C++

CÓMO PROGRAMAR

NOVENA EDICIÓN

PAUL DEITEL
HARVEY DEITEL

ALWAYS LEARNING

PEARSON

ACCESO A LOS CAPÍTULOS ADICIONALES DEL LIBRO

Para acceder a los capítulos 14 a 17 (en español) y 18 a 23 (en inglés) mencionados en el texto, visite el sitio web del libro:

www.pearsonenespañol.com/deitel

Utilice una moneda para descubrir el código de acceso.
(No use objetos filosos porque podría dañarlo).

Deitel / Cómo programar en C++

IMPORTANTE:

¡Este código de acceso tiene vigencia de 2 días!
Asegúrese que el código no aparezca dañado ya que sólo puede usarse una vez y no será reemplazado en ningún caso.

C++

CÓMO PROGRAMAR

NOVENA EDICIÓN

Paul Deitel
Deitel & Associates, Inc.

Harvey Deitel
Deitel & Associates, Inc.

TRADUCCIÓN
Alfonso Vidal Romero Elizondo
Ingeniero en Sistemas Electrónicos
Tecnológico de Monterrey, Campus Monterrey

REVISIÓN TÉCNICA
Sergio Fuenlabrada Velázquez
Edna Martha Miranda Chávez
José Luis López Goytia
Judith Sonck Ledezma
Mario Alberto Sesma Martínez
Mario Oviedo Galdeano
Oskar Armando Gómez Coronel
Academia de Computación
Unidad Profesional Interdisciplinaria de Ingeniería
y Ciencias Sociales y Administrativas (UPIICSA)
Instituto Politécnico Nacional, México

Sandra Luz Gómez Coronel
Academia de Electrónica
Unidad Profesional Interdisciplinaria de Ingeniería
y Tecnologías Avanzadas (UPIITA)
Instituto Politécnico Nacional, México

PEARSON

Datos de catalogación bibliográfica

DEITEL, PAUL y HARVEY DEITEL

Cómo programar en C++
Novena edición

PEARSON EDUCACIÓN, México, 2014

ISBN: 978-607-32-2739-1

Formato: 20 × 25.5 cm Páginas: 720

Edición en español

Dirección General:	Philip de la Vega
Dirección Educación Superior:	Santiago Gutiérrez
Editor Sponsor:	Luis M. Cruz Castillo
	luis.cruz@pearson.com
Editor de Desarrollo:	Bernardino Gutiérrez Hernández
Supervisor de Producción:	Enrique Trejo Hernández
Gerencia Editorial Educación Superior:	Marisa de Anta

Novena edición, 2014

D.R. © 2014 por Pearson Educación de México, S.A. de C.V.
Atlacomulco 500-5o. piso
Col. Industrial Atoto
53519, Naucalpan de Juárez, Estado de México

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

ISBN 978-607-32-2739-1
ISBN e-book 978-607-32-2740-7
ISBN e-chapter 978-607-32-2741-4

Impreso en México. *Printed in Mexico*.
1 2 3 4 5 6 7 8 9 0 - 17 16 15 14

PEARSON

www.pearsonenespañol.com **ISBN 978-607-32-2739-1**

En memoria de Dennis Ritchie,
creador del lenguaje de programación C,
uno de los lenguajes clave que inspiraron a C++.

Paul y Harvey Deitel

Marcas registradas

Contenido

Los capítulos 15, 16 y 17
se encuentran, en español en el sitio web del libro

Los capítulos 18, 19, 20, 21, 22 y 23 se encuentran, en inglés en el sitio web del libro

Prefacio

"El principal mérito del lenguaje es la claridad…"
—Galen

Bienvenido al lenguaje de programación C++ y a *Cómo programar en C++, novena edición*. Este libro, que presenta las tecnologías de computación de vanguardia, es apropiado para secuencias de cursos introductorios basados en las recomendaciones curriculares de dos organizaciones profesionales clave: la ACM y el IEEE.

La base del libro es el reconocido *método de código activo* de Deitel: los conceptos se presentan en el contexto de programas funcionales completos seguidos de ejecuciones de ejemplo, en lugar de hacerlo a través de fragmentos separados de código. En la sección Antes de empezar (vea el sitio web de este libro) encontrará información para configurar su computadora basada en Linux, Windows o Apple OS X para ejecutar los ejemplos de código. Todo el código fuente está disponible en el sitio web de este libro (si requiere los códigos en inglés, puede obtenerlos de: `www.deitel.com/books/cpphtp9` y en `www.pearsonhighered.com/deitel`). Use el código fuente que le proporcionamos para *ejecutar cada programa* a medida que lo vaya estudiando.

Creemos que este libro y sus materiales de apoyo le brindarán una introducción informativa, desafiante y entretenida a C++. Si surge alguna duda o pregunta a medida que lea este libro, puede comunicarse con nosotros en `deitel@deitel.com`; le responderemos a la brevedad. Síganos en Facebook (`www deitel.com/deitelfan`) y Twitter (`@deitel`); también puede suscribirse al boletín de correo electrónico *Deitel® Buzz Online* (`www.deitel.com/newsletter/subscribe.html`).

Estándar C++ 11

El nuevo estándar C++11 que se publicó en 2011 nos motivó a escribir esta nueva edición de *Cómo programar en C++*. A lo largo del libro, cada una de las nuevas características de C++11 se identifica con el icono "11" como se muestra aquí en el margen. A continuación le presentamos algunas de las características clave de C++11 en esta nueva edición:

- ***Cumple con el nuevo estándar C++11.*** Extensa cobertura de las nuevas características de C++11 (figura 1).
- ***El código se probó de manera exhaustiva en tres compiladores de C++ populares a nivel industrial.*** Probamos los ejemplos de código en GNU™ C++ 4.7, Microsoft® Visual C++® 2012 y Apple® LLVM en Xcode® 4.5.
- ***Apuntadores inteligentes.*** Los apuntadores inteligentes nos ayudan a evitar errores de administración de memoria dinámica al proveer una funcionalidad adicional a la de los apuntadores integrados. En el capítulo 17 (en el sitio web) hablaremos sobre `unique_ptr`.

Características de C++11 en *Cómo programar en C++, novena edición*		
Algoritmo `all_of`	Heredar constructores de la clase base	Generación de números aleatorios no deterministicos
Algoritmo `any_of`	Funciones miembro de contenedor `insert` devuelven iteradores	Algoritmo `none_of`
Contenedor `array`	Algoritmo `is_heap`	Funciones de conversión numérica
`auto` para inferencia de tipos	Algoritmo `is_heap_until`	`nullptr`
Funciones `begin/end`	Palabras clave nuevas en C++11	Palabra clave `override`
Funciones miembro de contenedor `cbegin/cend`	Expresiones lambda	Instrucción `for` basada en rangos
Compilador fix for >> en tipos de plantillas	Inicialización de listas de pares clave-valor	Expresiones regulares
Algoritmo `copy_if`	Inicialización de listas de objetos `pair`	Referencias rvalue
Algoritmo `copy_n`	Inicialización de listas de valores de retorno	Enumeraciones (`enum`) con alcance
Funciones miembro de contenedor `crbegin/cend`	Lista inicializadora de un arreglo asignado en forma dinámica	Apuntador inteligente `shared_ptr`
`decltype`	Lista inicializadora de un `vector`	Función miembro `shrink_to_fit` `vector/deque`
Argumentos de tipo predeterminado en plantillas de funciones	Inicializadores de listas en llamadas a constructores	Especificar el tipo de las constantes de una enumeración
Funciones miembro predeterminadas (`default`)	Tipo `long long inte`	Objetos `static_assert` para nombres de archivo
Delegación de constructores	Algoritmos `min` y `max` con parámetros `initializer_list`	Objetos `string` para nombres de archivo
Funciones miembro eliminadas (`delete`)	Algoritmo `minmax`	Función no miembro `swap`
Operadores de conversión `explicit`	Algoritmo `minmax_element`	Funciones `for` con tipos de retorno al final
Clases `final`	Algoritmo `move`	Plantilla variádica `tuple`
Funciones miembro `final`	Operadores de asignación de movimiento	Apuntador inteligente `unique_ptr`
Algoritmo `find_if_not`	Algoritmo `move_backward`	`long long int` sin signo
Contenedor `forward_list`	Constructores de movimiento	Apuntador inteligente `weak_ptr`
Claves inmutables en contenedores asociativos	`noexcept`	
Inicializadores en la clase		

Fig. 1 | Una muestra de las características de C++11 en *Cómo programar en C++, novena edición*.

- ***Cobertura anticipada de los contenedores, iteradores y algoritmos de la Biblioteca Estándar, mejorada con capacidades de C++11.*** Transferimos el tratamiento de los contenedores, iteradores y algoritmos de la Biblioteca de plantillas Estándar del capítulo 22, en la edición anterior, a los capítulos 15 y 16, y lo mejoramos con las características adicionales de C++11. La gran mayoría de las necesidades de los programadores en cuanto a estructuras de datos pueden satisfacerse mediante la *reutilización* de estas capacidades de la Biblioteca Estándar. En el capítulo 19 (en inglés en el sitio web) le indicaremos cómo crear sus propias estructuras de datos *personalizadas*.

- ***Generación de números aleatorios, simulación y juegos.*** Para ayudar a que los programas sean más seguros, agregamos un tratamiento de las nuevas herramientas de generación de números aleatorios no deterministas.

Programación orientada a objetos

- ***Metodología de presentación de objetos en los primeros capítulos.*** El libro presenta los conceptos básicos y la terminología de la tecnología de objetos en el capítulo 1. Usted desarrollará sus primeras clases y objetos personalizados en el capítulo 3; de esta manera hacemos que usted "piense acerca de los objetos" de inmediato y domine estos conceptos con más profundidad.[1]

- ***Objetos `string` de la Biblioteca Estándar de C++.*** C++ ofrece *dos* tipos de cadenas: los objetos de la clase `string` (que comenzaremos a usar en el capítulo 3) y las cadenas de C. Reemplazamos la mayoría de las ocurrencias de las cadenas de C con instancias de la clase `string` de C++ para que los programas sean más robustos y para eliminar muchos de los problemas de seguridad de las cadenas de C. Seguiremos hablando sobre las cadenas de C más adelante en el libro para preparar al lector a trabajar con el código heredado que encontrará en la industria. En cuanto al desarrollo de nuevos programas, es preferible usar objetos `string`.

- ***Objetos `array` de la Biblioteca Estándar de C++.*** Nuestro principal tratamiento de los arreglos usa ahora la plantilla de la clase `array` de la Biblioteca Estándar en vez de los arreglos integrados estilo C, basados en apuntadores. De todas formas cubriremos los arreglos integrados ya que siguen siendo útiles en C++ y para que el lector pueda leer el código heredado. C++ ofrece *tres* tipos de arreglos: objetos `array` y `vector` (que comenzaremos a usar en el capítulo 7) y arreglos estilo C basados en apuntadores, que veremos en el capítulo 8. Según sea apropiado, usaremos la plantilla de clase `array` en vez de arreglos de C a lo largo del libro. En cuanto al desarrollo de nuevos programas, es preferible usar objetos de la plantilla de clase `array`.

- ***Creación de clases valiosas.*** Un objetivo clave de este libro es preparar al lector para crear clases valiosas. En el ejemplo práctico del capítulo 10, usted creará su propia clase `Array` personalizada (y luego en los ejercicios del capítulo 18, en inglés, la convertirá en una plantilla de clase). Aquí es donde apreciará en verdad el concepto de las clases. El capítulo 10 comienza con una prueba práctica de la plantilla de clase `string`, de modo que pueda ver un uso elegante de la sobrecarga de operadores antes de implementar su propia clase personalizada con operadores sobrecargados.

- ***Ejemplos prácticos en programación orientada a objetos.*** Ofrecemos ejemplos prácticos que abarcan varias secciones y capítulos, además de cubrir el ciclo de vida de desarrollo de software. Entre éstos se incluye la clase `LibroCalificaciones` en los capítulos 3 a 7, la clase `Tiempo` en el capítulo 9 y la clase `Empleado` en los capítulos 11 y 12. El capítulo 12 contiene un diagrama detallado y una explicación de cómo puede C++ implementar internamente el polimorfismo, las funciones `virtual` y la vinculación dinámica.

- ***Ejemplo práctico opcional: uso de UML para desarrollar un diseño orientado a objetos y la implementación en C++ de un ATM.*** El UML™ (Lenguaje unificado de modelado™) es el lenguaje gráfico estándar de la industria para los sistemas de modelado orientados a objetos. Presentamos el UML en los primeros capítulos. Diseñamos e implementamos el software para un cajero automático simple (ATM). Analizamos un documento de requerimientos típico que

1 En los cursos que requieren una metodología en la que se presenten los objetos en capítulos posteriores, le recomendamos el libro *C++ How to Program, Late Objects* (versión en inglés), el cual empieza con los primeros seis capítulos sobre los fundamentos de la programación (incluyendo dos sobre instrucciones de control) y continúa con siete capítulos que introducen los conceptos de programación orientada a objetos en forma gradual.

especifica el sistema a construir. Determinamos las clases necesarias para implementar ese sistema, los atributos que necesitan tener las clases, los comportamientos que deben exhibir y especificamos la forma en que deben interactuar esas clases entre sí para cumplir con los requerimientos del sistema. A partir del diseño producimos una implementación completa en C++. A menudo los estudiantes reportan que el ejemplo práctico les ayuda a "enlazarlo todo" y comprender en verdad la orientación a objetos.

- ***Manejo de excepciones.*** Integramos el manejo básico de excepciones *en los primeros capítulos* del libro. Los instructores pueden extraer con facilidad más material del capítulo 17, Manejo de excepciones: un análisis detallado.
- ***Estructuras de datos basadas en plantillas personalizadas.*** Ofrecemos un tratamiento extenso de varios capítulos sobre las estructuras de datos: consulte el módulo Estructuras de datos en el gráfico de dependencia de los capítulos (figura 6).
- ***Tres paradigmas de programación.*** Hablamos sobre la *programación estructurada*, la *programación orientada a objetos* y la *programación genérica*.

Características pedagógicas

- ***Extensa cobertura de los fundamentos de C++.*** Incluimos un tratamiento conciso de dos capítulos sobre las instrucciones de control y el desarrollo de algoritmos.
- ***El capítulo 2 ofrece una introducción simple a la programación en C++.***
- ***Ejemplos:*** incluimos un amplio rango de programas de ejemplo seleccionados de temas como ciencias computacionales, negocios, simulación, juegos y demás (figura 2).

Ejemplos	
Ejemplo práctico de la clase `Array`	Simulación del juego de dados "craps"
Clase `Autor`	Programa de consulta de crédito
Programa de cuenta bancaria	Clase `Fecha`
Programa para imprimir gráficos de barras	Conversión de tipos descendente (downcasting) e información de tipos en tiempo de ejecución
Clase `EmpleadoBaseMasComision`	Clase `Empleado`
Creación y recorrido de árboles binarios	Constructor `explicit`
Programa de prueba `BusquedaBinaria`	Función `fibonacci`
Barajar y repartir cartas	Algoritmos `fill`
Clase `DatosCliente`	Especializaciones de plantillas de funciones de la plantilla de función `imprimirArreglo`
Clase `EmpleadoPorComision`	Algoritmos `generate`
Comparación de objetos `string`	Clase `LibroCalificaciones`
Proceso de compilación y vinculación	Inicialización de un arreglo en una declaración
Cálculos del interés compuesto con `for`	Entrada a partir de un objeto `istringstream`
Conversión de objetos `string` a cadenas de C	Solución de factorial iterativa
Repetición controlada por contador	

Fig. 2 | Una muestra de los ejemplos del libro (parte 1 de 2).

Ejemplos	
Expresiones lambda	Clase `Vendedor`
Manipulación de listas enlazadas	Algoritmos de búsqueda y ordenamiento de la Biblioteca Estándar
Plantilla de clase `map`	Archivos secuenciales
Algoritmos matemáticos de la Biblioteca Estándar	Plantilla de clase `set`
Plantilla de función `maximum`	Programa `shared_ptr`
Programa de ordenamiento por combinación	Clase de adaptador `stack`
Plantilla de clase `multiset`	Clase `Stack`
`new` lanza `bad_alloc` durante una falla	Desenredo de pila (stack unwinding)
Clase `NumeroTelefonico`	Programa de la clase `string` de la Biblioteca Estándar
Programa de análisis de encuestas	Manipulador de flujos `showbase`
Demostración del polimorfismo	Asignación y concatenación de objetos `string`
Preincremento y postincremento	Función miembro `substr` de `string`
Clase de adaptador `priority_queue`	Suma de enteros con la instrucción `for`
Clase de adaptador `queue`	Clase `Tiempo`
Archivos de acceso aleatorio	Objeto `unique_ptr` que administra la memoria asignada en forma dinámica
Generación de números aleatorios	Validar la entrada del usuario con expresiones regulares
Función recursiva `factorial`	Plantilla de clase `vector`
Tirar un dado 6,000,000 de veces	
Clase `EmpleadoAsalariado`	

Fig. 2 | Una muestra de los ejemplos del libro (parte 2 de 2).

- ***Audiencia.*** Los ejemplos son accesibles para estudiantes de ciencias computacionales, tecnología de la información, ingeniería de software y de negocios en cursos de C++ para estudiantes de nivel principiante a intermedio, aunque también puede ser de gran utilidad para los programadores profesionales.

- ***Ejercicios de autoevaluación y respuestas.*** Se incluyen ejercicios extensos de autoevaluación *con sus* respuestas para autoestudio.

- ***Ejercicios interesantes, entretenidos y desafiantes.*** Cada capítulo concluye con un amplio conjunto de ejercicios: recordatorios simples de la terminología y los conceptos importantes, que sirven para identificar los errores en ejemplos de código, escribir instrucciones individuales de programas, escribir pequeñas porciones de clases de C++ y funciones tanto miembro como no miembro, escribir programas completos e implementar proyectos grandes. La figura 3 contiene una lista de muestra de los ejercicios del libro, incluyendo nuestros ejercicios *Hacer la diferencia*, que animan al lector a usar las computadoras e Internet para investigar y resolver problemas sociales importantes. Esperamos que usted los aborde con *sus propios* valores, políticas y creencias.

Ejercicios		
Sistema de reservaciones de una aerolínea	Ordenamiento de burbuja	Cálculo de los salarios
Ejercicios avanzados de manipulación de cadenas	Cree su propio compilador	Clase abstracta de la huella de carbono: polimorfismo
	Cree su propia computadora	

Fig. 3 | Una muestra de los ejercicios del libro (parte 1 de 2).

Ejercicios		
Barajar y repartir cartas	Ocho reinas	Triples de Pitágoras
Instrucción asistida por computadora	Respuesta de emergencia	Calculadora de salarios
Instrucción asistida por computadora: niveles de dificultad	Implementar la privacidad con la criptografía	Criba de Eratóstenes
Instrucción asistida por computadora: monitoreo del desempeño de los estudiantes	Crecimiento de la base de usuarios de Facebook	Desencriptado simple
Instrucción asistida por computadora: reducción de la fatiga de los estudiantes	Serie de Fibonacci	Encriptado simple
Instrucción asistida por computadora: variación de los tipos de problemas	Kilometraje de gasolina	Lenguaje SMS
Cocinar con ingredientes más saludables	Cuestionario de hechos de advertencia global	Explorador de "spam"
Modificación del juego de "craps"	Juego de adivinar el número	Corrector ortográfico
Límites de crédito	Juego del colgado	Calculadora de la frecuencia cardiaca esperada
Generador de crucigramas	Registros médicos, Paseo del caballo	Alternativas para el plan fiscal: el "impuesto justo"
Criptogramas	Quintillas	Generador de palabras de números telefónicos
Leyes de De Morgan	Recorrido de laberinto: generación de laberintos al azar	Canción "Los doce días de Navidad"
Tiro de dados	Código Morse	Simulación de la tortuga y la liebre
	Modificación al sistema de nóminas	Torres de Hanoi
	Problema de Peter Minuit	Crecimiento de la población mundial
	Explorador de "phishing"	
	Latín cerdo	
	Programa bancario polimórfico mediante el uso de la jerarquía de cuentas	

Fig. 3 | Una muestra de los ejercicios del libro (parte 2 de 2).

- ***Ilustraciones y figuras.*** Se incluyen muchas tablas, dibujos lineales, diagramas de UML, programas y resultados de los programas. La figura 4 incluye una muestra de los dibujos y diagramas del libro.

Dibujos y diagramas		
Dibujos y diagramas del texto principal		
Jerarquía de datos	Diagrama de actividad de UML de la instrucción de repetición `while`	Análisis de un programa de paso por valor y paso por referencia
Proceso de compilación y vinculación para programas con varios archivos de código fuente	Diagrama de actividad de UML de la instrucción de repetición `for`	Diagramas de la jerarquía de herencia
Orden en el que se evalúa un polinomio de segundo grado	Diagrama de actividad de UML de la instrucción de repetición `do...while`	Pila de llamadas a funciones y registros de activación
Diagramas de la clase `LibroCalificaciones`	Diagrama de actividad de la instrucción `switch` de selección múltiple	Llamadas recursivas a la función `fibonacci`
Diagrama de actividad de la instrucción `if` de selección simple	Instrucciones de secuencia, selección y repetición de una sola entrada/una sola salida de C++	Diagramas de aritmética de apuntadores
Diagrama de actividad de la instrucción `if...else` de selección doble		Jerarquía de herencia de `MiembroDeLaComunidad`
		Jerarquía de herencia de `Figura`

Fig. 4 | Una muestra de los dibujos y diagramas del libro (parte 1 de 2).

Dibujos y diagramas		
Herencia `public`, `protected` y `private`	Representación gráfica de una lista	Operación `quitarDelFinal` representada en forma gráfica
Diagrama de clases de UML de la jerarquía `Empleado`	Operación `insertarAlFrente` representada en forma gráfica	Lista circular enlazada simple
Cómo funcionan las llamadas a funciones `virtual`	Operación `insertarAlFinal` representada en forma gráfica	Lista doblemente enlazada
Jerarquía de plantillas de E/S de flujos	Operación `quitarDelFrente` representada en forma gráfica	Lista circular doblemente enlazada
Dos objetos de clases autorreferenciadas enlazados entre sí		Representación gráfica de un árbol binario
Dibujos y diagramas del ejemplo práctico del ATM		
Diagrama de caso-uso para el sistema del ATM desde la perspectiva del Usuario	Clases en el sistema del ATM con atributos y operaciones	Diagrama de clases que muestra las relaciones de composición de una clase `Auto`
Diagrama de clases que muestra una asociación entre clases	Diagrama de comunicaciones del ATM ejecutando una solicitud de saldo	Diagrama de clases para el modelo del sistema del ATM que incluye la clase `Deposito`
Diagrama de clases que muestra relaciones de composición	Diagrama de comunicación para ejecutar una solicitud de saldo	Diagrama de actividad para una transacción `Deposito`
Diagrama de clases para el modelo del sistema del ATM	Diagrama de secuencia que modela la ejecución de un `Retiro`	Diagrama de secuencia que modela la ejecución de un `Deposito`
Clases con atributos	Diagrama de caso-uso para una versión modificada de nuestro sistema ATM que también permite a los usuarios transferir dinero entre cuentas	
Diagrama de estado para el ATM		
Diagrama de actividad para una transacción `SolicitudSaldo`		
Diagrama de actividad para una transacción `Retiro`		

Fig. 4 | Una muestra de los dibujos y diagramas del libro (parte 2 de 2).

Otras características

- ***Apuntadores.*** Ofrecemos una cobertura extensa de las herramientas integradas de apuntadores y la relación íntima entre los apuntadores integrados, las cadenas de C y los arreglos integrados.
- ***Presentación visual de la búsqueda y el ordenamiento con una explicación simple de Big O.***
- ***Material adicional en línea.*** Se incluyen varios capítulos (unos en español y otros en inglés) disponibles en formato PDF con capacidad de búsqueda en el sitio Web de este libro, consulte con su proveedor de Pearson cómo acceder a ellos.

Programación segura en C++

Es difícil crear sistemas con solidez industrial que resistan los ataques de los virus, gusanos y otros tipos de "malware". En la actualidad dichos ataques pueden ser instantáneos y de alcance global debido a Internet. Al integrar la seguridad en el software desde el comienzo del ciclo de desarrollo es posible reducir la vulnerabilidad de manera considerable.

El Centro de coordinación CERT® (`www.cert.org`) se creó para analizar y responder a los ataques de manera oportuna. CERT (el equipo de respuesta a emergencias cibernéticas) es una organización financiada por el gobierno de Estados Unidos, dentro del Carnegie Mellon University Software Engineering Institute™. CERT publica y promueve estándares seguros de codificación para varios lenguajes de programación populares para ayudar a los desarrolladores de software a implementar sistemas con solidez industrial que eviten las prácticas de programación que dejan los sistemas expuestos a ataques.

Agradecemos el apoyo de Robert C. Seacord, gerente de codificación segura en CERT y profesor adjunto en la Carnegie Mellon University School of Computer Science. El señor Seacord fue revisor técnico de nuestro libro *Cómo programar en C, séptima edición*, y revisó nuestros programas en C desde una perspectiva de seguridad, y recomendó que nos adhiriéramos al *Estándar de codificación segura en C de CERT*.

Con este libro hicimos lo mismo: nos adherimos al *Estándar de codificación segura en C++ de CERT*, el cual puede encontrar en:

```
www.securecoding.cert.org
```

Nos llenó de orgullo descubrir que ya estábamos recomendando muchas de esas prácticas de codificación en nuestros libros. Actualizamos nuestro código y los análisis del mismo para cumplir con estas prácticas, según lo apropiado para un libro de texto de nivel introductorio/intermedio. Si piensa crear sistemas en C++ con solidez industrial, considere leer el libro *Secure Coding in C and C++, Second Edition* (de Robert Seacord, Addison-Wesley Professional).

Contenido en línea

Para acceder al sitio Web del libro, vaya a

```
www.pearsonenespañol.com/deitel
```

En el sitio web encontrará los siguientes capítulos en formato PDF con capacidad de búsqueda. Los tres primeros se encuentran en español, y los restantes en idioma inglés:

- 15 Contenedores e iteradores de la biblioteca estándar
- 16 Algoritmos de la biblioteca estándar
- 17 Manejo de excepciones: un análisis más detallado
- 18 Introduction to Custom Templates
- 19 Custom Templatized Data Structures
- 20 Seearching and Sorting
- 21 Class `string` and String Stream Processing: A Deeper Look
- 22 Bits, Characters, C Strings and `structs`
- 23 Other Topics

El sitio web también incluye:

- Descripciones del ejercicio Build Your Own Compiler del capítulo 19.
- Prueba práctica del capítulo 1 para Mac OS X.

Gráfico de dependencias

El gráfico de la siguiente página (figura 6) muestra la dependencia entre los capítulos de este libro, el cual puede ser de mucha ayuda para los profesores que desean planear sus programas de estudio con este texto. El gráfico muestra la organización modular del libro.

Métodos de enseñanza

Esta edición contiene una extensa colección de ejemplos. Hacemos hincapié en la claridad de los programas y nos concentramos en crear software bien diseñado.

Método de código activo. El libro está lleno de ejemplos de "código activo": la mayoría de los nuevos conceptos se presentan en *aplicaciones de C++ funcionales completas*, seguidos de una o más ejecuciones que muestran las entradas y salidas de los programas. En algunos casos, en donde utilizamos un fragmento de código, para asegurar que esté correcto lo evaluamos en un programa funcional completo para después copiarlo y pegarlo en el libro.

Coloreo de sintaxis. Para mejorar la legibilidad, resaltamos la sintaxis de todo el código de C++, en tonos de gris, de manera similar a como lo hace la mayoría de los entornos de desarrollo integrados de C++ y los editores de código. Nuestras convenciones de sintaxis son las siguientes:

```
los comentarios aparecen así
las palabras clave aparecen así
las constantes y los valores literales aparecen así
el resto del código aparece en este tipo
```

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

Uso de fuentes para dar énfasis. Colocamos la ocurrencia de definición de cada término clave en **negritas** para facilitar su referencia. Enfatizamos los componentes en pantalla en la fuente **Helvetica en negritas** (por ejemplo, el menú **Archivo**) y el texto del programa de C++ en la fuente `Lucida` (por ejemplo, `int x = 5;`).

Objetivos. Las citas de apertura van seguidas de una lista de objetivos del capítulo.

Tips de programación. Incluimos tips de programación para ayudarle a enfocarse en los aspectos importantes del desarrollo de programas. Estos tips y prácticas representan lo mejor que hemos podido recabar a lo largo de siete décadas combinadas de experiencia en la programación y la enseñanza.

Buenas prácticas de programación

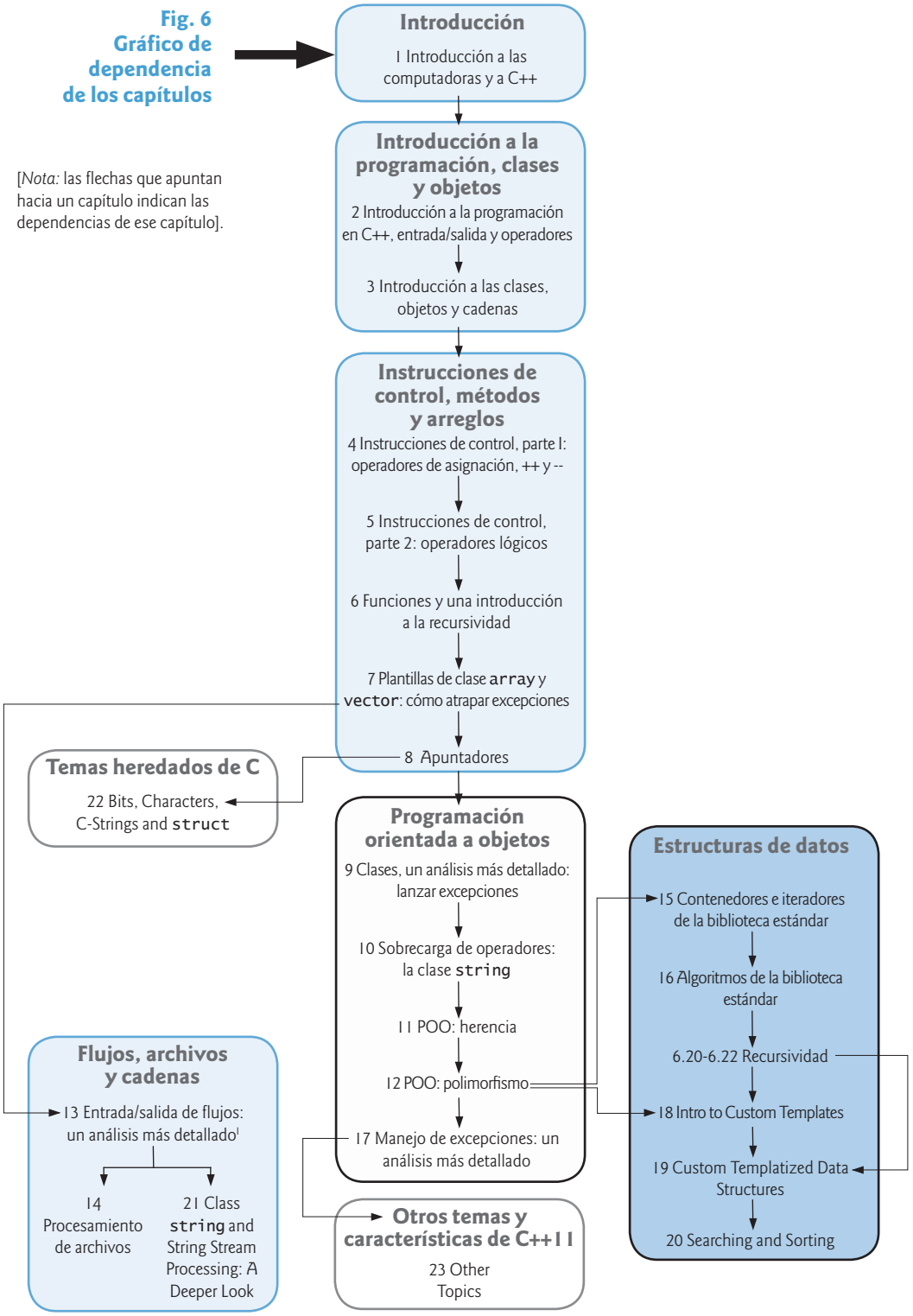
Las Buenas prácticas de programación *llaman la atención hacia técnicas que le ayudarán a producir programas más claros, comprensibles y fáciles de mantener.*

Errores comunes de programación

Al poner atención en estos Errores comunes de programación *se reduce la probabilidad de que pueda cometerlos.*

Fig. 6 Gráfico de dependencia de los capítulos

[*Nota:* las flechas que apuntan hacia un capítulo indican las dependencias de ese capítulo].



1. La mayoría del capítulo 13 puede leerse después del capítulo 7. Una pequeña parte requiere los capítulos 11 y 18.

Tips para prevenir errores

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

Tips de rendimiento

Estos tips resaltan las oportunidades para hacer que sus programas se ejecuten más rápido, o para minimizar la cantidad de memoria que ocupan.

Tips de portabilidad

Los Tips de portabilidad *le ayudan a escribir código que pueda ejecutarse en varias plataformas.*

Observaciones de Ingeniería de Software

Las Observaciones de Ingeniería de Software *resaltan los asuntos de arquitectura y diseño, lo cual afecta la construcción de los sistemas de software, especialmente los de gran escala.*

Viñetas de resumen. Presentamos un resumen detallado del capítulo, estilo lista con viñetas, sección por sección. Para facilitar su consulta, incluimos el número de página donde aparece la definición de los términos clave del capítulo.

Índice. Incluimos un amplio índice en donde las definiciones de los términos clave se resaltan con un número en **negritas**.

Cómo obtener el software utilizado en este libro

Escribimos los ejemplos de código en *Cómo programar en C++, novena edición* mediante el uso de las siguientes herramientas de desarrollo de C++:

- El programa gratuito Visual Studio Express 2012 de Microsoft para Windows Desktop, que incluye Visual C++ junto con otras herramientas de desarrollo de Microsoft. Se ejecuta en Windows 7 y 8, y está disponible para su descarga en

```
www.microsoft.com/visualstudio/esn/downloads#
  d-express-windows-desktop
```

- El programa gratuito GNU C++ de GNU (`gcc.gnu.org/install/binaries.html`), que ya se encuentra instalado en la mayoría de los sistemas Linux y puede instalarse también en sistemas Mac OS X y Windows.
- El programa Xcode gratuito de Apple, que los usuarios de Mac OS X pueden descargar de la App Store de Mac.

Suplementos para el profesor (todos en inglés)

Los siguientes suplementos están disponibles *sólo para profesores calificados* a través del Centro de recursos para el profesor de Pearson (`http://www.pearsonhighered.com/deitel/`):

- ***Manual de soluciones:*** contiene soluciones para la *mayoría* de los ejercicios de final de capítulo. Agregamos muchos ejercicios *Making a Difference* (*Hacer la diferencia)*, la mayoría con soluciones. **El acceso está restringido para los profesores que lleven este libro como texto en su curso.** Los profesores pueden obtener acceso a través de los representantes de Pearson. Si usted no es un miembro docente registrado, póngase en contacto con alguno de nuestros representantes o contacte al editor. *No* se proporcionan soluciones a los ejercicios

de "proyectos". Consulte nuestro Centro de recursos de proyectos de programación, en donde encontrará muchos ejercicios y proyectos adicionales:

```
www.deitel.com/ProgrammingProjects
```

- ***Test Item File (Archivo de prueba)*** de preguntas de opción múltiple (aproximadamente dos por cada sección del libro).
- ***Diapositivas de PowerPoint® personalizables*** que contienen todo el código (en inglés) y las figuras del texto (también en inglés), además de elementos en viñetas que sintetizan los puntos clave.

Agradecimientos

Queremos agradecer a Abbey Deitel y Barbara Deitel, de Deitel & Associates, Inc., por la gran cantidad de horas que dedicaron a este proyecto. Abbey fue coautora del capítulo 1; además, junto con Barbara investigaron de manera minuciosa las nuevas herramientas de C++11.

Somos afortunados al haber trabajado con el dedicado equipo de editores profesionales en Pearson. Apreciamos la orientación, inteligencia y energía de Tracy Johnson, editor ejecutivo de ciencias computacionales. Carole Snyder hizo un extraordinario trabajo de reclutar a los revisores del libro y se hizo cargo del proceso de revisión. Bob Engelhardt hizo un maravilloso trabajo para llevar el libro a su publicación.

