo de ejecución de qué función se deberá llamar para un objeto particular.

Lista Colección lineal de longitud variable de componentes homogéneos.

Lista de argumentos Mecanismo mediante el cual las funciones se comunican entre sí.

Lista ligada Lista en que el orden de los componentes es determinado por un miembro de enlace explícito en cada nodo, en lugar de por el orden secuencial de los componentes en la memoria.

Lista ligada dinámica Lista ligada compuesta por nodos dinámicamente asignados que son ligados por apuntadores.

Llamada de función (a una función vacía) Sentencia que transfiere el control a una función vacía. En C++, esta sentencia es el nombre de la función, seguido por una lista de argumentos.

Llamada de función (invocación de función) Mecanismo que transfiere el control a una función.

Llamada recursiva Llamada de función en que la función que se llama es la misma que la que hace la llamada.

Longitud Número de valores actualmente almacenados en una lista.

Manejador Función `main` que se usa para llamar a una función en prueba. El uso de un manejador permite el control directo del proceso de prueba.

Manipulador (manejador) de excepción Sección de código de programa que se ejecuta cuando ocurre una determinada excepción.

Metalenguaje Lenguaje que se emplea para escribir las reglas de sintaxis para otro lenguaje.

Miembro de clase Componente de una clase. Miembros de clase pueden ser datos o funciones.

Módulo Colección independiente de pasos que resuelve un problema o subproblema; puede contener pasos tanto abstractos como concretos.

Moldeo de tipos Conversión explícita de un valor de un tipo de datos a otro; conocida también como conversión de tipos.

Objeto de clase (instancia de clase) Variable de un tipo `class`.

Objeto inaccesible Variable dinámica en el dispositivo de almacenamiento libre sin apuntador que esté apuntando a ella.

Observador Operación que permite observar el estado de una instancia de un ADT sin cambiarlo.

Ocultación de información Encapsulación y ocultación de detalles de implementación para evitar que el usuario de una abstracción dependa de estos detalles o los manipule incorrectamente.

Operación agregada Operación en una estructura de datos en conjunto, a diferencia de una operación en un componente individual de la estructura de datos.

Operación polimorfa Operación que tiene múltiples significados, dependiendo del tipo del objeto al que está unida al tiempo de ejecución.

Operador binario Operador que tiene dos operandos.

Operador unario Operador que sólo tiene un operando.

Ordenación Arreglo de componentes de una lista en un orden (por ejemplo, palabras en orden alfabético o números en orden ascendente o descendente).

Palabra reservada Palabra que tiene un significativo especial en C++; no se podrá usar como un identificador definido por el programador.

Parámetro Variable declarada en un encabezado de función; también se denomina *argumento formal* o *parámetro formal*.

Parámetro de referencia Parámetro que recibe la ubicación (dirección de memoria) del argumento del invocador.

Parámetro de valor Parámetro que recibe una copia del valor del argumento correspondiente.

Paso concreto Paso para el cual los detalles de implementación están completamente especificados.

Paso de abstracción Paso para el cual algunos detalles de implementación permanecen sin especificación.

Piratería de software Copiado no autorizado de software, ya sea para uso personal o por otros.

Plan de prueba Documento que estipula cómo se debe realizar la prueba de un programa.

Plantilla de clase Concepto del lenguaje C++ que permite al compilador generar múltiples versiones de una clase admitiendo tipos de datos parametrizados.

Plantilla de función Concepto del lenguaje C++ que permite que el compilador genere múltiples versiones de una función, permitiendo tipos de datos parametrizados.

Poscondición Afirmación que deberá ser verdadera después de que ejecutó el módulo.

Precedencia de nombre Precedencia que tiene un identificador local en una función sobre un identificador global con el mismo nombre en todas las referencias que la función hace a dicho identificador; también se denomina *ocultación de nombre*.

Precisión Número máximo de dígitos significativos.

Precondición Afirmación que deberá ser verdadera antes de que el módulo empiece la ejecución.

Programa de computadora Secuencia de instrucciones que realizará una computadora.

Programa fuente Programa escrito en un lenguaje de programación de alto nivel.

Programa objeto Versión del lenguaje de máquina de un programa fuente.

Programación Planeación o planificación de la realización de una tarea o suceso.

Programación en computadora Proceso de planificar una secuencia de pasos para que los desarrolle una computadora.

Programación estructurada (de procedimiento) Elaboración de programas que son colecciones de funciones o procesos en interacción.

Programación orientada a objetos (OOP) Uso de abstracción de datos, herencia y enlace dinámico para elaborar programas que son colecciones de objetos en interacción.

Promoción (ensanchamiento) Conversión de un valor del tipo "inferior" a un tipo "superior", de acuerdo con la precedencia de tipos de datos de un lenguaje de programación.

Prototipo de función Enunciado de función sin el cuerpo de la función.

Prueba de ciclo Punto donde la expresión While es evaluada y se toma la decisión de empezar una nueva iteración o pasar a la sentencia que sigue inmediatamente al ciclo.

Prueba del estado de un flujo Uso de un objeto de flujo de C++ en una expresión lógica como si fuera una variable booleana; el resultado es `true` si la última operación de I/O en este flujo tuvo éxito, y `false` en caso contrario.

Recursión de cola Algoritmo recursivo en que no se ejecuta ninguna sentencia después del retorno de la llamada recursiva.

Recursión infinita Situación en que una función se llama a sí misma una y otra vez sin parar.

Registro (estructura, en C++) Tipo de datos estructurados con un número fijo de componentes accedidos por nombre. Los componentes podrán ser heterogéneos (de tipos diferentes).

Registro jerárquico Registro en el cual por lo menos uno de los componentes es un registro en sí mismo.

Reglas de alcance Reglas que determinan dónde, en el programa, se puede accesar a un identificador, dado el punto donde dicho identificador es declarado.

Representación de datos Forma concreta de datos que se usa para representar los valores abstractos de un tipo de datos abstractos.

Representación externa Forma imprimible (carácter) de un valor de datos.

Representación interna Modo en que un valor de datos es almacenado dentro de la unidad de memoria.

Salida de ciclo Punto donde la repetición del cuerpo del ciclo termina y el control pasa a la primera sentencia que sigue al ciclo.

Selector de miembro Expresión que se usa para acceder a componentes de un *struct* o variable de clase. Se forma usando el nombre del *struct* o variable de clase y el nombre del miembro, separado por un punto (dot).

Semántica Conjunto de reglas que determinan el significado de instrucciones escritas en un lenguaje de programación.

Sentencia de asignación Sentencia que almacena el valor de una expresión en una variable.

Sentencia de expresión Sentencia formada por añadidura de un punto y coma a una expresión.

Sintaxis Reglas formales que rigen cómo se deben escribir instrucciones válidas en un lenguaje de programación.

Sistema interactivo Sistema que permite la comunicación directa entre el usuario y la computadora.

Sistema operativo Conjunto de programas que administra los recursos de la computadora.

Sobrecarga de función Uso del mismo nombre para diferentes funciones de C++, distinguidas una de la otra por sus listas de parámetros.

Software Programas de computación; el conjunto de todos los programas disponibles en la computadora.

Talón Función ficticia (de prueba) que ayuda en la prueba de un programa. Un talón tiene el mismo nombre e interfaz que una función que en realidad sería llamada por la parte del programa que se pone a prueba, pero en general es mucho más sencillo.

Tiempo de vida Periodo durante la ejecución del programa cuando un identificador tiene memoria asignada a él.

Tipo anónimo Tipo que no tiene un identificador de tipo asociado.

Tipo de apuntador Tipo de datos simples que consiste en un conjunto ilimitado de valores, cada uno de los cuales indica por direccionamiento, o de otra manera, la ubicación de una variable de un tipo determinado. Entre las operaciones definidas en variables de apuntadores se encuentra la asignación y prueba por igualdad.