Revisores

Queremos agradecer los esfuerzos de nuestros revisores. Este libro fue revisado por los miembros actuales y anteriores del comité de estándares de C++ que desarrolló C++11, académicos que imparten cursos sobre C++ y expertos en la industria. Todos ellos proporcionaron incontables sugerencias para mejorar la presentación. Cualquier error en el libro es por culpa nuestra.

Revisores de la novena edición: Dean Michael Berris (Google, Miembro del comité ISO C++), Danny Kalev (experto en C++, analista de sistemas certificado y ex miembro del Comité de estándares de C++), Linda M. Krause (Elmhurst College), James P. McNellis (Microsoft Corporation), Robert C. Seacord (gerente de codificación segura en SEI/CERT, autor de *Secure Coding in C and C++*) y José Antonio González Seco (Parlamento de Andalucía).

Revisores de ediciones anteriores: Virginia Bailey (Jackson State University), Thomas J. Borrelli (Rochester Institute of Technology), Ed Brey (Kohler Co.), Chris Cox (Adobe Systems), Gregory Dai (eBay), Peter J. DePasquale (The College of New Jersey), John Dibling (SpryWare), Susan Gauch (University of Arkansas), Doug Gregor (Apple, Inc.), Jack Hagemeister (Washington State University), Williams M. Higdon (University of Indiana), Anne B. Horton (Lockheed Martin), Terrell Hull (Logicalis Integration Solutions), Ed James Beckham (Borland), Wing-Ning Li (University of Arkansas), Dean Mathias (Utah State University), Robert A. McLain (Tidewater Community College), Robert Myers (Florida State University), Gavin Osborne (Saskatchewan Inst. of App. Sci. and Tech.), Amar Raheja (California State Polytechnic University, Pomona), April Reagan (Microsoft), Raymond Stephenson (Microsoft), Dave Topham (Ohlone College), Anthony Williams (autor y miembro del Comité de estándares de C++) y Chad Willwerth (University Washington, Tacoma).

Apreciaremos con sinceridad sus comentarios, críticas, correcciones y sugerencias para mejorar el texto. Dirija toda su correspondencia a:

```
deitel@deitel.com
```

Le responderemos a la brevedad posible. Disfrutamos mucho el escribir *Cómo programar en C++, novena edición*. ¡Esperamos que usted disfrute el leerlo!

Paul Deitel
Harvey Deitel

Acerca de los autores

Paul J. Deitel, CEO y Director Técnico de Deitel & Associates, Inc., es egresado del MIT, en donde estudió Tecnología de la Información. A través de Deitel & Associates, Inc., ha impartido cientos de cursos de programación a empresas de la industria como: Cisco, IBM, Siemens, Sun Microsystems, Dell, Fidelity, NASA (en el Centro Espacial Kennedy), National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, SunGard Higher Education, Nortel Networks, Puma, iRobot, Invensys, entre muchos más. Él y su coautor, el Dr. Harvey M. Deitel, son autores de los libros de texto, libros profesionales y videos sobre lenguajes de programación más vendidos en el mundo.

Dr. Harvey M. Deitel, Presidente y Consejero de Estrategia de Deitel & Associates, Inc., tiene 50 años de experiencia en el campo de la computación. El Dr. Deitel tiene una licenciatura y una maestría en ingeniería eléctrica por el MIT y un doctorado en matemáticas de la Boston University. Tiene muchos años de experiencia como profesor universitario, la cual incluye un puesto vitalicio y el haber sido presidente del departamento de Ciencias de la computación en el Boston College antes de fundar, con su hijo Paul J. Deitel, Deitel & Associates, Inc. Los textos de los Deitel se han ganado el reconocimiento internacional y han sido traducidos al chino, coreano, japonés, alemán, ruso, español, francés, polaco, italiano, portugués, griego, urdú y turco. El Dr. Deitel ha impartido cientos de cursos de programación a clientes corporativos, académicos, gubernamentales y militares.

Capacitación corporativa de Deitel & Associates, Inc.

Deitel & Associates, Inc., fundada por Paul Deitel y Harvey Deitel, es una empresa reconocida a nivel mundial, dedicada al entrenamiento corporativo y la creación de contenido, especializada en lenguajes de programación de computadora, tecnología de objetos, desarrollo de aplicaciones móviles y tecnología de software de Internet y Web. Sus clientes incluyen muchas de las empresas más grandes del mundo, agencias gubernamentales, sectores del ejército e instituciones académicas. La empresa proporciona cursos, en las instalaciones de sus clientes en todo el mundo, sobre la mayoría de los lenguajes y plataformas de programación, como C++, Visual C++®, C, Java™, Visual C#®, Visual Basic®, XML®, Python®, tecnología de objetos, programación en Internet y Web, desarrollo de aplicaciones para Android, desarrollo de aplicaciones de Objective-C y para iPhone, y una lista cada vez mayor de cursos adicionales de programación y desarrollo de software.

A lo largo de su sociedad editorial de más de 36 años con Prentice Hall/Pearson, Deitel & Associates, Inc. ha publicado libros de texto de vanguardia sobre programación, libros profesionales y cursos en video de *LiveLessons*. Puede contactarse con Deitel & Associates, Inc. y con los autores por medio del correo electrónico:

```
deitel@deitel.com
```

Para conocer más sobre la Serie de Capacitación Corporativa *Dive Into*® de Deitel, visite:

```
www.deitel.com/training
```

Para solicitar una propuesta de capacitación impartida en el sitio de su organización, en cualquier parte del mundo, envíe un correo a `deitel@deitel.com`.

Quien desee comprar libros de Deitel y cursos en video de *LiveLessons* puede hacerlo a través de `www.deitel.com`. Las empresas, el gobierno, las instituciones militares y académicas que deseen realizar pedidos en masa deben hacerlo directamente con Pearson. Para obtener más información, visite

```
www.pearsonenespañol.com
```

Introducción a las computadoras y a C++

1

El hombre sigue siendo la computadora más extraordinaria de todas.

—John F. Kennedy

Un buen diseño es un buen negocio.

—Thomas J. Watson, fundador de IBM

Qué maravilloso es que nadie necesite esperar un solo momento para empezar a mejorar el mundo.

—Anne Frank

Objetivos

En este capítulo aprenderá a:

- Familiarizarse con los emocionantes y recientes desarrollos en el campo de las computadoras.
- Conocer los fundamentos del hardware, software y redes de computadoras.
- Emplear la jerarquía de datos.
- Reconocer distintos tipos de lenguajes de programación.
- Familiarizarse con algunos conceptos básicos de tecnología de objetos.
- Apreciar algunos fundamentos de Internet y World Wide Web.
- Conocer un entorno de desarrollo común de programas en C++.
- Probar una aplicación en C++.
- Familiarizarse con algunas de las recientes tecnologías clave de software.
- Comprender cómo pueden las computadoras ayudarlo a hacer la diferencia.

Plan general

1.1 Introducción

Bienvenido a C++: un poderoso lenguaje de programación de computadoras apropiado para las personas con orientación técnica con poca o ninguna experiencia de programación, y para los programadores experimentados que desarrollan sistemas de información de tamaño considerable. Usted ya está familiarizado con las poderosas tareas que realizan las computadoras. Mediante este libro aprenderá a escribir instrucciones para ordenar a las computadoras que realicen esos tipos de tareas. El *software* (es decir, las instrucciones que usted escribe) controla el *hardware* (es decir, las computadoras).

Aprenderá sobre la *programación orientada a objetos*: la metodología de programación clave de la actualidad. En este texto creará muchos *objetos de software* que modelan las *cosas* del mundo real.

C++ es uno de los lenguajes de desarrollo de software más populares. Este libro le proporciona una introducción a la programación en C++11: la versión más reciente estandarizada mediante la **Organización Internacional para la Estandarización (ISO)** y la **Comisión Electrotécnica Internacional (IEC)**.

En la actualidad hay en uso más de mil millones de computadoras de propósito general, además de miles de millones de teléfonos celulares, teléfonos inteligentes (smartphones) y dispositivos portátiles (como las computadoras tipo tableta). De acuerdo con un estudio realizado por eMarketer, el número de usuarios móviles de Internet sobrepasará los 134 millones en 2014.[1] Las ventas de teléfonos inteligentes excedieron a las ventas de computadoras personales en 2011.[2] Se espera que para 2015, las ventas de las tabletas representen cerca del 20% de todas las ventas de computadoras personales.[3] Se espera que en 2014 el mercado de las aplicaciones de teléfonos inteligentes exceda los $40 mil millones.[4] Este explosivo crecimiento está creando oportunidades importantes para la programación de aplicaciones móviles.

1 www.circleid.com/posts/mobile_internet_users_to_reach_134_million_by_2013/.
2 www.mashable.com/2012/02/03/smartphone-sales-overtake-pcs.
3 www.forrester.com/ER/Press/Release/0,1769,1340,00.html.
4 *Inc.* diciembre de 2010/enero de 2011, páginas 116-123.

1.2 Las computadoras e Internet en la industria y la investigación

Éstos son tiempos emocionantes en el campo de la computación. Muchas de las empresas más influyentes y exitosas de las últimas dos décadas son compañías de tecnología, como Apple, IBM, Hewlett Packard, Dell, Intel, Motorola, Cisco, Microsoft, Google, Amazon, Facebook, Twitter, Groupon, Foursquare, Yahoo!, eBay y muchas más. Estas empresas son importantes fuentes de empleo para las personas que estudian ciencias computacionales, ingeniería computacional, sistemas de información o disciplinas relacionadas. En 2013, Apple era la compañía más valiosa del mundo. La figura 1.1 provee unos cuantos ejemplos de las formas en que las computadoras están mejorando las vidas de las personas en la investigación, la industria y la sociedad.

Nombre	Descripción
Registros de salud electrónicos	Podrían incluir el historial médico de un paciente, prescripciones, vacunas, resultados de laboratorio, alergias, información de seguros y demás. Al poner esa información a disposición de los proveedores de servicios médicos a través de una red segura, logramos mejorar el cuidado de los pacientes, se reduce la probabilidad de error y en general, aumenta la eficiencia del sistema de servicios médicos.
Proyecto Genoma Humano	El Proyecto Genoma Humano se fundó para identificar y analizar los más de 20 000 genes en el ADN humano. El proyecto utilizó programas de computadora para analizar datos genéticos complejos, determinar las secuencias de los miles de millones de pares base químicos que conforman el ADN humano y almacenar la información en bases de datos disponibles a través de Internet para los investigadores en muchos campos.
AMBER™ Alert	El sistema de alerta AMBER (Niños norteamericanos extraviados: transmisión de respuesta a emergencias) se utiliza para buscar niños secuestrados. Las autoridades notifican a los funcionarios de transporte estatal y a las transmisoras de TV y radio, quienes a su vez transmiten alertas en TV, radio, señales de tráfico computarizadas, Internet y dispositivos inalámbricos. AMBER Alert se asoció hace poco con Facebook, cuyos usuarios pueden "dar Like" a las páginas de AMBER Alert por ubicación para recibir alertas en sus fuentes de noticias.
World Community Grid	Personas de todo el mundo pueden donar su poder de procesamiento de cómputo que no utilicen, mediante la instalación de un programa de software seguro gratuito que permite a World Community Grid (`www.worldcommunitygrid.org`) aprovechar la capacidad que no se utilice. Este poder de cómputo, al cual se accede a través de Internet, se utiliza en lugar de costosas supercomputadoras para realizar proyectos científicos de investigación que están haciendo la diferencia: ya sea proporcionar agua potable a países del tercer mundo, combatir el cáncer, cultivar arroz más nutritivo para regiones que combaten el hambre y otras cosas más.
Computación en nube	La **computación en nube** nos permite usar software, hardware e información almacenada en la "nube" (es decir, se accede desde computadoras remotas a través de Internet y está disponible bajo demanda) en vez de tener que almacenarla en nuestra computadora personal. Estos servicios, que le permiten aumentar o disminuir los recursos para satisfacer sus necesidades en un momento dado, son por lo general más efectivos en costos que comprar hardware costoso para asegurarse de tener el suficiente almacenamiento y poder de procesamiento para satisfacer sus necesidades en sus niveles máximos. Al usar los servicios de computación en nube traspasamos la carga de tener que manejar estas aplicaciones de nuestra empresa al proveedor de servicios, con lo cual ahorramos dinero.

Fig. 1.1 | Unos cuantos usos para las computadoras (parte 1 de 3).

Nombre	Descripción
Imágenes para diagnóstico médico	Las exploraciones por tomografía computarizada (CT) con rayos X, también conocidas como CAT (tomografía axial computarizada), toman rayos X del cuerpo desde cientos de ángulos distintos. Se utilizan computadoras para ajustar la intensidad del rayo X, con lo cual se optimiza la exploración para cada tipo de tejido, para después combinar toda la información y crear una imagen tridimensional (3D). Los escáneres MRI usan una técnica conocida como imágenes de resonancia magnética, también para producir imágenes internas de una manera no invasiva.
GPS	Los dispositivos con Sistema de posicionamiento global (GPS) utilizan una red de satélites para obtener información basada en la ubicación. Varios satélites envían señales con etiquetas de tiempo al dispositivo GPS, el cual calcula la distancia hacia cada satélite con base en la hora en que la señal salió del satélite y la hora en que se recibió la señal. Esta información se utiliza para determinar la ubicación exacta del dispositivo. Los dispositivos GPS pueden proveer indicaciones paso a paso y ayudarle a localizar negocios cercanos (restaurantes, gasolineras, etc.) y puntos de interés. El sistema GPS se utiliza en numerosos servicios de Internet basados en la ubicación, como las aplicaciones de registro (check-in) en línea, para que usted pueda encontrar a sus amigos (por ejemplo, Foursquare y Facebook), en aplicaciones para hacer ejercicio como RunKeeper, que rastrean el tiempo, la distancia y la velocidad promedio de su rutina de trotar en exteriores, aplicaciones de citas que le ayudan a buscar una pareja cercana y aplicaciones que actualizan en forma dinámica las condiciones cambiantes del tráfico.
Robots	Los robots se pueden utilizar para tareas diarias (por ejemplo, la aspiradora Roomba de iRobot), de entretenimiento (como las mascotas robóticas), combate militar, exploración espacial y en la profundidad del océano (como el trotamundos Curiosity de la NASA), y otras más. RoboEarth (`www.roboearth.org`) es "una World Wide Web para robots". Permite a los robots aprender unos de otros mediante la compartición de información, con lo cual mejoran sus habilidades para realizar tareas, navegar, reconocer objetos y demás.
Correo electrónico, mensajería instantánea, chat de video y FTP	Los servidores basados en Internet soportan toda su mensajería en línea. Los mensajes de correo electrónico pasan por un servidor de correo que también almacena esos mensajes. Las aplicaciones de mensajería instantánea (IM) y chat de video, como AIM, Skype, Yahoo! Messenger, Google Talk, Trillian, Microsoft Messenger y otras más, le permiten comunicarse con otras personas en tiempo real, mediante el envío de mensajes y video a través de los servidores. FTP (protocolo de transferencia de archivos) le permite intercambiar archivos entre varias computadoras (por ejemplo, una computadora cliente como su escritorio y un servidor de archivos) a través de Internet.
TV por Internet	Los dispositivos de TV por Internet (como Apple TV, Google TV y TiVo) le permiten acceder a una enorme cantidad de contenido bajo demanda, como juegos, noticias, películas, programas de televisión, etcétera, además de que le ayudan a asegurar que el contenido se transmita en flujo continuo hacia su TV sin problemas.
Servicios de transmisión de música por flujo continuo	Los servicios de transmisión de música por flujo continuo (como Pandora, Spotify, Last.fm y más) le permiten escuchar grandes catálogos de música a través de Web, crear "estaciones de radio" personalizadas y descubrir nueva música con base en la información que usted les retroalimente.

Fig. 1.1 | Unos cuantos usos para las computadoras (parte 2 de 3).

Nombre	Descripción
Programación de juegos	Los analistas esperan que los ingresos mundiales por juegos de video lleguen a $91 mil millones en 2015 (`www.vg247.com/2009/06/23/global-industry-analysts-predicts-gaming-market-to-reach-91-billion-by-2015/`). El desarrollo de los juegos más sofisticados puede costar hasta $100 millones. *Call of Duty: Black Ops de Activision* (uno de los juegos más vendidos de todos los tiempos) ¡obtuvo $360 millones en sólo un día (`www.forbes.com/sites/insertcoin/2011/03/11/call-of-duty-black-ops-now-the-best-selling-video-game-of-all-time/`)! Los *juegos sociales* en línea, que permiten a usuarios en todo el mundo competir entre sí a través de Internet, están creciendo con rapidez. Zynga (creador de juegos en línea populares, como *Words with Friends*, *CityVille* y otros) se fundó en 2007 y ya tiene más de 300 millones de usuarios al mes. Para dar cabida al aumento en tráfico, ¡Zynga agregará cerca de 1 000 servidores cada semana (`techcrunch.com/2010/09/22/zynga-moves-1-petabyte-of-data-daily-adds-1000-servers-a-week/`)!

Fig. 1.1 | Unos cuantos usos para las computadoras (parte 3 de 3).

1.3 Hardware y software

Las computadoras pueden realizar cálculos y tomar decisiones lógicas con una rapidez increíblemente mayor que los humanos. Actualmente, muchas de las computadoras personales pueden realizar miles de millones de cálculos en un segundo —más de lo que un humano podría realizar en toda su vida. ¡Las *supercomputadoras* ya pueden realizar *miles de billones* de instrucciones por segundo! ¡La supercomputadora Sequoia de IBM puede realizar más de 16 mil billones de cálculos por segundo (16.32 *petaflops*)![5] Dicho de otra forma, *¡la supercomputadora Sequoia de IBM puede realizar en un segundo alrededor de 1.5 millones de cálculos para cada uno de los habitantes del planeta!* ¡Y estos "límites superiores" están aumentando con rapidez!

Las computadoras procesan datos bajo el control de conjuntos de instrucciones conocidas como **programas de computadora**. Estos programas guían a la computadora a través de conjuntos ordenados de acciones especificadas por gente conocida como **programadores** de computadoras. A los programas que se ejecutan en una computadora se les denomina **software**. En este libro aprenderá una metodología de programación clave que mejora la productividad del programador, con lo cual se reducen los costos de desarrollo del software: *programación orientada a objetos*.

Una computadora consiste en varios dispositivos conocidos como hardware (teclado, pantalla, ratón, discos duros, memoria, unidades de DVD y unidades de procesamiento). Los costos de las computadoras *han disminuido en forma espectacular*, debido a los rápidos desarrollos en las tecnologías de hardware y software. Las computadoras que ocupaban grandes habitaciones y que costaban millones de dólares hace algunas décadas, ahora pueden colocarse en las superficies de chips de silicio más pequeños que una uña, y con un costo de quizá unos cuantos dólares cada uno. Aunque suene irónico, el silicio es uno de los materiales más abundantes en el planeta: es un ingrediente en la arena común. La tecnología de los chips de silicio ha vuelto tan económica a la tecnología de la computación que en la actualidad las computadoras se han vuelto un producto básico.

5 `www.top500.org/`.

1.3.1 La Ley de Moore

Es probable que cada año espere pagar por lo menos un poco más por la mayoría de los productos y servicios. En el caso de los campos de las computadoras y las comunicaciones se ha dado lo opuesto, en especial con relación a los costos del hardware que da soporte a estas tecnologías. Los costos del hardware han disminuido con rapidez durante varias décadas. Aproximadamente, cada uno o dos años, las capacidades de las computadoras se *duplican* sin que el precio se incremente. Esta notable tendencia se conoce en el ámbito común como la **Ley de Moore**, y debe su nombre a la persona que la identificó en la década de 1960: Gordon Moore, cofundador de Intel —el principal fabricante de procesadores para las computadoras y los sistemas incrustados de la actualidad. La Ley de Moore y las observaciones relacionadas son especialmente ciertas en cuanto a la cantidad de memoria que tienen las computadoras para los programas, la cantidad de almacenamiento secundario (como el almacenamiento en disco) que tienen para guardar los programas y datos durante periodos extendidos, y las velocidades de sus procesadores —las velocidades con que las computadoras ejecutan sus programas (es decir, realizan su trabajo). Se ha producido un crecimiento similar en el campo de las comunicaciones, en donde los costos se han desplomado a medida que la enorme demanda por el ancho de banda de las comunicaciones (es decir, la capacidad de transmisión de información) atrae una competencia intensa. No conocemos otros campos en los que la tecnología mejore con tanta rapidez y los costos disminuyan de una manera tan drástica. Dicha mejora fenomenal está fomentando sin duda la *Revolución de la información*.

1.3.2 Organización de la computadora

Independientemente de las diferencias en su apariencia *física*, las computadoras pueden concebirse como divididas en varias **unidades lógicas** o secciones (figura 1.2).

Unidad lógica	Descripción
Unidad de entrada	Esta sección "receptora" obtiene información (datos y programas de cómputo) de los **dispositivos de entrada** y pone esta información a disposición de las otras unidades para que pueda procesarse. La mayor parte de la información se introduce en las computadoras a través de los teclados y ratones. Hay otras formas de entrada: recibir comandos de voz, digitalizar imágenes y códigos de barra, leer desde dispositivos de almacenamiento secundarios (como discos duros, unidades de DVD, unidades de Blu-ray Disc™ y unidades flash USB), recibir video de una cámara Web y hacer que su computadora reciba información de Internet (como cuando transmite videos en flujo continuo desde YouTube™ o descarga libros electrónicos de Amazon). Las formas más recientes de entrada incluyen los datos de posición de un dispositivo GPS, la información de movimiento y orientación de un acelerómetro en un Smartphone o controlador de juegos (como Microsoft® Kinect™, Wii™ Remote y Move de Sony PlayStation®).
Unidad de salida	Esta sección de "embarque" toma información que ya ha sido procesada por la computadora y la coloca en varios **dispositivos de salida**, para que esté disponible fuera de la computadora. Hoy en día, la mayor parte de la información de salida de las computadoras se despliega en pantallas, se imprime en papel (el "movimiento ecológico" desaprueba esta opción), se reproduce como audio o video en computadoras personales y reproductores de medios (como los populares iPod de Apple) y pantallas gigantes en estadios deportivos, se transmite a través de Internet o se usa para controlar otros dispositivos, como robots y aparatos "inteligentes".

Fig. 1.2 | Unidades lógicas de una computadora (parte 1 de 2).

Unidad lógica	Descripción
Unidad de memoria	Esta sección de "almacén" de acceso rápido, pero con relativa baja capacidad, retiene la información que se introduce a través de la unidad de entrada, para que pueda estar disponible de manera inmediata para procesarla cuando sea necesario. La unidad de memoria también retiene la información procesada hasta que la unidad de salida pueda colocarla en los dispositivos de salida. La información en la unidad de memoria es *volátil*: por lo general se pierde cuando se apaga la computadora. Con frecuencia, a esta unidad de memoria se le llama **memoria** o **memoria primaria**. Muchas memorias en computadoras de escritorio y tipo notebook contienen comúnmente hasta 16 GB (esta medida significa gigabytes; un gigabyte equivale aproximadamente mil millones de bytes).
Unidad aritmética y lógica (ALU)	Esta sección de "manufactura" realiza *cálculos* como suma, resta, multiplicación y división.También contiene los mecanismos de *decisión* que permiten a la computadora hacer cosas como, por ejemplo, comparar dos elementos de la unidad de memoria para determinar si son iguales o no. Por lo general, en los sistemas actuales la ALU se implementa como parte de la siguiente unidad lógica, la CPU.
Unidad central de procesamiento (CPU)	Ésta sección "administrativa" coordina y supervisa la operación de las demás secciones. La CPU le indica a la unidad de entrada cuándo debe grabarse la información dentro de la unidad de memoria, a la ALU cuándo debe utilizarse la información de la unidad de memoria para los cálculos, y a la unidad de salida cuándo enviar la información desde la unidad de memoria hasta ciertos dispositivos de salida. Muchas de las computadoras actuales contienen múltiples CPU y, por lo tanto, pueden realizar muchas operaciones de manera simultánea. Un **procesador multinúcleo** implementa varios procesadores en un solo chip de circuitos integrados; un *procesador de doble núcleo* (dual-core) tiene dos CPU y un *procesador de cuádruple núcleo* (quad-core) tiene cuatro CPU. Las computadoras de escritorio de la actualidad tienen procesadores que pueden ejecutar miles de millones de instrucciones por segundo.
Unidad de almacenamiento secundario	Ésta es la sección de "almacén" de alta capacidad y de larga duración. Los programas o datos que no utilizan las demás unidades con frecuencia se colocan en dispositivos de almacenamiento secundario (por ejemplo, el *disco duro*) hasta que se requieran de nuevo, lo cual puede llegar a ser horas, días, meses o incluso años después. La información en los dispositivos de almacenamiento secundario es *persistente*: se conserva aun y cuando se apaga la computadora. El tiempo para acceder a la información en almacenamiento secundario es mucho mayor que el necesario para acceder a la de la memoria principal, pero el costo por unidad de memoria secundaria es mucho menor que el correspondiente a la unidad de memoria principal. Las unidades de CD, DVD y Flash USB son ejemplos de dispositivos de almacenamiento secundario, algunos de los cuales pueden contener hasta 768 GB.Los discos duros típicos en las computadoras de escritorio y portátiles pueden contener hasta 2 TB (TB se refiere a terabytes; un terabyte equivale aproximadamente a un billón de bytes).

Fig. 1.2 | Unidades lógicas de una computadora (parte 2 de 2).

1.4 Jerarquía de datos

Los elementos de datos que procesan las computadoras forman una **jerarquía de datos** que se vuelve cada vez más grande y compleja en estructura, a medida que progresamos primero a bits, luego a caracteres, después a campos y así en lo sucesivo. La figura 1.3 ilustra una porción de la jerarquía de datos. La figura 1.4 sintetiza los niveles de la jerarquía de datos.

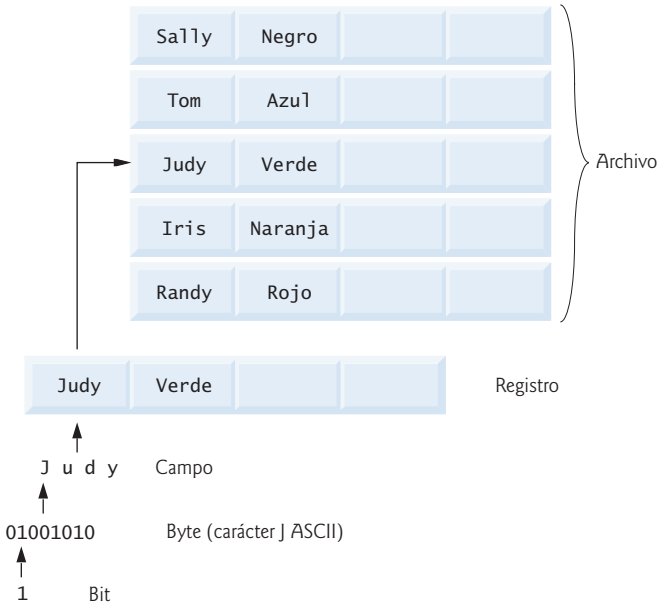


Fig. 1.3 | Jerarquía de datos.

Nivel	Descripción
Bits	El elemento de datos más pequeño en una computadora puede asumir el valor `0` o el valor `1`. A dicho elemento de datos se le denomina **bit** (abreviación de "dígito binario": un dígito que puede asumir uno de dos valores). Es notable que las impresionantes funciones que realizan las computadoras sólo impliquen las manipulaciones más simples de `0`s y `1`s: *examinar el valor de un bit, establecer el valor de un bit e invertir el valor de un bit* (de `1` a `0` o de `0` a `1`).
Caracteres	Es tedioso para las personas trabajar con datos en el formato de bajo nivel de los bits. En cambio, prefieren trabajar con *dígitos decimales* (0–9), *letras* (A–Z y a–z) y *símbolos especiales* (por ejemplo, $, @, %, &, *, (,), –, +, ", :, ? y /). Los dígitos, letras y símbolos especiales se conocen como **caracteres**. El **conjunto de caracteres** de la computadora es el conjunto de todos los caracteres que se utilizan para escribir programas y representar elementos de datos. Las computadoras sólo procesan `1`s y `0`s, por lo que cada carácter se representa como un patrón de `1`s y `0`s. El conjunto de caracteres **Unicode®** contiene caracteres para muchos de los idiomas en el mundo. C++ soporta varios conjuntos de caracteres, incluyendo los caracteres Unicode® de 16 bits que están compuestos de dos **bytes**, cada uno de los cuales se compone a su vez de ocho bits. En el apéndice B obtendrá más información sobre el conjunto de caracteres **ASCII** (**Código estándar estadounidense para el intercambio de información**): el popular subconjunto de Unicode que representa las letras mayúsculas y minúsculas, los dígitos y algunos caracteres especiales comunes.
Campos	Así como los caracteres están compuestos de bits, los **campos** están compuestos de caracteres o bytes.Un campo es un grupo de caracteres o bytes que transmiten un significado. Por ejemplo, un campo compuesto de letras mayúsculas y minúsculas se puede usar para representar el nombre de una persona, y un campo compuesto de dígitos decimales podría representar la edad de esa persona.

Fig. 1.4 | Niveles de la jerarquía de datos (parte 1 de 2).

Nivel	Descripción
Registros	Se pueden usar varios campos relacionados para componer un **registro**. Por ejemplo, en un sistema de nómina, el registro de un empleado podría consistir en los siguientes campos (los posibles tipos para estos campos se muestran entre paréntesis): • Número de identificación del empleado (un número entero) • Nombre (una cadena de caracteres) • Dirección (una cadena de caracteres) • Salario por horas (un número con punto decimal) • Ingresos del año a la fecha (un número con punto decimal) • Monto de impuestos retenidos (un número con punto decimal) Así, un registro es un grupo de campos relacionados. En el ejemplo anterior, todos los campos pertenecen al mismo empleado. Una compañía podría tener muchos empleados y un registro de nómina para cada uno.
Archivos	Un **archivo** es un grupo de registros relacionados. [*Nota*: dicho en forma más general, un archivo contiene datos arbitrarios en formatos arbitrarios. En algunos sistemas operativos, un archivo se ve tan sólo como una *secuencia de bytes*: cualquier organización de esos bytes, como cuando se organizan los datos en registros, es una vista creada por el programador de la aplicación]. Es muy común que una organización tenga muchos archivos, algunos de los cuales pueden contener miles de millones, o incluso billones de caracteres de información.
Base de datos	Una **base de datos** es una colección electrónica de datos organizada para facilitar su acceso y manipulación. El modelo de base de datos más popular es la base de datos relacional, en la cual los datos se almacenan en simples *tablas*. Una tabla incluye *registros* y *campos*. Por ejemplo, una tabla de estudiantes podría incluir el primer nombre, apellido paterno, carrera, número de ID de estudiante y promedio de calificaciones. Los datos para cada estudiante conforman un registro y las piezas individuales de información en cada registro son los campos. Es posible realizar búsquedas, ordenar y manipular los datos con base en su relación con varias tablas o bases de datos. Por ejemplo, una universidad podría usar datos de la base de datos de estudiantes en combinación con las bases de datos de los cursos, alojamiento en el campus, planes de alimentación, etc.

Fig. 1.4 | Niveles de la jerarquía de datos (parte 2 de 2).

1.5 Lenguajes máquina, lenguajes ensambladores y lenguajes de alto nivel

Los programadores escriben instrucciones en diversos lenguajes de programación, algunos de los cuales los comprende directamente la computadora, mientras que otros requieren pasos intermedios de *traducción*.

Lenguajes máquina

Cualquier computadora puede entender de manera directa sólo su propio **lenguaje máquina** (también conocido como *código máquina*), el cual se define según su diseño de hardware. Por lo general, los lenguajes máquina consisten en cadenas de números (que finalmente se reducen a 1s y 0s). Dichos lenguajes son difíciles de comprender para los humanos.

Lenguajes ensambladores

La programación en lenguaje máquina era demasiado lenta y tediosa para la mayoría de los programadores. Por lo tanto, empezaron a utilizar *abreviaturas* del inglés para representar las operaciones

elementales. Estas abreviaturas formaron la base de los **lenguajes ensambladores**. Se desarrollaron *programas traductores* conocidos como **ensambladores** para convertir los programas que se encontraban en lenguaje ensamblador a lenguaje máquina. Aunque el código en lenguaje ensamblador es más claro para los humanos, las computadoras no lo pueden entender sino hasta que se traduce en lenguaje máquina.

Lenguajes de alto nivel

Para agilizar el proceso de programación se desarrollaron los **lenguajes de alto nivel**, en donde podían escribirse instrucciones individuales para realizar tareas importantes. Los lenguajes de alto nivel, como C++, Java, C# y Visual Basic nos permiten escribir instrucciones que son muy similares al inglés y contienen expresiones matemáticas de uso común. Los programas traductores llamados **compiladores** convierten los programas que se encuentran en lenguaje de alto nivel a programas en lenguaje máquina.

El proceso de compilación de un programa escrito en lenguaje de alto nivel a un lenguaje máquina puede tardar un tiempo considerable en la computadora. Los programas **intérpretes** se desarrollaron para ejecutar programas en lenguaje de alto nivel de manera directa (sin el retraso de la compilación), aunque con más lentitud de la que se ejecutan los programas compilados. Los **lenguajes de secuencias de comandos**, como los populares lenguajes JavaScript y PHP para Web, son procesados por intérpretes.

Tip de rendimiento 1.1

Los intérpretes tienen una ventaja sobre los compiladores en las secuencias de comandos de Internet. Un programa interpretado puede comenzar a ejecutarse tan pronto como se descarga en la máquina cliente, sin necesidad de compilarse antes de poder ejecutarse. Por otra parte, las secuencias de comandos interpretadas generalmente se ejecutan con más lentitud que el código compilado.

1.6 C++

C++ evolucionó a partir de C, que fue desarrollado por Dennis Ritchie en los laboratorios Bell. C está disponible para la mayoría de las computadoras y es independiente del hardware. Con un diseño cuidadoso, es posible escribir programas en C que sean **portables** para la mayoría de las computadoras.

Por desgracia, el amplio uso de C con diversos tipos de computadoras (a las que algunas veces se les denomina **plataformas de hardware**) produjo muchas variaciones. Era necesaria una versión estándar de C. El Instituto nacional estadounidense de estándares (ANSI) cooperó con la Organización internacional para la estandarización (ISO) para estandarizar C a nivel mundial; el documento estándar colectivo se publicó en 1990, y se conoce como *ANSI/ISO 9899: 1990*.

C11 es el estándar más reciente de ANSI para el lenguaje de programación C. Se desarrolló para que C evolucionara y se mantuviera a la par con el hardware cada vez más poderoso y los requerimientos de los usuarios, que cada vez son más exigentes. C11 también hace a C más consistente con C++. Para obtener más información sobre C y C11, consulte nuestro libro *C How to Program, séptima edición* y nuestro Centro de recursos de C (que se encuentra en `www.deitel.com/C`).

El lenguaje C++, que es una extensión de C, lo desarrolló Bjarne Stroustrup en 1979, en los laboratorios Bell. Conocido en un principio como "C con clases", se cambió su nombre a C++ a principios de la década de 1980. C++ ofrece varias características que "pulen" al lenguaje C pero, lo más importante es que proporciona las capacidades de una programación orientada a objetos.

El lector comenzará a desarrollar objetos y clases personalizadas reutilizables en el capítulo 3, Introducción a las clases, objetos y cadenas. El libro está orientado a objetos, en donde sea apropiado, desde el principio y a lo largo del texto.

También proporcionamos un caso de estudio *opcional* sobre un cajero automático (ATM) en los capítulos 25 y 26, el cual contiene una implementación completa en C++. El caso de estudio presenta una introducción cuidadosamente pautada al diseño orientado a objetos mediante el UML: un lenguaje de modelado gráfico estándar en la industria para desarrollar sistemas orientados a objetos. Lo guiamos a través de una experiencia de diseño amigable enfocada hacia el principiante.

Biblioteca estándar de C++

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

Observación de Ingeniería de Software 1.1

Utilice un método de "construcción en bloques" para crear programas. Evite reinventar la rueda. Use piezas existentes siempre que sea posible. Esta práctica denominada ***reutilización de software*** *es el fundamento de la programación orientada a objetos.*

Observación de Ingeniería de Software 1.2

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

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

Tip de rendimiento 1.2

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

Tip de portabilidad 1.1

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

1.7 Lenguajes de programación

En esta sección proporcionaremos comentarios breves sobre varios lenguajes de programación populares (figura 1.5).

Lenguaje de programación	Descripción
Fortran	Fortran (FORmula TRANslator, Traductor de fórmulas) fue desarrollado por IBM Corporation a mediados de la década de 1950 para utilizarse en aplicaciones científicas y de ingeniería que requerían cálculos matemáticos complejos. Aún se utiliza mucho y sus versiones más recientes soportan la programación orientada a objetos.
COBOL	COBOL (COmmon Business Oriented Language, Lenguaje común orientado a negocios) fue desarrollado a finales de la década de 1950 por fabricantes de computadoras, el gobierno estadounidense y usuarios de computadoras de la industria, con base en un lenguaje desarrollado por Grace Hopper, un oficial de la Marina de Estados Unidos y científico informático. COBOL aún se utiliza mucho en aplicaciones comerciales que requieren de una manipulación precisa y eficiente de grandes volúmenes de datos. Su versión más reciente soporta la programación orientada a objetos.
Pascal	La investigación en la década de 1960 dio como resultado la *programación estructurada*: un método disciplinado para escribir programas que sean más claros, fáciles de probar y depurar, y más fáciles de modificar que los programas extensos producidos con técnicas anteriores. Uno de los resultados más tangibles de esta investigación fue el desarrollo de Pascal por el profesor Niklaus Wirth en 1971. Se diseñó para la enseñanza de la programación estructurada y fue popular en los cursos universitarios durante varias décadas.
Ada	Ada, un lenguaje basado en Pascal, se desarrolló bajo el patrocinio del Departamento de Defensa (DOD) de Estados Unidos durante la década de 1970 y a principios de la década de 1980. El DOD quería un solo lenguaje que pudiera satisfacer la mayoría de sus necesidades. El nombre de este lenguaje basado en Pascal es en honor de Lady Ada Lovelace, hija del poeta Lord Byron. A ella se le atribuye el haber escrito el primer programa para computadoras en el mundo, a principios de la década de 1800 (para la Máquina Analítica, un dispositivo de cómputo mecánico diseñado por Charles Babbage). Ada también soporta la programación orientada a objetos.
Basic	Basic se desarrolló en la década de 1960 en el Dartmouth College, para que los principiantes se familiarizaran con las técnicas de programación. Muchas de sus versiones más recientes son orientadas a objetos.
C	C fue implementado en 1972 por Dennis Ritchie en los laboratorios Bell. En un principio se hizo muy popular como el lenguaje de desarrollo del sistema operativo UNIX. En la actualidad, la mayoría del código para los sistemas operativos de propósito general se escribe en C o C++.
Objective-C	Objective-C es un lenguaje orientado a objetos basado en C. Se desarrolló a principios de la década de 1980 y después fue adquirido por la empresa NeXT, que a su vez fue adquirida por Apple. Se ha convertido en el lenguaje de programación clave para el sistema operativo Mac OS X y todos los dispositivos operados por el iOS (como los dispositivos iPod, iPhone e iPad).

Fig. 1.5 | Algunos otros lenguajes de programación (parte 1 de 3).

Lenguaje de programación	Descripción
Java	Sun Microsystems patrocinó en 1991 un proyecto de investigación corporativo interno dirigido por James Gosling, que resultó en el lenguaje de programación orientado a objetos basado en C++, conocido como Java. Un objetivo clave de Java es poder escribir programas que se ejecuten en una gran variedad de sistemas de computadora y dispositivos para controlar computadoras. A esto se le conoce algunas veces como "escribir una vez, ejecutar en donde sea". Java se utiliza para desarrollar aplicaciones empresariales a gran escala, mejorar la funcionalidad de servidores Web (las computadoras que proveen el contenido que vemos en nuestros navegadores Web), proveer aplicaciones en dispositivos para el consumidor (smartphones, tabletas, receptores digitales multimedia, aparatos, automóviles y otros más) y para muchos otros propósitos. Java es también el lenguaje clave para desarrollar aplicaciones Android para smartphones y tabletas.
Visual Basic	El lenguaje Visual Basic de Microsoft se introdujo a principios de la década de 1990 para simplificar el desarrollo de aplicaciones para Microsoft Windows. Sus versiones más recientes soportan la programación orientada a objetos.
C#	Los tres principales lenguajes de programación orientados a objetos de Microsoft son Visual Basic (basado en el Basic original), Visual C++ (basado en C++) y C# (basado en C++ y Java; desarrollado para integrar Internet y Web en las aplicaciones de computadora).
PHP	PHP es un lenguaje orientado a objetos de "secuencias de comandos" y "código fuente abierto" (vea la sección 1.11.2), el cual recibe soporte por medio de una comunidad de usuarios y desarrolladores; se utiliza en numerosos sitios Web, incluyendo Wikipedia y Facebook. PHP es independiente de la plataforma —existen implementaciones para todos los principales sistemas operativos UNIX, Linux, Mac y Windows. PHP también soporta muchas bases de datos, incluyendo MySQL.
Perl	Perl (Lenguaje práctico para la extracción e informes), uno de los lenguajes de secuencias de comandos orientados a objetos más utilizados para la programación Web, fue desarrollado en 1987 por Larry Wall. Cuenta con extensas herramientas de procesamiento de texto y mucha flexibilidad.
Python	Python, otro lenguaje de secuencias de comandos orientado a objetos, se liberó al público en 1991. Fue desarrollado por Guido van Rossum del Instituto Nacional de Investigación para las Matemáticas y Ciencias Computacionales en Amsterdam (CWI); la mayor parte de Python se basa en Modula-3 —un lenguaje de programación de sistemas. Python es "extensible": puede extenderse a través de clases e interfaces de programación.
JavaScript	JavaScript es el lenguaje de secuencias de comandos más utilizado en el mundo. Su principal uso es para agregar capacidad de programación a las páginas Web; por ejemplo, animaciones e interactividad con el usuario. Se incluye en todos los principales navegadores Web.

Fig. 1.5 | Algunos otros lenguajes de programación (parte 2 de 3).

Lenguaje de programación	Descripción
Ruby on Rails	Ruby fue creado a mediados de la década de 1990 por Yukihiro Matsumoto; es un lenguaje de programación orientado a objetos de código fuente abierto, con una sintaxis simple que es similar a Perl y Python. Ruby onRails combina el lenguaje de secuencias de comandos Ruby con el marco de trabajo de aplicaciones Web Rails, desarrollado por 37Signals. Su libro, *Getting Real* (disponible sin costo en `gettingreal.37signals.com/toc.php`), es una lectura obligatoria para los desarrolladores Web. Muchos desarrolladores de Ruby onRails han reportado ganancias de productividad superiores a las de otros lenguajes, al utilizar aplicaciones Web que trabajan de manera intensiva con bases de datos. Ruby on Rails se utilizó para crear la interfaz de usuario de Twitter.
Scala	Scala (`www.scala-lang.org/node/273`) abreviación en inglés de "lenguaje escalable", fue diseñado por Martin Odersky, un profesor en la École Polytechnique Fédérale de Lausanne (EPFL) en Suiza. Se lanzó al público en 2003; utiliza los paradigmas de orientación a objetos y de programación funcional, y está diseñado para integrarse con Java. Si programa en Scala, podrá reducir de manera considerable la cantidad de código en sus aplicaciones. Twitter y Foursquare usan Scala.

Fig. 1.5 | Algunos otros lenguajes de programación (parte 3 de 3).

1.8 Introducción a la tecnología de objetos

Crear software en forma rápida, correcta y económica sigue siendo un objetivo difícil de alcanzar en una época en que la demanda de software nuevo y más poderoso va en aumento.Los *objetos*, o dicho en forma más precisa (como veremos en el capítulo 3), las *clases* de las que provienen los objetos, son en esencia componentes de software *reutilizables*. Existen objetos de fecha, de hora, de audio, de video, de automóviles, de personas, etc. Casi cualquier *sustantivo* se puede representar de manera razonable como un objeto de software en términos de sus *atributos* (como el nombre, color y tamaño) y *comportamientos* (por ejemplo, calcular, moverse y comunicarse). Los desarrolladores de software han descubierto que al usar una metodología de diseño e implementación orientada a objetos y modular, pueden crear grupos de desarrollo de software más productivos de lo que era posible con las técnicas anteriores; por lo general los programas orientados a objetos son más fáciles de comprender, corregir y modificar.

El automóvil como un objeto

Empecemos con una analogía simple. Suponga que desea *conducir un automóvil y hacer que vaya más rápido al oprimir el pedal del acelerador*. ¿Qué debe ocurrir para que usted pueda hacer esto? Bueno, antes de que pueda conducir un automóvil, alguien tiene que *diseñarlo*. Por lo general, un automóvil empieza en forma de dibujos de ingeniería, similares a los *planos de construcción* que describen el diseño de una casa. Estos dibujos de ingeniería incluyen el diseño del pedal del acelerador. El pedal *oculta* al conductor los complejos mecanismos que se encargan de que el automóvil aumente su velocidad, de igual forma que el pedal del freno *oculta* los mecanismos que disminuyen la velocidad del automóvil y el volante *oculta* los mecanismos que hacen que el automóvil dé vuelta. Esto permite que las personas con poco o nada de conocimiento acerca de cómo funcionan los motores, los frenos y los mecanismos de la dirección puedan conducir un automóvil con facilidad.

Antes de poder conducir un automóvil, éste debe *construirse* a partir de los dibujos de ingeniería que lo describen. Un automóvil completo tendrá un pedal acelerador *verdadero* para hacer que aumente su velocidad, pero aun así no es suficiente; el automóvil no acelerará por su propia cuenta (¡esperemos que así sea!), así que el conductor debe *oprimir* el pedal del acelerador para aumentar la velocidad del automóvil.

Funciones miembro y clases

Ahora vamos a utilizar nuestro ejemplo del automóvil para introducir algunos conceptos clave de la programación orientada a objetos. Para realizar una tarea en una aplicación se requiere una **función miembro**. Esa función miembro aloja las instrucciones del programa que se encargan de realizar sus tareas. Oculta al usuario estas tareas, de la misma forma que el pedal del acelerador de un automóvil oculta al conductor los mecanismos para hacer que el automóvil vaya más rápido. En C++ creamos una unidad de programa llamada clase para alojar el conjunto de funciones miembro que realizan las tareas de esa **clase**. Por ejemplo, una clase que representa a una cuenta bancaria podría contener una función miembro para *depositar* dinero en una cuenta, otra para *retirar* dinero de una cuenta y una tercera para *solicitar* el saldo actual de la cuenta. Una clase es similar en concepto a los dibujos de ingeniería de un automóvil, que contienen el diseño de un pedal acelerador, volante de dirección, etcétera.

Instanciamiento

Así como alguien tiene que *construir un automóvil* a partir de sus dibujos de ingeniería para que alguien pueda conducirlo después, también es necesario *crear un objeto* de una clase para que un programa pueda realizar las tareas definidas por los métodos de esa clase. Al proceso de hacer esto se le denomina *instanciamiento*. Entonces, un objeto viene siendo una **instancia** de su clase.

Reutilización

Así como los dibujos de ingeniería de un automóvil se pueden *reutilizar* muchas veces para construir muchos automóviles, también es posible *reutilizar* una clase muchas veces para crear muchos objetos. Al reutilizar las clases existentes para crear nuevas clases y programas, ahorramos tiempo y esfuerzo. La reutilización también nos ayuda a crear sistemas más confiables y efectivos, debido a que con frecuencia las clases y los componentes existentes pasan por un extenso proceso de *prueba, depuración* y *optimización del desempeño*. De la misma manera en que la noción de *piezas intercambiables* fue crucial para la Revolución Industrial, las clases reutilizables son cruciales para la revolución de software incitada por la tecnología de objetos.

Mensajes y llamadas a funciones miembro

Cuando conduce un automóvil, al oprimir el pedal del acelerador envía un *mensaje* al automóvil para que realice una tarea: aumentar la velocidad. De manera similar, es posible *enviar mensajes a un objeto*. Cada mensaje se implementa como **llamada a función miembro**, para indicar a una función miembro del objeto que realice su tarea. Por ejemplo, un programa podría llamar al método *depositar* de un objeto cuenta de banco específico para incrementar el saldo de esa cuenta.

Atributos y miembros de datos

Además de tener capacidades para realizar tareas, un automóvil también tiene *atributos*: color, número de puertas, cantidad de gasolina en el tanque, velocidad actual y registro del total de kilómetros recorridos (es decir, la lectura de su velocímetro). Al igual que sus capacidades, los atributos del automóvil se representan como parte de su diseño en sus diagramas de ingeniería (que, por ejemplo, incluyen un velocímetro y un indicador de combustible). Al conducir un automóvil real, estos atributos se llevan junto con el automóvil. Cada automóvil mantiene sus *propios* atributos. Por ejemplo, cada uno sabe cuánta gasolina hay en su tanque, pero *no* cuánta hay en los tanques de *otros* automóviles.

De manera similar, un objeto tiene atributos que lleva consigo a medida que se utiliza en un programa. Estos atributos se especifican como parte de la clase del objeto. Por ejemplo, un objeto cuenta bancaria tiene un *atributo saldo* que representa la cantidad de dinero en la cuenta. Cada objeto cuenta bancaria conoce el saldo de la cuenta que representa, pero *no* los saldos de las *otras* cuentas en el banco. Los atributos se especifican mediante los **miembros de datos** de la clase.

Encapsulamiento

Las clases **encapsulan** (envuelven) los atributos y las funciones miembro en objetos; los atributos y las funciones miembro de un objeto están muy relacionados entre sí. Los objetos se pueden comunicar entre sí, pero por lo general no se les permite saber cómo están implementados otros objetos; los detalles de implementación están *ocultos* dentro de los mismos objetos. Este **ocultamiento de información** es, como veremos, crucial para la buena ingeniería de software.

Herencia

Es posible crear una nueva clase de objetos con rapidez y de manera conveniente mediante la **herencia**: la nueva clase absorbe las características de una clase existente, con la posibilidad de personalizarlas y agregar características únicas propias. En nuestra analogía del automóvil, sin duda un objeto de la clase "convertible" *es un* objeto de la clase más *general* llamada "automóvil" pero, de manera más *específica*, el techo puede ponerse o quitarse.

Análisis y diseño orientado a objetos (A/DOO)

Pronto escribirá programas en C++. ¿Cómo creará el **código** (es decir, las instrucciones) para sus programas? Tal vez, al igual que muchos programadores, sólo encenderá su computadora y empezará a escribir. Quizás este método funcione para pequeños programas (como los que presentamos en los primeros capítulos del libro), pero ¿qué tal si le pidieran crear un sistema de software para controlar miles de cajeros automáticos para un banco importante? O ¿qué tal si le piden que trabaje con un equipo de miles de desarrolladores de software para crear el nuevo sistema de control de tráfico aéreo en Estados Unidos? Para proyectos tan grandes y complejos, no es conveniente tan sólo sentarse y empezar a escribir programas.

Para crear las mejores soluciones, debe seguir un proceso de **análisis** detallado para determinar los **requerimientos** de su proyecto (definir *qué* se supone que debe hacer el sistema) y desarrollar un **diseño** que los satisfaga (decidir *cómo* debe hacerlo el sistema). Lo ideal sería pasar por este proceso y revisar el diseño con cuidado (además de pedir a otros profesionales de software que revisen su diseño) antes de escribir cualquier código. Si este proceso implica analizar y diseñar su sistema desde un punto de vista orientado a objetos, se denomina **proceso de análisis y diseño orientado a objetos (A/DOO)**. Los lenguajes como C++ son orientados a objetos. La programación en un lenguaje de este tipo, conocida como **programación orientada a objetos (POO)**, le permite implementar un diseño orientado a objetos como un sistema funcional.

El UML (Lenguaje Unificado de Modelado)

Aunque existen muchos procesos de A/DOO distintos, hay un solo lenguaje gráfico para comunicar los resultados de *cualquier* proceso de A/DOO que se utiliza en la mayoría de los casos. Este lenguaje, conocido como Lenguaje Unificado de Modelado (UML), es en la actualidad el esquema gráfico más utilizado para modelar sistemas orientados a objetos. Presentamos nuestros primeros diagramas de UML en los capítulos 3 y 4; después los utilizamos en nuestro análisis más detallado de la programación orientada a objetos en el capítulo 12. En nuestro caso de estudio *opcional* de ingeniería de software del ATM en los capítulos 25 y 26 presentamos un subconjunto simple de las características del UML, mientras lo guiamos por una experiencia de diseño orientada a objetos.

1.9 Entorno de desarrollo común en C++

Por lo general, los sistemas de C++ consisten en tres partes: un entorno de desarrollo de programas, el lenguaje y la Biblioteca estándar de C++. Comúnmente, los programas en C++ pasan a través de seis fases: edición, preprocesamiento, compilación, enlace, carga y ejecución. A continuación explicaremos un típico entorno de desarrollo de programas en C++.

Fase 1: Edición (Creación) de un programa
La fase 1 consiste en editar un archivo con un *programa de edición*, conocido comúnmente como *editor* (figura 1.6). Usted escribe un programa en C++ (conocido por lo general como **código fuente**) utilizando el editor, realiza las correcciones necesarias y guarda el programa en un dispositivo de almacenamiento secundario, tal como su disco duro. A menudo, los nombres de archivos de código fuente en C++ terminan con las extensiones `.cpp`, `.cxx`, `.cc` o `.C` (observe que `C` está en mayúsculas), para indicar que un archivo contiene código fuente en C++. Consulte la documentación para su compilador de C++ si desea obtener más información acerca de las extensiones de nombres de archivos.

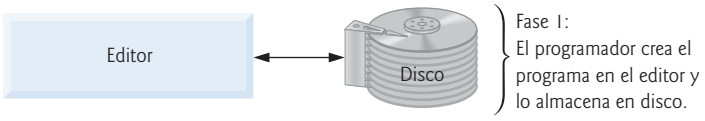


Fig. 1.6 | Entorno de desarrollo común en C++: fase de edición.

Dos de los editores que se utilizan ampliamente en sistemas Linux son `vi` y `emacs`. Los paquetes de software de C++ para Microsoft Windows tales como Microsoft Visual C++ (`microsoft.com/express` en inglés, y `www.microsoft.com/visualstudio/esn#products/visual-studio-express-products` en español) tienen editores integrados en el entorno de programación. También puede utilizar un editor de texto simple, como el Bloc de notas en Windows, para escribir su código de C++.

Para las organizaciones que desarrollan sistemas de información de un tamaño considerable, hay **entornos integrados de desarrollo (IDE)** disponibles de muchos de los principales proveedores de software. Los IDE ofrecen herramientas para apoyar el proceso de desarrollo de software, incluyendo editores para escribir y editar programas, y depuradores para localizar **errores lógicos**: errores que provocan que los programas se ejecuten en forma incorrecta. Los IDE populares son: Microsoft® Visual Studio 2012 Express Edition, Dev C++, NetBeans, Eclipse, Xcode de Apple y CodeLite.

Fase 2: Preprocesamiento de un programa en C++
En la fase 2, se introduce el comando para **compilar** el programa (figura 1.7). En un sistema de C++, un programa **preprocesador** se ejecuta de manera automática antes de que empiece la fase de traducción del compilador (por lo que a la fase 2 la llamamos preprocesamiento, y a la fase 3 la llamamos compilación). El preprocesador de C++ obedece a comandos denominados **directivas del preprocesador**, las cuales indican que deben realizarse ciertas manipulaciones en el programa, antes de compilarlo. Por lo general, estas manipulaciones incluyen la compilación de otros archivos de texto y realizan varios reemplazos de texto. Las directivas más comunes del preprocesador se describen en los primeros capítulos; en el apéndice E, Preprocesador, aparece una discusión detallada acerca de las características del preprocesador.

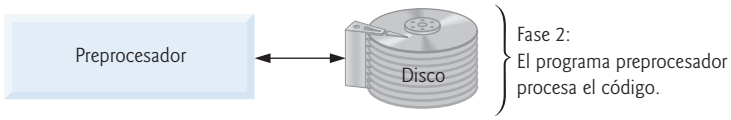


Fig. 1.7 | Entorno de desarrollo común en C++: fase del preprocesador.

Fase 3: Compilación de un programa de C++
En la fase 3, el compilador traduce el programa de C++ en código de lenguaje máquina (también conocido como código objeto) (figura 1.8).

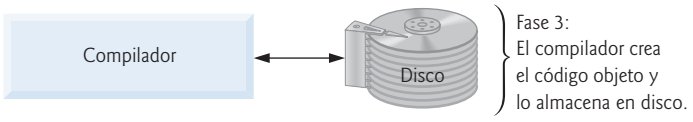


Fig. 1.8 | Entorno de desarrollo común en C++: fase de compilación.

Fase 4: Enlace
A la fase 4 se le llama **enlace**. Por lo general, los programas en C++ contienen referencias a funciones y datos definidos en otra parte, como en las bibliotecas estándar o en las bibliotecas privadas de grupos de programadores que trabajan sobre un proyecto específico (figura 1.9). El código objeto producido por el compilador de C++ comúnmente contiene "huecos", debido a estas partes faltantes. Un **enlazador** relaciona el código objeto con el código para las funciones faltantes, de manera que se produzca un **programa ejecutable** (sin piezas faltantes). Si el programa se compila y se enlaza de manera correcta, se produce una imagen ejecutable.

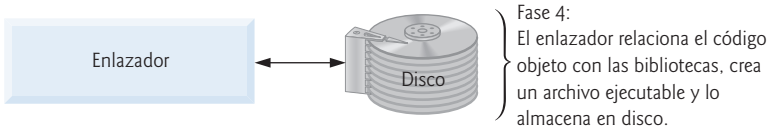


Fig. 1.9 | Entorno de desarrollo común en C++: fase de enlace.

Fase 5: Carga
A la fase 5 se le conoce como **carga**. Antes de poder ejecutar un programa, primero se debe colocar en la memoria (figura 1.10). Esto se hace mediante el **cargador**, que toma la imagen ejecutable del disco y la transfiere a la memoria. También se cargan los componentes adicionales de bibliotecas compartidas que dan soporte al programa.

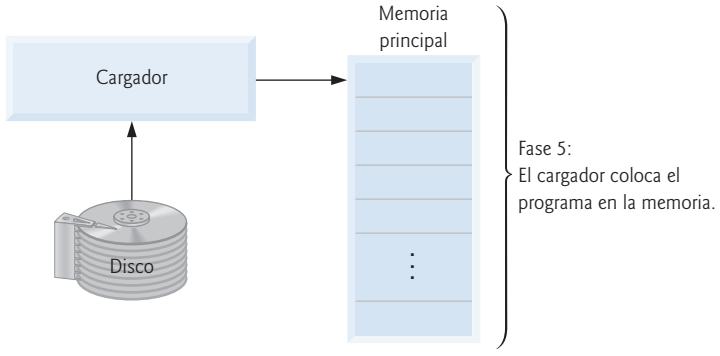


Fig. 1.10 | Entorno de desarrollo común en C++: fase de carga.

Fase 6: Ejecución
Por último, la computadora, bajo el control de su CPU, ejecuta el programa una instrucción a la vez (figura 1.11). Algunas arquitecturas de cómputo modernas pueden ejecutar varias instrucciones en paralelo.

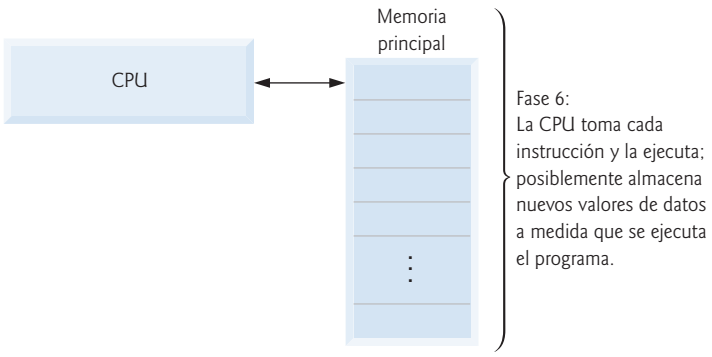


Fig. 1.11 | Entorno de desarrollo común en C++: fase de ejecución.

Problemas que pueden ocurrir en tiempo de ejecución
Es probable que los programas no funcionen la primera vez. Cada una de las fases anteriores puede fallar, debido a diversos errores que describiremos en este texto. Por ejemplo, un programa en ejecución podría intentar una división entre cero (una operación ilegal para la aritmética con números enteros en C++). Esto haría que el programa de C++ mostrara un mensaje de error. Si esto ocurre, tendría que regresar a la fase de edición, hacer las correcciones necesarias y proseguir con las fases restantes de nuevo, para determinar que las correcciones hayan resuelto el(los) problema(s). [*Nota:* la mayoría de los programas en C++ reciben y/o producen datos. Ciertas funciones de C++ toman su entrada de `cin` (el **flujo estándar de entrada**; se pronuncia "c-in"), que por lo general es el teclado, pero `cin` puede redirigirse a otro dispositivo. A menudo los datos se envían a `cout` (el **flujo estándar de salida**), que por lo general es la pantalla de la computadora, pero `cout` puede redirigirse a otro dispositivo. Cuando decimos que un programa imprime un resultado, por lo general nos referimos a que el resultado se despliega en una pantalla. Los datos pueden enviarse a otros dispositivos, como los discos y las impresoras. También hay un **flujo estándar de error**, conocido como **`cerr`**. El flujo `cerr` (que por lo general se conecta a la pantalla) se utiliza para mostrar mensajes de error.

Error común de programación 1.1
Los errores, como la división entre cero, ocurren a medida que se ejecuta un programa, por lo cual se les llama ***errores en tiempo de ejecución****. Los* ***errores fatales en tiempo de ejecución*** *hacen que los programas terminen inmediatamente, sin haber realizado correctamente su trabajo. Los errores no fatales en tiempo de ejecución permiten a los programas ejecutarse hasta terminar su trabajo, lo que a menudo produce resultados incorrectos.*

1.10 Prueba de una aplicación de C++

En esta sección, ejecutará su primera aplicación en C++ e interactuará con ella. Empezará ejecutando un divertido juego de "adivinar el número", el cual elije un número del 1 al 1 000 y le pide que lo adivine. Si su elección es la correcta, el juego termina. Si no es correcta, la aplicación le indica si su elección

es mayor o menor que el número correcto. No hay límite en cuanto al número de intentos que puede realizar [*Nota:* sólo para esta prueba hemos modificado esta aplicación del ejercicio que le pediremos que cree en el capítulo 6, Funciones y una introducción a la recursividad. Por lo general, esta aplicación selecciona al azar la respuesta correcta cuando usted ejecuta el programa. La aplicación modificada utiliza la misma respuesta correcta cada vez que el programa se ejecuta (aunque esto podría variar según el compilador), por lo que puede utilizar las *mismas* elecciones que utilizamos en esta sección, y verá los *mismos* resultados a medida que interactúe con su primera aplicación en C++].

Demostraremos cómo ejecutar una aplicación de C++ mediante el **Símbolo del sistema** de Windows y mediante un shell en Linux. La aplicación se ejecuta de manera similar en ambas plataformas. Hay muchos entornos de desarrollo disponibles en los cuales los lectores pueden compilar, generar y ejecutar aplicaciones de C++, como GNU C++, Microsoft Visual C++, Apple Xcode, Dev C++, CodeLite, NetBeans, Eclipse, etc. Consulte con su instructor para obtener información acerca de su entorno de desarrollo específico.

En los siguientes pasos, ejecutará la aplicación y escribirá varios números para adivinar el número correcto. Los elementos y la funcionalidad que puede ver en esta aplicación son típicos de los que aprenderá a programar en este libro. A lo largo del mismo, utilizamos distintos tipos de letra para diferenciar entre las características que se pueden ver en la pantalla (como el **Símbolo del sistema**) y los elementos que no están directamente relacionados con la pantalla. Enfatizamos las características de la pantalla, como los títulos y los menús (como el menú **Archivo** menu) en un tipo de letra **Helvetica sans-serif** en semi-negritas, y enfatizamos los nombres de archivo, el texto desplegado por una aplicación y los valores que debe introducir en una aplicación (como `AdivinarNumero` o `500`) en un tipo de letra `Lucida sans-serif`. Como tal vez ya se ha dado cuenta, la **ocurrencia de definición** de cada término se establece en negritas. En las figuras en esta sección, señalamos las partes importantes de la aplicación. Para aumentar la visibilidad de estas características, modificamos el color de fondo de la ventana del **Símbolo del sistema** (sólo para la prueba en Windows). Para modificar los colores del **Símbolo del sistema** en su sistema, abra una ventana **Símbolo del sistema** seleccionando **Inicio > Todos los programas > Accesorios > Símbolo del sistema**, después haga clic con el botón derecho del ratón en la barra de título y seleccione **Propiedades**. En el cuadro de diálogo **Propiedades de "Símbolo del sistema"** que aparezca, haga clic en la ficha **Colores** y seleccione sus colores de texto y fondo preferidos.

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

1. ***Revise su configuración***. Es importante que lea la sección Antes de empezar en `www.deitel.com/books/cpphtp9/` para confirmar que haya copiado correctamente los ejemplos del libro en su disco duro.

2. ***Localice la aplicación completa***. Abra una ventana **Símbolo del sistema** window. Para cambiar al directorio de la aplicación completa **AdivinarNumero**, escriba **`cd C:\ejemplos\cap01\AdivinarNumero\Windows`** y después oprima *Intro* (figura 1.12). El comando `cd` se utiliza para cambiar de directorio.

Fig. 1.12 | Abrir una ventana **Símbolo del sistema** y cambiar de directorio.

3. ***Ejecute la aplicación*** *AdivinarNumero*. Ahora que se encuentra en el directorio que contiene la aplicación **AdivinarNumero** escriba el comando **`AdivinarNumero`** (figura 1.13) y oprima *Intro*. [*Nota:* `AdivinarNumero.exe` es el nombre real de la aplicación; sin embargo, Windows asume la extensión `.exe` de manera predeterminada].

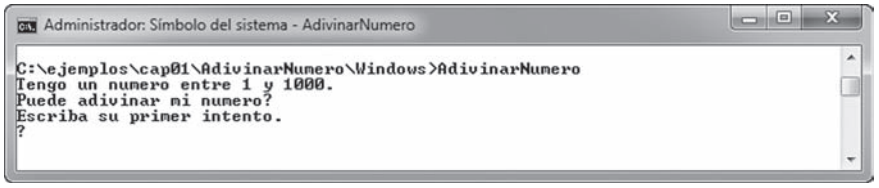


Fig. 1.13 | Ejecución de la aplicación **AdivinarNumero**.

4. ***Escriba su primer intento***. La aplicación muestra el mensaje `"Escriba su primer intento."` y después muestra un signo de interrogación (?) como un indicador en la siguiente línea (figura 1.13). En el indicador, escriba **200** (figura 1.14).

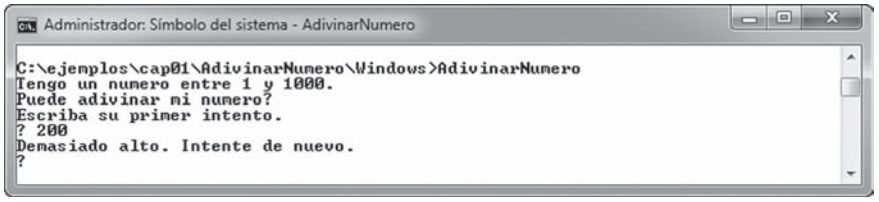


Fig. 1.14 | Escriba su primer intento.

5. ***Intente de nuevo***. La aplicación muestra `"Demasiado alto. Intente de nuevo."` lo cual significa que el valor que escribió es mayor que el número que eligió la aplicación como la respuesta correcta. Por lo tanto, debe escribir un número menor como su siguiente intento. En el indicador, escriba **100** (figura 1.15). La aplicación muestra de nuevo el mensaje `"Demasiado alto. Intente de nuevo."`, ya que el valor que escribió sigue siendo mayor que el número que eligió la aplicación como la respuesta correcta.

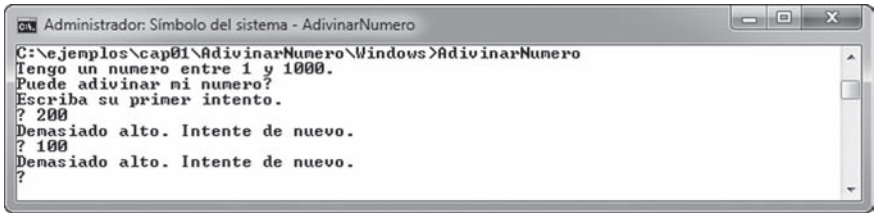


Fig. 1.15 | Intente nuevamente y reciba retroalimentación.

6. ***Escriba más elecciones***. Continúe el juego, escribiendo valores hasta que adivine el número correcto. La aplicación mostrará el mensaje `"Excelente! Adivino el numero!"` (figura 1.16).

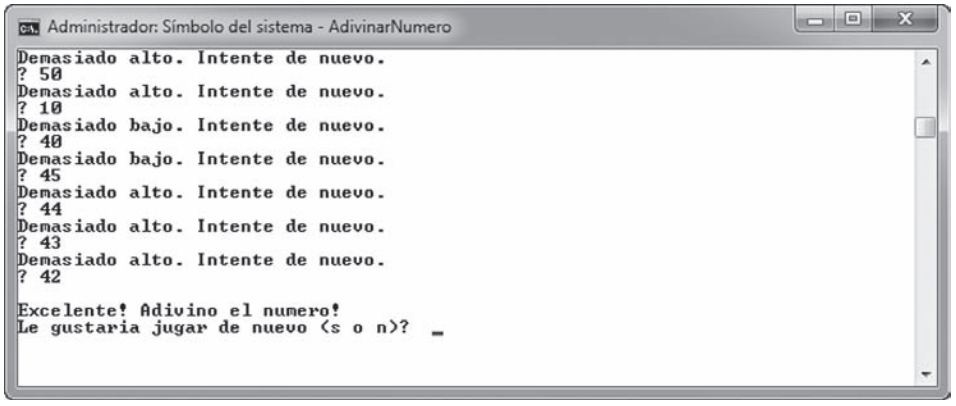


Fig. 1.16 | Escriba más elecciones y adivine el número correcto.

7. ***Juegue de nuevo o salga de la aplicación.*** Una vez que adivine la respuesta correcta, la aplicación le preguntará si desea jugar otra vez (figura 1.16). En el indicador `"Le gustaria jugar de nuevo (s o n)?"` al escribir el carácter **s** la aplicación elegirá un nuevo número y mostrará el mensaje `"Escriba su primer intento."` seguido de un signo de interrogación como indicador (figura 1.17), de manera que pueda escribir su primer intento en el nuevo juego. Al escribir el carácter **n** la aplicación termina y el sistema nos regresa al directorio de la aplicación en el **Símbolo del sistema** (figura 1.18). Cada vez que ejecute esta aplicación desde el principio (es decir, desde el *paso 3*), elegirá los mismos números para que usted adivine.

8. ***Cierre la ventana*** *Símbolo del sistema.*

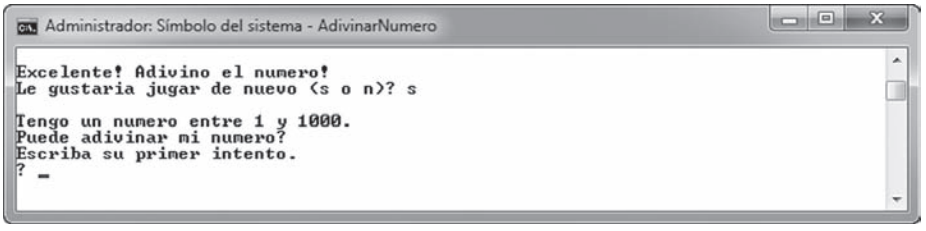


Fig. 1.17 | Juegue de nuevo.

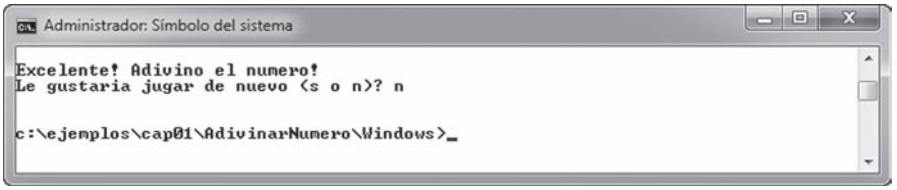


Fig. 1.18 | Salga del juego.

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

1. ***Localice la aplicación completa.*** Desde un shell en Linux, cambie al directorio de la aplicación **AdivinarNumero** completa (figura 1.19); para ello escriba

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

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

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

Fig. 1.19 | Cambie al directorio de la aplicación **AdivinarNumero**.

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

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

como en la figura 1.20. Este comando compila la aplicación y produce un archivo ejecutable, llamado `AdivinarNumero`.

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

Fig. 1.20 | Compile la aplicación **AdivinarNumero** usando el comando g++.

3. ***Ejecute la aplicación*** *AdivinarNumero*. Para ejecutar el archivo `AdivinarNumero`, escriba `./AdivinarNumero` en el siguiente indicador y luego oprima *Intro* (figura 1.21).