Tipo de datos Conjunto específico de valores de datos, agrupado a un conjunto de operaciones en esos valores.

Tipo de datos abstractos Tipo de datos cuyas propiedades (dominio y operaciones) se especifican independientemente de cualquier implementación particular.

Tipo de datos estructurados Tipo de datos en que cada valor es una colección de componentes y cuya organización es caracterizada por el método que se usa para acceder componentes individuales. Las operaciones permisibles en un tipo de datos estructurados incluyen el almacenamiento y la recuperación de componentes individuales.

Tipo de datos genéricos Tipo para el cual las operaciones son definidas, pero no lo son los tipos de datos de los ítems que son manipulados.

Tipo de datos simples (atómicos) Tipo de datos en que cada valor es atómico (indivisible).

Tipo de enumeración Tipo de datos definido por el usuario cuyo dominio es un conjunto ordenado de valores literales expresados como identificadores.

Tipo de referencia Tipo de datos simples que consiste en un conjunto ilimitado de valores, cada uno de los cuales es la dirección de una variable de un tipo dado. La única operación definida en una variable de referencia es la inicialización, después de la cual cada apariencia de la variable es implícitamente desreferenciada.

Tipo de valor de función Tipo de datos del valor de resultado devuelto por una función.

Tipo nombrado Tipo definido por el usuario cuya declaración incluye un identificador de tipo que le proporciona un nombre al tipo.

Transformador Operación que determina un nuevo valor del ADT, dados uno o varios valores previos de este tipo.

Unidad aritmética/lógica (ALU) Componente de la unidad central de procesamiento que realiza operaciones aritméticas y lógicas.

Unidad central de procesamiento (CPU) Parte de la computadora que ejecuta las instrucciones (programa) almacenadas en la memoria; está conformada por la unidad aritmética/lógica y la unidad de control.

Unidad de control Componente de la unidad central de procesamiento que controla las acciones de los otros componentes para que se ejecuten las instrucciones (el programa) en el orden correcto.

Unidad de memoria Almacenamiento interno de datos en una computadora.

Valor literal Cualquier valor constante escrito en un programa.

Variable Ubicación en la memoria, referenciada por un identificador, que contiene un valor de datos que puede ser modificado.

Variable automática Variable para la cual se asigna y desasigna memoria cuando el control entra en y sale del bloque donde se declara.

Variable estática Variable para la cual la memoria permanece asignada a través de la ejecución del programa completo.

Variable local Variable declarada dentro de un bloque y no accesible fuera de dicho bloque.

Virus Programa de computadora que se reproduce por sí mismo, a menudo con el propósito de invadir otras computadoras sin autorización y tal vez con la intención de hacer daño.

Respuestas a ejercicios selectos

Capítulo 1 Ejercicios de preparación para examen

2. Análisis y especificación, solución general (algoritmo), verificar.
4. El paso de análisis y especificación dentro de la fase de resolución de problemas.
6. ***a***) vi, ***b***) ii, ***c***) v, ***d***) i, ***e***) vii, ***f***) iv.
8. La unidad de control dirige las acciones de los demás componentes en la computadora para ejecutar las instrucciones de programa en el orden apropiado.
10. Falso. Dispositivos periféricos son externos al CPU y su memoria principal.
12. ***a***) ii, ***b***) v, ***c***) iii, ***d***) vii, ***e***) i, ***f***) vi, ***g***) iv.

Capítulo 1 Ejercicios de calentamiento para programación

2. Siga el algoritmo separado para llenar el vaso de agua y colocarlo sobre la barra.

Si es diestro,
  levante el tubo de pasta de dientes con la mano izquierda y desatornille la tapa girando en sentido opuesto al de las manecillas del reloj
  ponga la tapa del tubo de pasta de dientes sobre la barra
  transfiera el tubo de pasta de dientes a la mano derecha y levante el cepillo de dientes con la mano izquierda, sosteniéndolo por el mango
  coloque el extremo abierto del tubo de pasta de dientes contra las puntas de las cerdas del cepillo y apriete el tubo de pasta de dientes justo lo suficiente para crear una extrusión de 0.5 pulgadas de pasta de dientes sobre el cepillo
  quite el tubo de pasta de dientes del cepillo y coloque el tubo sobre la barra
  transfiera el tubo de pasta de dientes a la mano derecha, sosteniéndolo por su mango
  abra la boca e inserte el cepillo, colocando las cerdas cubiertas de pasta de dientes contra un diente
  refriegue el diente con el cepillo moviendo éste hacia arriba y hacia abajo 10 veces
  reposicione el cepillo en algún diente no cepillado
  repita los pasos anteriores hasta que se hayan cepillado todos los dientes
  escupa la pasta de dientes al lavabo
  coloque el cepillo de dientes en el lavabo
  levante el vaso con la mano derecha
  llene la boca con agua del vaso
  mueva el agua en la boca durante cinco segundos
  escupa el agua de la boca en el lavabo
  repita los pasos anteriores tres veces

De otro modo (en caso de ser zurdo)
levante el tubo de pasta de dientes con la mano derecha y desatornille la tapa girando en sentido opuesto al de las manecillas del reloj
ponga la tapa del tubo de pasta de dientes sobre la barra
transfiera el tubo de pasta de dientes a la mano izquierda y levante el cepillo de dientes con la mano derecha, sosteniéndolo por el mango
coloque el extremo abierto del tubo de pasta de dientes contra las puntas de las cerdas del cepillo y apriete el tubo de pasta de dientes justo lo suficiente para crear una extrusión de 0.5 pulgadas de pasta de dientes sobre el cepillo
quite el tubo de pasta de dientes del cepillo y coloque el tubo sobre la barra
transfiera el tubo de pasta de dientes a la mano izquierda, sosteniéndolo por su mango
abra la boca e inserte el cepillo, colocando las cerdas cubiertas de pasta de dientes contra un diente
refriegue el diente con el cepillo moviendo el cepillo hacia arriba y hacia abajo 10 veces
reposicione el cepillo en algún diente no cepillado
repita los pasos anteriores hasta que se hayan cepillado todos los dientes
escupa la pasta de dientes al lavabo
coloque el cepillo de dientes en el lavabo
levante el vaso con la mano izquierda
llene la boca con agua del vaso
mueva el agua en la boca durante cinco segundos
escupa el agua de la boca en el lavabo
repita los pasos anteriores tres veces
Coloque el vaso sobre la barra
Siga un algoritmo separado para limpiar después de cepillarse los dientes

4. Cambie el paso u a:
u. Repite el paso t nueve veces.

Capítulo 2 Ejercicios de preparación para examen

2. ***a***) vi, ***b***) ix, ***c***) iv, ***d***) v, ***e***) vii, ***f***) ii, ***g***) i, ***h***) x, ***i***) iii, ***j***) viii.
4. La plantilla permite que un identificador empiece con un dígito. Identificadores pueden empezar con una letra o una raya de subrayado, y dígitos podrán aparecer sólo en la segunda posición de carácter y después.
6. Verdadero.
8. Observe que la segunda línea concatena dos palabras sin espacio separador.

```
Four score and
seven years agoour fathers
brought forth on this
continent a new nation...
```

10. Precediéndolo por un carácter de diagonal invertida (\").
12. La forma // de comentario no debe abarcar más de una línea. Tampoco puede ser insertada en medio de una línea de código porque entonces todo lo que se encuentra a la derecha de // se convierte en comentario.
14. No. El identificador `endl` es un manipulador y no es un valor de cadena.
16. `std::cout << "Hello everybody!" << std::endl;`
18. Partiéndolo en partes que caben en una línea y conectándolos con el operador de concatenación.