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

Fig. 1.21 | Ejecución de la aplicacióne **AdivinarNumero**.

4. ***Escriba su primer intento.*** La aplicación muestra el mensaje `"Escriba su primer intento."`, y después muestra un signo de interrogación (?) como un indicador en la siguiente línea (figura 1.21). En el indicador, escriba **100** (figura 1.22). [*Nota:* ésta es la misma aplicación que modificamos y probamos para Windows, pero los resultados podrían variar, dependiendo del compilador que se utilice].

5. ***Intente de nuevo.*** La aplicación muestra `"Demasiado alto. Intente de nuevo."`, lo cual significa que el valor que escribió es mayor que el número que eligió la aplicación como la respuesta correcta (figura 1.22). En el siguiente indicador, escriba **30** (figura 1.23). Esta vez la aplicación muestra el mensaje `"Demasiado bajo. Intente de nuevo."`, ya que el valor que escribió es menor que el número que la respuesta correcta.

6. ***Escriba más elecciones.*** Continúe el juego (figura 1.24), escribiendo valores hasta que adivine el número correcto. Cuando adivine la respuesta correcta, la aplicación mostrará el mensaje `"Excelente! Adivino el numero."`

```
~/ejemplos/cap01/AdivinarNumero/GNU_Linux$ ./AdivinarNumero
Tengo un numero entre 1 y 1000.
Puede adivinar mi numero?
Escriba su primer intento.
? 100
Demasiado alto. Intente de nuevo.
?
```

Fig. 1.22 | Escriba su elección inicial.

```
~/ejemplos/cap01/AdivinarNumero/GNU_Linux$ ./AdivinarNumero
Tengo un numero entre 1 y 1000.
Puede adivinar mi numero?
Escriba su primer intento.
? 100
Demasiado alto. Intente de nuevo.
? 30
Demasiado bajo. Intente de nuevo.
?
```

Fig. 1.23 | Intente de nuevo y reciba retroalimentación.

```
Demasiado bajo. Intente de nuevo.
? 40
Demasiado bajo. Intente de nuevo.
? 60
Demasiado alto. Intente de nuevo.
? 50
Demasiado alto. Intente de nuevo.
? 45
Demasiado alto. Intente de nuevo.
? 41
Demasiado bajo. Intente de nuevo.
? 44
Demasiado alto. Intente de nuevo.
? 43
Demasiado alto. Intente de nuevo.
? 42
Excelente! Adivino el numero!
Le gustaria jugar de nuevo (s o n)?
```

Fig. 1.24 | Escriba más elecciones y adivine el número correcto.

7. ***Juegue de nuevo o salga de la aplicación.*** Una vez que adivine la respuesta correcta, la aplicación le preguntará si desea jugar otra vez. En el indicador `"Le gustaria jugar de nuevo (s o n)?"` al escribir el carácter **`s`** la aplicación elegirá un nuevo número y mostrará el mensaje "`Escriba su primer intento.`" seguido de un signo de interrogación como indicador (figura 1.25), de manera que pueda escribir su primera elección en el nuevo juego. Al escribir el carácter **`n`** la aplicación termina y el sistema nos regresa al directorio de la aplicación en el shell (figura 1.26). Cada vez que ejecute esta aplicación desde el principio (es decir, desde el *paso 3*), elegirá los mismos números para que usted adivine.

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

Tengo un numero entre 1 y 1000.
Puede adivinar mi numero?
Escriba su primer intento.
?
```

Fig. 1.25 | Juegue de nuevo.

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

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

Fig. 1.26 | Salga del juego.

1.11 Sistemas operativos

Los **sistemas operativos** son sistemas de software que se encargan de hacer más conveniente el uso de las computadoras para los usuarios, desarrolladores de aplicaciones y administradores de sistemas. Proveen servicios que permiten a cada aplicación ejecutarse en forma segura, eficiente y *concurrente* (es decir, en paralelo) con otras aplicaciones. El software que contiene los componentes básicos del sistema operativo se denomina **kernel**. Los sistemas operativos de escritorio populares son: Linux, Windows y OS X (conocido anteriormente como Mac OS X); utilizamos estos tres en el desarrollo del libro. Los sistemas operativos móviles populares que se utilizan en smartphones y tabletas son: Android de Google, Apple iOS (para sus dispositivos iPhone, iPad e iPod Touch), BlackBerry OS y Windows Phone. Es posible desarrollar aplicaciones en C++ para los siguientes sistemas operativos clave, incluyendo varios de los más recientes sistemas operativos móviles.

1.11.1 Windows: un sistema operativo propietario

A mediados de la década de 1980 Microsoft desarrolló el **sistema operativo Windows**, el cual consiste en una interfaz gráfica de usuario creada sobre DOS: un sistema operativo de computadora personal muy popular con el que los usuarios interactuaban al *teclear* comandos. Windows tomó prestados muchos conceptos (como los iconos, menús y ventanas) desarrollados en un principio por Xerox PARC y que se hicieron populares gracias a los primeros sistemas operativos Apple Macintosh. Windows 8 es el

sistema operativo más reciente de Microsoft; sus características incluyen mejoras en la interfaz de usuario, un arranque más veloz, un mayor grado de refinamiento en cuanto a las características de seguridad, soporte para pantalla táctil y multitáctil, y otras cosas más. Windows es un sistema operativo *propietario*; está bajo el control exclusivo de Microsoft. Windows es por mucho el sistema operativo de escritorio más utilizado en el mundo.

1.11.2 Linux: un sistema operativo de código fuente abierto

El sistema operativo Linux es tal vez el más grande éxito del movimiento de *código fuente abierto*. El **software de código fuente abierto** se desvía del estilo de desarrollo de software *propietario*, el cual predominó durante los primeros años del software. Con el desarrollo de código fuente abierto, individuos y compañías *contribuyen* sus esfuerzos para desarrollar, mantener y evolucionar el software a cambio del derecho de usarlo para sus propios fines, por lo general *sin costo*. A menudo el código fuente abierto es escudriñado por una audiencia mucho mayor que la del software propietario, de modo que casi siempre los errores se eliminan con más rapidez. El código fuente abierto también fomenta una mayor innovación. Las compañías de sistemas empresariales como IBM, Oracle y muchas otras, han realizado inversiones considerables en el desarrollo del código fuente abierto de Linux.

Algunas organizaciones clave en la comunidad de código fuente abierto son: la fundación Eclipse (el Entorno integrado de desarrollo Eclipse ayuda a los programadores a desarrollar software de manera conveniente), la fundación Mozilla (creadores del navegador Web Firefox), la fundación de software Apache (creadores del servidor Web Apache que se utiliza para desarrollar aplicaciones basadas en Web) y SourceForge (quien proporciona las herramientas para administrar proyectos de código fuente abierto; cientos de miles de estos proyectos en desarrollo). Las rápidas mejoras en la computación y las comunicaciones, la reducción en costos y el software de código fuente abierto han logrado que sea mucho más fácil y económico crear un negocio basado en software en la actualidad de lo que era hace tan sólo una década. Facebook es un gran ejemplo de ello; este sitio se inició desde un dormitorio universitario y se creó con software de código fuente abierto.

El kernel de **Linux** es el núcleo del sistema operativo de código fuente abierto más popular y lleno de funcionalidades, que se distribuye en forma gratuita. Es desarrollado por un equipo de voluntarios organizados de manera informal; es popular en servidores, computadoras personales y sistemas incrustados. A diferencia de los sistemas operativos propietarios como Microsoft Windows y Apple OS X, el código fuente de Linux (el código del programa) está disponible al público para que lo examinen y modifiquen; además se puede descargar e instalar sin costo. Como resultado, los usuarios de Linux se benefician de una comunidad de desarrolladores que depuran y mejoran el kernel de manera continua, y de la habilidad de poder personalizar el sistema operativo para cumplir necesidades específicas.

Son varias cuestiones —el poder de mercado de Microsoft, el pequeño número de aplicaciones Linux amigables para los usuarios y la diversidad de distribuciones de Linux, tales como Red Hat Linux, Ubuntu Linux y muchas más— las que han impedido que se popularice el uso de Linux en las computadoras de escritorio. Pero este sistema operativo se ha vuelto muy popular en servidores y sistemas incrustados, como los teléfonos inteligentes basados en Android de Google.

1.11.3 OS X de Apple: iOS de Apple para dispositivos iPhone®, iPad® y iPod Touch®

Steve Jobs y Steve Wozniak fundaron Apple en 1976, que se convirtió rápidamente en líder en la computación personal. En 1979, Jobs y varios empleados de Apple visitaron Xerox PARC (Centro de investigación de Palo Alto) para aprender sobre la computadora de escritorio de Xerox que contaba con una interfaz gráfica de usuario (GUI). Esa GUI sirvió como inspiración para la Apple Macintosh, que se lanzó con muchas fanfarrias en un memorable anuncio del Súper Tazón de 1984.

El lenguaje de programación Objective-C, creado por Brad Cox y Tom Love en Stepstone a principios de la década de 1980, agregó capacidades de programación orientada a objetos (OOP) al lenguaje de programación C. Al momento de escribir este libro, Objective-C se comparaba en popularidad con C++.[6] Steve Jobs dejó Apple en 1985 y fundó NeXT, Inc. En 1988, NeXT obtuvo la licencia para Objective-C de StepStone y desarrolló un compilador de Objective-C además de varias bibliotecas, lo cual se usó como la plataforma para la interfaz de usuario el sistema operativo NEXTSTEP y para Interface Builder (que se usaba para construir interfaces gráficas de usuario).

Jobs regresó a Apple en 1996, cuando esta empresa compró NeXT. El sistema operativo OS X de Apple es descendiente de NeXTSTEP. El sistema operativo propietario de Apple, **iOS**, se deriva del Apple OS X y se utiliza en los dispositivos iPhone, iPad y iPod Touch.

1.11.4 Android de Google

Android —el sistema operativo para dispositivos móviles y smartphones, cuyo crecimiento ha sido el más rápido hasta ahora— está basado en el kernel de Linux y en Java. Los programadores experimentados de Java no tienen problemas para entrar y participar con rapidez en el desarrollo de aplicaciones para Android. Un beneficio de desarrollar este tipo de aplicaciones es el grado de apertura de la plataforma. El sistema operativo es gratuito y de código fuente abierto.

El sistema operativo Android fue desarrollado por Android, Inc., compañía que adquirió Google en 2005. En 2007 se formó la Alianza para los dispositivos móviles abiertos™ (OHA) —un consorcio de 34 compañías en un principio que creció a 84 para 2011—, para continuar con el desarrollo de Android. Al mes de junio de 2012, ¡se activaban más de 900 000 teléfonos inteligentes con Android a diario![7] Ahora los smartphones Android se venden más que los iPhone en Estados Unidos.[8] El sistema operativo Android se utiliza en varios smartphones (Motorola Droid, HTC One S, Samsung Galaxy Nexus y muchos más), dispositivos lectores electrónicos (como el Kindle Fire y el Nook™ de Barnes and Noble), computadoras tipo tableta (como Dell Streak y Samsung Galaxy Tab), quioscos con pantallas táctiles dentro de las tiendas, automóviles, robots, reproductores multimedia y demás.

1.12 Internet y World Wide Web

Internet (una red global de computadoras) se hizo posible gracias a la *convergencia de las tecnologías de la computación y las comunicaciones.* A finales de la década de 1960, ARPA (la Agencia de proyectos de investigación avanzados) extendió los planos para conectar en red los sistemas de cómputo principales de alrededor de una docena de universidades e instituciones de investigación patrocinadas por la ARPA. La investigación académica estaba a punto de dar un gigantesco paso hacia delante. La ARPA procedió a implementar la **ARPANET**, que eventualmente se convirtió en la **Internet** que conocemos. Pronto se volvió claro que el primer beneficio clave de ARPANET era la capacidad de comunicarse con rapidez y facilidad mediante correo electrónico. Esto es cierto incluso en Internet de la actualidad, ya que facilita las comunicaciones de todo tipo entre los usuarios de Internet a nivel mundial.

Conmutación de paquetes

Uno de los principales objetivos de ARPANET fue permitir que *múltiples* usuarios enviaran y recibieran información al mismo tiempo y a través de las *mismas* rutas de comunicaciones (por ejemplo, las líneas

6 www.tiobe.com/index.php/content/paperinfo/tpci/index.html.
7 mashable.com/2012/06/11/900000-android-devices/.
8 www.pcworld.com/article/196035/android_outsells_the_iphone_no_big_surprise.html.

telefónicas). La red operaba mediante una técnica conocida como **conmutación de paquetes**, en donde los datos digitales se enviaban en pequeños grupos llamados **paquetes**. Los paquetes contenían información de *dirección, control de errores* y *secuencia*. La información de la dirección permitía *enrutar* los paquetes hacia sus destinos. La información de secuencia ayudaba a *volver a ensamblar* los paquetes —ya que debido a los complejos mecanismos de enrutamiento podrían llegar desordenados— en su orden original para presentarlos al receptor. Los paquetes de distintos emisores se entremezclaban en las *mismas* líneas para usar con eficiencia el ancho de banda disponible. Esta técnica de conmutación de paquetes redujo de manera considerable los costos de transmisión, en comparación con el costo de las líneas de comunicaciones *dedicadas*.

La red se diseñó para operar sin un control centralizado. Si fallaba una parte de la red, el resto de las porciones funcionales podrían seguir enrutando paquetes de los emisores a los receptores a través de rutas alternativas para mejorar la confiabilidad.

TCP/IP

El protocolo (conjunto de reglas) para comunicarse a través de ARPANET se conoció como **TCP**: **Protocolo de control de transmisión**. TCP aseguraba que los mensajes se enrutaran en forma correcta del emisor hacia el receptor, y que llegaran intactos.

A medida que evolucionó Internet, organizaciones de todo el mundo estaban implementando sus propias redes. Un desafío fue lograr que estas distintas redes se comunicaran. La ARPA lo logró con el desarrollo de **IP**: el **Protocolo de Internet**, con lo que verdaderamente creó una red de redes, la arquitectura actual de Internet. El conjunto combinado de protocolos se conoce ahora comúnmente como **TCP/IP**.

World Wide Web, HTML, HTTP

World Wide Web nos permite localizar y ver documentos basados en multimedia sobre casi cualquier tema en Internet. La Web es una creación relativamente reciente. En 1989, Tim Berners-Lee de CERN (la Organización Europea para la Investigación Nuclear) empezó a desarrollar una tecnología para compartir información a través de documentos de texto con hipervínculos. A esta invención, Berners-Lee la llamó **Lenguaje de marcado de hipertexto (HTML)**. También escribió protocolos de comunicación para formar la espina dorsal de su nuevo sistema de información, al cual llamó World Wide Web. En especial escribió el **Protocolo de transferencia de hipertexto (HTTP)**: un protocolo de comunicaciones utilizado para enviar información a través de Web. El **URL (Localizador uniforme de recursos)** especifica la dirección (ubicación) de la página Web que se visualiza en la ventana del navegador. Cada página Web en Internet se asocia con un URL único. El **Protocolo seguro de transferencia de hipertexto (HTTPS)** es el estándar para transferir datos cifrados en Web.

Mosaic, Netscape, surgimiento de Web 2.0

El uso de Web explotó con la disponibilidad en 1993 del navegador Mosaic, el cual incluía una interfaz gráfica amigable para el usuario. Marc Andreessen, cuyo equipo en el Centro nacional de aplicaciones de supercomputación (NCSA) desarrolló Mosaic, fue el fundador de Netscape, la empresa que muchas personas consideran fue la mecha que encendió la explosiva economía de Internet a finales de la década de 1990.

En 2003 hubo un cambio considerable en cuanto a la forma en que las personas y las empresas usaban la Web y desarrollaban aplicaciones basadas en Web. Dale Dougherty de O'Reilly Media[9] inventó el término **Web 2.0** en 2003 para describir esta tendencia. Por lo general, las compañías Web 2.0 usan la Web como una plataforma para crear sitios colaborativos basados en comunidades (como sitios de redes sociales, blogs y wikis).

---

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

Algunas empresas con características de Web 2.0 son: Google (búsqueda Web), YouTube (compartición de videos), Facebook (redes sociales), Twitter (microblogs), Groupon (comercio social), Foursquare (registro móvil), Salesforce (software de negocios que se ofrece en forma de servicios en línea "en la nube"), Craigslist (en su mayoría, anuncios clasificados gratuitos), Flickr (compartición de fotografías), Skype (telefonía, videollamadas y conferencias por Internet) y Wikipedia (una enciclopedia gratuita en línea).

Web 2.0 *involucra* a los usuarios; no sólo crean contenido con frecuencia, sino que ayudan a organizarlo, compartirlo, volver a mezclarlo, criticarlo, actualizarlo, etc. Web 2.0 es una *conversación*, en donde todos tienen la oportunidad de hablar y compartir opiniones. Las empresas que entienden la Web 2.0 saben que sus productos y servicios son conversaciones también.

Arquitectura de participación

La Web 2.0 abarca una **arquitectura de participación**: un diseño que fomenta la interacción del usuario y las contribuciones comunitarias. Usted el usuario es el aspecto más importante de Web 2.0; de hecho, es tan importante que en 2006 la "Persona del año" de la revista *TIME* fue "usted".[10] El artículo reconoció el fenómeno social de Web 2.0: el cambio de *unos cuantos poderosos* a *muchos empoderados*. Ahora varios blogs populares compiten con los poderosos medios tradicionales y muchas empresas Web 2.0 están basadas casi por completo en contenido generado por el usuario. Para sitios Web como Facebook, Twitter, YouTube, eBay y Wikipedia, los usuarios crean el contenido, mientras que las empresas proporcionan las *plataformas* en las cuales se va a introducir, manipular y compartir la información.

1.13 Cierta terminología clave de desarrollo de software

La figura 1.27 muestra una lista de palabras de moda que escuchará en la comunidad de desarrollo de software. Creamos Centros de Recursos sobre la mayoría de estos temas, y hay muchos por venir.

Tecnología	Descripción
Ajax	**Ajax** es una de las tecnologías de software Premier de Web 2.0. Ajax ayuda a que las aplicaciones basadas en Internet se ejecuten como aplicaciones de escritorio; una tarea difícil, dado que dichas aplicaciones sufren de retrasos en la transmisión a medida que los datos se intercambian entre su computadora y los servidores en Internet.
Desarrollo ágil de software	El **desarrollo ágil de software** es un conjunto de metodologías que tratan de implementar software con más rapidez y menos recursos que las metodologías anteriores. Visite los sitios de Agile Alliance (`www.agilealliance.org`) y Agile Manifesto (`www.agilemanifesto.org`). También puede visitar el sitio en español (`www.agile-spain.com`).
Refactorización	La **refactorización** implica reformular programas para hacerlos más claros y fáciles de mantener, al tiempo que se preserva su funcionalidad e integridad. Es muy utilizado en las metodologías de desarrollo ágil. Muchos IDE contienen *herramientas de refactorización* integradas para realizar la mayor parte del proceso de refactorización de manera automática.

Fig. 1.27 | Tecnologías de software (parte 1 de 2).

10 `www.time.com/time/magazine/article/0,9171,1570810,00.html`.

Tecnología	Descripción
Patrones de diseño	Los **patrones de diseño** son arquitecturas probadas para construir software orientado a objetos flexible y que pueda mantenerse. El campo de los patrones de diseño trata de enumerar a los patrones recurrentes, y de alentar a los diseñadores de software para que los reutilicen y puedan desarrollar un software de mejor calidad con menos tiempo, dinero y esfuerzo.
LAMP	**LAMP** es un acrónimo para el conjunto de tecnologías de código fuente abierto que usan muchos desarrolladores en la creación de aplicaciones Web: se refiere a Linux, Apache, MySQL y PHP (o Perl, o Python; otros dos lenguajes que se usan para propósitos similares). MySQL es un sistema de administración de bases de datos de código fuente abierto. PHP es el lenguaje de "secuencias de comandos" del lado servidor de código fuente abierto más popular para el desarrollo de aplicaciones basadas en Internet.
Software como un servicio (SaaS)	Por lo general, el software siempre se ha visto como un producto; la mayoría del software aún se ofrece de esta forma. Para ejecutar una aplicación, hay que comprar el paquete a un distribuidor de software; por lo general un CD, DVD o descarga Web. Después instalamos ese software en la computadora y la ejecutamos cuando sea necesario. A medida que aparecen nuevas versiones, actualizamos el software, lo cual genera con frecuencia un tiempo y un gasto considerables. Este proceso puede ser incómodo para las organizaciones con decenas de miles de sistemas, a los que se debe dar mantenimiento en una diversa selección de equipo de cómputo. En el **Software como un servicio (SaaS)**, el software se ejecuta en servidores ubicados en cualquier parte de Internet. Cuando se actualizan esos servidores, los clientes en todo el mundo ven las nuevas capacidades; no se requiere una instalación local. Podemos acceder al servicio a través de un navegador. Los navegadores son bastante portables, por lo que podemos ver las mismas aplicaciones en una amplia variedad de computadoras desde cualquier parte del mundo. Salesforce.com, Google, Microsoft Office Live y Windows Live ofrecen SaaS. Esta tecnología es una herramienta de la computación en nube.
Plataforma como un servicio (SaaS)	La **Plataforma como un servicio (PaaS)**, otra herramienta de la computación en nube, provee una plataforma de cómputo para desarrollar y ejecutar aplicaciones como un servicio a través de Web, en vez de instalar las herramientas en su computadora. Los proveedores de PaaS más importantes son: Google App Engine, Amazon EC2, Bungee Labs, entre otros.
Kit de desarrollo de software (SDK)	Los **Kits de desarrollo de software (SDK)** incluyen tanto las herramientas como la documentación que utilizan los desarrolladores para programar aplicaciones.

Fig. 1.27 | Tecnologías de software (parte 2 de 2).

La figura 1.28 describe las categorías de liberación de versiones de los productos de software.

Versión	Descripción
Alpha	El software *alfa* es la primera versión de un producto de software cuyo desarrollo aún se encuentra activo. Por lo general las versiones alfa tienen muchos errores, son incompletas e inestables; además se liberan a un pequeño número de desarrolladores para que evalúen las nuevas características, para obtener retroalimentación lo más pronto posible, etc.

Fig. 1.28 | Terminología de liberación de versiones de productos de software (parte 1 de 2).

Versión	Descripción
Beta	Las versiones *beta* se liberan a un número mayor de desarrolladores en una etapa posterior del proceso de desarrollo, una vez que se ha corregido la mayoría de los errores importantes y las nuevas características están casi completas. El software beta es más estable, pero todavía puede sufrir cambios.
Candidatos para liberación (Release Candidates)	En general, los *candidatos para liberación* tienen todas sus *características completas*, están (supuestamente) libres de errores y listos para que la comunidad los utilice, con lo cual se logra un entorno de prueba diverso —el software se utiliza en distintos sistemas, con restricciones variables y para muchos fines diferentes. Cualquier error que aparezca se corrige y, en un momento dado, el producto final se libera al público en general. A menudo, las compañías de software distribuyen actualizaciones incrementales a través de Internet.
Beta permanente	El software que se desarrolla mediante este método por lo general no tiene números de versión (por ejemplo, la búsqueda de Google o Gmail). Este software, que se aloja en la nube (no se instala en su computadora), evoluciona de manera constante de modo que los usuarios siempre dispongan de la versión más reciente.

Fig. 1.28 | Terminología de liberación de versiones de productos de software (parte 2 de 2).

1.14 C++11 y las bibliotecas Boost de código fuente abierto

C++11 (anteriormente conocido como C++0x) es el estándar más reciente del lenguaje de programación C++. Las organizaciones ISO/IEC lo publicaron en 2011. Bjarne Stroustrup, el creador de C++, expresó su visión para el futuro del lenguaje: las principales metas eran facilitar el aprendizaje de C++, mejorar las herramientas para generar bibliotecas e incrementar la compatibilidad con el lenguaje de programación C. El nuevo estándar extiende la Biblioteca estándar de C++ e incluye varias herramientas y mejoras para incrementar el rendimiento y la seguridad. Los principales distribuidores de compiladores de C++ ya implementaron muchas de las nuevas características de C++11 (figura 1.29). A lo largo del libro hablaremos sobre varias características clave de C++11. Para obtener más información, visite el sitio Web del Comité de Estándares de C++ en `www.open-std.org/jtc1/sc22/wg21/` e `isocpp.org`. Puede comprar copias de la especificación del lenguaje C++11 (ISO/IEC 14882:2011) en:

```
http://bit.ly/CPlusPlus11Standard
```

Compilador de C++	URL de las descripciones de las características de C++11
Características de C++11 implementadas en cada uno de los principales compiladores de C++.	`wiki.apache.org/stdcxx/C%2B%2B0xCompilerSupport`
Microsoft® Visual C++	`msdn.microsoft.com/en-us/library/hh567368.aspx`
Colección de compiladores de GNU (g++)	`gcc.gnu.org/projects/cxx0x.html`

Fig. 1.29 | Compiladores de C++ que implementaron las principales porciones de C++11 (parte 1 de 2).

Compilador de C++	URL de las descripciones de las características de C++11
Compilador Intel® C++	`software.intel.com/en-us/articles/c0x-features-supported-by-intel-c-compiler/`
IBM® XL C/C++	`www.ibm.com/developerworks/mydeveloperworks/blogs/5894415f-be62-4bc0-81c5-3956e82276f3/entry/xlc_compiler_s_c_11_support50?lang=en`
Clang	`clang.llvm.org/cxx_status.html`
EDG ecpp	`www.edg.com/docs/edg_cpp.pdf`

Fig. 1.29 | Compiladores de C++ que implementaron las principales porciones de C++11 (parte 2 de 2).

Bibliotecas Boost de C++

Las **Bibliotecas Boost de C++** son bibliotecas gratuitas de código fuente abierto, creadas por miembros de la comunidad de C++. Son revisadas por expertos y portables entre muchos compiladores y plataformas. El tamaño de Boost ha crecido hasta más de 100 bibliotecas, y se agregan más con regularidad. Hoy en día hay miles de programadores en la comunidad de código fuente de Boost. Boost ofrece a los programadores de C++ bibliotecas útiles que funcionan bien con la Biblioteca estándar de C++ existente. Las bibliotecas Boost las pueden utilizar los programadores de C++ que trabajan en una amplia variedad de plataformas, con muchos compiladores distintos. Algunas de las nuevas características de la Biblioteca estándar de C++11 se derivaron de las bibliotecas de Boost correspondientes. Nosotros veremos las generalidades sobre las bibliotecas y proporcionaremos ejemplos de código para las bibliotecas de "expresiones regulares" y "apuntadores inteligentes", entre otras.

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

Muchos errores comunes en el código de C y C++ están relacionados con los apuntadores, una poderosa herramienta de programación que C++ absorbió de C. Como veremos, los **apuntadores inteligentes** nos ayudan a evitar errores asociados con los apuntadores tradicionales.

1.15 Mantenerse actualizado con las tecnologías de la información

La figura 1.30 muestra una lista de las publicaciones técnicas y comerciales clave que le ayudarán a permanecer actualizado con la tecnología, las noticias y las tendencias más recientes. También encontrará una lista cada vez más grande de Centros de recursos relacionados con Internet y Web en `www.deitel.com/resourcecenters.html`.

Publicación	URL
ACM TechNews	`technews.acm.org/`
ACM Transactions on Accessible Computing	`www.gccis.rit.edu/taccess/index.html`
ACM Transactions on Internet Technology	`toit.acm.org/`
Bloomberg BusinessWeek	`www.businessweek.com`

Fig. 1.30 | Publicaciones técnicas y comerciales (parte 1 de 2).

Publicación	URL
CNET	`news.cnet.com`
Communications of the ACM	`cacm.acm.org/`
Computerworld	`www.computerworld.com`
Engadget	`www.engadget.com`
eWeek	`www.eweek.com`
Fast Company	`www.fastcompany.com/`
Fortune	`money.cnn.com/magazines/fortune/`
IEEE Computer	`www.computer.org/portal/web/computer`
IEEE Internet Computing	`www.computer.org/portal/web/internet/home`
InfoWorld	`www.infoworld.com`
Mashable	`mashable.com`
PCWorld	`www.pcworld.com`
SD Times	`www.sdtimes.com`
Slashdot	`slashdot.org/`
Smarter Technology	`www.smartertechnology.com`
Technology Review	`technologyreview.com`
Techcrunch	`techcrunch.com`
Wired	`www.wired.com`

Fig. 1.30 | Publicaciones técnicas y comerciales (parte 2 de 2).

1.16 Recursos Web

Esta sección proporciona vínculos a nuestros Centros de recursos sobre C++ y temas relacionados que le serán de utilidad a medida que aprenda C++. Los sitios blogs, artículos, documentos técnicos, compiladores, herramientas de desarrollo, descargas, preguntas frecuentes (FAQ), tutoriales, webcasts, wikis y vínculos a recursos de programación de juegos en C++. Para actualizaciones sobre publicaciones Deitel, Centros de recursos, cursos de capacitación, ofertas para socios y demás, síganos por Facebook® en `www.facebook.com/deitelfan/`, Twitter® `@deitel`, Google+ en `gplus.to/deitel` y LinkedIn en `bit.ly/DeitelLinkedIn`.

Sitios Web de Deitel & Associates

`www.deitel.com/books/cpphtp9/`
El sitio Web de *Cómo programar en C++, 9/e* de Deitel & Associates. Aquí encontrará vínculos a los ejemplos del libro y otros recursos.

`www.deitel.com/cplusplus/`
`www.deitel.com/visualcplusplus/`
`www.deitel.com/codesearchengines/`
`www.deitel.com/programmingprojects/`
Revise estos Centros de Recursos en busca de compiladores, descargas de código, tutoriales, documentación, libros, libros electrónicos, artículos, blogs, canales (feeds) RSS y demás, que le ayudarán a desarrollar aplicaciones de C++.

`www.deitel.com`
Revise este sitio en busca de actualizaciones, correcciones y recursos adicionales para todas las publicaciones Deitel.

`www.deitel.com/newsletter/subscribe.html`
Suscríbase al boletín de correo electrónico gratuito *Deitel® Buzz Online*, para seguir el programa de publicaciones de Deitel & Associates, incluyendo actualizaciones y fe de erratas para este libro.

Ejercicios de autoevaluación

1.1 Complete las siguientes oraciones:

a) Las computadoras procesan los datos bajo el control de conjuntos de instrucciones llamadas _____.

b) Las unidades lógicas clave de la computadora son _____, _____, _____, _____, _____ y _____.

c) Los tres tipos de lenguajes descritos en este capítulo son _____, _____ y _____.

d) Los programas que traducen programas de lenguaje de alto nivel a lenguaje máquina se denominan _____ _____.

e) _____ es un sistema operativo para dispositivos móviles, basado en el kernel de Linux y en Java.

f) Por lo general, el software _____ tiene todas sus características completas y (supuestamente) está libre de errores, listo para que la comunidad lo utilice.

g) El Wii Remote, así como muchos smartphones, usa un(a) _____ que permite al dispositivo responder al movimiento.

1.2 Complete cada una de las siguientes oraciones relacionadas con el entorno de C++:

a) Por lo general, los programas de C++ se escriben en una computadora mediante el uso de un programa _____.

b) En un sistema de C++, un programa _____ se ejecuta antes de que empiece la fase de traducción del compilador.

c) El programa _____ combina la salida del compilador con varias funciones de biblioteca para producir una imagen ejecutable.

d) El programa _____ transfiere el programa ejecutable del disco a la memoria.

1.3 Complete cada una de las siguientes oraciones (basándose en la sección 1.8):

a) Los objetos tienen una propiedad que se conoce como _____; aunque éstos pueden saber cómo comunicarse con los demás objetos a través de interfaces bien definidas, generalmente no se les permite saber cómo están implementados los otros objetos.

b) Los programadores de C++ se concentran en crear _____, que contienen miembros de datos y las funciones miembro que manipulan a esos miembros de datos y proporcionan servicios a los clientes.

c) Al proceso de analizar y diseñar un sistema desde un punto de vista orientado a objetos se le conoce como _____.

d) Con la _____, se derivan nuevas clases de objetos absorbiendo las características de las clases existentes, y después se agregan sus propias características.

e) _____ es un lenguaje gráfico, que permite a las personas que diseñan sistemas de software utilizar una notación estándar en la industria para representarlos.

f) El tamaño, forma, color y peso de un objeto se consideran _____ de la clase de ese objeto.

Respuestas a los ejercicios de autoevaluación

1.1 a) programas. b) unidad de entrada, unidad de salida, unidad de memoria, unidad central de procesamiento, unidad aritmética y lógica, unidad de almacenamiento secundario. c) lenguajes máquina, lenguajes ensambladores y lenguajes de alto nivel. d) compiladores. e) Android. f) Candidato para liberación. g) acelerómetro.

1.2 a) editor. b) preprocesador. c) enlazador. d) cargador.

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

Ejercicios

1.4 Complete cada una de las siguientes oraciones:

a) La unidad lógica de la computadora que recibe información desde el exterior de la computadora para que ésta la utilice es la _____.

b) El proceso de indicar a la computadora cómo resolver un problema se llama _____.

c) _____ es un tipo de lenguaje computacional que utiliza abreviaturas del inglés para las instrucciones de lenguaje máquina.

d) _____ es una unidad lógica de la computadora envía información, que ya ha sido procesada por la computadora, a varios dispositivos, de manera que la información pueda utilizarse fuera de la computadora.
e) _____ y _____ son unidades lógicas de la computadora que retienen información.
f) _____ es una unidad lógica de la computadora que realiza cálculos.
g) _____ es una unidad lógica de la computadora que toma decisiones lógicas.
h) Los lenguajes _____ son más convenientes para que el programador pueda escribir programas rápida y fácilmente.
i) Al único lenguaje que una computadora puede entender directamente se le conoce como el _____ _____ de esa computadora.
j) _____ es una unidad lógica de la computadora que coordina las actividades de todas las demás unidades lógicas.

1.5 Complete los siguientes enunciados:
a) _____ en un principio se hizo popular como el lenguaje de desarrollo del sistema operativo Unix.
b) El lenguaje de programación _____ fue desarrollado por Bjarne Stroustrup a principios de la década de 1980 en los laboratorios Bell.

1.6 Complete los siguientes enunciados:
a) Por lo general los programas de C++ pasan por seis fases: _____, _____, _____, _____, _____ y _____.
b) Un(a) _____ proporciona muchas herramientas para apoyar el proceso de desarrollo de software, como editores para escribir y editar programas, depuradores para localizar errores lógicos en los programas y muchas otras características.

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

Hacer la diferencia

Hemos incluido en este libro ejercicios Hacer la diferencia, en los que le pediremos que trabaje con problemas que son de verdad importantes para individuos, comunidades, países y el mundo. Para obtener más información sobre las organizaciones a nivel mundial que trabajan para hacer la diferencia, y para obtener ideas sobre proyectos de programación relacionados, visite nuestro Centro de recursos para hacer la diferencia en `www.deitel.com/makingadifference`.

1.8 ***(Prueba práctica: calculadora de impacto ambiental del carbono)*** Algunos científicos creen que las emisiones de carbono, sobre todo las que se producen al quemar combustibles fósiles, contribuyen de manera considerable al calentamiento global y que esto se puede combatir si las personas tomamos conciencia y limitamos el uso de los combustibles basados en carbono. Varias organizaciones e individuos se preocupan cada vez más por el "impacto ambiental debido al carbono". Los sitios Web como Terra Pass

```
www.terrapass.com/carbon-footprint-calculator/
```

y Carbon Footprint

```
www.carbonfootprint.com/calculator.aspx
```

ofrecen calculadoras de impacto ambiental del carbono. Pruebe estas calculadoras para determinar el impacto que provoca usted en el ambiente debido al carbono. Los ejercicios en capítulos posteriores le pedirán que programe su propia calculadora de impacto ambiental del carbono. Como preparación, le sugerimos investigar las fórmulas para calcular el impacto ambiental del carbono.

1.9 ***(Prueba práctica: calculadora del índice de masa corporal)*** Según las estimaciones recientes, dos terceras partes de las personas que viven en Estados Unidos padecen de sobrepeso; la mitad de estas personas son obesas. Esto provoca aumentos considerables en el número de personas con enfermedades como la diabetes y las cardiopatías. Para determinar si una persona tiene sobrepeso o padece de obesidad, puede usar una medida conocida como índice de

masa corporal (IMC). El Departamento de salud y servicios humanos de Estados Unidos proporciona una calculadora del IMC en `www.nhlbisupport.com/bmi/`. Úsela para calcular su propio IMC. Un ejercicio del capítulo 2 le pedirá que programe su propia calculadora del IMC. Como preparación, le sugerimos investigar las fórmulas para calcular el IMC.

1.10 ***(Atributos de los vehículos híbridos)*** En este capítulo aprendió sobre los fundamentos de las clases. Ahora empezará a describir con detalle los aspectos de una clase conocida como "Vehículo híbrido". Los vehículos híbridos se están volviendo cada vez más populares, puesto que por lo general pueden ofrecer mucho más kilometraje que los vehículos operados sólo por gasolina. Navegue en Web y estudie las características de cuatro o cinco de los automóviles híbridos populares en la actualidad; después haga una lista de todos los atributos relacionados con sus características de híbridos que pueda encontrar. Por ejemplo, algunos de los atributos comunes son los kilómetros por litro en ciudad y los kilómetros por litro en carretera. También puede hacer una lista de los atributos de las baterías (tipo, peso, etc.).

1.11 ***(Neutralidad de género)*** Muchas personas desean eliminar el sexismo de todas las formas de comunicación. Usted ha recibido la tarea de crear un programa que pueda procesar un párrafo de texto y reemplazar palabras que tengan un género específico con palabras neutrales en cuanto al género. Suponiendo que recibió una lista de palabras con género específico y sus reemplazos con neutralidad de género (por ejemplo, reemplace "esposa" por "cónyuge", "hombre" por "persona", "hija" por "descendiente", y así en lo sucesivo), explique el procedimiento que utilizaría para leer un párrafo de texto y realizar estos reemplazos en forma manual. ¿Cómo podría su procedimiento generar un término extraño tal como "woperchild", que aparece listado en el Diccionario urbano (`www.urbandictionary.com`)? En el capítulo 4 aprenderá que un término más formal para "procedimiento" es "algoritmo", y que un algoritmo especifica los pasos a realizar, además del orden en el que se deben llevar a cabo.

1.12 ***(Privacidad)*** Algunos servicios de correo electrónico en línea guardan toda la correspondencia electrónica durante cierto periodo de tiempo. Suponga que un empleado disgustado de uno de estos servicios de correo electrónico en línea publicara en Internet todas las correspondencias de correo electrónico de millones de personas, incluyendo la suya. Analice las consecuencias.

1.13 ***(Responsabilidad ética y legal del programador)*** Como programador en la industria, tal vez llegue a desarrollar software que podría afectar la salud de otras personas, o incluso sus vidas. Suponga que un error de software en uno de sus programas provocara que un paciente de cáncer recibiera una dosis excesiva durante la terapia de radiación y resultara gravemente lesionada o muriera. Analice las consecuencias.

1.14 ***(El "Flash Crash" de 2010)*** Un ejemplo de las consecuencias de nuestra excesiva dependencia con respecto a las computadoras es el denominado "flash crash", que ocurrió el 6 de mayo de 2010, cuando el mercado de valores de Estados Unidos se derrumbó de manera precipitada en cuestión de minutos, al borrarse billones de dólares de inversiones que se volvieron a recuperar pocos minutos después. Use Internet para investigar las causas de este derrumbe y analice las consecuencias que genera.

Recursos para hacer la diferencia

La *Copa Imagine de Microsoft* es una competencia global en la que los estudiantes usan la tecnología para intentar resolver algunos de los problemas más difíciles del mundo, como la sostenibilidad ambiental, acabar con la hambruna, la respuesta a emergencias, la alfabetización, combatir el HIV/SIDA y otros más. Visite `www.imaginecup.com/about` para obtener más información sobre la competencia y para aprender sobre los proyectos desarrollados por los anteriores ganadores. También encontrará varias ideas de proyectos enviadas por organizaciones de caridad a nivel mundial.

Si desea obtener ideas adicionales para proyectos de programación que puedan hacer la diferencia, busque en Web el tema "hacer la diferencia" y visite

```
www.un.org/millenniumgoals
```

El proyecto Milenio de Naciones Unidas busca soluciones para los principales problemas mundiales, como la sostenibilidad ambiental, la igualdad de sexos, la salud infantil y materna, la educación universal y otros más.

`www.ibm.com/smarterplanet/`
El sitio Web Smarter Planet de IBM® habla sobre cómo es que IBM utiliza la tecnología para resolver problemas relacionados con los negocios, la computación en nube, la educación, la sostenibilidad y otros más.

`www.gatesfoundation.org/Pages/home.aspx`
La Fundación Bill y Melinda Gates ofrece becas a las organizaciones que trabajan para mitigar el hambre, la pobreza y las enfermedades en los países en desarrollo. En Estados Unidos, la fundación se enfoca en mejorar la educación pública, en especial para las personas con bajos recursos.

`www.nethope.org/`
NetHope es una colaboración de organizaciones humanitarias en todo el mundo, que trabajan para resolver los problemas relacionados con la tecnología, como la conectividad y la respuesta a las emergencias, entre otros.

`www.rainforestfoundation.org/home`
La Fundación Rainforest trabaja para preservar los bosques tropicales y proteger los derechos de los indígenas que consideran a estos bosques como su hogar. El sitio incluye una lista de cosas que usted puede hacer para ayudar.

`www.undp.org/`
El Programa de las Naciones Unidas para el Desarrollo (UNDP) busca soluciones a los desafíos globales, como la prevención y recuperación de crisis, la energía y el ambiente, la gobernanza democrática y otros más.

`www.unido.org`
La Organización de las Naciones Unidas para el Desarrollo Industrial (UNIDO) busca reducir la pobreza, dar a los países en desarrollo la oportunidad de participar en el comercio global y promover tanto la eficiencia de la energía como la sostenibilidad.

`www.usaid.gov/`
USAID promueve la democracia global, la salud, el crecimiento económico, la prevención de conflictos y la ayuda humanitaria, entre otras cosas.

`www.toyota.com/ideas-for-good/`
El sitio Web Ideas for Good de Toyota describe varias tecnologías de esta empresa que están haciendo la diferencia; entre éstas se incluyen su Sistema avanzado de asistencia de estacionamiento (Advanced Parking Guidance System), la tecnología Hybrid Synergy Drive®, el Sistema de ventilación operado por energía solar (Solar Powered Ventilation System), el modelo T.H.U.M.S. (Modelo humano total para la seguridad) y Touch Tracer Display. Usted puede participar en el desafío de Ideas for Good; envíe un breve ensayo o un video que describa cómo se pueden usar estas tecnologías para otros buenos propósitos.

2

Introducción a la programación en C++, entrada/salida y operadores

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

—William Shakespeare

Los pensamientos elevados deben tener un lenguaje elevado.

—Aristófanes

Una persona puede hacer una diferencia y todas las personas deberían intentarlo.

—John F. Kennedy

Objetivos

En este capítulo aprenderá a:

- Escribir programas de cómputo simples en C++.
- Escribir instrucciones simples de entrada y salida.
- Utilizar los tipos fundamentales.
- Familiarizarse con los conceptos básicos de la memoria de la computadora.
- Utilizar los operadores aritméticos.
- Reconocer precedencia de los operadores aritméticos.
- Escribir instrucciones simples para tomar decisiones.

Plan general

2.1 Introducción

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

Compilar y ejecutar programas

En `www.deitel.com/books/cpphtp9`, publicamos videos que demuestran cómo compilar y ejecutar programas en Microsoft Visual C++, GNU C++ y Xcode.

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

Considere un programa simple que imprime una línea de texto (figura 2.1). Este programa ilustra varias características importantes del lenguaje C++. El texto en las líneas 1 a 11 es el *código fuente* (o *código*) del programa. Los números de línea no son parte del código fuente.

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

```
Bienvenido a C++!
```

Fig. 2.1 | Programa para imprimir texto.

Comentarios

Las líneas 1y 2

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

comienzan con **//**, lo cual indica que el resto de cada línea es un **comentario**. Insertamos comentarios para *documentar* nuestros programas y ayudar a que otras personas puedan leerlos y comprenderlos. Los comentarios no hacen que la computadora realice ninguna acción cuando se ejecuta el programa; el compilador de C++ los *ignora* y *no* genera ningún código objeto en lenguaje máquina. El comentario `Programa para imprimir texto` describe el propósito del programa. A un comentario que empieza con `//` se le llama **comentario de una sola línea**, ya que termina al final de la línea actual. [*Nota:* también puede usar comentarios que contienen una o más líneas encerradas entre `/*` y `*/`].

Buena práctica de programación 2.1

Todo programa debe comenzar con un comentario que describa su propósito.

La directiva de preprocesamiento `#include`

La línea 3

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

es una **directiva del preprocesador**, la cual es un mensaje para el preprocesador de C++ (que presentamos en la sección 1.9). Las líneas que empiezan con **#** son procesadas por el preprocesador *antes* de que se compile el programa. Esta línea indica al preprocesador que debe incluir en el programa el contenido del **encabezado de flujos de entrada/salida `<iostream>`**. Este encabezado es un archivo que contiene información que el compilador usa al compilar cualquier programa que muestre datos en la pantalla o que reciba datos del teclado, mediante el uso de la entrada/salida de flujos al estilo C++. El programa de la figura 2.1 muestra datos en la pantalla, como pronto veremos. En el capítulo 6 hablaremos con más detalle sobre encabezados, y en el capítulo 13 explicaremos el contenido de `<iostream>`.

Error común de programación 2.1

Olvidar incluir el archivo de encabezado `<iostream>` en un programa que reciba datos del teclado, o que envíe datos a la pantalla, hace que el compilador genere un mensaje de error.

Líneas en blanco y espacio en blanco

La línea 4 es simplemente una *línea en blanco*. Los programadores usan líneas en blanco, *caracteres de espacio* y *caracteres de tabulación* (es decir, "tabuladores") para facilitar la lectura de los programas. En conjunto, estos caracteres se conocen como **espacio en blanco**. Por lo general, el compilador *ignora* los caracteres de espacio en blanco.

La función `main`

La línea 5

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

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

La línea 6

```
int main()
```

forma parte de todo programa en C++. Los paréntesis después de **`main`** indican que éste es un bloque de construcción denominado **function**. Los programas en C++ comúnmente consisten en una o más funciones y clases (como aprenderá en el capítulo 3). Exactamente *una* función en cada programa *debe* ser `main`. La figura 2.1 contiene sólo una función. Los programas en C++ empiezan a ejecutarse en la función `main`, aún si `main` *no* es la primera función definida en el programa. La palabra clave `int` a la izquierda de `main` indica que "devuelve" un valor entero. Una **palabra clave** es una palabra en código reservada para C++, para un uso específico. En la figura 4.3 encontrará la lista completa de palabras clave de C++. Explicaremos lo que significa que una función "devuelva un valor" cuando le demostremos cómo crear sus propias funciones en la sección 3.3. Por ahora, simplemente incluya la palabra clave `int` a la izquierda de `main` en todos sus programas.

La **llave izquierda**, **`{`**, (línea 7) debe *comenzar* el **cuerpo** de toda función. Su correspondiente **llave derecha**, **`}`**, (línea 11) debe *terminar* el cuerpo de cada función.

Una instrucción de salida

La línea 8

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

indica a la computadora que debe **realizar una acción**; a saber, imprimir los caracteres contenidos entre las comillas dobles. En conjunto, a las comillas y los caracteres entre ellas se les conoce como **cadena (string)**, **cadena de caracteres** o **literal de cadena**. En este libro nos referimos a los caracteres entre comillas dobles simplemente como cadenas. El compilador no ignora los caracteres de espacio en blanco en las cadenas.

A la línea 8 completa, incluyendo `std::cout`, el **operador `<<`**, la cadena "`Bienvenido a C++!\n`" y el **punto y coma** (**`;`**), se le conoce como **instrucción**. La mayoría de las instrucciones en C++ terminan con un punto y coma, lo que también se conoce como **terminador de instrucciones** (pronto veremos algunas excepciones a esto). Las directivas del preprocesador (como `#include`) no terminan con un punto y coma. Por lo general, en C++ las operaciones de entrada y salida se realizan mediante **flujos** de caracteres. Por ende, cuando se ejecuta la instrucción anterior, envía el flujo de caracteres `Bienvenido a C++!\n` al **objeto flujo estándar de salida** (**`std::cout`**) el cual por lo general está "conectado" a la pantalla.

Error común de programación 2.2

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

Buena práctica de programación 2.2

Aplique sangría al cuerpo de cada función un nivel dentro de las llaves que delimitan el cuerpo de la función. Esto hace que la estructura funcional de un programa resalte y hace que sea más fácil de leer.

Buena práctica de programación 2.3

Establezca una convención para el tamaño de la sangría que prefiera y luego aplíquela de manera uniforme. La tecla de tabulación puede usarse para crear sangrías, pero los topes de tabulación pueden variar. Nosotros preferimos tres espacios por cada nivel de sangría.

El espacio de nombres `std`

La instrucción `std::` antes de `cout` se requiere cuando utilizamos nombres que hemos traído al programa por la directiva del preprocesador `#include <iostream>`. La notación `std::cout` especifica que estamos usando un nombre (en este caso `cout`) que pertenece al espacio de nombres `std`. Los nombres `cin` (el flujo de entrada estándar) y `cerr` (el flujo de error estándar), que presentamos en el capítulo 1, también pertenecen al espacio de nombres `std`. (Los espacios de nombres son una característica avanzada de C++ que puede consultar en el capítulo 23, Other Topics, que se encuentra en inglés en el sitio web de este libro). Por ahora, simplemente debe recordar incluir `std::` antes de cada mención de `cout`, `cin` y `cerr` en un programa. Esto puede ser incómodo; en el siguiente ejemplo introduciremos las declaraciones `using` y la directiva `using`, la cual nos permitirá omitir `std::` antes de cada uso de un nombre en el espacio de nombres `std`.

El operador de inserción de flujo y las secuencias de escape

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

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

Fig. 2.2 | Secuencias de escape.

La instrucción `return`

La línea 10

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

es uno de varios medios que utilizaremos para **salir de una función**. Cuando se utiliza la **instrucción `return`** al final de `main`, como se muestra aquí, el valor `0` indica que el programa ha *terminado correctamente*. La llave derecha, }, (línea 11) indica el fin de la función `main`. De acuerdo con el estándar de C++,

si la ejecución del programa llega al final de `main` sin encontrar una instrucción `return` se asume que el programa terminó con éxito: exactamente como cuando la última instrucción en `main` es una instrucción `return` con el valor `0`. Por esta razón, *omitiremos* la instrucción `return` al final de `main` en el resto de los programas.

Una observación sobre los comentarios
Al escribir un nuevo programa o modificar uno existente, debe *mantener actualizados sus comentarios* con el código del programa. A *menudo* será necesario realizar cambios en los programas existentes; por ejemplo, para corregir errores (comúnmente conocidos como *bugs*) que evitan que un programa funcione de manera correcta, o para mejorar un programa. Al actualizar sus comentarios a medida que realiza cambios en el código, ayuda a asegurar que los comentarios reflejen con precisión lo que hace el código. Esto hará que sus programas sean más fáciles de comprender y modificar en el futuro.

2.3 Modificación de nuestro primer programa en C++

Ahora le presentaremos dos ejemplos que modifican el programa de la figura 2.1 para imprimir texto en una línea utilizando varias instrucciones, y para imprimir texto en varias líneas utilizando una sola instrucción.

Cómo mostrar una sola línea de texto con varias instrucciones
`Bienvenido a la programacion en C++!` puede mostrarse en varias formas. Por ejemplo, la figura 2.3 realiza la inserción de flujos en varias instrucciones (líneas 8 y 9), y produce el mismo resultado que el programa de la figura 2.1. [*Nota:* de aquí en adelante, utilizaremos un *fondo color gris claro* para resaltar las características clave que se introduzcan en cada programa]. Cada inserción de flujo reanuda la impresión en donde se detuvo la anterior. La primera inserción de flujo (línea 8) imprime la palabra `Bienvenido` seguida de un espacio, y puesto que esta cadena no terminó con `\n`, la segunda inserción de flujo (línea 9) empieza a imprimir en la *misma* línea, justo después del espacio.

```
1  // Fig. 2.3: fig02_03.cpp
2  // Imprimir una línea de texto con varias instrucciones.
3  #include <iostream> // permite que el programa envíe datos a la pantalla
4
5  // la función main comienza la ejecución del programa
6  int main()
7  {
8     std::cout << "Bienvenido ";
9     std::cout << "a C++!\n";
10 } // fin de la función main
```

```
Bienvenido a C++!
```

Fig. 2.3 | Impresión de una línea de texto con varias instrucciones.

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

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

```
Bienvenido
a

C++!
```

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

2.4 Otro programa en C++: suma de enteros

Nuestro siguiente programa obtiene dos enteros escritos por el usuario mediante el teclado, calcula la suma de esos valores e imprime el resultado mediante el uso de `std::cout`. En la figura 2.5 se muestra el programa, junto con los datos de entrada y salida de ejemplo. En la ejecución de ejemplo resaltamos la entrada del usuario en negritas. El programa empieza a ejecutarse con la función `main` (línea 6). La llave izquierda (línea 7) marca el inicio del cuerpo de `main` y la correspondiente llave derecha (línea 22) marca el fin.

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

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

Fig. 2.5 | Programa que muestra la suma de dos enteros.

Declaraciones de variables
Las líneas 9 a 11

```
int numero1 = 0; // primer entero a sumar (se inicializa con 0)
int numero2 = 0; // segundo entero a sumar (se inicializa con 0)
int suma = 0; // suma de numero1 y numero2 (se inicializa con 0)
```

son **declaraciones**. Los identificadores `numero1`, `numero2` y `suma` son nombres de **variables**. Una variable es una ubicación en la memoria de la computadora, en la que puede almacenarse un valor para usarlo mediante un programa. Estas declaraciones especifican que las variables `numero1`, `numero2` y `suma` son datos de tipo `int`, lo cual significa que estas variables contienen valores **enteros**; es decir, números enteros tales como 7, –11, 0 y 31914. Las declaraciones también inicializan cada una de estas variables con `0`.

Tip para prevenir errores 2.1
Aunque no siempre es necesario inicializar cada variable de manera explícita, hacerlo evitará muchos tipos de problemas.

Todas las variables se *deben* declarar con un *nombre* y un *tipo de datos* antes de poder usarlas en un programa. Pueden declararse diversas variables del mismo tipo en una declaración, o en varias declaraciones. Podríamos haber declarado las tres variables en una declaración mediante una **lista separada por comas**, como se muestra a continuación:

```
int numero1 = 0, numero2 = 0, suma = 0;
```

Esto reduce la legibilidad del programa y evita que podamos proporcionar comentarios que describan el propósito de cada variable.

Buena práctica de programación 2.4
Declare sólo una variable en cada declaración y proporcione un comentario que explique el propósito de la variable en el programa.

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

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

Tip de portabilidad 2.1
C++ permite identificadores de cualquier longitud, pero tal vez la implementación de C++ que usted utilice imponga algunas restricciones en cuanto a la longitud de los identificadores. Use identificadores de 31 caracteres o menos, para asegurar la portabilidad.

Buena práctica de programación 2.5

Seleccionar nombres significativos para los identificadores ayuda a que un programa se ***autodocumente****: que una persona pueda comprender el programa con sólo leerlo, en lugar de tener que consultar los comentarios o la documentación del programa.*

Buena práctica de programación 2.6

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

Buena práctica de programación 2.7

No use identificadores que empiecen con guiones bajos y dobles guiones bajos, ya que los compiladores de C++ pueden, de manera interna, utilizar nombres como esos para sus propios fines. Esto evitará que los nombres que usted elija se confundan con los nombres que elijan los compiladores.

Colocación de las declaraciones de variables

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

```
int numero1 = 0; // primer entero a sumar (se inicializa con 0)
```

se podría haber colocado justo antes de la línea 14

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

la declaración en la línea 10

```
int numero2 = 0; // segundo entero a sumar (se inicializa con 0)
```

se podría haber colocado justo antes de la línea 17

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

y la declaración en la línea 11

```
int suma = 0; // suma de numero1 y numero2 (se inicializa con 0)
```

se podría haber colocado justo antes de la línea 19

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

Obtener el primer valor del usuario

La línea 13

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

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

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

utiliza el **objeto flujo de entrada estándar `cin`** (del espacio de nombres `std`) y el **operador de extracción de flujo, `>>`**, para obtener un valor del teclado. Al usar el operador de extracción de flujo con

`std::cin` se toma la entrada de caracteres del flujo de entrada estándar, que por lo general es el teclado. Nos gusta pronunciar la instrucción anterior como "`std::cin` *proporciona* un valor a `numero1`", o simplemente como "`std::cin` *proporciona* `numero1`".

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

Los objetos flujo `std::cout` y `std::cin` facilitan la interacción entre el usuario y la computadora.

Claro que los usuarios pueden introducir datos *inválidos* desde el teclado. Por ejemplo, cuando su programa espera que el usuario introduzca un entero, éste podría introducir caracteres alfabéticos, símbolos especiales (como # o @) o un número con un punto decimal (como 73.5), entre otros. En estos primeros programas asumimos que el usuario introduce datos *válidos*. A medida que progrese por el libro, aprenderá varias técnicas para lidiar con el amplio rango de posibles problemas de introducción de datos.

Obtener el segundo valor del usuario

La línea 16

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

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

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

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

Calcular la suma de los valores introducidos por el usuario

La instrucción de asignación en la línea 19

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

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

Buena práctica de programación 2.8

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

Mostrar el resultado

La línea 21

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

muestra a la cadena de caracteres `La suma es`, seguida del valor numérico de la variable `suma` seguido de `std::endl`, a éste último se le conoce como **manipulador de flujos**. El nombre `endl` es una abreviación

de "fin de línea" ("end of line", en inglés) y pertenece al espacio de nombres `std`. El manipulador de flujos `std::endl` imprime una nueva línea y después "vacía el búfer de salida". Esto simplemente significa que, en algunos sistemas en donde los datos de salida se acumulan en el equipo hasta que haya suficientes como para que "valga la pena" mostrarlos en pantalla, `std::endl` obliga a que todos los datos de salida acumulados se muestren en ese momento. Esto puede ser importante cuando los datos de salida piden al usuario una acción, como introducir datos.

La instrucción anterior imprime múltiples valores de distintos tipos. El operador de inserción de flujo "sabe" cómo imprimir cada tipo de datos. Al uso de varios operadores de inserción de flujo (<<) en una sola instrucción se le conoce como **operaciones de inserción de flujo en cascada**, **concatenamiento** o **encadenamiento**.

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

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

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

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

2.5 Conceptos acerca de la memoria

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

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

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

en la línea 14, el entero que escribe el usuario se coloca en una ubicación de memoria a la que el compilador haya asignado el nombre `numero1`. Suponga que el usuario introduce el número `45` como el valor para `numero1`. La computadora colocará el `45` en la ubicación `numero1`, como se muestra en la figura 2.6. Cuando se coloca un valor en una ubicación en memoria, ese valor *sobrescribe* al valor anterior en esa ubicación; por ende, se dice que la acción de colocar un nuevo valor en una ubicación en memoria es una operación **destructiva**.

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

Regresando a nuestro programa de suma, suponga que el usuario introduce el valor 72 cuando se ejecuta la instrucción

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

Este valor se coloca en la ubicación `numero2`, y la memoria aparece como se muestra en la figura 2.7. Las ubicaciones de las variables no necesariamente están adyacentes en la memoria.

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

```
suma = numero1 + numeror2; // suma los números; almacena el resultado en suma
```

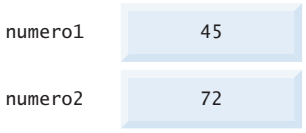


Fig. 2.7 | Ubicaciones de memoria, después de almacenar valores en las variables para `numero1` y `numero2`.

reemplaza el valor que estaba almacenado en `suma`. La suma calculada de `numero1` y `numero2` se coloca en la variable `suma` sin importar qué valor haya tenido `suma` anteriormente; ese valor se *pierde*. Después de calcular `suma`, la memoria aparece como se muestra en la figura 2.8. Los valores de `numero1` y `numero2` aparecen exactamente como estaban antes del cálculo. Se utilizaron estos valores pero *no* se destruyeron, en el momento en el que la computadora realizó el cálculo. Por ende, cuando se *lee* un valor de una ubicación de memoria, la operación es **no destructiva**.

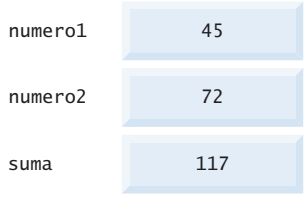


Fig. 2.8 | Ubicaciones de memoria, después de calcular y almacenar la `suma` `numero1` y `numero2`.

2.6 Aritmética

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

La **división de enteros** (es decir, en donde tanto el numerador como el denominador son enteros) produce un cociente entero; por ejemplo, la expresión `7 / 4` da como resultado `1` y la expresión `17 / 5` da

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

Fig. 2.9 | Operadores aritméticos.

como resultado 3. *Cualquier parte fraccionaria en una división de enteros se* ***trunca*** *(es decir, se descarta); no ocurre un redondeo.*

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

Expresiones aritméticas en formato de línea recta

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

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

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

Paréntesis para agrupar subexpresiones

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

Reglas de precedencia de operadores

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

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

   ```
   ( a * ( b + c ) )
   ```

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

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

3. Las operaciones de suma y resta se aplican al último. Si una expresión contiene varias de esas operaciones, los operadores se aplican de *izquierda a derecha*. Los operadores de suma y resta tienen el *mismo* nivel de precedencia.

Las reglas de precedencia de operadores definen el orden en el que C++ aplica los operadores. Cuando decimos que ciertos operadores se aplican de izquierda a derecha, nos referimos a su **asociatividad**. Por ejemplo, los operadores de suma (+) en la expresión

```
a + b + c
```

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

Operador(es)	Operación(es)	Orden de evaluación (precedencia)
`( )`	Paréntesis	Se evalúa primero. Si los paréntesis son *anidados*, como en la expresión `( a * ( b + c / d + e ) )`, la expresión en el par *más interno* se evalúa primero. [*Precaución:* si tiene una expresión como `(a + b) * (c - d)` en donde dos conjuntos de paréntesis no están anidados, pero aparecen "en el mismo nivel", en C++ estándar *no* no especifica el orden en el que se evaluarán estas subexpresiones entre paréntesis].

`*`
`/`
`%`
	Multiplicación
	División
	Módulo
		Se evalúan en segundo lugar. Si hay varios operadores de este tipo, se evalúan de izquierda a derecha.

`+` `-`	Suma Resta	Se evalúan al último. Si hay varios operadores de este tipo, se evalúan de izquierda a derecha.

Fig. 2.10 | Precedencia de los operadores aritméticos.

Ejemplos de expresiones algebraicas y de C++

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

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

C++:
```
m = ( a + b + c + d + e ) / 5;
```

Los paréntesis son obligatorios, ya que la división tiene una *mayor* precedencia que la suma. La cantidad *completa* `( a + b + c + d + e )` va a dividirse entre `5`. Si los paréntesis se omiten por error, obtenemos `a + b + c + d + e / 5`, que se evalúa incorrectamente como:

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

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

Álgebra: $y = mx + b$

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

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

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

Álgebra: $z = pr\%q + w/x - y$

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

6 1 2 4 3 5

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

Evaluación de un polinomio de segundo grado

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

```
y = a * x * x + b * x + c;
    6   1   2   4   3   5
```

Los números dentro de los círculos debajo de la instrucción indican el orden en el que C++ aplica los operadores. *En C++ no hay operador aritmético para la exponenciación*, por lo que hemos representado a x^2 como `x * x`. En el capítulo 5 hablaremos sobre la función `pow` ("power"o potencia) de la biblioteca estándar que realiza la exponenciación.

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

```
Paso 1.  y = 2 * 5 * 5 + 3 * 5 + 7;      (Multiplicación de más a la izquierda)
             2 * 5 es 10

Paso 2.  y = 10 * 5 + 3 * 5 + 7;         (Multiplicación de más a la izquierda)
             10 * 5 es 50

Paso 3.  y = 50 + 3 * 5 + 7;             (Multiplicación antes de la suma)
                 3 * 5 es 15

Paso 4.  y = 50 + 15 + 7;                (Suma de más a la izquierda)
             50 + 15 es 65

Paso 5.  y = 65 + 7;                     (Última suma)
             65 + 7 es 72

Paso 6.  y = 72                          (Última operación; colocar 72 en y)
```

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

Paréntesis redundantes

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

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

2.7 Toma de decisiones: operadores de igualdad y relacionales

Ahora presentaremos una versión simple de la **instrucción `if`** de C++, la cual permite que un programa tome una acción alternativa, con base en la verdad o falsedad de cierta **condición**. Si la condición es *verdadera*, se ejecuta la instrucción que está en el cuerpo de la instrucción `if`. Si la condición es *falsa*, el cuerpo *no se* ejecuta. En breve veremos un ejemplo.

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

Operador algebraico de igualdad o relacional	Operador de igualdad o relacional de C++	Ejemplo de condición en C++	Significado de la condición en C++
Operadores relacionales			
>	>	x > y	x es mayor que y
<	<	x < y	x es menor que y
≥	>=	x >= y	x es mayor o igual que y
≤	<=	x <= y	x es menor o igual que y
Operadores de igualdad			
=	==	x == y	x es igual a y
≠	!=	x != y	x no es igual a y

Fig. 2.12 | Operadores de igualdad y relacionales.

Error común de programación 2.3

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

Error común de programación 2.4

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

Uso de la instrucción `if`

El siguiente ejemplo (figura 2.13) utiliza seis instrucciones `if` para comparar dos números introducidos por el usuario. Si se satisface la condición en cualquiera de estas instrucciones `if`, se ejecuta la instrucción de salida asociada con esa instrucción `if`.

```
1  // Fig. 2.13: fig02_13.cpp
2  // Comparación de enteros mediante instrucciones if, operadores
3  // relacionales y operadores de igualdad.
4  #include <iostream> // permite al programa realizar operaciones de entrada y
                       salida
5
6  using std::cout; // el programa usa cout
7  using std::cin; // el programa usa cin
8  using std::endl; // el programa usa endl
9
10 // la función main empieza la ejecución del programa
11 int main()
12 {
13    int numero1 = 0; // primer entero a comparar (se inicializa con 0)
14    int numero2 = 0; // segundo entero a comparar (se inicializa con 0)
15
16    cout << "Escriba dos enteros a comparar: "; // pide los datos al usuario
17    cin >> numero1 >> numero2; // lee dos enteros del usuario
18
19    if ( numero1 == numero2 )
20       cout << numero1 << " == " << numero2 << endl;
21
22    if ( numero1 != numero2 )
23       cout << numero1 << " != " << numero2 << endl;
24
25    if ( numero1 < numero2 )
26       cout << numero1 << " < " << numero2 << endl;
27
28    if ( numero1 > numero2 )
29       cout << numero1 << " > " << numero2 << endl;
30
31    if ( numero1 <= numero2 )
32       cout << numero1 << " <= " << numero2 << endl;
33
34    if ( numero1 >= numero2 )
35       cout << numero1 << " >= " << numero2 << endl;
36 } // fin de la función main
```

```
Escriba dos enteros a comparar: 3 7
3 != 7
3 < 7
3 <= 7
```

```
Escriba dos enteros a comparar: 22 12
22 != 12
22 > 12
22 >= 12
```

Fig. 2.13 | Comparación de enteros mediante instrucciones `if`, operadores relacionales y operadores de igualdad (parte 1 de 2).

```
Escriba dos enteros a comparar: 7 7
7 == 7
7 <= 7
7 >= 7
```

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

Declaraciones `using`

Las líneas 6 a 8:

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

son **declaraciones `using`** que eliminan la necesidad de repetir el prefijo `std::` que usamos en programas anteriores. Ahora podemos escribir `cout` en vez de `std::cout`, `cin` en vez de `std::cin` y `endl` en vez de `std::endl`, respectivamente, en el resto del programa.

En lugar de las líneas 6 a 8, muchos programadores prefieren usar la **directiva `using`**:

```
using namespace std;
```

la cual permite que un programa utilice *todos* los nombres en cualquier encabezado estándar de C++ (como `<iostream>`) que un programa pudiera incluir. Desde este punto en adelante en el libro, usaremos la directiva anterior en nuestros programas.[1]

Declaraciones de variables y lectura de las entradas del usuario

Las líneas 13 y 14:

```
int numero1 = 0; // primer entero a comparar (se inicializa con 0)
int numero2 = 0; // segundo entero a comparar (se inicializa con 0)
```

declaran las variables utilizadas en el programa y las inicializan con `0`.

El programa usa operaciones de extracción de flujos en cascada (línea 17) para recibir dos enteros como entrada. Recuerde que podemos escribir `cin` (en vez de `std::cin`) debido a la línea 7. Primero se lee un valor y se coloca en la variable `number1`, luego se lee un valor y se coloca en la variable `number2`.

Comparación de números

La instrucción `if` en las líneas 19 y 20:

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

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

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

---

1 En el capítulo 23, Other Topics (en inglés, que se encuentra en el sitio web), hablaremos sobre algunas directivas `using` en sistemas de gran escala.

Buena práctica de programación 2.9

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

Error común de programación 2.5

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

Espacio en blanco

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

Buena práctica de programación 2.10

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

Precedencia de operadores

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

Operadores	Asociatividad	Tipo
`()`	*[vea la precaución en la figura 2.10]*	paréntesis para agrupar
`*` `/` `%`	izquierda a derecha	multiplicativa
`+` `-`	izquierda a derecha	suma
`<<` `>>`	izquierda a derecha	inserción/extracción de flujo
`<` `<=` `>` `>=`	izquierda a derecha	relacional
`==` `!=`	izquierda a derecha	igualdad
`=`	derecha a izquierda	asignación

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

Buena práctica de programación 2.11

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

2.8 Conclusión

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

Las aplicaciones no orientadas a objetos que presentamos en este capítulo lo introdujeron a los conceptos básicos de programación. Como verá en el capítulo 3, las aplicaciones de C++ por lo general contienen sólo unas cuantas líneas de código en la función `main`: estas instrucciones normalmente crean los objetos que realizan el trabajo de la aplicación y después los objetos "se hacen cargo a partir de aquí". En el capítulo 3 aprenderá a implementar sus propias clases y a usar objetos de esas clases en las aplicaciones.

Resumen

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

- Los comentarios de una sola línea (pág. 40) comienzan con //. Insertamos comentarios para documentar los programas y mejorar su legibilidad.
- Los comentarios no provocan que la computadora realice ninguna acción (pág. 41) cuando se ejecuta el programa: el compilador los ignora y no provocan que se genere ningún código objeto en lenguaje máquina.
- Una directiva de preprocesamiento (pág. 40) comienza con # y es un mensaje para el preprocesador de C++. Las directivas de preprocesamiento se procesan antes de que se compile el programa.
- La línea `#include <iostream>` (pág. 40) indica al preprocesador de C++ que debe incluir el contenido del encabezado del flujo de entrada/salida, que contiene la información necesaria para compilar programas que usen `std::cin` (pág. 46) y `std::cout` (pág. 41) junto con los operadores de inserción de flujo (<<, pág. 42) y de extracción de flujo (>>, pág. 46).
- El espacio en blanco (es decir líneas en blanco, caracteres de espacio y caracteres de tabulación, pág. 40) hace que los programas sean más fáciles de leer. El compilador ignora los caracteres de espacio en blanco que se encuentran fuera de las literales de cadena.
- Los programas en C++ empiezan a ejecutarse en `main` (pág. 41), incluso aunque `main` no aparezca primero en el programa.
- La palabra clave `int` a la izquierda de `main` indica que esta función "devuelve" un valor entero.

- El cuerpo (pág. 41) de toda función debe estar encerrado entre llaves ({ y }).
- Una cadena (string) (pág. 41) entre comillas dobles se conoce algunas veces como cadena de caracteres, mensaje o literal de cadena. El compilador *no* ignora los caracteres de espacio en blanco en las cadenas.
- La mayoría de las instrucciones de C++ (pág. 41) terminan con un punto y coma, también conocido como terminador de instrucción (pronto veremos algunas excepciones a esto).
- En C++, las operaciones de entrada y salida se realizan mediante flujos (pág. 41) de caracteres.
- El objeto flujo de salida `std::cout` (que por lo general está conectado a la pantalla) se utiliza para mostrar datos. Es posible mostrar múltiples elementos de datos mediante la concatenación de operadores de inserción de flujo (<<).
- El objeto flujo de entrada `std::cin` (que por lo general está conectado al teclado) se utiliza para introducir datos. Es posible introducir múltiples elementos de datos mediante la concatenación de operadores de extracción de flujo (>>).
- La notación `std::cout` especifica que utilizamos `cout` del "espacio de nombres" `std`.
- Cuando hay una barra diagonal (es decir, un carácter de escape) en una cadena de caracteres, el siguiente carácter se combina con la barra diagonal para formar una secuencia de escape (pág. 42).
- La secuencia de escape de nueva línea `\n` (pág. 42) mueve el cursor al principio de la siguiente línea en la pantalla.
- Un mensaje que indica al usuario que realice una acción específica se conoce como indicador (prompt) (pág. 46).
- La palabra `return` de C++ (pág. 42) es uno de varios medios para salir de una función

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

- Hay que declarar todas las variables (pág. 45) en un programa en C++ antes de poder usarlas.
- El nombre de una variable es cualquier identificador válido (pág. 45) que no sea una palabra clave. Un identificador es una serie de caracteres que consisten en letras, dígitos y guiones bajos (_). Los identificadores no pueden comenzar con un dígito. Su nombre puede ser de cualquier longitud, aunque tal vez algunos sistemas o implementaciones de C++ impongan restricciones en cuanto al largo.
- C++ es sensible al uso de mayúsculas y minúsculas (pág. 45).
- La mayoría de los cálculos se realizan en instrucciones de asignación (pág. 47).
- Una variable es una ubicación en memoria (pág. 48) en donde se puede almacenar un valor para que lo utilice un programa.
- Las variables de tipo `int` (pág. 45) contienen valores enteros; es decir, números integer como 7, –11, 0, 31914.

Sección 2.5 Conceptos acerca de la memoria

- Toda variable almacenada en la memoria de la computadora tiene un nombre, valor, tipo y tamaño.
- Cada vez que se coloca un nuevo valor en una ubicación de memoria, el proceso es destructivo (pág. 48); es decir, el nuevo valor reemplaza al valor anterior en esa ubicación. El valor anterior se pierde.
- Cuando se lee un valor de la memoria, el proceso es no destructivo (pág. 49); es decir, se lee una copia del valor y el original queda sin ser perturbado en la ubicación de memoria.
- El manipulador de flujos `std::endl` (pág. 47) imprime una nueva línea y luego "vacía el búfer de salida".

Sección 2.6 Aritmética

- C++ evalúa las expresiones aritméticas (pág. 49) en una secuencia precisa determinada por las reglas de precedencia de operadores (pág. 50) y la asociatividad (pág. 50).
- Pueden usarse paréntesis para agrupar expresiones.
- La división de enteros (pág. 49) produce un cociente entero. Cualquier parte fraccional en la división de enteros se trunca.
- El operador módulo `%` (pág. 50) produce el residuo después de la división de enteros.

Sección 2.7 Toma de decisiones: operadores de igualdad y relacionales

- La instrucción `if` (pág. 53) permite a un programa tomar una acción alternativa con base en el cumplimiento de una condición. El formato para una instrucción `if` es:

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

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

- Por lo general, las condiciones en las instrucciones `if` se forman mediante el uso de operadores de igualdad y relacionales (pág. 53). El resultado de usar esos operadores siempre es un valor verdadero o falso.
- La siguiente declaración `using` (pág. 55):

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

  informa al compilador en dónde encontrar `cout` (namespace `std`) y elimina la necesidad de repetir `std::` prefijo. La siguiente directiva `using` (pág. 55):