Capítulo 2 Ejercicios de calentamiento para programación

2. Observe que la secuencia \" es necesaria en seis lugares en el listado siguiente:

```
cout << "He said, \"How is that possible?\"" << endl
     << "She replied, \"Using manipulators.\"" << endl
     << "\"Of course,\" he exclaimed!" << endl;
```

4.
```
const string FIRST = "Your first name inserted here";
const string LAST = "Your last name inserted here";
// Insert your middle initial in place of A:
const char MIDDLE = 'A';
```
6. `PART1 + PART2 + PART1 + PART3`
8.
```
#include <iostream>
#include <string>

using namespace std;

const string TITLE = "Rev.";
const char FIRST = 'H';
const char MID 'G';
const char DOT = '.';

int main()
{
    cout << TITLE << FIRST << DOT << MID << DOT
         << " Jones";
}
```

Capítulo 3 Ejercicios de preparación para examen

2. `char, short, int, long.`
4. E significa un exponente en notación científica. Los dígitos a la izquierda de E son multiplicados por 10 elevado a la potencia dada por los dígitos a la derecha de E.
6. División del entero con un resultado entero, y división flotante con un resultado flotante.
8. `()  unary -  [ *  /  % ]  [ +  - ]`
10. *a*) vi, *b*) ii, *c*) vii, *d*) iv, *e*) i, *f*) v, *g*) iii.
12. Se puede escribir el nombre del tipo de datos, seguido por una expresión entre paréntesis.
14. `215.00`
16. `string::size_type`
18. Le dice al flujo que imprima números de punto flotante sin usar la notación científica.

Capítulo 3 Ejercicios de calentamiento para programación

2. *a*) `days / 7`
   b) `days % 7`
4.
```
float(dollars * 100 + quarters * 25 + dimes * 10 +
        nickels * 5 + pennies) / 100.0
```
6. *a*) `3 * X + Y`
   b) `A * A + 2 * B + C`
   c) `((A + B)/(C - D)) * (X / Y)`
   d) `((A * A + 2 * B + C)/D) / (X * Y)`
   e) `sqrt(fabs(A - B))`
   f) `pow(X, -cos(Y))`
8. `startOfMiddle = name.find(' ') + 1;`
10. `cout << setprecision(5) << setw(15) << distance;`
12.
```
//*********************************************************
// Programa Celsius
// Este programa produce la temperatura Celsius
// correspondiente a una determinada temperatura Fahrenheit
//*********************************************************
```

```
#include <iostream>

using namespace std;

int main()
{
    const float FAHRENHEIT = 72.0;
    float celsius;

    celsius = 5/9 * (FAHRENHEIT - 32);
    cout << fixed << "Celsius equivalent of Fahrenheit "
         << FAHRENHEIT << " is " << celsius << " degrees.";
    return 0;
}
```

Capítulo 4 Ejercicios de preparación para examen

2. ***a***) El operador de inserción no se puede usar con `cin`.
b) El operador de extracción no se puede usar con `cout`.
c) El operando derecho del operador de extracción debe ser una variable.
d) Los argumentos están en orden inverso.
e) El nombre de función no deberá tener una L mayúscula, y los argumentos están invertidos.

4. a = 70, b = 80, c = 30, d = 0, e = 20, f = 30.

6. ***a***) `string1 = "January 25, 2005"`
b) El marcador de lectura está al principio de la siguiente línea.

8. Devuelve `\n`.

10. Algo de variación en la respuesta es aceptable, mientras quede claro que el mensaje al usuario inexperto debe ser más detallado.

a)
```
cout << "Introducir una fecha en el formato de mm/dd/yyyy."
     << " Por ejemplo, para octubre 18, 1989, introduzca 10/18/1989.";
```
b)
```
cout << "Introducir la fecha, formato mm/dd/yyyy.";
```

12. `ifstream` y `ofstream`.

14. La corrección es convertir el nombre en una cadena C.

```
ifstream inData;
string name;

cout << "Introducir el nombre del archivo: ");
cin >> name;
infile.open(name.c_str());
```

16. Falso. El archivo entra inmediatamente en estado de falla, y las operaciones I/O en él simplemente se ignoran.

18. La respuesta puede variar, pero al menos deberá incluir el vehículo, cliente y agencia. Con un poco más de esmero se podría añadir cochera, taller de reparación, etcétera.

20. ***a***) Paso para el cual los detalles de implementación están especificados.
b) Paso para el cual algunos detalles de implementación quedan sin especificar.
c) Serie independiente de pasos que resuelve un determinado problema o subproblema.
d) Módulo que realiza la misma operación que el paso abstracto que define.
e) Propiedad de un módulo en el cual todos los pasos concretos están dirigidos a resolver un solo problema, y todos los subproblemas significativos están escritos como pasos abstractos.

22. ***a***) no; ***b***) sí; ***c***) sí; ***d***) no; ***e***) no.

Capítulo 4 Ejercicios de calentamiento para programación

2.
```
cout << "Introducir un nombre en el formato: nombre apellido paterno
apellido materno:";
cin >> first >> middle >> last;
```

4.
```
float number1;
float number2;
float number3;
float average;

cout << "Introducir el primer número y presionar retorno:";
cin >> number1;
cout << "Introducir el segundo número y presionar retorno:";
cin >> number2;
cout << "Introducir el tercer número y oprimir retorno:";
cin >> number3;
average = (number1 + number2 + number3) / 3.0;
cout << "El promedio es: " << average << endl;
```

6. Los nombres de las variables podrán ser diferentes de los que se muestran aquí. En lugar de usar la función ignorar para saltar un carácter individual, también es posible usar la lectura a una variable ficticia (dummy) `char`.

a)
```
int int1;
char char1;
float float1;

cin >> int1 >> char1 >> float1;
```

b)
```
string string1;
string string2;
int int1;
int int2;

cin >> string1 >> int1 >> string2 >> int2;
```

c)
```
int int1;
int int2;
float float1;

cin >> int1;
cin.ignore(1, ',');
cin >> int2;
cin.ignore(1, ',');
cin >> float1;
```

d)
```
char char1;
char char2;
char char3;
char char4;

cin >> char1;
cin.ignore(1, ' ');
cin >> char2;
cin.ignore(1, ' ');
cin >> char3;
cin.ignore(1, ' ');
cin >> char4;
cin.ignore(1, ' ');
```

e) `float float1;`

```
cin.ignore(1, '$');
cin >> float1;
```

8.
```
#include <fstream>
fstream temps;
temps.open("temperatures.dat");
```

10.
```
#include <iostream>
#include <fstream>

using namespace std;

fstream inData;
fstream outData;
const float PI = 3.14159265;
float radius;
float circumference;
float area;

int main()
 {
  inData.open("indata.dat");
  outData.open("outdata.dat");
  inData >> radius;
  circumference = radius * 2 * PI;
  area = radius * radius * PI;
  cout << Para el primer círculo, la circunferencia es "
       << circumference << " y el área es " << area << endl;
  outData << radius << " " << circumference << " " << area << endl;
  inData >> radius;
  circumference = radius * 2 * PI;
  area = radius * radius * PI;
  cout << "Para el segundo círculo, la circunferencia es "
       << circumference << " y el área es " << area << endl;
  outData << radius << " " << circumference << " " << area << endl;
 }
```

12.
```
inFile >> int1 >> int2 >> int3;
```

14.

Nivel superior:

Escribir carta
Enviar carta

Escribir carta **Nivel 2**

Escribir dirección de remitente y fecha
Escribir dirección de la empresa
Escribir encabezamiento
Escribir cuerpo de la carta
Escribir saludo
Firmar carta

Enviar carta

Escribir dirección en el sobre
Escribir dirección del remitente
Pegar timbre al sobre
Doblar carta en tres
Insertar carta en el sobre
Sellar sobre
Colocar sobre en el buzón de correo

Capítulo 5 Ejercicios de preparación para examen

2. Falso. Son constantes predefinidas.
4. Porque en la secuencia de comprobación del conjunto de caracteres todas las letras mayúsculas vienen antes de las minúsculas.
6. *a*) verdadero
   b) verdadero
   c) verdadero
   d) verdadero
   e) verdadero
   f) falso
   g) falso
8. Verdadero.
10. Produce `"The data doesn't make sense."`
12. Nada.
14. Produce `Very good`
16. Añade llaves para encerrar la sentencia If-Then.
18. No hay límite, aunque para un lector humano puede ser difícil seguir demasiados niveles.

Capítulo 5 Ejercicios de calentamiento para programación

2.
```
!inFile1 && !inFile2
```
4.
```
if (year1 < year2 && month1 < month2 && day1 < day2)
    cout << month1 << "/" << day1 << "/" year1
         << " va antes de "
         << month2 << "/" << day2 << "/" << year2;
else
    cout << month1 << "/" << day1 << "/" year1
         << " no va antes de "
         << month2 << "/" << day2 << "/" << year2;
```
6.
```
bool1 || bool2 && !(bool1 && bool2)
```
8.
```
if (score < 0 || score > 100)
    cout << "La puntuación está fuera del intervalo";
else
{
    scoreTotal = scoreTotal + score;
    scoreCount++;
}
```
10.
```
if (score > 100)
    cout << "Duffer.";
else if (score > 80)
    cout << "Weekend regular.";
else if (score > 72)
    cout << "Competitive player.";
```

```
else if (score > 68)
    cout << "Turn pro!";
else
    cout << "Time to go on tour!";
```

12. Las pruebas condicionales son el uso de = en lugar de ==. Ambas ramas tienen este error, pero el segundo no produce el mensaje porque el resultado de la expresión de asignación es 0, lo que la rama interpreta como falso.

```
maximum = 75;
minimum = 25;
if (maximum == 100)
    cout << "Error in maximum: " << maximum << endl;
if (minimum == 0)
    cout << "Error in minimum: " << minimum << endl;
```

14. Los valores de temp que se deberán intentar son: 213, 212, 211, 33, 32, 31.

Capítulo 6 Ejercicios de preparación para examen

2. Falso.

4. ***a***) ii, ***b***) ix, ***c***) iv, ***d***) viii, ***e***) vi, ***f***) i, ***g***) v, ***h***) vii, ***i***) iii.

6. Doce veces.

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
```

8. La salida es:

```
     @
     @@
    @@@
    @@@@
   @@@@@
   @@@@@@
  @@@@@@@
  @@@@@@@@
 @@@@@@@@@
 @@@@@@@@@@
@@@@@@@@@@@
```

10. El código no procesa el primer valor en el archivo. El código corregido es:

```
sum = 0;
indata >> number;
while (indata)
{
    sum = sum + number;
    indata >> number;
}
```

12. La salida es:

```
3 5 7 9 11 13 15 17 19
```

El código correcto es:

```
number = 0;
while (number < 10)
{
```

```
    cout << number * 2 - 1 << " ";
    number++;
}
```

14. ¿Cuál es la condición que termina el bucle? ¿Cómo se deberá inicializar la condición? ¿Cómo se deberá actualizar la condición? ¿Cuál es el proceso que se repite? ¿Cómo se deberá inicializar el proceso? ¿Cómo se deberá actualizar el proceso? ¿Cuál es el estado del programa a la salida del bucle?

Capítulo 6 Ejercicios de calentamiento para programación

2.
```
count = 1;
sum = 0;
while (sum <= 10000)
{
    sum = sum + count;
    count++;
}
cout << count - 1;
```

4.
```
count = 0;
getline(chapter6, line);
while (chapter6)
{
    if (line.find("code segment") < string::npos)
      count++;
    getline(chapter6, line);
}
cout << "The string was found " << count << " times."
```

6.
```
cout << "Sun  Mon  Tue  Wed  Thu  Fri  Sat" << endl;
count = 1;
while (count <= startDay)  //Imprimir espacios en blanco en la primera
                             semana
{
    cout << "     ";
    count++;
}
dayNumber = 1;
while (dayNumber <= days)  //Imprimir el número de días para el mes
{
    while (count <= 7)     //Imprimir para una semana en el mes
    {
        if (dayNumber <= days)
            cout << setw(3) << dayNumber << "  ";
        count++;
        dayNumber++;
    }
    cout << endl;
    count = 1;
}
```

8.
```
char oneChar;
int eCount;
int charCount;
eCount = 0;
```

```
textData >> oneChar;
charCount = 0;
while (textData)
{
    charCount++;
    if (oneChar == 'z')
        eCount++;
    textData >> oneChar;
}
cout << "Porcentaje de letra 'z'; "
     << float(eCount) / charCount * 100;
```

10.
```
first = 1;
second = 1;
cout << first << endl << second << endl;
while (second < 30000)
{
    next = first + second;
    first = second;
    second = next;
    cout << second << endl;
}
```

12.
```
cout << "Introducir el número de estrellas: ";
cin >> number;
count = 1;
while (count <= number)
{
    cout << '*';
    count++;
)
cout << endl;
```

14.
```
indata >> number;
while (indata)
{
    count = 1;
    while (count <= number)
    {
        cout << '*';
        count++;
    }
    cout << endl;
    indata >> number;
}
```

Capítulo 7 Ejercicios de preparación para examen

2. Falso. El nombre de parámetro es opcional.

4. ***a***) v, ***b***) iii, ***c***) vi, ***d***) i, ***e***) iv, ***f***) viii, ***g***) ii, ***h***) vii.

6. La llamada debe tener seis argumentos, a menos que se usen parámetros de omisión (en este libro no se estudia su uso, pero se mencionan).

8. El tipo debe venir antes del nombre de cada parámetro en lugar de después. El `&` se deberá añadir al tipo en lugar del nombre de parámetro. El prototipo deberá terminar con un punto y coma.

10. Ocultar una implementación de módulo en un bloque separado con una interfaz formalmente especificada.

12. No deberá contener una sentencia `return`.
14. El resultado no se devuelve al llamador; `result` deberá ser un parámetro de referencia en lugar de un parámetro de valor.
16. Falso. Como cualquier otra declaración de variable, se puede acceder sólo mediante enunciados dentro del bloque que sigue a su declaración.
18. Que el archivo contenga datos válidos (sólo enteros), y que tenga por lo menos un valor (el promedio está indefinido para un conjunto vacío de datos).

Capítulo 7 Ejercicios de calentamiento para programación

2.
```
void Max (int, int, int&);
```
4.
```
void GetLeast (/* inout */ ifstream& infile,
               /* out */   int&      lowest)
```
6.
```
void GetLeast (/* inout */ ifstream& infile,
               /* out */   int&      lowest)
{
    int number;
    infile >> number;       // Obtener un número
    lowest = INT_MAX;       // Usar al inicio el entero máximo
    while (infile)
    {
        if (number < lowest)
            lowest = number;   // Recordar el número menor
        infile >> number;      // Obtener el siguiente número
    }
}
```
8.
```
void Reverse ( /* in */  string  original,
            /* out */ string& lanigiro )
```
10.
```
void Reverse ( /* in */  string  original,
               /* out */ string& lanigiro )
{
    int count;
    count = 1;
    lanigiro = "";
    while (count <= original.length())
    {
        lanigiro = lanigiro +
            original.substr(original.length() - count, 1);
        count++;
    }
)
```
12.
```
void LowerCount ( /* out */ int& count)
{
    char inChar;
    count = 0;
    cin >> inChar;
    while (cin && inChar != '\n')
    {
      if (islower(inChar))
          count++;
      cin >> inChar;
    }
}
```

14.
```
void GetNonemptyLine ( /* inout */ ifstream& infile,
                       /* out */   string&   line)
{
 getline(infile, line);
    while (infile && line == "")
        getline(infile, line);
}
```

16.
```
void TimeAdd ( /* inout */ int& days,
               /* inout */ int& hours,
               /* inout */ int& minutes,
               /* in */    int  addDays,
               /* in */    int  addHours,
               /* in */    int  addMinutes)
{
    int extraHour;
    int extraDay;
    minutes = (minutes + addMinutes) % 60;
    extraHour = (minutes + addMinutes) / 60;
    hours = (hours + addHours + extraHour) % 24;
    extraDay = (hours + addHours + extraHour) / 24;
    days = days + addDays + extraDay;
}
```

18.
```
void SeasonPrint (/* in */ int month,
                  /* in */ int day)
{
    if (month == 12 && day >= 21)
        cout << "Invierno";
    else if ((month == 9 and day >= 21) || month > 9)
        cout << "Otoño";
    else if ((month == 6 and day >= 21) || month > 6)
        cout << "Verano";
    else if ((month == 3 and day >= 21) || month > 3)
        cout << "Primavera";
    else
        cout << "Invierno";
}
```

Capítulo 8 Ejercicios de preparación para examen

2. Falso. Es local para el cuerpo de la función.
4. Verdadero.
6. La combinación de `param2`, que es un parámetro de referencia, y el uso erróneo de una expresión de asignación en lugar de una prueba de igualdad en la primera declaración If ocasiona que al argumento a `param2` se le asigne el valor `param1` cada vez que se llama la función.
8. El espacio de nombre (namespace) está en el espacio global.
10. ***a***) Estático; ***b***) Automático; ***c***) Estático.
12. Se inicializa una vez cuando el control llega por primera vez a la sentencia.
14. Falso.
16. La expresión `return` es del tipo `float`, y el tipo de devolución de la función es `int`.
18. De esta manera, un error de efecto secundario es más probable.

Capítulo 8 Ejercicios de calentamiento para programación

2.
```
bool Equals (/* in */ float x,
             /* in */ float y)
```

4.
```
bool Equals (/* in */ float x,
             /* in */ float y)
{
    return abs(x - y) < 0.00000001;
}
```

6.
```
float ConeVolume (/* in */ float radius,
                  /* in */ float height);
{
    return 1.0/3.0 * 3.14159265 * radius * radius * height;
}
```

8.
```
string Reverse (/* in */  string  original)
{
    string lanigiro;
    int count;
    count = 1;
    lanigiro = "";
    while (count <= original.length())
    {
        lanigiro = lanigiro +
            original.substr(original.length() - count, 1);
        count++;
    }
    return lanigiro;
)
```

10.
```
float SquareKm (/* in */ float length,
                /* in */ float width);
{
    return length * 1.6 * width * 1.6;
}
```

12.
```
string MonthAbbrev (/* in */ int month)
{
    if (month == 12) return "Dec";
    else if (month == 11) return "Nov";
    else if (month == 10) return "Oct";
    else if (month == 9) return "Sep";
    else if (month == 8) return "Aug";
    else if (month == 7) return "Jul";
    else if (month == 6) return "Jun";
    else if (month == 5) return "May";
    else if (month == 4) return "Apr";
    else if (month == 3) return "Mar";
    else if (month == 2) return "Feb";
    else return "Jan";
}
```

14.
```
float RunningAvg (/* in */ float value)
{
    static float total = 0.0;
```