```
using namespace std;
```

  permite al programa usar en el encabezado todos los nombres incluidos de cualquier biblioteca estándar de C++.

Ejercicios de autoevaluación

2.1 Complete las siguientes oraciones:

a) Todo programa en C++ empieza su ejecución en la función _____.
b) Un(a) _____ empieza el cuerpo de toda función y un(a) _____ lo termina.
c) La mayoría de las instrucciones de C++ terminan con un(a) _____.
d) La secuencia de escape `\n` representa el carácter _____, el cual hace que el cursor se posicione al principio de la siguiente línea en la pantalla.
e) La instrucción _____ se utiliza para tomar decisiones.

2.2 Indique si cada una de las siguientes instrucciones es *verdadera* o *falsa*. Si es *falsa*, explique por qué. Asuma que se usa la instrucción `using std::cout`.

a) Los comentarios hacen que la computadora imprima, en la pantalla, el texto que va después de los caracteres / /, al ejecutarse el programa.
b) Cuando la secuencia de escape `\n`, se imprime con `cout` y el operador de inserción de flujo causa que el cursor se posicione al principio de la siguiente línea en la pantalla.
c) Todas las variables deben declararse antes de utilizarlas.
d) Todas las variables deben ser de un tipo al declararlas.
e) C++ considera que las variables `numero` y `NuMeRo` son idénticas.
f) Las declaraciones pueden aparecer casi en cualquier parte del cuerpo de una función de C++.
g) El operador módulo (`%`) se puede utilizar sólo con operandos enteros.
h) Los operadores aritméticos `*`, `/`, `%`, `+` y `–` tienen todos el mismo nivel de precedencia.
i) Un programa en C++ que imprime tres líneas de salida debe contener tres instrucciones en las que se utilicen `cout` y el operador de inserción de flujo.

2.3 Escriba una sola instrucción en C++ para realizar cada una de las siguientes tareas (suponga que no se han utilizado declaraciones `using` ni la directiva `using`):

a) Declarar las variables `c`, `estaEsUnaVariable`, `q76354` y `numero` como de tipo `int` (en una instrucción).
b) Pedir al usuario que introduzca un entero. Termine su mensaje con un signo de dos puntos (`:`) seguido de un espacio, y deje el cursor posicionado después del espacio.
c) Recibir un entero como entrada del usuario mediante el teclado y almacenarlo en la variable entera `edad`.
d) Si la variable `numero` no es igual a `7`, imprimir `"La variable numero no es igual a 7"`.
e) Imprimir en una línea el mensaje `"Este es un programa en C++"`.
f) Imprimir en dos líneas el mensaje `"Este es un programa en C++"`. Termine la primera línea con `C++`.

g) Imprimir el mensaje `"Este es un programa en C++"` con cada palabra en una línea separada.
h) Imprimir el mensaje `"Este es un programa en C++"`. Separe una palabra de otra mediante un tabulador.

2.4 Escriba una declaración (o comentario) para realizar cada una de las siguientes tareas (suponga que se han utilizado declaraciones `using` para `cin`, `cout` y `endl`):

a) Indicar que un programa calculará el producto de tres enteros.
b) Declarar las variables `x`, `y`, `z` y `resultado` de tipo `int` (en instrucciones separadas) e inicializar cada una con 0.
c) Pedir al usuario que escriba tres enteros.
d) Recibir tres enteros del teclado y almacenarlos en las variables `x`, `y` y `z`.
e) Calcular el producto de los tres enteros contenidos en las variables `x`, `y` y `z`, y asignar el resultado a la variable `resultado`.
f) Imprimir `"El producto es "` seguido del valor de la variable `resultado`.
g) Devolver un valor de `main`, para indicar que el programa terminó correctamente.

2.5 Utilizando las instrucciones que escribió en el ejercicio 2.4, escriba un programa completo que calcule e imprima el producto de tres enteros. Agregue comentarios al código donde sea apropiado. [*Nota:* necesitará escribir las declaraciones `using` o la directiva `using` que sean necesarias].

2.6 Identifique y corrija los errores en cada una de las siguientes instrucciones (suponga que se utiliza la instrucción `using std::cout;`):

a)
```
if ( c < 7 );
   cout << "c es menor que 7\n";
```
b)
```
if ( c => 7 )
   cout << "c es igual o mayor que 7\n";
```

Respuestas a los ejercicios de autoevaluación

2.1 a) `main`. b) llave izquierda (`{`), llave derecha (`}`). c) punto y coma. d) nueva línea. e) `if`.

2.2 a) Falso. Los comentarios no producen ninguna acción cuando el programa se ejecuta. Se utilizan para documentar programas y mejorar su legibilidad.
b) Verdadero.
c) Verdadero.
d) Verdadero.
e) Falso. C++ es sensible a mayúsculas y minúsculas, por lo que estas variables son diferentes.
f) Verdadero.
g) Verdadero.
h) Falso. Los operadores `*`, `/` y `%` se encuentran en el mismo nivel de precedencia, y los operadores `+` y `-` se encuentran en un nivel menor de precedencia.
i) Falso. Una instrucción con `cout` y varias secuencias de escape `\n` puede imprimir varias líneas.

2.3 a)
```
int c, estaEsUnaVariable, q76354, numero;
```
b)
```
std::cout << "Escriba un entero: ";
```
c)
```
std::cin >> edad;
```
d)
```
if ( numero != 7 )
   std::cout << "La variable numero no es igual a 7\n";
```
e)
```
std::cout << "Este es un programa en C++\n";
```
f)
```
std::cout << "Este es un\n programa en C++\n";
```
g)
```
std::cout << "Este\nes\nun\nprograma\nen\nC++\n";
```
h)
```
std::cout << "Este\tes\tun\tprograma\ten\tC++\n";
```

2.4 a) `// Calcula el producto de tres enteros`

b)
```
int x = 0;
int y = 0;
int z = 0;
int resultado = 0;
```

c) `cout << "Escriba tres enteros: ";`

d) `cin >> x >> y >> z;`

e) `resultado = x * y * z;`

f) `cout << "El producto es " << resultado << endl;`

g) `return 0;`

2.5 (Vea el siguiente programa).

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

2.6 a) *Error:* punto y coma después del paréntesis derecho de la condición en la instrucción `if`.
Corrección: elimine el punto y coma después del paréntesis derecho. [*Nota:* el resultado de este error es que la instrucción de salida se ejecuta sin importar que la condición en la instrucción `if` sea verdadera o no]. El punto y coma después del paréntesis derecho es una instrucción nula (o vacía) que no hace nada. Aprenderemos más sobre la instrucción nula en el capítulo 4.

b) *Error:* el operador relacional `=>`.
Corrección: cambie `=>` a `>=` y tal vez quiera cambiar "igual o mayor que" a "mayor o igual que" también.

Ejercicios

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

a) `std::cin`

b) `std::cout`

2.8 Complete las siguientes oraciones:

a) _____ se utilizan para documentar un programa y mejorar su legibilidad.

b) El objeto que se utiliza para imprimir información en la pantalla es _____ .

c) Una instrucción de C++ que toma una decisión es _____.

d) La mayoría de los cálculos se realizan normalmente mediante instrucciones _____.

e) El objeto _____ recibe valores de entrada del teclado.

2.9 Escriba una sola instrucción o línea en C++ que realice cada una de las siguientes tareas:
a) Imprimir el mensaje `"Escriba dos números"`.
b) Asignar el producto de las variables `b` y `c` a la variable `a`.
c) Indicar que un programa va a realizar un cálculo de nómina (es decir, usar texto que ayude a documentar un programa).
d) Recibir tres valores de entrada del teclado y colocarlos en las variables enteras `a`, `b` y `c`.

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

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

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

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

2.13 ¿Cuáles de las siguientes instrucciones de C++ contienen variables cuyos valores se reemplazan?
a) `cin >> b >> c >> d >> e >> f;`
b) `p = i + j + k + 7;`
c) `cout << "variables cuyos valores se reemplazan";`
d) `cout << "a = 5";`

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

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

2.15 ***(Orden de evaluación)*** Indique el orden de evaluación de los operadores en cada una de las siguientes instrucciones en C++ y muestre el valor de `x` después de ejecutar cada una de ellas.

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

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

2.17 ***(Impresión)*** Escriba un programa que imprima los números 1 a 4 en la misma línea con cada par de números adyacentes separado por un espacio. Haga esto de varias formas:

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

2.18 ***(Comparación de enteros)*** Escriba un programa que pida al usuario que escriba dos enteros, obtenga los números del usuario y luego imprima el número más grande, seguido de las palabras `"es más grande"`. Si los números son iguales, imprima el mensaje `"Estos numeros son iguales"`.

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

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

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

2.21 ***(Mostrar figuras con asteriscos)*** Escriba un programa que imprima un cuadro, un óvalo, una flecha y un diamante como se indica a continuación:

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

2.22 ¿Qué imprime el siguiente código?

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

2.23 ***(Enteros mayor y menor)*** Escriba un programa que lea cinco enteros, determine e imprima los enteros mayor y menor en el grupo. Use sólo las técnicas de programación que aprendió en este capítulo.

2.24 ***(Impar o par)*** Escriba un programa que lea un entero, determine e imprima si es impar o par. [*Sugerencia:* use el operador módulo. Un número par es un múltiplo de dos. Cualquier múltiplo de dos deja un residuo de cero cuando se divide entre 2].

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

2.26 ***(Patrón de tablero de damas)*** Muestre el siguiente patrón de tablero de damas con ocho instrucciones de salida, y después muestre el mismo patrón utilizando el menor número de instrucciones posible.

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

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

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

Puede imprimir el equivalente entero de un carácter mediante el uso de using `static_cast`, como en el siguiente ejemplo:

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

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

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

```
4   2   3   3   9
```

2.29 ***(Tabla)*** Utilice las técnicas de este capítulo para escribir un programa que calcule los cuadrados y cubos de los enteros del 0 al 10. Use tabuladores para imprimir la siguiente tabla ordenada de valores:

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

Hacer la diferencia

2.30 ***(Calculadora del índice de masa corporal)*** En el ejercicio 1.9 introdujimos la calculadora del índice de masa corporal (BMI). Las fórmulas para calcular el BMI son

$$BMI = \frac{pesoEnLibras \times 703}{alturaEnPulgadas \times alturaEnPulgadas}$$

o

$$BMI = \frac{pesoEnKilogramos}{alturaEnMetros \times alturaEnMetros}$$

Cree una aplicación de calculadora del BMI que lea el peso del usuario en libras y la altura en pulgadas (o, si lo prefiere, el peso del usuario en kilogramos y la altura en metros), para que luego calcule y muestre el índice de masa corporal del usuario. La aplicación debe mostrar además la siguiente información del Departamento de Salud y Servicios Humanos/Instituto Nacional de Salud para que el usuario pueda evaluar su BMI:

```
VALORES DE BMI
Bajo peso:   menos de 18.5
Normal:      entre 18.5 y 24.9
Sobrepeso:   entre 25 y 29.9
Obeso:       30 o más
```

[*Nota:* en este capítulo aprendió a usar el tipo `int` para representar números enteros. Cuando se realizan los cálculos del BMI con valores `int` se producen resultados en números enteros. En el capítulo 4 aprenderá a usar el tipo `double`, para representar a los números con puntos decimales. Cuando se realizan los cálculos del BMI con valores `double`, producen números con puntos decimales; a éstos se les conoce como números de "punto flotante"].

2.31 ***(Calculadora de ahorro por viajes compartidos en automóvil)*** Investigue varios sitios Web de viajes compartidos en automóvil. Cree una aplicación que calcule su costo diario al conducir su automóvil, de modo que pueda estimar cuánto dinero puede ahorrar si comparte los viajes en automóvil, lo cual también tiene otras ventajas, como la reducción de las emisiones de carbono y la reducción de la congestión de tráfico. La aplicación debe recibir como entrada la siguiente información y mostrar el costo por día para el usuario por conducir al trabajo:

a) Total de kilómetros conducidos por día.
b) Costo por litro de gasolina.
c) Promedio de kilómetros por litro.
d) Cuotas de estacionamiento por día.
e) Peaje por día.

3

Introducción a las clases, objetos y cadenas

Nada puede tener valor sin ser un objeto de utilidad.
—Karl Marx

Sus sirvientes públicos le sirven bien.
—Adlai E. Stevenson

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

Objetivos

En este capítulo aprenderá a:

- Definir una clase y utilizarla para crear un objeto.
- Implementar los comportamientos de una clase como funciones miembro.
- Implementar los atributos de una clase como datos miembros.
- Llamar a una función miembro de un objeto para realizar una tarea.
- Diferenciar entre los datos miembros de una clase y las variables locales de una función.
- Utilizar un constructor para inicializar los datos de un objeto al momento de crear el objeto.
- Maquinar la interfaz de la implementación de una clase y fomentar la reutilización.
- Usar objetos de la clase `string`.

Plan general

3.1 Introducción

En el capítulo 2 creó programas simples que mostraban mensajes al usuario, obtenían información del usuario, realizaban cálculos y tomaban decisiones. En este capítulo empezará a escribir programas que emplean los conceptos básicos de la *programación orientada a objetos* que presentamos en la sección 1.8. Una característica común de todos los programas del capítulo 2 fue que todas las instrucciones que ejecutaban tareas se encontraban en la función `main`. Por lo general, los programas que desarrollará en este libro consistirán de la función `main` y una o más *clases*, cada una de las cuales puede contener *datos miembros* y *funciones miembro*. Si usted se integra a un equipo de desarrollo en la industria, podría trabajar en sistemas de software que contengan cientos, o incluso miles, de clases. En este capítulo desarrollaremos un marco de trabajo simple y bien maquinado para organizar los programas orientados a objetos en C++.

Presentaremos una secuencia cuidadosamente pautada de programas funcionales completos para demostrarle cómo crear y utilizar sus propias clases. Estos ejemplos empiezan nuestro caso de estudio integrado acerca de cómo desarrollar una clase tipo libro de calificaciones, que los instructores pueden utilizar para mantener las calificaciones de las pruebas de sus estudiantes. También presentaremos la clase `string` de la biblioteca estándar de C++.

3.2 Definición de una clase con una función miembro

Vamos a empezar con un ejemplo (figura 3.1) que consiste en la clase `LibroCalificaciones` (líneas 8 a 16) que, cuando se desarrolle en su totalidad en el capítulo 7, representará a un libro de calificaciones que un instructor puede utilizar para mantener las calificaciones de los exámenes de sus estudiantes, y una función `main` (líneas 19 a 23) que crea un objeto `LibroCalificaciones`. La función `main` utiliza este objeto y su función miembro `mostrarMensaje` (líneas 12 a 15) para mostrar un mensaje en la pantalla y dar la bienvenida al instructor al programa del libro de calificaciones.

```
1  // Fig. 3.1: fig03_01.cpp
2  // Define la clase LibroCalificaciones con una función miembro llamada
   mostrarMensaje
3  // crea un objeto LibroCalificaciones y llama a su función mostrarMensaje.
4  #include <iostream>
5  using namespace std;
```

Fig. 3.1 | Define la clase `LibroCalificaciones` con una función miembro llamada `mostrarMensaje`, crea un objeto `LibroCalificaciones` y llama a su función `mostrarMensaje` (parte 1 de 2).

```
6
7  // Definición de la clase LibroCalificaciones
8  class LibroCalificaciones
9  {
10 public:
11    // función que muestra un mensaje de bienvenida para el usuario de
         LibroCalificaciones
12    void mostrarMensaje() const
13    {
14       cout << "Bienvenido al Libro de calificaciones!" << endl;
15    } // fin de la función mostrarMensaje
16 }; // fin de la clase LibroCalificaciones
17
18 // la función main empieza la ejecución del programa
19 int main()
20 {
21    LibroCalificaciones miLibroCalificaciones; // crea un objeto
      LibroCalificaciones llamado miLibroCalificaciones
22    miLibroCalificaciones.mostrarMensaje(); // llama a la función
                                                 mostrarMensaje del objeto
23 } // fin de main
```

```
Bienvenido al Libro de calificaciones!
```

Fig. 3.1 | Define la clase `LibroCalificaciones` con una función miembro llamada `mostrarMensaje`, crea un objeto `LibroCalificaciones` y llama a su función `mostrarMensaje` (parte 2 de 2).

La clase `LibroCalificaciones`

Antes de que la función `main` (líneas 19 a 23) pueda crear un objeto de la clase `LibroCalificaciones`, debemos indicar al compilador qué funciones miembro y cuáles datos miembros pertenecen a la clase. La **definición de la clase** `LibroCalificaciones` (líneas 8 a 16) contiene una función miembro llamada `mostrarMensaje` (líneas 12 a 15), la cual muestra un mensaje en la pantalla (línea 14). Necesitamos crear un objeto de la clase `LibroCalificaciones` (línea 21) y llamar a su función miembro `mostrarMensaje` (línea 22) para hacer que se ejecute la línea 14 y se muestre el mensaje de bienvenida. Pronto explicaremos las líneas 21 y 22 con detalle.

La definición de la clase empieza en la línea 8 con la palabra clave `class`, seguida del nombre de la clase `LibroCalificaciones`. Por convención, el nombre de una clase definida por el usuario empieza con una letra mayúscula, y por legibilidad, cada palabra subsiguiente en el nombre de la clase empieza con una letra mayúscula. Este estilo de capitalización se conoce comúnmente como **nomenclatura de Pascal**, debido a que esta convención era muy utilizada en el lenguaje de programación Pascal. Las ocasionales letras mayúsculas se asemejan a las jorobas de un camello. En términos más generales, el estilo de capitalización conocido como **nomenclatura de camello** permite que la primera letra sea mayúscula o minúscula (como `miLibroCalificaciones` en la línea 21).

El **cuerpo** de cada clase va encerrado entre un par de llaves izquierda y derecha (`{` y `}`), como en las líneas 9 y 16. La definición de la clase termina con un punto y coma (línea 16).

Error común de programación 3.1

Olvidar el punto y coma al final de la definición de una clase es un error de sintaxis.

Recuerde que la función `main` siempre se llama de manera automática cuando ejecutamos un programa. La mayoría de las funciones *no* se llaman de manera automática. Como pronto veremos, debemos llamar a la función miembro `mostrarMensaje` de manera *explícita* para indicarle que debe realizar su tarea.

La línea 10 contiene la palabra clave `public`, que es un **especificador de *acceso***. En las líneas 12 a 15 se define la función miembro `mostrarMensaje`. Esta función miembro aparece *después* del especificador de acceso `public:` para indicar que la función está "disponible para el público"; es decir, otras funciones en el programa (como `main`) la pueden llamar, y también las funciones miembro de otras clases (si las hay). Los especificadores de acceso siempre van seguidos de un signo de dos puntos (`:`). En el resto del libro, cuando hagamos referencia al especificador de acceso `public`, omitiremos el punto y coma como en esta oración. En la sección 3.4 presentaremos el especificador de acceso `private`. Más adelante en el libro estudiaremos el especificador de acceso `protected`.

Cada función en un programa realiza una tarea y puede *devolver un valor* cuando complete su tarea; por ejemplo, una función podría realizar un cálculo y después devolver el resultado del mismo. Al definir una función, debemos especificar un **tipo de valor de retorno** para indicar el tipo de valor que devuelve la función cuando completa su tarea. En la línea 12, la palabra clave **`void`** a la izquierda del nombre de la función `mostrarMensaje` es el tipo de valor de retorno de ésta. El tipo de valor de retorno `void` indica que `mostrarMensaje` *no* devolverá datos a la **función que la llamó** (en este ejemplo, la línea 22 de `main`, como veremos en unos momentos) cuando complete su tarea. En la figura 3.5 veremos un ejemplo de una función que *sí* devuelve un valor.

El nombre de la función miembro, `mostrarMensaje`, va después del tipo de valor de retorno (línea 12). Por convención, los nombres de nuestras funciones usan el estilo de *nomenclatura de camello* con la primera letra en minúscula. Los paréntesis después del nombre de la función miembro indican que ésta es una *función*. Un conjunto vacío de paréntesis, como se muestra en la línea 12, indica que esta función miembro *no* requiere datos adicionales para realizar su tarea. En la sección 3.3 veremos un ejemplo de una función miembro que *sí* requiere datos adicionales.

Declaramos la función miembro `mostrarMensaje` como **`const`** en la línea 12 debido a que, en el proceso de mostrar el mensaje `"Bienvenido al libro de calificaciones!"` la función *no* modifica (y *no debería* hacerlo) el objeto `LibroCalificaciones` sobre el cual se llama. Al declarar `mostrarMensaje` como `const` indicamos al compilador que "esta función *no* debe modificar el objeto sobre el cual se llama; y si lo hace, hay que generar un error de compilación". Esto puede ayudar al programador a localizar errores en caso de insertar por accidente código en `mostrarMensaje` que *pudiera* modificar el objeto. La línea 12 se conoce comúnmente como **encabezado de función**.

El *cuerpo* de todas las funciones está delimitado por las llaves izquierda y derecha (`{` y `}`), como en las líneas 13 y 15. El *cuerpo de la función* contiene instrucciones que realizan la tarea de esa función. En este caso, la función miembro `mostrarMensaje` contiene una instrucción (línea 14) que muestra el mensaje `"Bienvenido al Libro de calificaciones!"`. Una vez que se ejecute esta instrucción, la función habrá completado su tarea.

Prueba de la clase `LibroCalificaciones`

A continuación nos gustaría utilizar la clase `LibroCalificaciones` en un programa. Como aprendió en el capítulo 2, la función `main` (líneas 19 a 23) empieza la ejecución de todos los programas.

En este programa queremos llamar a la función miembro `mostrarMensaje` de la clase `LibroCalificaciones` para mostrar el mensaje de bienvenida. Por lo general, no podemos llamar a una función miembro de una clase, sino hasta *crear un objeto* de esa clase. (Como veremos en la sección 9.14, las funciones miembro `static` son una excepción). En la línea 21 se crea un objeto de la clase `LibroCalificaciones`, llamado `miLibroCalificaciones`. El tipo de la variable es `LibroCalificaciones`: la clase que definimos en las líneas 8 a 16. Cuando declaramos variables de tipo `int`, como en el capítulo 2, el compilador sabe lo que es `int`: un *tipo fundamental* que está "integrado" a C++. Sin embargo, en la línea 21 el compilador *no* sabe automáticamente qué tipo corresponde a `LibroCalificaciones`: es un **tipo definido por el usuario**. Para indicar al compilador qué es `LibroCalificaciones`, incluimos la *definición de la clase* (líneas 8 a 16). Si omitiéramos estas líneas, el compilador generaría un mensaje de error. Cada nueva clase que creamos se convierte en un nuevo *tipo* que puede usarse para crear objetos. Los programadores pueden definir nuevos tipos de clases según lo necesiten; ésta es una razón por la cual C++ se conoce como un **lenguaje de programación extensible**.

En la línea 22 se hace una *llamada* a la función miembro `mostrarMensaje`, usando la variable `miLibroCalificaciones` seguida del **operador punto** (`.`), el nombre de la función `mostrarMensaje` y un conjunto vacío de paréntesis. Esta llamada hace que la función `mostrarMensaje` realice su tarea. Al principio de la línea 22, "`miLibroCalificaciones`" indica que `main` debe usar el objeto `LibroCalificaciones` que se creó en la línea 21. Los *paréntesis vacíos* en la línea 12 indican que la función miembro `mostrarMensaje` *no* requiere datos adicionales para realizar su tarea, razón por la cual llamamos a esta función con paréntesis vacíos en la línea 22 (en la sección 3.3 veremos cómo pasar datos a una función). Cuando `mostrarMensaje` completa su tarea, el programa llega al final de `main` (línea 23) y termina.

Diagrama de clases de UML para la clase `LibroCalificaciones`

En la sección 1.8 vimos que UML es un lenguaje gráfico estandarizado, utilizado por los desarrolladores de software para representar sistemas orientados a objetos. En UML, cada clase se modela en un **diagrama de clases de UML** en forma de un *rectángulo* con tres *compartimientos*. La figura 3.2 presenta un diagrama de clases para la clase `LibroCalificaciones` (figura 3.1). El *compartimiento superior* contiene el nombre de la clase, centrado en forma horizontal y en negrita. El *compartimiento de en medio* contiene los atributos de la clase, que en C++ corresponden a los datos miembros. En estos momentos el compartimiento está vacío, ya que la clase `LibroCalificaciones` no tiene atributos todavía (en la sección 3.4 presentaremos una versión de la clase `LibroCalificaciones` con un atributo). El *compartimiento inferior* contiene las operaciones de la clase, que en C++ corresponden a las funciones miembro. Para modelar las operaciones, UML lista el nombre de la operación seguido de un conjunto de paréntesis. La clase `LibroCalificaciones` tiene una sola función miembro llamada `mostrarMensaje`, por lo que el compartimiento inferior de la figura 3.2 lista una operación con este nombre. La función miembro `mostrarMensaje` *no* requiere información adicional para realizar sus tareas, por lo que los paréntesis que van después de `mostrarMensaje` en el diagrama de clases están *vacíos*, de igual forma que como aparecieron en el encabezado de la función miembro, en la línea 12 de la figura 3.1. El *signo más (+)* que va antes del nombre de la operación indica que `mostrarMensaje` es una operación *public* en UML (es decir, una función miembro `public` en C++).

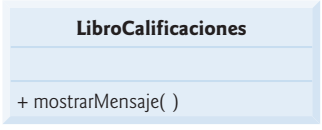


Fig. 3.2 | Diagrama de clases de UML, el cual indica que la clase `LibroCalificaciones` tiene una operación `mostrarMensaje` pública.

3.3 Definición de una función miembro con un parámetro

En nuestra analogía del automóvil de la sección 1.8, hablamos sobre el hecho de que al oprimir el pedal del acelerador se envía un *mensaje* al automóvil para que realice una tarea: hacer que vaya más rápido. Pero *¿qué tan rápido* debería acelerar el automóvil? Como sabe, entre más oprima el pedal, mayor será la aceleración del automóvil. Por lo tanto, el mensaje para el automóvil en realidad incluye *tanto* la *tarea a realizar* como *información adicional que ayuda al automóvil a realizar su tarea*. A la información adicional se le conoce como **parámetro**; el *valor* del parámetro ayuda al automóvil a determinar qué tan rápido debe acelerar. De manera similar, una función miembro puede requerir uno o más parámetros que representan la información adicional que necesita para realizar su tarea. La llamada a una función proporciona valores llamados **argumentos** para cada uno de los parámetros de esa función. Por ejemplo, para realizar un depósito en una cuenta bancaria, suponga que una función miembro llamada `depositar` de una clase `Cuenta` especifica un parámetro que representa el *monto a depositar*. Cuando se hace una lla-

mada a la función miembro `deposito`, se copia al parámetro de la función miembro un valor como argumento, que representa el monto a depositar. Después, la función miembro suma esa cantidad al saldo de la cuenta.

Definición y prueba de la clase `LibroCalificaciones`

Nuestro siguiente ejemplo (figura 3.3) redefine la clase `LibroCalificaciones` (líneas 9 a 18) con una función miembro `mostrarMensaje` (líneas 13 a 17) que muestra el nombre del curso como parte del mensaje de bienvenida. La nueva versión de `mostrarMensaje` requiere un *parámetro* (`nombreCurso` en la línea 13) que representa el nombre del curso a imprimir en pantalla.

```
1  // Fig. 3.3: fig03_03.cpp
2  // Define la clase LibroCalificaciones con una función miembro que recibe un
      parámetro,
3  // crea un objeto LibroCalificaciones y llama a su función mostrarMensaje.
4  #include <iostream>
5  #include <string> // el programa usa la clase string estándar de C++
6  using namespace std;
7
8  // definición de la clase LibroCalificaciones
9  class LibroCalificaciones
10 {
11 public:
12    // función que muestra un mensaje de bienvenida para el usuario de
         LibroCalificaciones
13    void mostrarMensaje( string nombreCurso ) const
14    {
15       cout << "Bienvenido al libro de calificaciones para\n" << nombreCurso << "!"
16          << endl;
17    } // fin de la función mostrarMensaje
18 }; // fin de la clase LibroCalificaciones
19
20 // la función main empieza la ejecución del programa
21 int main()
22 {
23    string nombreDelCurso; // cadena de caracteres que almacena el nombre del curso
24    LibroCalificaciones miLibroCalificaciones;
      // crea un objeto LibroCalificaciones llamado mi LibroCalificaciones
25
26    // pide y recibe el nombre del curso como entrada
27    cout << "Escriba el nombre del curso:" << endl;
28    getline( cin, nombreDelCurso );
      // lee el nombre de un curso con espacios en blanco
29    cout << endl; // imprime una línea en blanco
30
31    // llama a la función mostrarMensaje de miLibroCalificaciones
32    // y pasa nombreDelCurso como argumento
33    miLibroCalificaciones.mostrarMensaje( nombreDelCurso );
34 } // fin de main
```

Fig. 3.3 | Define la clase `LibroCalificaciones` con una función miembro que recibe un parámetro, crea un objeto `LibroCalificaciones` y llama a su función `mostrarMensaje` (parte 1 de 2).