```
    static int count = 0;
    total = total + value;
    count++;
    return total/float(count);
}
```

Capítulo 9 Ejercicios de preparación para examen

2. Falso. Tiene alcance local.
4. Falso. Tanto Break como Continue están permitidos en cualquiera de las sentencias de bucle.
6. El control salta a `case`, que corresponde a la expresión `switch`, pero luego procede a ejecutar las sentencias restantes en todos los `case` sucesivos.
8. 1 000 veces.
10. Una sentencia While sería la mejor opción.
12. `August`
14. `2`
    `1`
    `0`
    `3`
    `2`
    `1`
    `4`
    `3`
    `2`
16. `How now brown cow?`
18. `12`

Capítulo 9 Ejercicios de calentamiento para programación

2.
```
switch (day)
{
    case 0 : cout << "Sunday"; break;
    case 1 : cout << "Monday"; break;
    case 2 : cout << "Tuesday"; break;
    case 3 : cout << "Wednesday"; break;
    case 4 : cout << "Thursday"; break;
    case 5 : cout << "Friday"; break;
    case 6 : cout << "Saturday"; break;
    default: cout << "Error";
}
```

4.
```
for (int day = 6, day >= 0, day- -)
    cout << DayOfWeek(day) << endl;
```

6.
```
char response;
do
{
    cout << "Please enter 'Y' or 'N': ";
    cin >> response;
    if (response != 'Y' && response != 'N')
        cout << "Invalid response. ";
} while (response != 'Y' && response != 'N');
```

8.
```
for (int row = 1; row <= 10; row++)
{
```

```
    for (int col = 1; col <= 10; col++)
        cout << setw(5) << row * col;
    cout << endl;
}
```

10.
```
int line;
int star;
line = 1;
do
{
    star = 1;
    do
    {
        cout << '*';
        star++;
    } while (star <= line);
    cout << endl;
    line++;
} while (line <= 10);
```

12.
```
void Rectangle (/* in */ int height,
                /* in */ int width)
{
    for (int col = 1; col <= width; col++)
        cout << '*';
    cout << endl;
    for (int row = 2; row <= height - 1; row++)
    {
        cout << '*';
        for (int col = 2; col < width - 1; col++)
            cout << ' ';
        cout << '*' << endl;
    }
    for (int col = 1; col <= width; col++)
        cout << '*';
    cout << endl;
}
```

14.
```
int startHour = 3;
int endHour = 7;
int startMin = 15;
int endMin = 30;
int beginMin;
int lastMin;

for (int hour = startHour; hour <= endHour; hour++)
{
    if (hour == startHour)
        beginMin = startMin;
    else
        beginMin = 0;
    if (hour == endHour)
        lastMin = endMin;
    else
        lastMin = 59;
```

```
        for (int min = beginMin; min <= lastMin; min++)
            for (int sec = 0; sec <= 59; sec++)
                cout << hour << ':' << min << ':' << sec << endl;
}
```

Capítulo 10 Ejercicios de preparación para examen

2. Verdadero.
4. Verdadero.
6. `char`, `short`, `int`, `long`, `bool`.
8. Los operadores de asignación.
10. Cuando el nombre de tipo no es una palabra individual, como en `unsigned int`.
12. `'g'`
14. Error absoluto es un valor fijo de error permisible, mientras que error relativo multiplica el error permisible por uno de los valores que se están comparando.
16. El valor de la expresión es 3.
18. Porque es necesario usar el mismo nombre de tipo para argumentos, parámetros y prototipos de función. Si el tipo no tiene nombre, no se puede usar en este contexto.

Capítulo 10 Ejercicios de calentamiento para programación

2. ***a***) 3.14159265F
   b) 3.14159265
   c) 3.14159265L
   d) 3.14159265E0
4. `sizeof(long) * 8`
6.
```
cin >> charIn;
if (isdigit( charIn))
  numIn = charIn - '0';
```
8.
```
int MonthNum (/* in */ string month)
{
    switch (tolower(month[0]))
    {
        case 'a': switch (tolower(month[1]))
                  {
                      case 'p': return 4;
                      case 'u': return 8;
                      default : return 0;
                  }
        case 'd': return 12;
        case 'f': return 2;
        case 'j': switch (tolower(month[3]))
                  {
                      case 'e': return 6;
                      case 'u': return 1;
                      case 'y': return 7;
                      default : return 0;
                  }
        case 'm': switch (tolower(month[2]))
                  {
                      case 'r': return 3;
                      case 'y': return 5;
                      default : return 0;
                  }
```

```
            case 'n': return 11;
            case 'o': return 10;
            case 's': return 9;
            default : return 0;
        }
    }
```

10.
```
abs(balance - audit) <= 0.001
```

12.
```
enum Planets {MERCURY, VENUS, EARTH, MARS,
              JUPITER, SATURN, URANUS, NEPTUNE, PLUTO};
```

14.
```
string PlanetToString (/* in */ Planet sphere)
{
    switch (sphere)
    {
        case MERCURY: return "Mercury";
        case VENUS: return "Venus";
     case EARTH: return "Earth";
     case MARS: return "Mars";
     case JUPITER: return "Jupiter";
     case SATURN: return "Saturn";
     case URANUS: return "Uranus";
     case NEPTUNE: return "Neptune";
     case PLUTO: return "Pluto";
     default : return "Error";
    }
}
```

Capítulo 11 Ejercicios de preparación de examen

2. Verdadero.
4. Verdadero.
6. Asignación, pasando como parámetro, valor de devolución de una función.
8. ***a)***
```
sally.studentName.first = "Sally";
sally.studentName.middle = "Ellen";
sally.studentName.last = "Strong";
```

b)
```
sally.gradeNumber = 7;
```

c)
```
spring = sally.grades[3]
```

10. ***a)*** Hace que el tipo de `grade` sea `char`, e introduce un `char` a `grade`.
b) Compara el valor en `grade` para ver si se encuentra en el rango de `'A'` a `'D'`.
c) Calcula el entero equivalente de la letra `grade`, cambia el tipo de `grade` a `int`, y le asigna el valor numérico.
12. La representación de datos es una manera concreta del dominio, expresada en términos de estructuras y tipos soportados por el lenguaje de programación.
14.
```
current.plus(period);
```
16. Emparejando el número, orden y tipos de los argumentos con la lista de parámetros.
18. Se escribe con una tilde antes del nombre.

Capítulo 11 Ejercicios de calentamiento para programación

2.
```
someTime.minutes = 6;
someTime.seconds = 54;
```

4.
```
Song mySong;

mySong.title = "Long Run for a Collie Dog";
mySong.album = "Timmy's Nightmare"
mySong.artist = "Rebekah MacIntash";
mySong.playTime = someTime;
mySong.type = BLUES;
```

6.
```
union Temporal
{
    string asString;
    int asInteger;
    Time asTime;
}
```

8.
```
//***********************************************
// ARCHIVO DE ESPECIFICACIÓN (Period)
// Este archivo da la especificación de una clase
// para periodos de tiempo geológico
//***********************************************
class Period
{
public:
    enum PeriodName { PRECAMBRIAN, CAMBRIAN, ORDOVICIAN,
                      SILURIAN, DEVONIAN, CARBONIFEROUS,
                      PERMIAN, TRIASSIC, JURASSIC,
                      CRETACEOUS, TERTIARY, QUATERNARY};

    Period();
    // Constructor por omisión
    // Crea el objeto con el valor PRECAMBRIAN

    Period(PeriodName);
    // Constructor
    // Crea el objeto con el valor especificado

    String ToString() const;
    // Observador
    // Devuelve el nombre del objeto como una cadena

    int ToInt() const;
    // Observador
    // Devuelve el valor del objeto como un int, 0 = PRECAMBRIAN

    float StartDate() const;
    // Observador
    // Devuelve la fecha de inicio del periodo en millones de años atrás

    void Increment();
    // Transformador
    // Adelanta el valor del objeto al siguiente periodo más reciente
    // No avanzará más allá de QUATERNARY

private:

    PeriodName thisPeriod;

}
```

10.
```
Period::Period()
// Constructor por omisión
// Crea el objeto con el valor PRECAMBRIAN
{
    PeriodName = PRECAMBRIAN;
}

Period::Period(/* in */ PeriodName aPeriod)
// Constructor
// Crea el objeto con el valor especificado
{
    thisPeriod = aPeriod;
}
```

12. El archivo se llamaría `Period.h`.

14.
```
//****************************************
// ARCHIVO DE ESPECIFICACIÓN (Money)
// Este archivo da la especificación de una
// clase para representar dinero como dólares
// y centavos usando enteros.
//****************************************

class Money
{
public:

    Money();
    // Constructor por omisión
    // Crea un objeto con valor $0.00

    Money(int, int);
    // Constructor
    // Crea el objeto con el valor especificado
    // en dólares y centavos

    int Dollars() const;
    // Observador
    // Devuelve la porción en dólares del objeto como un entero

    int Cents() const;
    // Observador
    // Devuelve la porción en centavos del objeto como un entero

    float Amount() const;
    // Observador
    // Devuelve el valor del dinero como un float

    void Add(Money);
    // Transformador
    // Añade el valor del objeto dinero en el parámetro

    void Sub(Money);
    // Transformador
    // Resta el valor del objeto dinero en el parámetro

private:
```

```
    int dollars;
    int cents;
}
```

16. El nombre del archivo sería `Money.h`.
18.
```
float Money::Amount() const
// Observador
// Devuelve el valor del dinero como un float
{
    return float(dollars) + float(cents)/100.0;
}
```

Capítulo 12 Ejercicios de preparación para examen

2. Verdadero.
4. Verdadero.
6. C++ no reporta un acceso fuera de límite como error. El programa accede la ubicación que precede al primer elemento del arreglo, que podrá ser un valor sin ninguna relación en la memoria.
8. La dirección base es la ubicación de memoria del primer elemento de un arreglo. Es el valor que se pasa a un parámetro de referencia en una llamada de función.
10. El bucle interior incrementaría el índice de columna, manteniendo el renglón constante. El bucle exterior incrementaría el índice de renglón para desplazarse al siguiente renglón después que se procesa cada renglón.
12. El bucle For corre de 1 a 100 en lugar de 0 a 99. La última iteración asignará cero a una ubicación fuera de límite.
14. Las funciones no pueden devolver tipos de arreglo.
16. Porque el valor de índice se introduce y luego se usa sin prueba alguna, será probable un acceso de arreglo.
18.
```
 * **
* *
 * **
* *
 * **
```
20.
```
* * *
 * *
* * *
 * *
```

Capítulo 12 Ejercicios de calentamiento para programación

2.
```
typedef float DataSet[5];
DataSet input;
DataSet output;
DataSet working;
```

4.
```
for (int row = 0; row < 3; row++)
    for (int col = 0; col < 5; col++)
        set[row][col] = 0.0;
```

6.
```
bool Equals (/* in */ const DataSet first,
             /* in */ const DataSet second)
{
    for(int index = 0; index < 5; index++)
        if (first[index] != second[index])
            return false;
```

```
    // Sólo llega al final del bucle si todos son iguales
    return true;
}
```

8. *a)* `currentList[36]`
   b) `currentList[11].title`
   c) `currentList[84].size.width`
   d) `currentList[119].room`
   e) `currentList[77].artist[0]`

10.
```
float total = 0.0;
for (int index = 0; index < numPieces; index++)
    total = total + currentList[index].price;
```

12.
```
float total = 0.0;
for (int index = 0; index < numPieces; index++)
    if(currentList[index].medium == OIL)
        if (currentList[index].size.width *
              currentList[index].size.height > 400)
        total = total + currentList[index].price;
```

14.
```
for (int octave = 0; octave < 8; octave++)
    for (Notes note = A; note <= GSHARP; note = Notes(note  + 1))
        cin >> scale[octave][note];
```

16.
```
for (int octave = 0; octave < 8; octave++)
    cout << scale[octave][C] << endl;
```

18.
```
void Reset (float humidity[10][52][50][3])
{
  for (int year = 0; year < 10; year++)
      for (int week = 0; week < 52; week++)
          for (int state = 0; state < 50; state++)
              for (Type type = MAX; type <= AVERAGE;
                   type = Type(type + 1));
                  humidity = 0.0;
}
```

20.
```
for (int year = 5; year < 10; year++)
    for (int week = 0; week < 52; week++)
        cout << humidity[year][week][22][AVERAGE] << endl;
```

Capítulo 13 Ejercicios de preparación para examen

2. Que todos los componentes tengan el mismo tipo.
4. `while (index < length && fabs(item –data[index]) >= EPSILON)`
   Es necesario cambiarlo porque números de punto flotante no pueden ser confiablemente comparados para una igualdad exacta.
6. Ha identificado el valor más pequeño en la parte de la lista que falta por ordenar.
8. Verdadero.
10. 5
12. Dos iteraciones. El medio es inicialmente 7, luego la ubicación 11 (que guarda el decimosegundo valor) se convierte en medio.

Capítulo 13 Ejercicios de calentamiento para programación

2.
```
void List::Insert( /* in */ ItemType item)