```
Escriba el nombre del curso:
CS101 Introduccion a la programacion en C++

Bienvenido al libro de calificaciones para
CS101 Introduccion a la programacion en C++!
```

Fig. 3.3 | Define la clase `LibroCalificaciones` con una función miembro que recibe un parámetro, crea un objeto `LibroCalificaciones` y llama a su función `mostrarMensaje` (parte 2 de 2).

Antes de hablar sobre las nuevas características de la clase `LibroCalificaciones`, veamos cómo se utiliza la nueva clase en `main` (líneas 21 a 34). En la línea 23 se crea una variable de tipo **`string`** llamada `nombreDelCurso`, la cual se utilizará para almacenar el nombre del curso que escriba el usuario. Una variable de tipo `string` representa a una cadena de caracteres tal como `"CS101 Introduccion a la programacion en C++"`. En realidad, una cadena es un *objeto* de la clase `string`, de la Biblioteca estándar de C++. Esta clase se define en el **encabezado `<string>`**, y el nombre `string` (al igual que `cout`) pertenece al espacio de nombres `std`. Para que las líneas 13 y 23 se puedan compilar, en la línea 5 se *incluye* el encabezado `<string>`. La directiva `using` en la línea 6 nos permite escribir simplemente `string` en la línea 23, en vez de `std::string`. Por ahora, puede considerar a las variables `string` como las variables de otros tipos como `int`. En la sección 3.8 y en el capítulo 21 (en inglés, en el sitio web) aprenderá acerca de las herramientas adicionales de `string`.

En la línea 24 se crea un objeto de la clase `LibroCalificaciones`, llamado `miLibroCalificaciones`. En la línea 27 se pide al usuario que escriba el nombre de un curso. En la línea 28 se lee el nombre del usuario y se asigna a la variable `nombreDelCurso`, usando la función de biblioteca **`getline`** para llevar a cabo la entrada. Antes de explicar esta línea de código, veamos por qué no podemos simplemente escribir

```
cin >> nombreDelCurso;
```

para obtener el nombre del curso.

En la ejecución de ejemplo de nuestro programa, utilizamos el nombre "`CS101 Introduccion a la programacion en C++`", el cual contiene varias palabras *separadas por espacios en blanco* (recuerde que resaltamos la entrada que suministra el usuario en negrita). Cuando se lee un objeto `string` con el operador de extracción de flujo, `cin` lee caracteres *hasta que se llega al primer carácter de espacio en blanco*. Por ende, la instrucción anterior sólo leería "`CS101`". El resto del nombre del curso tendría que leerse mediante operaciones de entrada subsiguientes.

En este ejemplo, nos gustaría que el usuario escribiera el nombre completo del curso y que oprimiera *Intro* para enviarlo al programa, y nos gustaría almacenar el nombre *completo* del curso en la variable `string` llamada `nombreDelCurso`. La llamada a la función `getline( cin, nombreDelCurso)` en la línea 28 lee caracteres (*incluyendo* los caracteres de espacio que separan las palabras en la entrada) del objeto flujo de entrada estándar `cin` (es decir, el teclado) hasta encontrar el carácter de *nueva línea*, coloca los caracteres en la variable `string` llamada `nombreDelCurso` y *descarta* el carácter de nueva línea. Al oprimir *Intro* mientras se introducen los datos, se inserta una nueva línea en el flujo de entrada. Debe incluirse el encabezado `<string>` en el programa para usar la función `getline`, que pertenece al espacio de nombres `std`.

En la línea 33 se hace una llamada a la función miembro `mostrarMensaje` de `miLibroCalificaciones`. La variable `nombreDelCurso` entre paréntesis es el *argumento* que se pasa a la función miembro `mostrarMensaje` para que pueda realizar su tarea. El valor de la variable `nombreDelCurso` en `main` se *copia* al parámetro `nombreCurso` de la función miembro `mostrarMensaje` en la línea 13. Al ejecutar este programa, la función miembro `mostrarMensaje` imprime, como parte del mensaje de bienvenida, el

nombre del curso que usted escriba (en nuestra ejecución de ejemplo, `CS101 Introduccion a la programacion en C++`).

Más sobre los argumentos y los parámetros

Para especificar en la definición de una función que ésta requiere datos para realizar su tarea, hay que colocar información adicional en la **lista de parámetros** de la función, la cual se encuentra en los paréntesis que van después del nombre de la función. La lista de parámetros puede contener *cualquier* número de parámetros, incluso *ninguno* (lo que se representa mediante los paréntesis vacíos, como en la línea 12 de la figura 3.1), para indicar que una función *no* requiere parámetros. La lista de parámetros de la función miembro `mostrarMensaje` (figura 3.3, línea 13) declara que la función requiere un parámetro. Cada parámetro debe especificar un *tipo* y un *identificador*. El tipo `string` y el identificador `nombreCurso` indican que la función miembro `mostrarMensaje` requiere un objeto `string` para realizar su tarea. El cuerpo de la función miembro utiliza el parámetro `nombreDelCurso` para acceder al valor que se pasa a la función en la llamada (línea 33 en `main`). En las líneas 15 y 16 se muestra el valor del parámetro `nombreDelCurso` como parte del mensaje de bienvenida. El nombre de la variable de parámetro (`nombreCurso` en la línea 13) puede ser *igual* o *distinto* al nombre de la variable de argumento (`nombreDelCurso` en la línea 33); en el capítulo 6 aprenderá por qué.

Una función puede especificar múltiples parámetros; sólo hay que separar un parámetro de otro mediante una coma. El número y el orden de los argumentos en la llamada a una función *deben coincidir* con el número y orden de los parámetros en la lista de parámetros del encabezado de la función miembro que se llamó. Además, los tipos de los argumentos en la llamada a la función deben ser consistentes con los tipos de los parámetros correspondientes al encabezado de la función (como veremos en capítulos posteriores, no siempre se requiere que el tipo de un argumento y el tipo de su correspondiente parámetro sean *idénticos*, pero deben ser "consistentes"). En nuestro ejemplo, el único argumento `string` en la llamada a la función (es decir, `nombreDelCurso`) *coincide exactamente* con el único parámetro `string` en la definición de la función miembro (es decir, `nombreCurso`).

Diagrama de clases de UML actualizado para la clase `LibroCalificaciones`

El diagrama de clases de UML de la figura 3.4 modela la clase `LibroCalificaciones` de la figura 3.3. Al igual que la clase `LibroCalificaciones` definida en la figura 3.1, esta clase `LibroCalificaciones` contiene la función miembro `public` llamada `mostrarMensaje`. Sin embargo, esta versión de `mostrarMensaje` tiene un *parámetro*. Para modelar un parámetro, UML lista el nombre del parámetro, seguido de dos puntos y del tipo del parámetro entre paréntesis, después del nombre de la operación. UML tiene sus *propios* tipos de datos similares a los de C ++. UML es *independiente del lenguaje* —se utiliza con muchos lenguajes de programación distintos— por lo que su terminología no coincide exactamente con la de C++. Por ejemplo, el tipo `String` de UML corresponde al tipo `string` de C++. La función miembro `mostrarMensaje` de la clase `LibroCalificaciones` (figura 3.3, líneas 13 a 17) tiene un parámetro `string` llamado `nombreCurso`, por lo que en la figura 3.4 se lista a `nombreCurso : String` entre los paréntesis que van después del nombre de la operación `mostrarMensaje`. Esta versión de la clase `LibroCalificaciones` aún *no* tiene datos miembros.

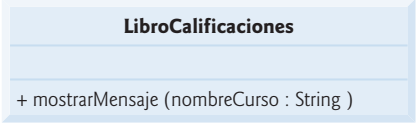


Fig. 3.4 | Diagrama de clases de UML, que indica que la clase `LibroCalificaciones` tiene una operación pública llamada `mostrarMensaje`, con un parámetro llamado `nombreCurso` de tipo `String` de UML.

3.4 Datos miembros, funciones *establecer* y *obtener*

En el capítulo 2 declaramos todas las variables de un programa en su función `main`. Las variables que se declaran en el cuerpo de la definición de una función se conocen como **variables locales**, y *sólo* se pueden utilizar desde la línea de su declaración en la función, hasta la llave derecha de cierre (`}`) del bloque en el que se declaran. Una variable local se debe declarar *antes* de poder utilizarla en una función. No se puede acceder a una variable local *fuera* de la función en la que está declarada. *Cuando una función termina, se pierden los valores de sus variables locales* (en el capítulo 6 veremos una excepción a esto, cuando hablemos sobre las variables locales `static`).

Por lo general, una clase consiste en una o más funciones miembro que manipulan los atributos pertenecientes a un objeto específico de la clase. Los atributos se representan como variables en la definición de una clase. Dichas variables se llaman **datos miembros** y se declaran *dentro* de la definición de una clase, pero *fuera* de los cuerpos de las definiciones de las funciones miembro de la clase. Cada objeto de una clase mantiene sus propios atributos en memoria. Estos atributos existen durante toda la vida del objeto. El ejemplo en esta sección demuestra una clase `LibroCalificaciones`, que contiene un dato miembro llamado `nombreCurso` para representar el nombre del curso de un objeto `LibroCalificaciones` específico. Si crea más de un objeto `LibroCalificaciones`, cada uno tendrá su propio miembro de datos `nombreCurso`, y éstos pueden tener *diferentes* valores.

La clase `LibroCalificaciones` *con un dato miembro, y funciones* establecer *y* obtener

En nuestro siguiente ejemplo, la clase `LibroCalificaciones` (figura 3.5) mantiene el nombre del curso como un *dato miembro*, para que pueda *usarse* o *modificarse* durante la ejecución de un programa. Esta clase contiene las funciones miembro `establecerNombreCurso`, `obtenerNombreCurso` y `mostrarMensaje`. La función miembro `establecerNombreCurso` *almacena* el nombre de un curso en un miembro de datos de `LibroCalificaciones`. La función miembro `obtenerNombreCurso` *obtiene* el nombre del curso de ese dato miembro. La función miembro `mostrarMensaje`, que en este caso *no especifica parámetros*, sigue mostrando un mensaje de bienvenida que incluye el nombre del curso. Pero como veremos más adelante, la función ahora *obtiene* el nombre del curso mediante una llamada a otra función en la misma clase: `obtenerNombreCurso`.

```
1  // Fig. 3.5: fig03_05.cpp
2  // Define la clase LibroCalificaciones que contiene un miembro de datos
3  // nombreCurso y funciones miembro para establecer y obtener su valor;
4  // Crea y manipula un objeto LibroCalificaciones con estas funciones.
5  #include <iostream>
6  #include <string> // el programa usa la clase string estándar de C++
7  using namespace std;
8
9  // definición de la clase LibroCalificaciones
10 class LibroCalificaciones
11 {
12 public:
13    // función que establece el nombre del curso
14    void establecerNombreCurso( string nombre )
15    {
16       nombreCurso = nombre; // almacena el nombre del curso en el objeto
17    } // fin de la función establecerNombreCurso
```

Fig. 3.5 | Definición y prueba de la clases `LibroCalificaciones` con un miembro de datos y funciones *establecer* y *obtener* (parte 1 de 2).

```
18
19    // función que obtiene el nombre del curso
20    string obtenerNombreCurso() const
21    {
22       return nombreCurso; // devuelve el nombreCurso del objeto
23    } // fin de la función obtenerNombreCurso
24
25    // función que muestra un mensaje de bienvenida
26    void mostrarMensaje() const
27    {
28       // esta instrucción llama a obtenerNombreCurso para obtener el
29       // nombre del curso que representa este LibroCalificaciones
30       cout << "Bienvenido al libro de calificaciones para\n"
         << obtenerNombreCurso() << "!"
31          << endl;
32    } // fin de la función mostrarMensaje
33 private:
34    string nombreCurso; // nombre del curso para este LibroCalificaciones
35 }; // fin de la clase LibroCalificaciones
36
37 // la función main empieza la ejecución del programa
38 int main()
39 {
40    string nombreDelCurso; // cadena de caracteres para almacenar el nombre del curso
41    LibroCalificaciones miLibroCalificaciones;
      // crea un objeto LibroCalificaciones llamado miLibroCalificaciones
42
43    // muestra el valor inicial de nombreCurso
44    cout << "El nombre inicial del curso es: "
      << miLibroCalificaciones.obtenerNombreCurso()
45       << endl;
46
47    // pide, recibe y establece el nombre del curso
48    cout << "\nEscriba el nombre del curso:" << endl;
49    getline( cin, nombreDelCurso );
      // lee el nombre de un curso con espacios en blanco
50    miLibroCalificaciones.establecerNombreCurso( nombreDelCurso )
      // establece el nombre del curso
51
52    cout << endl; // imprime una línea en blanco
53    miLibroCalificaciones.mostrarMensaje();
      // muestra un mensaje con el nuevo nombre del curso
54 } // fin de main
```

```
El nombre inicial del curso es:

Escriba el nombre del curso:
CS101 Introduccion a la programacion en C++

Bienvenido al libro de calificaciones parar
CS101 Introduccion a la programacion en C++!
```

Fig. 3.5 | Definición y prueba de la clases `LibroCalificaciones` con un miembro de datos y funciones *establecer* y *obtener* (parte 2 de 2).

Un instructor típico enseña *varios* cursos, cada uno con su propio nombre. En la línea 34 se declara que `nombreCurso` es una variable de tipo `string`. Como la variable se declara en la definición de la clase (líneas 10 a 35) pero fuera de los cuerpos de las definiciones de las funciones miembro de ésta (líneas 14 a 17, 20 a 23 y 26 a 32), la variable es un dato *miembro*. Cada instancia (es decir, objeto) de la clase `LibroCalificaciones` contiene cada uno de los datos miembros de la clase; si hay dos objetos `LibroCalificaciones`, cada objeto tiene su *propio* `nombreCurso` (uno por cada objeto), como veremos en el ejemplo de la figura 3.7. Un beneficio de hacer de `nombreCurso` un dato miembro es que *todas* las funciones miembro de la clase pueden manipular cualquier dato miembro que aparezca en la definición de la clase (en este caso, `nombreCurso`).

Los especificadores de acceso `public` *y* `private`

La mayoría de las declaraciones de datos miembros aparecen después del especificador de accesos **`private`**. Las variables o funciones declaradas después del especificador de acceso `private` (y *antes* del siguiente especificador de acceso, si hay uno) son accesibles sólo para las funciones miembro de la clase en la que se declaran (o para las "amigas" de la clase, como veremos en el capítulo 9). Así, el miembro de datos `nombreCurso` *sólo* puede utilizarse en las funciones miembro `establecerNombreCurso`, `obtenerNombreCurso` y `mostrarMensaje` de la clase `LibroCalificaciones` (o en las "amigas" de la clase, si las hay).

Tip para prevenir errores 3.1

Al hacer que los datos miembros de una clase sean `private` *y las funciones miembro de la clase sean* `public`*, se facilita la depuración debido a que los problemas con las manipulaciones de datos se localizan, ya sea en las funciones miembro de la clase o en las amigas de ésta.*

Error común de programación 3.2

Si una función, que no sea miembro de una clase específica (o amiga de esa clase), intenta acceder a un miembro `private` *de esa clase, se produce un error de compilación.*

El *acceso predeterminado* para los datos miembros es `private`, de manera que todos los miembros *después* del encabezado de la clase y *antes* del primer especificador de acceso son `private`. Los especificadores de acceso `public` y `private` pueden repetirse, pero esto es innecesario y puede ser confuso.

El proceso de declarar datos miembros con el modificador de acceso `private` se conoce como **ocultamiento de datos**. Cuando un programa crea un objeto de la clase `LibroCalificaciones`, el dato miembro `nombreCurso` se *encapsula* (oculta) en el objeto, y sólo está accesible para las funciones miembro de la clase de ese objeto. En la clase `LibroCalificaciones`, las funciones miembro `establecerNombreCurso` y `obtenerNombreCurso` manipulan al dato miembro `nombreCurso` directamente.

Las funciones miembro `establecerNombreCurso` *y* `obtenerNombreCurso`

La función miembro `obtenerNombreCurso` (líneas 14 a 17) no *devuelve* datos cuando completa su tarea, por lo que su tipo de valor de retorno es `void`. La función miembro *recibe* un parámetro `nombre`, el cual representa el nombre del curso que recibirá como argumento (como veremos en la línea 50 de `main`). La línea 16 asigna `nombre` al dato miembro `nombreCurso`, con lo cual se *modifica* el objeto; por esta razón *no* declaramos a `establecerNombreCurso` como `const`. En este ejemplo, `establecerNombreCurso` no *valida* el nombre del curso; es decir, la función *no* verifica que el nombre del curso se apegue a cierto formato en especial, o que siga cualquier otra regla en relación con la apariencia que debe tener un nombre de curso "válido". Por ejemplo, suponga que una universidad puede imprimir certificados de los estudiantes que contengan los nombres de cursos de sólo 25 caracteres o menos. En este caso, es conveniente que la clase `LibroCalificaciones` asegure que su dato miembro `nombreCurso` nunca contendrá más de 25 caracteres. En la sección 3.8 hablaremos sobre la validación.

La función miembro `obtenerNombreCurso` (definida en las líneas 20 a 23) *devuelve* un valor de `nombreCurso` de un objeto `LibroCalificaciones` específico, *sin* modificar el objeto; por esta razón declaramos a `obtenerNombreCurso` como `const`. La función miembro tiene una *lista de parámetros*

vacía, por lo que *no* requiere información adicional para realizar su tarea. La función específica que devuelve un objeto `string`. Cuando se hace una llamada a una función que especifica un tipo de valor de retorno distinto de `void`, y ésta completa su tarea, la función usa una **instrucción `return`** (como en la línea 22) para *devolver un resultado* a la función que la llamó. Por ejemplo, cuando usted va a un cajero automático (ATM) y solicita el saldo de su cuenta, espera que el ATM le devuelva un valor que representa su saldo. De manera similar, cuando una instrucción llama a la función miembro `obtenerNombreCurso` en un objeto `LibroCalificaciones`, la instrucción espera recibir el nombre del curso de `LibroCalificaciones` (en este caso, un objeto `string`, como se especifica en el tipo de valor de retorno de la función).

Si tiene una función llamada `cuadrado` que devuelve el cuadrado de su argumento, es de esperarse que la instrucción

```
resultado = cuadrado( 2 );
```

devuelva `4` de la función `cuadrado` y asigne el valor `4` a la variable `resultado`. Si tiene una función llamada `maximo` que devuelve el mayor de tres argumentos enteros, la siguiente instrucción

```
mayor = maximo( 27, 114, 51 );
```

devuelve `114` de la función `maximo` y asigna este valor a la variable `mayor`.

Las instrucciones en las líneas 16 y 22 utilizan la variable `nombreCurso` (línea 34), aun y cuando esta variable *no* se declaró en ninguna de las funciones miembro. Podemos hacer esto ya que `nombreCurso` es un *miembro de datos* de la clase y éstos son accesibles desde las funciones miembro de una clase.

La función miembro `mostrarMensaje`

La función miembro `mostrarMensaje` (líneas 26 a 32) *no* devuelve datos cuando completa su tarea, por lo que su tipo de valor de retorno es `void`. Esta función *no* recibe parámetros, por lo que su lista de parámetros está vacía. En las líneas 30 y 31 se imprime un mensaje de bienvenida, que incluye el valor del dato miembro `nombreCurso`. La línea 30 llama a la función miembro `obtenerNombreCurso` para obtener el valor de `nombreCurso`. La función miembro `mostrarMensaje` también podría acceder al dato miembro `nombreCurso` directamente, así como las funciones miembro `establecerNombreCurso` y `obtenerNombreCurso`. En breve explicaremos por qué es preferible, desde la perspectiva de ingeniería de software, llamar a la función miembro `obtenerNombreCurso` para obtener el valor de `nombreCurso`.

Prueba de la clase `LibroCalificaciones`

La función `main` (líneas 38 a 54) crea un objeto de la clase `LibroCalificaciones` y utiliza cada una de sus funciones miembro. En la línea 41 se crea un objeto `LibroCalificaciones` llamado `miLibroCalificaciones`. En las líneas 44 a 45 se muestra el nombre inicial del curso, llamando a la función miembro `obtenerNombreCurso` del objeto. La primera línea de la salida no muestra un nombre de curso, ya que al principio el dato miembro `nombreCurso` del objeto (es decir, un `string`) está vacío; de manera predeterminada, el valor inicial de un objeto `string` es lo que se denomina **cadena vacía** (una cadena que no contiene caracteres). No aparece nada en la pantalla cuando se muestra una cadena vacía.

En la línea 48 se pide al usuario que escriba el nombre de un curso. La variable `string` local `nombreDelCurso` (declarada en la línea 40) se inicializa con el nombre del curso que escribió el usuario, el cual se devuelve mediante la llamada a la función `getline` (línea 49). En la línea 50 se hace una llamada a la función miembro `establecerNombreCurso` del objeto `miLibroCalificaciones` y se provee `nombreDelCurso` como argumento para la función. Cuando se hace la llamada a la función, el valor del argumento se copia al parámetro `nombre` (línea 14) de la función miembro `establecerNombreCurso`. Después, el valor del parámetro se asigna al dato miembro `nombreCurso` (línea 16). En la línea 52 se salta una línea en la salida; después en la línea 53 se hace una llamada a la función `mostrarMensaje` del objeto `miLibroCalificaciones` para mostrar en pantalla el mensaje de bienvenida, que contiene el nombre del curso.

Ingeniería de software mediante las funciones establecer y obtener

Los datos miembros `private` de una clase pueden manipularse *sólo* mediante las funciones miembro de esa clase (y por los "amigos" de la clase, como veremos en el capítulo 9). Por lo tanto, un **cliente de un objeto** (es decir, cualquier instrucción que llame a las funciones miembro del objeto desde su *exterior*) llama a las funciones miembro `public` de la clase para solicitar los servicios de la clase para objetos específicos de ésta. Esto explica por qué las instrucciones en la función `main` llaman a las funciones miembro `establecerNombreCurso`, `obtenerNombreCurso` y `mostrarMensaje` en un objeto `LibroCalificaciones`. A menudo, las clases proporcionan funciones miembro `public` para permitir a los clientes de la clase ***establecer*** (es decir, asignar valores a) u ***obtener*** (es decir, obtener los valores de) datos miembros `private`. Los nombres de estas funciones miembro no necesitan empezar con `establecer` u `obtener`, pero esta convención de nomenclatura es común. En este ejemplo, la función miembro que *establece* el dato miembro `nombreCurso` se llama `establecerNombreCurso`, y la función miembro que *obtiene* el valor del miembro de datos `nombreCurso` se llama `obtenerNombreCurso`. Las funciones *establecer* se conocen también como **mutadores** (porque mutan, o modifican, valores) y las funciones *obtener* se conocen también como **accesores** (porque acceden a los valores).

Recuerde que al declarar datos miembros con el especificador de acceso `private` se cumple con la ocultación de datos. Al proporcionar funciones *establecer* y *obtener* `public`, permitimos que los clientes de una clase accedan a los datos ocultos, pero sólo en forma *indirecta*. El cliente sabe que está intentando modificar u obtener los datos de un objeto, pero *no* sabe *cómo* el objeto lleva a cabo estas operaciones. En algunos casos, una clase puede representar *internamente* una pieza de datos de cierta forma, pero puede exponer esos datos a los clientes de una forma distinta. Por ejemplo, suponga que una clase `Reloj` representa la hora del día como un miembro de datos `private int` llamado `hora`, que almacena el número de segundos transcurridos desde media noche. Sin embargo, cuando un cliente llama a la función miembro `obtenerHora` de un objeto `Reloj`, el objeto podría devolver la hora con horas, minutos y segundos en un objeto `string`, en el formato `"HH:MM:SS"`. De manera similar, suponga que la clase `Reloj` cuenta con una función *establecer* llamada `establecerHora`, que recibe un parámetro `string` en el formato `"HH:MM:SS"`. Mediante el uso de las herramientas de la clase `string` (que puede ver en el capítulo 21, el cual se encuentra en inglés en el sitio web), la función `establecerHora` podría convertir este objeto `string` en un número de segundos, que la función almacena en su dato miembro `private`. La función *establecer* también podría verificar que el valor que recibe represente una hora válida (por ejemplo, `"12:30:45"` es válida, pero `"42:85:70"` no). Las funciones *establecer* y *obtener* permiten que un cliente interactúe con un objeto, pero los datos `private` del objeto permanecen *encapsulados* (ocultos) de una manera segura dentro del mismo objeto.

Las funciones *establecer* y *obtener* de una clase también deben utilizarlas otras funciones miembro *dentro* de la clase, para manipular los datos `private` de ésta, aunque estas funciones miembro *pueden* acceder a los datos `private` directamente. En la figura 3.5, las funciones miembro `establecerNombreCurso` y `obtenerNombreCurso` son funciones miembro `public`, por lo que están accesibles para los clientes de la clase, así como para la misma clase. La función miembro `mostrarMensaje` llama a la función miembro `obtenerNombreCurso` para obtener el valor del dato miembro `nombreCurso` y mostrarlo en pantalla, aun y cuando `mostrarMensaje` puede acceder directamente a `nombreCurso`; al acceder a un dato miembro a través de su función *obtener* se crea una clase más robusta y mejor (es decir, una clase que sea fácil de mantener y menos propensa a dejar de trabajar). Si decidimos cambiar el dato miembro `nombreCurso` en cierta forma, la definición de `mostrarMensaje` *no* requerirá modificación; sólo los cuerpos de las funciones *establecer* y *obtener*, que manipulan directamente al dato miembro, tendrán que cambiar. Por ejemplo, suponga que decidimos representar el nombre del curso como dos datos miembro separados: `nombreCurso` (por ejemplo, `"CS101"`) y `tituloCurso` (por ejemplo, `"Introduccion a la programacion en C++"`). La función miembro `mostrarMensaje` puede aún emitir una sola llamada a la función miembro `obtenerNombreCurso` para obtener el nombre completo del curso y mostrarlo como parte del mensaje de bienvenida. En este caso, `obtenerNombreCurso` necesitaría crear y devolver un objeto `string` que contenga el `nombreCurso` seguido del `tituloCurso`. La función miembro `mostrarMensaje` seguiría mostrando el título completo del curso "`CS101 Introduccion a la programacion`

en C++". Los beneficios de llamar a una función *establecer* desde otra función miembro de la clase se volverán más claros cuando hablemos sobre la validación en la sección 3.8.

Buena práctica de programación 3.1

Trate siempre de localizar *los efectos de las modificaciones a los datos miembros de una clase, utilizando y manipulando los datos miembros a través de sus correspondientes funciones* obtener *y* establecer.

Observación de Ingeniería de Software 3.1

Escriba programas que sean comprensibles y fáciles de mantener. El cambio es la regla, en vez de la excepción. Debemos anticipar que nuestro código será modificado en el futuro.

Diagrama de clases de UML para la clase `LibroCalificaciones` *con un dato miembro, y funciones* establecer *y* obtener

La figura 3.6 contiene un diagrama de clases de UML actualizado para la versión de la clase `LibroCalificaciones` de la figura 3.5. Este diagrama modela el miembro de datos `nombreCurso` de la clase `LibroCalificaciones` como un atributo en el compartimiento intermedio. UML representa a los datos miembros como atributos, listando el nombre del atributo, seguido de dos puntos y del tipo del atributo. El tipo de UML del atributo `nombreCurso` es `String`, que corresponde al tipo `string` en C++. El miembro de datos `nombreCurso` es `private` en C++, por lo que el diagrama de clases lista un *signo menos (-)* en frente del nombre del atributo correspondiente. La clase `LibroCalificaciones` contiene tres funciones miembro `public`, por lo que el diagrama de clases lista tres operaciones en el tercer compartimiento. La operación `establecerNombreCurso` tiene un parámetro `String` llamado `nombre`. UML indica el *tipo de valor de retorno* de una operación colocando dos puntos y el tipo de valor de retorno después de los paréntesis que le siguen al nombre de la operación. La función miembro `obtenerNombreCurso` de la clase `LibroCalificaciones` tiene un tipo de valor de retorno `string` en C++, por lo que el diagrama de clases muestra un tipo de valor de retorno `String` en UML. Las operaciones `establecerNombreCurso` y `mostrarMensaje` no devuelven valores (es decir, devuelven `void` en C++), por lo que el diagrama de clases de UML no especifica un tipo de valor de retorno después de los paréntesis de estas operaciones.

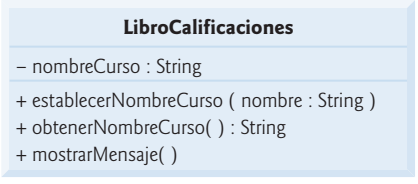


Fig. 3.6 | Diagrama de clases de UML para la clase `LibroCalificaciones`, con un atributo privado `nombreCurso` y operaciones públicas `establecerNombreCurso`, `obtenerNombreCurso` y `mostrarMensaje`.

3.5 Inicialización de objetos mediante constructores

Como mencionamos en la sección 3.4, cuando se crea un objeto de la clase `LibroCalificaciones` (figura 3.5), su miembro de datos `nombreCurso` se inicializa con la cadena vacía de manera predeterminada. ¿Qué pasa si usted desea proporcionar el nombre de un curso a la hora de *crear* un objeto `LibroCalificaciones`? Cada clase que usted declare puede proporcionar uno o más **constructores**, los cuales pueden utilizarse para inicializar un objeto de una clase al momento de crear ese objeto. Un constructor es una función miembro especial que debe definirse con el *mismo nombre que el de la clase*, de manera

que el compilador pueda diferenciarlo de las demás funciones miembro de la clase. Una importante diferencia entre los constructores y las otras funciones es que *los primeros no pueden devolver valores*, por lo cual *no pueden* especificar un tipo de valor de retorno (ni siquiera `void`). Normalmente los constructores son declarados `public`. En los primeros capítulos, por lo general nuestras clases tendrán un constructor; en capítulos posteriores veremos cómo crear clases con más de un constructor, mediante la técnica de *sobrecarga de funciones*, que presentaremos en la sección 6.18.

C++ llama de manera automática a un constructor para cada objeto que se crea, lo cual ayuda a asegurar que cada objeto se inicialice antes de utilizarlo en un programa. La llamada al constructor ocurre cuando se crea el objeto. Si una clase no incluye constructores en forma *explícita*, el compilador proporciona un **constructor predeterminado** *sin* parámetros. Por ejemplo, cuando en la línea 41 de la figura 3.5 se crea un objeto `LibroCalificaciones`, se hace una llamada al constructor predeterminado. Este constructor proporcionado por el compilador crea un objeto `LibroCalificaciones` sin proporcionar ningún valor inicial para los datos miembros de tipos fundamentales del objeto. Para los datos miembros que son objetos de otras clases, el constructor predeterminado llama de manera implícita al constructor predeterminado de cada dato miembro, para asegurar que ese dato miembro se inicialice en forma apropiada. Ésta es la razón por la cual el dato miembro `string` llamado `nombreCurso` (en la figura 3.5) se inicializó con la cadena vacía; el constructor predeterminado para la clase `string` asigna al valor del objeto `string` a la cadena vacía.

En el ejemplo de la figura 3.7, especificamos el nombre de un curso para un objeto `LibroCalificaciones` cuando se crea el objeto (línea 47). En este caso, el argumento `"CS101 Introduccion a la programacion en C++"` se pasa al constructor del objeto `LibroCalificaciones` (líneas 14 a 18) y se utiliza para inicializar el `nombreCurso`. En la figura 3.7 se define una clase `LibroCalificaciones` modificada, la cual contiene un constructor con un parámetro `string` que recibe el nombre inicial del curso.

```
1  // Fig. 3.7: fig03_07.cpp
2  // Creación de instancias de varios objetos de la clase LibroCalificaciones y uso
3  // de su constructor para especificar el nombre del curso
4  // cuando se crea cada objeto LibroCalificaciones.
5  #include <iostream>
6  #include <string> // el programa usa la clase string estándar de C++
7  using namespace std;
8
9  // definición de la clase LibroCalificaciones
10 class LibroCalificaciones
11 {
12 public:
13    // el constructor inicializa a nombreCurso con la cadena que se suministra
      como argumento
14    explicit LibroCalificaciones( string nombre )
15       : nombreCurso ( nombre ) // inicializador de miembro para inicializar
         nombreCurso
16    {
17       // cuerpo vacío
18    } // fin del constructor de LibroCalificaciones
19
20    // función para establecer el nombre del curso
21    void establecerNombreCurso( string nombre )
22    {
23       nombreCurso = nombre; // almacena el nombre del curso en el objeto
24    } // fin de la función establecerNombreCurso
```

Fig. 3.7 | Creación de instancias de varios objetos de la clase `LibroCalificaciones` y uso de su constructor para especificar el nombre del curso cuando se crea cada objeto `LibroCalificaciones` (parte 1 de 2).

```
25
26    // función para obtener el nombre del curso
27    string obtenerNombreCurso() const
28    {
29       return nombreCurso; // devuelve el nombreCurso del objeto
30    } // fin de la función obtenerNombreCurso
31
32    // muestra un mensaje de bienvenida para el usuario de LibroCalificaciones
33    void mostrarMensaje() const
34    {
35       // llama a obtenerNombreCurso para obtener el nombreCurso
36       cout << "Bienvenido al libro de calificaciones para\n"
         << obtenerNombreCurso()
37          << "!" << endl;
38    } // fin de la función mostrarMensaje
39 private:
40    string nombreCurso; // nombre del curso para este LibroCalificaciones
41 }; // fin de la clase LibroCalificaciones
42
43 // la función main empieza la ejecución del programa
44 int main()
45 {
46    // crea dos objetos LibroCalificaciones
47    LibroCalificaciones libroCalificaciones1( "CS101 Introduccion a la
      programacion en C++" );
48    LibroCalificaciones libroCalificaciones2( "CS102 Estructuras de datos en
      C++" );
49
50    // muestra el valor inicial de nombreCurso para cada LibroCalificaciones
51    cout << "libroCalificaciones1 se creo para el curso: "
      << libroCalificaciones1.obtenerNombreCurso()
52       << "\nlibroCalificaciones2 se creo para el curso: "
         << libroCalificaciones2.obtenerNombreCurso()
53       << endl;
54 } // fin de main
```

```
libroCalificaciones1 se creo para el curso: CS101 Introduccion a la programacion en
C++ libroCalificaciones2 se creo para el curso: CS102 Estructuras de datos en C++
```

Fig. 3.7 | Creación de instancias de varios objetos de la clase `LibroCalificaciones` y uso de su constructor para especificar el nombre del curso cuando se crea cada objeto `LibroCalificaciones` (parte 2 de 2).

Definición de un constructor

En las líneas 14 a 18 de la figura 3.7 se define un constructor para la clase `LibroCalificaciones`. El constructor tiene el *mismo* nombre que su clase, `LibroCalificaciones`. Un constructor especifica en su lista de parámetros los datos que requiere para realizar su tarea. Al crear un nuevo objeto, el programador coloca estos datos en los paréntesis que van después del nombre del objeto (como hicimos en las líneas 47 y 48). En la línea 14 se indica que el constructor de la clase `LibroCalificaciones` tiene un parámetro `string` llamado `nombre`. Declaramos este constructor como **`explicit`** debido a que recibe un *solo* parámetro: esto es importante por razones sutiles que conocerá en la sección 10.13. Por ahora, sólo declara *todos* los constructores de un solo parámetro como `explicit`. En la línea 14 *no* se especifica un tipo de valor de retorno, ya que los constructores *no pueden* devolver valores (ni siquiera `void`). Además, los constructores no pueden declararse como `const` (ya que al inicializar un objeto, éste se modifica).

El constructor usa una **lista de inicializadores de miembros** (línea 15) para inicializar el dato miembro `nombreCurso` con el valor del parámetro `nombre` del constructor. Los *inicializadores de miembros* aparecen entre la lista de parámetros de un constructor y la llave izquierda que comienza el cuerpo del constructor. La lista de inicializadores de miembros se separa de la lista de parámetros con un signo de *dos puntos (:)*. Un inicializador de miembro consiste en el *nombre de la variable* de un dato miembro seguido de paréntesis que contienen el *valor inicial* del miembro. En este ejemplo, `nombreCurso` se inicializa con el valor del parámetro `nombre`. Si una clase contiene más de un miembro de datos, el inicializador de cada miembro de datos se separa del siguiente con una coma. La lista de inicializadores de miembros se ejecuta *antes* de que se ejecute el cuerpo del constructor. Podemos realizar la inicialización en el cuerpo del constructor, pero más adelante aprenderá en el libro que es más eficiente hacerlo con inicializadores miembro; además, algunos tipos de datos miembros deben inicializarse de esta forma.

Observe que tanto el constructor (línea 14) como la función `establecerNombreCurso` (línea 21) utilizan un parámetro llamado `nombre`. Puede usar los *mismos* nombres de parámetros en *distintas* funciones, ya que los parámetros son *locales* para cada función; *no* interfieren unos con otros.

Prueba de la clase `LibroCalificaciones`

En las líneas 44 a 54 de la figura 3.7 se define la función `main` que prueba la clase `LibroCalificaciones` y demuestra cómo inicializar objetos `LibroCalificaciones` mediante el uso de un constructor. En la línea 47, se crea y se inicializa un objeto `LibroCalificaciones` llamado `libroCalificaciones1`. Cuando se ejecuta esta línea, se hace una llamada al constructor de `LibroCalificaciones` (líneas 14 a 18) con el argumento `"CS101 Introduccion a la programacion en C++"` para inicializar el nombre del curso de `libroCalificaciones1`. En la línea 48 se repite este proceso para el objeto `LibroCalificaciones` llamado `libroCalificaciones2`, pero esta vez se pasa el argumento `"CS102 Estructuras de datos en C++"` para inicializar el nombre del curso de `libroCalificaciones2`. En las líneas 51 y 52 se utiliza la función miembro `obtenerNombreCurso` de cada objeto para obtener los nombres de los cursos y mostrar que, sin duda, se inicializaron al momento de crear los objetos. La salida confirma que cada objeto `LibroCalificaciones` mantiene su *propia* copia del miembro de datos `nombreCurso`.

Formas de proporcionar un constructor predeterminado para una clase

Cualquier constructor que no recibe argumentos se llama constructor predeterminado. Una clase puede recibir un constructor predeterminado en una de varias formas:

1. El compilador crea de manera *implícita* un constructor predeterminado en cada clase que *no* tenga constructores definidos por el usuario. El constructor predeterminado *no* inicializa los datos miembros de la clase, pero *llama* al constructor predeterminado para cada dato miembro que sea un objeto de otra clase. Una variable sin inicializar contiene un valor indefinido ("basura").
2. Usted como programador define en forma *explícita* un constructor que no recibe argumentos. Dicho constructor llamará al constructor predeterminado para cada dato miembro que sea un objeto de otra clase y realizará la inicialización adicional especificada por usted.
3. *Si define constructores con argumentos, C++ no creará de manera implícita un constructor predeterminado para esa clase.* Más tarde le mostraremos que C++11 le permite forzar al compilador a crear el constructor predeterminado, incluso aunque no haya definido constructores no predeterminados.

Para cada versión de la clase `LibroCalificaciones` en las figuras 3.1, 3.3 y 3.5, el compilador definió de manera *implícita* un constructor predeterminado.

Tip para prevenir errores 3.2

A menos que no sea necesario inicializar los datos miembros de su clase (casi nunca), debe proporcionar constructores para asegurar que los datos miembros de su clase se inicialicen con valores significativos al momento de crear cada nuevo objeto de su clase.

Observación de Ingeniería de Software 3.2

Los datos miembros se pueden inicializar en un constructor, o sus valores pueden establecerse *más adelante, después de crear el objeto. Sin embargo, es una buena práctica de ingeniería de software asegurarse que un objeto esté inicializado por completo antes de que el código cliente invoque las funciones miembro de ese objeto. En general, no debemos depender del código cliente para asegurar que un objeto se inicialice de manera apropiada.*

Agregar el constructor al diagrama de clases de UML de la clase `LibroCalificaciones`

El diagrama de clases de UML de la figura 3.8 modela la clase `LibroCalificaciones` de la figura 3.7, la cual tiene un constructor con un parámetro `nombre` de tipo `string` (representado por el tipo `String` en UML). Al igual que las operaciones, el UML modela a los constructores en el tercer compartimiento de una clase en un diagrama de clases. Para diferenciar a un constructor de las operaciones de la clase, UML coloca la palabra "constructor" entre los signos « y » antes del nombre del constructor. Es costumbre enlistar el constructor de la clase *antes* de todas las operaciones en el tercer compartimiento.

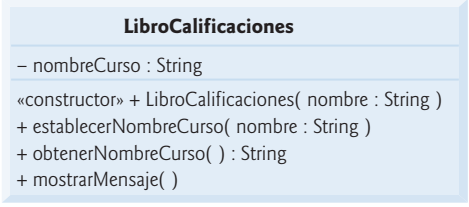


Fig. 3.8 | Diagrama de clases de UML, el cual indica que la clase `LibroCalificaciones` tiene un constructor con un parámetro llamado `nombre` de tipo `String` de UML.

3.6 Colocar una clase en un archivo separado para fines de reutilización

Uno de los beneficios de crear definiciones de clases es que, cuando se empaquetan en forma apropiada, nuestras clases pueden ser *reutilizadas* por otros programadores. Por ejemplo, podemos *reutilizar* el tipo `string` de la Biblioteca estándar de C++ en cualquier programa en C++ al incluir el encabezado `<string>` (y, como veremos, al poder enlazarnos con el código objeto de la biblioteca).

Los programadores que deseen utilizar nuestra clase `LibroCalificaciones` no pueden simplemente incluir el archivo de la figura 3.7 en otro programa. Como aprendió en el capítulo 2, la función `main` empieza la ejecución de todo programa, y cada programa debe tener *sólo y solamente una* función `main`. Si otros programadores incluyen el código de la figura 3.7, recibirán "equipaje" adicional (nuestra función `main`) y sus programas tendrán entonces dos funciones `main`. Si intentamos compilar un programa con dos funciones `main` se producirá un error. Por lo tanto, al colocar `main` en el mismo archivo con una definición de clase, *evitamos que esa clase pueda ser reutilizada* por otros programas. En esta sección demostraremos cómo hacer la clase `LibroCalificaciones` reutilizable, al *separarla* de la función `main` *y colocarla en otro archivo*.

Encabezados

Cada uno de los ejemplos anteriores en el capítulo consiste de un solo archivo `.cpp`, al cual se le conoce también como **archivo de código fuente**, y contiene la definición de la clase `LibroCalificaciones` y una función `main`. Al construir un programa en C++ orientado a objetos, es costumbre definir el código fuente *reutilizable* (como una clase) en un archivo que, por convención, tiene la extensión `.h`; a éste se le conoce como **encabezado**. Los programas utilizan las directivas del preprocesador `#include` para incluir encabezados y aprovechar los componentes de software reutilizables, como el tipo `string` que se

proporciona en la Biblioteca Estándar de C++, y los tipos definidos por el usuario como la clase `LibroCalificaciones`.

En nuestro siguiente ejemplo, separamos el código de la figura 3.7 en dos archivos: `LibroCalificaciones.h` (figura 3.9) y `fig03_10.cpp` (figura 3.10). Cuando analice el encabezado de la figura 3.9, observe que sólo contiene la definición de la clase `LibroCalificacioneS` (líneas 7 a 38) y los encabezados de los que depende la clase. La función `main` que *utiliza* a la clase `LibroCalificaciones` se define en el archivo de código fuente `fig03_10.cpp` (figura 3.10), en las líneas 8 a 18. Para ayudarlo a prepararse para los programas más extensos que encontrará más adelante en este libro y en la industria, a menudo utilizamos un archivo de código fuente separado que contiene la función `main` para probar nuestras clases (a éste se le conoce como **programa controlador**). Pronto aprenderá cómo un archivo de código fuente con `main` puede utilizar la definición de una clase que se encuentra en un encabezado para crear objetos de esa clase.