// Inserta el elemento en la lista sin duplicación

{
    if (!IsPresent(item))
    {
        data[length] = item;
        length++;
    }
}
```

4.
```
void DeleteAll( /* in */ ItemType item);
    // Precondición:
    //     NOT IsEmpty()
    //     && se asigna el elemento
    // Poscondición:
//      IF el elemento está presente en la lista
//         Todas las apariciones del elemento se borran de la lista
//         && Length() ==
//           Length()@entry - número de apariciones del elemento
//      ELSE
//         La lista permanece sin cambio
```

6.
```
void Replace( /* in */ ItemType item,
              /* in */ ItemType newItem);
// Precondición:
//      NOT IsEmpty()
//      && se asigna el oldItem
//      && se asigna el newItem
// Poscondición:
//      IF oldItem está presente en la lista
//        La primera aparición del oldItem se remplaza por el newItem
//      ELSE
//        La lista permanece sin cambio
```

8. Se deben cambiar las funciones `Insert` y `BinSearch` para habilitar el orden descendente. No es necesario cambiar `IsPresent` o `Delete` porque ambos usan `BinSearch` para localizar el ítem.

10.
```
void SortedList::DeleteAll( /* in */ ItemType item)

// Borra todas las apariciones del elemento en la lista

{
    int first;
    int index = 0;
    while (index < length && item != data[index])
        index++;
    first = index;
    while (index < length && item == data[index])
        index++;
    for (int position = index; position < length; position++)
        data[first + position - index] = data[position];
    length = length - (index - first);
}
```

12.
```
SortedList inData;
ifstream unsorted;
ItemType oneItem;

unsorted >> oneItem;
while (unsorted && !inData.IsFull())
{
    inData.Insert(oneItem);
    unsorted >> oneItem;
}
```

Capítulo 14 Ejercicios de preparación para examen

2. Verdadero.
4. Verdadero.
6. Declaración de un miembro de datos de clase para ser un objeto de la clase que se desea componer a ser la nueva clase que se define.
8. La declaración de clase para `DerivedClass` no especifica que hará los miembros de `BaseClass` públicos, así que no están disponibles para el código de cliente. El encabezado deberá cambiar a:

```
class DerivedClass : public BaseClass
```

10. ***a***) El constructor `BaseClass` es llamado primero.
b) El constructor `DerivedClass` es llamado al último.
12. La designación `virtual` sólo se deberá usar con la declaración de clase base de `BaseAlpha`.
14. c.

Capítulo 14 Ejercicios de calentamiento para programación

2.
```
#include "testscore.h"
#include <iostream>
#include <string>

using namespace std;

TestScore::TestScore(
    /* in */ string name,
    /* in */ int score)
{
    studentName = name;
    studentScore = score;
}

TestScore::string GetName() const
{
    return studentName;
}

TestScore::int GetScore() const
{
    return studentScore;
}
```

4.
```
#include "idscore.h"

class Exam
{
public:
  SetScore(
      /* in */ int location,
      /* in */ IDScore score );
  IDScore GetScore(
      /* in */ int location ) const;

private:
    IDScore examList[100];
}
```

6.
```
class InternPhone : public Phone
{
public
    InternPhone
        ( /* in */ int newCountry,
          /* in */ int newAreaCode,
          /* in */ int newNumber,
          /* in */ PhoneType newType );
    void Write () const;
private:
    int country;
}
```

8.
```
void InternPhone::Write() const
{
    cout << country;
    Phone::Write();
    cout << endl;
}
```

10.
```
class InstallRecord : public Computer
{
public:
    InstallRecord(
        /* in */ string newName,
        /* in */ string newBrand,
        /* in */ string newModel,
        /* in */ int    newSpeed,
        /* in */ string newSerial,
        /* in */ int    newNumber,
        /* in */ string newLocation,
        /* in */ SimpleDate newDate );
    void Write() const;
    string getNewLocation();
    SimpleDate getSimpleDate();

private:
    string location;
    SimpleDate installDate;
};
```

12.
```
InstallRecord::InstallRecord(
    /* in */ string newName,
    /* in */ string newBrand,
    /* in */ string newModel,
    /* in */ int    newSpeed,
    /* in */ string newSerial,
    /* in */ int    newNumber,
    /* in */ string newLocation,
    /* in */ SimpleDate newDate )
  : Computer( newName, newBrand, newModel,
              newSpeed, newSerial, newNumber )
{
    location = newLocation;
    installDate = newDate;
}
```

14.
```
void InstallRecord::Write() const;
{
    Computer::Write();
    cout << location << endl;
    installDate.Write();
}
```

Capítulo 15 Ejercicios de preparación para examen

2. ***a***) ii, ***b***) i, ***c***) iii, ***d***) v, ***e***) vi, ***f***) iv.
4. Verdadero.
6. Verdadero.
8. El (los) apuntador(es) se borra(n) pero no los datos a que se apunta. De esta manera se hace inaccesible y contribuye a la fuga de memoria.
10. Crea y referencia un apuntador suspendido (dangling pointer) en `germanShortHair`.
12. b.
14. `faxLog`
16. Un destructor.
18. `*libraryRecord`

Capítulo 15 Ejercicios de calentamiento para programación

2.
```
char* charArrPointer;
char[4] initials;
charArrPointer = initials;
charArrPointer[1] = 'A';
charArrPointer[2] = 'E';
charArrPointer[3] = 'W';
```

4.
```
struct Phone
{
    int country;
    int area;
    int number;
}
Phone newPhone;
Phone& structReference = newPhone;
structReference.country = 1;
structReference.area = 888;
structReference.number = 5551212;
```

6.
```
bool DeepCompare
  ( /* in */ structPointer first,
    /* in */ structPointer second )
{
    return (first->country == second->country &&
            first->area    == second->area &&
            first->number  == second->number);
}
```

8.
```
int Greatest(
  ( /* inout */ int data[],
    /* in */    int size )
{
    static max = data[0];
    for(int index = 1; index < size; index++)
        if (data[index] > max)
            max = data[index];
    delete [] data;
    return max;
}
```

10.
```
~Circuit()
{
    delete [] source;
    delete [] sink;
}
```

12.
```
if (oldValue != newValue)
 delete oldValue;
```

Capítulo 16 Ejercicios de preparación para examen

2. Falso.

4. ***a***) La dirección del siguiente nodo de la lista.
b) Los datos del nodo actual.
c) Los datos del siguiente nodo de la lista.
d) La dirección del nodo después del siguiente nodo.
e) Los datos del nodo después del siguiente nodo.
f) Los datos del cuarto nodo de la lista.

6. ***a***) Una lista vacía. ***b***) Que el procesamiento ha llegado al final de la lista.

8. Fije un apuntador temporal igual al campo de enlace del nodo actual, fije el campo de enlace del nodo actual igual al campo de enlace de su sucesor, y borre los datos a los que apunta el apuntador temporal.

10. Una representación directa de arreglo mantiene los elementos físicamente colindantes en el arreglo. Una lista ligada implementada con un arreglo permite que los elementos estén en cualquier orden físico en el arreglo y los conecta de manera lógica, usando un campo de enlace en cada elemento.

12. ***a***) Un array; la lista es pequeña y limitada, y con frecuencia se realizan búsquedas.
b) Una lista ligada; la lista varía de manera impredecible en cuanto a su tamaño y no requiere búsquedas frecuentes.
c) Una lista ligada; la lista es ilimitada, y el borrado está en la cabeza.

Capítulo 16 Ejercicios de calentamiento para programación

2. ***a)*** `head->component`
   b) `currPtr = currPtr->link`
   c) `(currPtr->link)->link`
   d) `((currPtr->link)->link)->link`

4.
```
newNodePtr = new NodeType;
newNodePtr->component = 212;
currPtr->link = newNodePtr;
currPtr = newNodePtr;
```

6.
```
NodePtr savePtr;
savePtr = head;
head = currPtr->link;
head->link = savePtr;
currPtr->link = currPtr->link->link;
```

8.
```
auxPtr = currPtr;
if (auxPtr != NULL)
    if (auxPtr->link == NULL)
        auxPtr = NULL;
    else
    {
        while (auxPtr->link != NULL)
            auxPtr= currPtr->link;
        auxPtr->NULL;
    }
```

10.
```
CopyReverse(NodeType* head, NodeType*& headR)
{
    headR = NULL;     // encabezado de la nueva lista

    NodePtr newNode;
    NodePtr tempPtr = head;          // empleado para pasaje
    while (tempPtr != NULL)
    {
        newNode = new NodeType;
        newNode->component = tempPtr->component;

        newNode->link = headR;
        headR = newNode;
        tempPtr = tempPtr->link
    }
}
```

Capítulo 17 Ejercicios de preparación para examen

2. Falso.
4. ***a)*** Clase de plantilla; ***b)*** Ninguno de estos (es un objeto de clase).
6. ***a)*** `float`, ***b)*** 3.85, ***c)*** Sí.
8. La sentencia Throw.
10. ***a)*** Falso. Puede haber muchas cláusulas de captura, mientras que sus parámetros sean diferentes.
    b) Verdadero.

c) Falso. Una sentencia Throw puede aparecer en cualquier parte. Una sentencia Throw con frecuencia se ubica en una función llamada desde una cláusula Try.

12. En la cláusula Catch, inserte `return 1;` después la sentencia de salida.

14. Relanza la excepción.

Capítulo 17 Ejercicios de calentamiento para programación

2.
```
PrintSquare<int>(10);
PrintSquare<long>(10L);
PrintSquare<float>(10.0);
```

o bien

```
PrintSquare(10);
PrintSquare(10L);
PrintSquare(10.0);
```

4. *a)*
```
int Twice( int num )
{
   return 2*num;
}
float Twice( float num )
{
    return 2.0*num;
}
```

b)
```
cout << Twice(someInt) << ' ' << Twice(someFloat) << endl;
```

6.
```
template<class SomeType>
void GetData( string promptStr, SomeType& data )
{
    cout << promptStr << ' ';
    cin >> data;
}
```

8. *a)*
```
GList<int>   intList;
GList<float> floatList;
```

b)
```
i = 10;
while (i <= 80 && !intList.IsEmpty())
{
    // A continuación, use "while", no "if", porque
    // se permiten duplicados en la lista

    while (intList.IsPresent(i))
    {
        intList.Delete(i);
        floatList.Insert(0.5 * float(i));
    }
    i++;
}
```

10.
```
MixedPair<int, float>  pair1(5, 29.48);
MixedPair<string, int> pair2("Book", 36);
```

12. ***a)***
```
class MathError
{};                    // No olvide el punto y coma
```
b)
```
throw MathError();     // No olvide los paréntesis
```

14.
```
class SumTooLarge   // Clase de excepción
{};
 :
int Sum( int int1, int int2 )
{
    if (int1 > INT_MAX - int2)
        throw SumTooLarge();
    return int1 + int2;
}
```

16. ***a)***
```
class OpenFailed   // Clase de excepción
{};

void OpenForInput( /* inout */ ifstream& someFile )

// Solicita al usuario el nombre de un archivo de entrada
// e intenta abrir el archivo

// Poscondición:
//     Se ha solicitado al usuario un nombre de archivo
//  && IF no se pudiera abrir el archivo
//           Se imprimiría un mensaje de error
//       && Se ha lanzado una excepción OpenFailed

{
    string fileName;    // Nombre de archivo especificado por el usuario

    cout << "Input file name: ";
    cin >> fileName;

    someFile.open(fileName.c_str());
    if ( !someFile )
    {
        cout << "** Can't open " << fileName << " **" << endl;
        throw OpenFailed();
    }
}
```

b) Observe que el código de llamada que sigue a continuación no imprime ningún mensaje de error. El mensaje de error ya ha sido impreso por `OpenForInput`, como queda avisado por su poscondición.

```
ifstream inFile;

try
{
    OpenForInput(inFile);
}
catch ( OpenFailed )
{
    return 1;
}
// Éxito en mantener la ejecución
```

Capítulo 18 Ejercicios de preparación para examen

2. Verdadero.
4. Falso. La recursión de extremo ocurre cuando no se hace nada después del retorno de la llamada recursiva.
6. F(3) = 8, F(4) = 16, F(5) = 32.
8. Se le acabará el espacio en la pila en tiempo de ejecución.
10. c. El valor de la variable.
12.
```
30
20
10
```
14.
```
abcc
d
e
```

Capítulo 18 Ejercicios de calentamiento para programación

2.
```
int OneDigit (int number)
{
    if (number <= 9)
        return number;
    else
        return OneDigit(DigitSum(number));
}
```

4.
```
void Ex4()
{
    int number;
    cout << "Introducir un número positivo, 0 al final: ";
    cin >> number;

    if (number != 0)
    {
        cout << number << endl;
        Ex4();
        cout << number << endl;
    }
}
```

6.
```
void Ex6(int sum, int& revSum)
{
    int number;
    cout << "Introducir un número positivo, 0 al final; ";
    cin >> number;

    if (number != 0)
    {
        cout << "Total: " << sum + number <<endl;
        Ex6(sum + number, revSum);
        revSum = revSum + number;
        cout << number << " Total: " << revSum << endl;
    }
}
```

8.
```
void Ex8(int& greatest)
{
    int number;
    cout << "Introducir un número positivo, 0 al final: ";
    cin >> number;

    if (number > greatest)
        greatest = number;

    if (number != 0)
    {
        cout << "Greatest: " << greatest << endl;
        Ex8(greatest);
        cout << number << endl;
    }

}
```

El más grande se debe escribir inmediatamente después de la llamada.

10.
```
void CopyReverse(PtrType head1, PtrType& head2)
// Suposición: head2 es originalmente NULL
{
    PtrType tempPtr;
    if (head1 != NULL)
    {
        tempPtr = new NodeType;
        tempPtr->info = head1->info;
        tempPtr->link = head2;
        head2 = tempPtr;
        CopyReverse(head1->link, head2);

    }
}
```

12.
```
void CopyDouble(PtrType head1, PtrType& head2)
// Suposición:   head2 es originalmente NULL
{

    PtrType tempPtr;
    if (head1 != NULL)
    {
        tempPtr = new NodeType;
        tempPtr->info = head1->info;
        tempPtr->link = head2;
        head2 = tempPtr;
        CopyReverse(head1->link, head2);
        if (head1 != NULL)
       {
            tempPtr = new NodeType;
            tempPtr->info = head1->info;
            tempPtr->link = head2;
            head2 = tempPtr;
            Copy(head1->link, head2->link);
```

```
        }
    }
}
```

o bien

```
void CopyDouble(PtrType head1, PtrType& head2)
{
    CopyReverse(head1, head2);
    Copy(head1, head2);
}
```

ÍNDICE

B

C

D

E

F

R

S

T

U

V

W

X