```
1  // Fig. 3.9: LibroCalificaciones.h
2  // Definición de la clase LibroCalificaciones en un archivo separado de main.
3  #include <iostream>
4  #include <string> // la clase LibroCalificaciones utiliza la clase string
                     estándar de C++
5
6  // definición de la clase LibroCalificaciones
7  class LibroCalificaciones
8  {
9  public:
10   // el constructor inicializa nombreCurso con la cadena que se suministra como
        argumento
11   explicit LibroCalificaciones( std::string nombre )
12      : nombreCurso( nombre ) // inicializador de miembro para inicializar
                                nombreCurso
13   {
14      // cuerpo vacío
15   } // fin del constructor de LibroCalificaciones
16
17   // función para establecer el nombre del curso
18   void establecerNombreCurso( std::string nombre )
19   {
20      nombreCurso = nombre; // almacena el nombre del curso en el objeto
21   } // fin de la función establecerNombreCurso
22
23   // función para obtener el nombre del curso
24   std::string obtenerNombreCurso() const
25   {
26      return nombreCurso; // devuelve el nombreCurso del objeto
27   } // fin de la función obtenerNombreCurso
28
29   // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
30   void mostrarMensaje() const
31   {
32      // llama a obtenerNombreCurso para obtener el nombreCurso
33      std::cout << "Bienvenido al libro de calificaciones para\n"
                  << obtenerNombreCurso()
34         << "!" << std::endl;
35   } // fin de la función mostrarMensaje
36 private:
37   std::string nombreCurso; // nombre del curso para este LibroCalificaciones
38 }; // fin de la clase LibroCalificaciones
```

Fig. 3.9 | Definición de la clase `LibroCalificaciones` en un archivo separado de `main`.

```
1  // Fig. 3.10: fig03_10.cpp
2  // Inclusión de la clase LibroCalificaciones del archivo LibroCalificaciones.h
      para usarla en main.
3  #include <iostream>
4  #include "LibroCalificaciones.h" // incluye la definición de la clase
                                    LibroCalificaciones
5  using namespace std;
6
7  // la función main empieza la ejecución del programa
8  int main()
9  {
10   // crea dos objetos LibroCalificaciones
11   LibroCalificaciones libroCalificaciones1( "CS101 Introduccion a la
                                                programacion en C++" );
12   LibroCalificaciones libroCalificaciones2( "CS102 Estructuras de datos en
                                                C++" );
13
14   // muestra el valor inicial de nombreCurso para cada LibroCalificaciones
15   cout << "libroCalificaciones1 creado para el curso: "
     << libroCalificaciones1.obtenerNombreCurso()
16      << "\nlibroCalificaciones2 creado para el curso: " << libroCalificaciones2.
                                                             obtenerNombreCurso()
17      << endl;
18 } // fin de main
```

```
libroCalificaciones1 creado para el curso: CS101 Introduccion a la programacion en
C++ libroCalificaciones2 creado para el curso: CS102 Estructuras de datos en C++
```

Fig. 3.10 | Inclusión de la clase `LibroCalificaciones` del archivo `LibroCalificaciones.h` para usarla en `main`.

Usar `std::` con los componentes de la biblioteca estándar en los encabezados

En el encabezado (figura 3.9) utilizamos `std::` al referirnos a un objeto `string` (líneas 11, 18, 24 y 37), `cout` (línea 33) y `endl` (línea 34). Por razones sutiles que explicaremos en un capítulo posterior, los encabezados *nunca* deben contener directivas `using` o declaraciones `using` (sección 2.7).

Incluir un archivo de encabezado que contiene una clase definida por el usuario

Un archivo de encabezado tal como `LibroCalificaciones.h` (figura 3.9) no puede usarse como un programa completo, ya que no contiene una función `main`. Para probar la clase `LibroCalificaciones` (definida en la figura 3.9), debe escribir un archivo de código fuente separado que contenga una función `main` (como la figura 3.10), la cual debe instanciar y utilizar objetos de la clase.

El compilador no sabe qué es un `LibroCalificaciones` ya que es un tipo definido por el usuario. De hecho, el compilador ni siquiera conoce las clases de la Biblioteca estándar de C++. Para ayudarlo a comprender cómo usar una clase, debemos proporcionar en forma explícita al compilador la definición de la clase; ésta es la razón por la que, para que un programa pueda usar un tipo `string`, debe incluir el encabezado `<string>`. Esto permite al compilador determinar la cantidad de memoria que debe reservar para cada objeto de la clase, y asegurar que un programa llame a las funciones miembro de la clase `string` de una forma correcta.

Para crear los objetos `LibroCalificaciones` llamados `libroCalificaciones1` y `libroCalificaciones2` en las líneas 11 y 12 de la figura 3.10, el compilador debe conocer el *tamaño* de un objeto `LibroCalificaciones`. Aunque en concepto los objetos contienen datos miembros y funciones miembro, los objetos de C++ *sólo* contienen datos. El compilador sólo crea *una* copia de las funciones miembro de la clase y *comparte* esa copia entre todos los objetos de la clase. Desde luego que cada objeto necesita sus propios datos

miembros, ya que su contenido puede variar de un objeto a otro (como dos objetos `CuentaBanco` distintos, que tienen dos saldos distintos). Sin embargo, el código de la función miembro *no se puede modificar*, por lo que puede compartirse entre todos los objetos de la clase. Por lo tanto, el tamaño de un objeto depende de la cantidad de memoria requerida para almacenar los datos miembros de la clase. Al incluir a `LibroCalificaciones.h` en la línea 4, proporcionamos acceso al compilador para que utilice la información que necesita (figura 3.9, línea 37) para determinar el tamaño de un objeto `LibroCalificaciones` y determinar si los objetos de la clase se utilizan correctamente (en las líneas 11 a 12 y 15 a 16 de la figura 3.10).

En la línea 4 se indica al preprocesador de C++ que reemplace la directiva con una copia del contenido de `LibroCalificaciones.h` (es decir, la definición de la clase `LibroCalificaciones`) *antes* de compilar el programa. Cuando se compila el archivo de código fuente `fig03_10.cpp`, ahora contiene la definición de la clase `LibroCalificaciones` (debido a la instrucción `#include`), y el compilador es capaz de crear objetos `LibroCalificaciones` y revisar que se hagan llamadas a sus funciones miembro en forma adecuada. Ahora que la definición de la clase está en un encabezado (sin una función `main`), podemos incluir ese encabezado en *cualquier* programa que necesite reutilizar nuestra clase `LibroCalificaciones`.

Cómo se localizan los encabezados

Observe que el nombre del encabezado `LibroCalificaciones.h` en la línea 4 de la figura 3.10 se encierra entre comillas (`" "`) en vez de usar los signos < y >. Por lo general, los archivos de código fuente de un programa y los encabezados definidos por el usuario se colocan en el *mismo* directorio. Cuando el preprocesador encuentra el nombre de un encabezado entre comillas, intenta localizar el encabezado en el mismo directorio que el archivo en el que aparece la directiva `#include`. Si el preprocesador no puede encontrar el encabezado en ese directorio, lo busca en la(s) misma(s) ubicación(es) que los encabezados de la Biblioteca Estándar de C++. Cuando el preprocesador encuentra el nombre de un encabezado entre los signos < y > (como `<iostream>`), asume que el encabezado forma parte de la Biblioteca Estándar de C++ y *no* busca en el directorio del programa que se está procesando.

Tip para prevenir errores 3.3

Para asegurar que el preprocesador pueda localizar los encabezados en forma correcta, en las directivas del preprocesador `#include` se deben colocar los nombres de los encabezados definidos por el usuario entre comillas (como `"LibroCalificaciones.h"`), y se deben colocar los nombres de los archivos de encabezado de la Biblioteca Estándar de C++ entre los signos < y > (como `<iostream>`).

Cuestiones adicionales sobre Ingeniería de Software

Ahora que la clase `LibroCalificaciones` está definida en un archivo de encabezado, puede *reutilizarse*. Por desgracia, al colocar la definición de una clase en un archivo de encabezado como en la figura 3.9, se sigue *revelando toda la implementación de la clase a los clientes de la misma*; `LibroCalificaciones.h` es simplemente un archivo de texto que cualquiera puede abrir y leer. La sabiduría de la Ingeniería de software convencional nos dice que para usar un objeto de la clase, el código cliente necesita saber sólo qué funciones miembro debe llamar, qué argumentos debe proporcionar a cada función miembro y qué tipo de valor de retorno debe esperar de cada función miembro. *El código cliente no necesita saber cómo se implementan esas funciones.*

Si el código cliente *sabe* cómo se implementa una clase, el programador podría escribir código cliente basado en los detalles de implementación de la clase. Lo ideal sería que, si cambia la implementación, los clientes de la clase no tengan que cambiar. *Al ocultar los detalles de implementación de la clase, facilitamos la tarea de cambiar la implementación de la clase al mismo tiempo que minimizamos (y con suerte, eliminamos) los cambios al código cliente.*

En la sección 3.7 le mostraremos cómo descomponer la clase `LibroCalificaciones` en dos archivos, de manera que:

1. la clase sea *reutilizable*,
2. los clientes de la clase sepan qué funciones miembro proporciona la clase, cómo llamarlas y qué tipo de valores de retorno esperar, y
3. los clientes *no* sepan cómo se implementan las funciones miembro de la clase.

3.7 Separar la interfaz de la implementación

En la sección anterior le mostramos cómo fomentar la reutilización de software al separar la definición de la clase del código cliente (por ejemplo, la función `main`) que utiliza esa clase. Ahora presentaremos otro principio fundamental de la buena Ingeniería de software: **separar la interfaz de la implementación**.

La interfaz de una clase

Las **interfaces** definen y estandarizan las formas en las que las personas y los sistemas interactúan entre sí. Por ejemplo, los controles de un radio sirven como una interfaz entre los usuarios del radio y sus componentes internos. Los controles permiten a los usuarios realizar un conjunto limitado de operaciones (como cambiar la estación, ajustar el volumen y elegir entre una estación en AM o una en FM). Varias radios pueden implementar estas operaciones de manera distinta; algunos proporcionan botones, otros perillas y algunas incluso soportan comandos de voz. La interfaz especifica *qué* operaciones permite realizar un radio a los usuarios, pero no especifica *cómo* se implementan estas operaciones en su interior.

De manera similar, la **interfaz de una clase** describe *qué* servicios pueden usar los clientes de la clase y cómo *solicitar* esos servicios, pero no *cómo* lleva a cabo la clase esos servicios. La interfaz `public` de una clase consiste en las funciones miembro `public` de la clase (también conocidas como **servicios públicos**). Por ejemplo, la interfaz de la clase `LibroCalificaciones` (figura 3.9) contiene un constructor y las funciones miembro `establecerNombreCurso`, `obtenerNombreCurso` y `mostrarMensaje`. Los clientes de `LibroCalificaciones` (por ejemplo, `main` en la figura 3.10) *utilizan* estas funciones para solicitar los servicios de la clase. Como pronto veremos, podemos especificar la interfaz de una clase al escribir una definición de clase que *sólo* enliste los nombres de las funciones miembro, los tipos de los valores de retorno y los tipos de los parámetros.

Separar la interfaz de la implementación

En nuestros ejemplos anteriores, la definición de cada clase contenía las definiciones completas de las funciones miembro `public` de la clase y las declaraciones de sus datos miembros `private`. Sin embargo, una mejor ingeniería de software es definir las funciones miembro *fuera* de la definición de la clase, de manera que sus detalles de implementación se puedan *ocultar* del código cliente. Esta práctica *asegura* que los programadores no escriban código cliente que dependa de los detalles de implementación de la clase.

El programa de las figuras 3.11 a 3.13 separa la interfaz de `LibroCalificaciones` de su implementación, para lo cual divide la definición de la clase de la figura 3.9 en dos archivos: el encabezado `LibroCalificaciones.h` (figura 3.11) en el que se define la clase `LibroCalificaciones`, y el archivo de código fuente `LibroCalificaciones.cpp` (figura 3.12) en el que se definen las funciones miembro de `LibroCalificaciones`. Por convención, las definiciones de las funciones miembro se colocan en un archivo de código fuente con el mismo nombre base (por ejemplo, `LibroCalificaciones`) que el encabezado de la clase, pero con una extensión de archivo `.cpp`. El archivo de código fuente `fig03_13.cpp` (figura 3.13) define la función `main` (el código cliente). El código y la salida de la figura 3.13 son idénticos a los de la figura 3.10. En la figura 3.14 se muestra cómo se compila este programa de tres archivos, desde las perspectivas del programador de la clase `LibroCalificaciones` y del programador del código cliente; explicaremos esta figura con detalle.

`LibroCalificaciones.h`: definición de la interfaz de una clase mediante prototipos de funciones

El archivo de encabezado `LibroCalificaciones.h` (figura 3.11) contiene otra versión de la definición de la clase `LibroCalificaciones` (líneas 8 a 17). Esta versión es similar a la de la figura 3.9, pero las definiciones de las funciones en la figura 3.9 se reemplazan aquí con **prototipos de funciones** (líneas 11 a 14) que *describen la interfaz `public` de la clase sin revelar las implementaciones de sus funciones miembro*. Un prototipo de función es una *declaración* de una función que indica al compilador el nombre de

la función, su tipo de valor de retorno y los tipos de sus parámetros. Además, el encabezado sigue especificando el dato miembro `private` de la clase (línea 16) también. De nuevo, el compilador *debe* conocer los datos miembros de la clase para determinar cuánta memoria debe reservar para cada objeto de la misma. Al incluir el encabezado `LibroCalificaciones.h` en el código cliente (línea 5 de la figura 3.13), el compilador obtiene la información que necesita para asegurar que el código cliente llame a las funciones miembro de la clase `LibroCalificaciones` en forma correcta.

```
1  // Fig. 3.11: LibroCalificaciones.h
2  // Definición de la clase LibroCalificaciones. Este archivo presenta la interfaz
3  // public de LibroCalificaciones sin revelar las implementaciones de sus funciones
4  // miembro, que están definidas en LibroCalificaciones.cpp.
5  #include <string> // la clase LibroCalificaciones utiliza la clase string
                     estándar de C++
6
7  // definición de la clase LibroCalificaciones
8  class LibroCalificaciones
9  {
10 public:
11    explicit LibroCalificaciones( std::string ); // constructor que inicializa a
                                                      nombreCurso
12    void establecerNombreCurso( std::string ); // establece el nombre del curso
13    std::obtenerNombreCurso() const; // obtiene el nombre del curso
14    void mostrarMensaje() const; // muestra un mensaje de bienvenida
15 private:
16    std::string nombreCurso; // nombre del curso para este LibroCalificaciones
17 }; // fin de la clase LibroCalificaciones
```

Fig. 3.11 | Definición de la clase `LibroCalificaciones` que contiene prototipos de funciones que especifican la interfaz de la clase.

El prototipo de función en la línea 11 (figura 3.11) indica que el constructor requiere un parámetro `string`. Recuerde que los constructores no tienen tipos de valores de retorno, por lo que no aparece ningún tipo de valor de retorno en el prototipo de la función. El prototipo de la función miembro `establecerNombreCurso` indica que requiere un parámetro `string` y no devuelve un valor (es decir, su tipo de valor de retorno es `void`). El prototipo de la función miembro `obtenerNombreCurso` indica que la función no requiere parámetros y devuelve un `string`. Por último, el prototipo de la función `mostrarMensaje` (línea 14) especifica que `mostrarMensaje` no requiere parámetros y no devuelve un valor. Estos prototipos de funciones son iguales que las primeras líneas de las correspondientes definiciones de funciones en la figura 3.9, sólo que los nombres de los parámetros (que son *opcionales* en los prototipos) no se incluyen y cada prototipo de función *debe* terminar con un punto y coma.

Buena práctica de programación 3.2

Aunque los nombres de los parámetros en los prototipos de funciones son opcionales (el compilador los ignora), muchos programadores utilizan estos nombres para fines de documentación.

`LibroCalificaciones.cpp`: definir las funciones miembro en un archivo de código fuente separado

El archivo de código fuente `LibroCalificaciones.cpp` (figura 3.12) *define* las funciones miembro de la clase `LibroCalificaciones`, que se *declararon* en las líneas 11 a 14 de la figura 3.11. Las definiciones aparecen en las líneas 9 a 33 y son casi idénticas a las definiciones de las funciones miembro en las líneas 11 a 35 de la figura 3.9. Cabe mencionar que la palabra clave `const` *debe* aparecer *tanto* en los prototipos

de función (figura 3.11, líneas 13 y 14) y las definiciones de las funciones `obtenerNombreCurso` y `mostrarMensaje` (líneas 22 y 28).

```
1  // Fig. 3.12: LibroCalificaciones.cpp
2  // Definiciones de las funciones miembro de LibroCalificaciones. Este archivo
      contiene
3  // implementaciones de las funciones miembro cuyo prototipo está en
      LibroCalificaciones.h.
4  #include <iostream>
5  #include "LibroCalificaciones.h" // incluye la definición de la clase
                                    LibroCalificaciones
6  using namespace std;
7
8  // el constructor inicializa nombreCurso con el objeto string suministrado como
      argumento
9  LibroCalificaciones::LibroCalificaciones( string nombre )
10    : nombreCurso( nombre ) // inicializador de miembro para inicializar nombreCurso
11 {
12    // cuerpo vacío
13 } // fin del constructor de LibroCalificaciones
14
15 // función para establecer el nombre del curso
16 void LibroCalificaciones::establecerNombreCurso( string nombre )
17 {
18    nombreCurso = nombre; // almacena el nombre del curso en el objeto
19 } // fin de la función establecerNombreCurso
20
21 // función para obtener el nombre del curso
22 string LibroCalificaciones::obtenerNombreCurso() const
23 {
24    return nombreCurso; // devuelve el nombreCurso del objeto
25 } // fin de la función obtenerNombreCurso
26
27 // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
28 void LibroCalificaciones::mostrarMensaje() const
29 {
30    // llama a obtenerNombreCurso para obtener el nombreCurso
31    cout << "Bienvenido al libro de calificaciones para\n" << obtenerNombreCurso()
32       << "!" << endl;
33 } // fin de la función mostrarMensaje
```

Fig. 3.12 | Las definiciones de las funciones miembro de `LibroCalificaciones` representan la implementación de la clase `LibroCalificaciones`.

Al nombre de cada función miembro (líneas 9, 16, 22 y 28) se le antepone el nombre de la clase y el símbolo `::`, que se conoce como el **operador de resolución de ámbito**. Esto "enlaza" a cada función miembro con la definición (ahora separada) de la clase `LibroCalificaciones` (figura 3.11), la cual declara las funciones miembro y los datos miembros. Sin "`LibroCalificaciones::`" antes de cada nombre de función, el compilador *no* las reconocería como funciones miembro de la clase `LibroCalificaciones`; las consideraría como funciones "libres" o "sueltas", al igual que `main`. A éstas también se les conoce como *funciones globales*. Dichas funciones no pueden acceder a los datos `private` de `LibroCalificaciones` ni llamar a las funciones miembro de la clase, sin especificar un objeto. Por lo tanto, el compilador *no* podría compilar estas funciones. Por ejemplo, en las líneas 18 y 24 en la figura 3.12 que acceden a la variable `nombreCurso` se producirían errores de compilación, ya que `nombreCurso` no está declarada como una variable local en cada función; el compilador no sabría que `nombreCurso` ya está declarada como dato miembro de la clase `LibroCalificaciones`.

Error común de programación 3.3

Al definir las funciones miembro de una clase fuera *de la misma, si se omite el nombre de la clase y el operador de resolución de ámbito (*`::`*) antes de los nombres de las funciones se producen errores de compilación.*

Para indicar que las funciones miembro en `LibroCalificaciones.cpp` forman parte de la clase `LibroCalificaciones`, debemos primero incluir el archivo de encabezado `LibroCalificaciones.h` (línea 5 de la figura 3.12), Esto nos permite acceder al nombre de la clase `LibroCalificaciones` en el archivo `LibroCalificaciones.cpp`. Al compilar `LibroCalificaciones.cpp`, el compilador utiliza la información en `LibroCalificaciones.h` para asegurar que

1. la primera línea de cada función miembro (líneas 9, 16, 22 y 28) coincida con su prototipo en el archivo `LibroCalificaciones.h`; por ejemplo, el compilador asegura que `obtenerNombreCurso` no acepte parámetros y devuelva un valor `string`, y que
2. cada función miembro sepa acerca de los datos miembros y otras funciones miembro de la clase; por ejemplo, en las líneas 18 y 24 se puede acceder a la variable `nombreCurso`, ya que está declarada en `LibroCalificaciones.h` como dato miembro de la clase `LibroCalificaciones`, y en la línea 31 se puede llamar a la función `obtenerNombreCurso`, ya que se declara como función miembro de la clase en `LibroCalificaciones.h` (y debido a que la llamada se conforma con el prototipo correspondiente).

Prueba de la clase `LibroCalificaciones`

En la figura 3.13 se realizan las mismas manipulaciones de objetos `LibroCalificaciones` que en la figura 3.10. Al separar la interfaz de `LibroCalificaciones` de la implementación de sus funciones miembro, *no* se ve afectada la forma en que este código cliente utiliza la clase. Sólo afecta la forma en que se compila y enlaza el programa, lo cual veremos en breve.

```
1  // Fig. 3.13: fig03_13.cpp
2  // Demostración de la clase LibroCalificaciones después de separar
3  // su interfaz de su implementación.
4  #include <iostream>
5  #include "LibroCalificaciones.h" // incluye la definición de la clase
                                    LibroCalificaciones GradeBook
6  using namespace std;
7
8  // la función main empieza la ejecución del programa
9  int main()
10 {
11    // crea dos objetos LibroCalificaciones
12    LibroCalificaciones libroCalificaciones1( "CS101 Introduccion a la
                                                 programacion en C++" );
13    LibroCalificaciones libroCalificaciones2( "CS102 Estructuras de datos en C++" );
14
15    // muestra el valor inicial de nombreCurso para cada LibroCalificaciones
16    cout << "libroCalificaciones1 creado para el curso: "
      << libroCalificaciones1.obtenerNombreCurso()
17       << "\nlibroCalificaciones2 creado para el curso: "
         << libroCalificaciones2.obtenerNombreCurso()
18       << endl;
19 } // fin de main
```

Fig. 3.13 | Demostración de la clase `LibroCalificaciones` después de separar la interfaz de su implementación (parte 1 de 2).

```
libroCalificaciones1 creado para el curso: CS101 Introduccion a la programacion en C++
libroCalificaciones2 creado para el curso: CS102 Estructuras de datos en C++
```

Fig. 3.13 | Demostración de la clase `LibroCalificaciones` después de separar la interfaz de su implementación (parte 2 de 2).

Al igual que en la figura 3.10, en la línea 5 de la figura 3.13 se incluye el archivo de encabezado `LibroCalificaciones.h`, de manera que el compilador pueda asegurar que los objetos `LibroCalificaciones` se creen y manipulen correctamente en el código cliente. Antes de ejecutar este programa, deben compilarse los archivos de código fuente de las figuras 3.12 y 3.13, y después se deben enlazar; es decir, las llamadas a las funciones miembro en el código cliente necesitan enlazarse con las implementaciones de las funciones miembro de la clase; un trabajo que realiza el enlazador.

El proceso de compilación y enlace

El diagrama de la figura 3.14 muestra el proceso de compilación y enlace que produce una aplicación `LibroCalificaciones` ejecutable, que los instructores pueden utilizar. Lo común es que un programador cree y compile la interfaz e implementación de una clase, y que otro programador implemente el código cliente para utilizar esa clase. Así, el diagrama muestra lo que requieren tanto el programador de la implementación de la clase, como el programador del código cliente. Las líneas punteadas en el diagrama muestran las piezas requeridas por el programador de la implementación de la clase, el programador del código cliente y el usuario de la aplicación `LibroCalificaciones`, respectivamente. [*Nota:* la figura 3.14 *no* es un diagrama de UML].

El programador de la implementación de una clase, responsable de crear una clase `LibroCalificaciones` reutilizable, crea el encabezado `LibroCalificaciones.h` y el archivo de código fuente `LibroCalificaciones.cpp` que incluye (mediante `#include`) el encabezado, y después compila el archivo de código fuente para crear el código objeto de `LibroCalificaciones`. Para ocultar los detalles de la implementación de las funciones miembro de `LibroCalificaciones`, el programador de la implementación de la clase proporciona al programador del código cliente el encabezado `LibroCalificaciones.h` (que especifica la interfaz y los datos miembros de la clase) y el código objeto de `LibroCalificaciones` (que contiene las instrucciones en lenguaje máquina que representan a las funciones miembro de `LibroCalificaciones`). El programador del código cliente *no* recibe `LibroCalificaciones.cpp`, por lo que desconoce cómo se implementan las funciones miembro de `LibroCalificaciones`.

El programador del código cliente sólo necesita conocer la interfaz de `LibroCalificaciones` para usar la clase, y debe tener la capacidad de enlazar su código objeto. Como la interfaz de la clase es parte de su definición en el encabezado `LibroCalificaciones.h`, el programador del código cliente debe tener acceso a este archivo e incluirlo (mediante `#include`) en el archivo de código fuente del cliente. Cuando se compila el código cliente, el compilador usa la definición de la clase en `LibroCalificaciones.h` para asegurar que la función `main` cree y manipule objetos de la clase `LibroCalificaciones` correctamente.

Para crear la aplicación `LibroCalificaciones` ejecutable, el último paso es enlazar

1. el código objeto para la función `main` (es decir, el código cliente),
2. el código objeto para las implementaciones de las funciones miembro de la clase `LibroCalificaciones` y
3. el código objeto de la Biblioteca estándar de C++ para las clases de C++ (como `string`) que utilicen tanto el programador de la implementación de la clase, como el programador del código cliente.

La salida del enlazador es la aplicación `LibroCalificaciones` *ejecutable*, que los instructores pueden utilizar para administrar las calificaciones de sus estudiantes. Por lo general los compiladores y los IDE invocan al enlazador por el programador después de compilar su código.

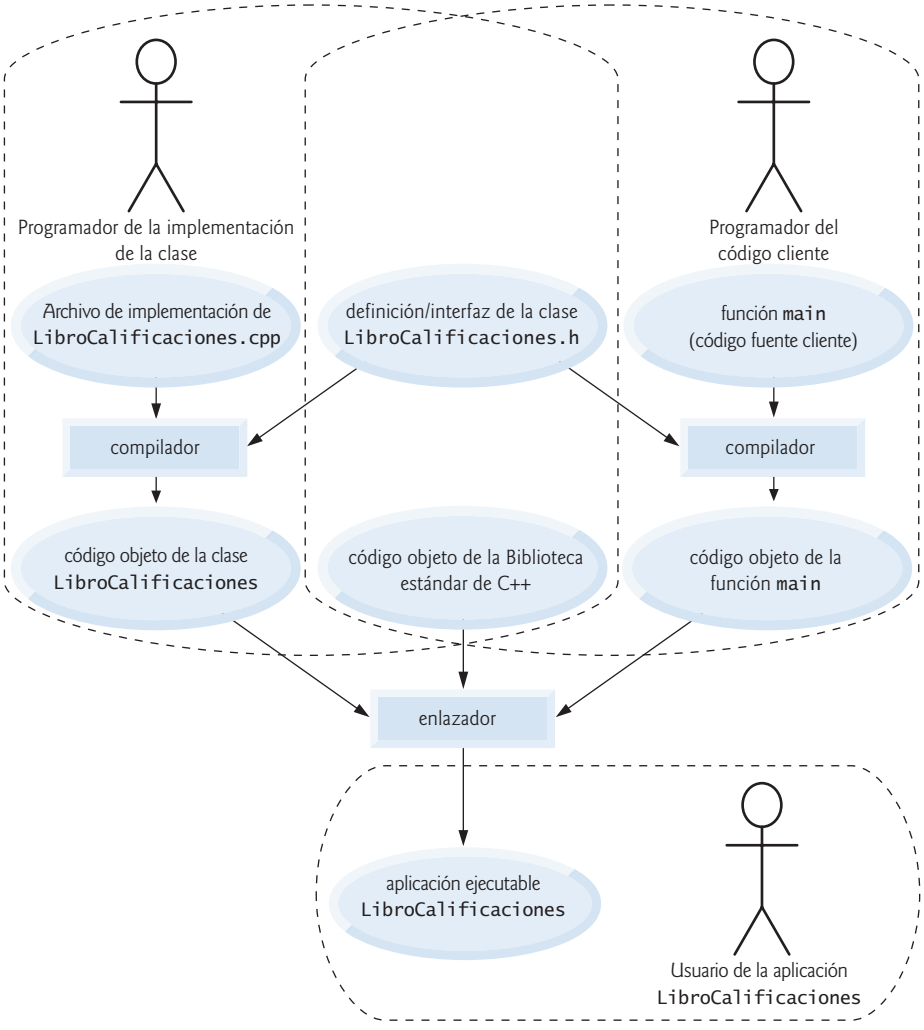


Fig. 3.14 | Proceso de compilación y enlace que produce una aplicación ejecutable.

Para obtener más información acerca de cómo compilar programas con varios archivos de código fuente, consulte la documentación de su compilador. En nuestro Centro de recursos de C++ en `www.deitel.com/cplusplus/` proporcionamos vínculos a varios compiladores de C++.

3.8 Validación de datos mediante funciones *establecer*

En la sección 3.4 presentamos funciones *establecer* para permitir a los clientes de una clase modificar el valor de un dato miembro `private`. En la figura 3.5, la clase `LibroCalificaciones` define la función miembro `establecerNombreCurso` tan sólo para asignar un valor recibido en su parámetro `nombre` al dato miembro `nombreCurso`. Esta función miembro no asegura que el nombre del curso se adhiera a algún formato específico, o que siga cualquier otra regla en relación con la apariencia que debe tener un nombre de curso "válido". Suponga que una universidad puede imprimir certificados de los estudiantes que contengan nombres de cursos con 25 caracteres o menos. Si la universidad utiliza un sistema que

contenga objetos `LibroCalificaciones` para generar los certificados, tal vez sea conveniente que la clase `LibroCalificaciones` asegure que su miembro de datos `nombreCurso` nunca contenga más de 25 caracteres. El programa de las figuras 3.15 a 3.17 mejora la función miembro `establecerNombreCurso` de la clase `LibroCalificaciones` para realizar esta **validación** (lo que también se conoce como **verificación de validez**).

Definición de la clase `LibroCalificaciones`

La definición de la clase `LibroCalificaciones` (figura 3.15) y por ende, su interfaz es idéntica a la de la figura 3.11. Como la interfaz permanece sin cambios, los clientes de esta clase no necesitan modificarse a la hora que se modifica la definición de la función `establecerNombreCurso`. Esto permite a los clientes aprovechar la clase `LibroCalificaciones` mejorada, con sólo enlazar el código cliente con el código objeto actualizado de `LibroCalificaciones`.

```
1  // Fig. 3.15: LibroCalificaciones.h
2  // Definición de la clase LibroCalificaciones presenta la interfaz public
3  // de la clase. Las definiciones de las funciones miembro aparecen en
      LibroCalificaciones.cpp.
4  #include <string> // el programa usa la clase string estándar de C++
5
6  // Definición de la clase LibroCalificaciones
7  class LibroCalificaciones
8  {
9  public:
10    explicit LibroCalificaciones( std::string ); // constructor que inicializa
                                                       nombreCurso
11    void establecerNombreCurso( std::string ); // establece el nombre del curso
12    std::string obtenerNombreCurso() const; // obtiene el nombre del curso
13    void mostrarMensaje() const; // muestra un mensaje de bienvenida
14 private:
15    std::string nombreCurso; // nombre del curso para este LibroCalificaciones
16 }; // fin de la clase LibroCalificaciones
```

Fig. 3.15 | La definición de la clase `LibroCalificaciones` presenta la interfaz `public` de la clase.

Validar el nombre del curso con la función miembro `establecerNombreCurso` *de* `LibroCalificaciones`

Las modificaciones a la clase `LibroCalificaciones` están en las definiciones del constructor (figura 3.16, líneas 9 a 12) y en `establecerNombreCurso` (líneas 16 a 29). En vez de usar un inicializador de miembro, el constructor ahora llama a `establecerNombreCurso`. En general, *todos* los datos miembros deberían inicializarse con *inicializadores de miembros*. Sin embargo, algunas veces un constructor debe también *validar* su(s) argumento(s); a menudo esto se maneja en el cuerpo del constructor (línea 11). La llamada a `establecerNombreCurso` *valida* el argumento del constructor y *establece* el dato miembro `nombreCurso`. En un principio, el valor de `nombreCurso` se establecerá en la cadena vacía *antes* de que se ejecute el cuerpo del constructor, y luego `establecerNombreCurso` *modificará* el valor de `nombreCurso`.

En `establecerNombreCurso`, la instrucción `if` en las líneas 18 y 19 determina si el parámetro `nombre` contiene un nombre de curso *válido* (es decir, un `string` de 25 caracteres o menos). Si el nombre del curso es válido, en la línea 19 se almacena el nombre del curso en el miembro de datos `nombreCurso`. Observe la expresión `nombre.size()` en la línea 18. Ésta es una llamada a la función miembro, justo igual que `miLibroCalificaciones.mostrarMensaje()`. La clase `string` de la biblioteca estándar

de C++ define a una función miembro **size** que devuelve el número de caracteres en un objeto `string`. El parámetro `nombre` es un objeto `string`, por lo que la llamada a `nombre.size()` devuelve el número de caracteres en `nombre`. Si este valor es menor o igual que `25`, `nombre` es válido y se ejecuta la línea 19.

```
1  // Fig. 3.16: LibroCalificaciones.cpp
2  // Implementaciones de las definiciones de las funciones miembro de
      LibroCalificaciones.
3  // La función establecerNombreCurso realiza la validación.
4  #include <iostream>
5  #include "LibroCalificaciones.h" // incluye la definición de la clase
                                    LibroCalificaciones
6  using namespace std;
7
8  // el constructor inicializa nombreCurso con la cadena que se suministra como
      argumento
9  LibroCalificaciones::LibroCalificaciones( string nombre )
10 {
11    establecerNombreCurso( nombre ); // valida y almacena nombreCurso
12 } // fin del constructor de LibroCalificaciones
13
14 // función que establece el nombre del curso;
15 // asegura que el nombre del curso tenga como máximo 25 caracteres
16 void LibroCalificaciones::establecerNombreCurso( string nombre )
17 {
18    if ( nombre.size() <= 25 ) // si nombre tiene 25 caracteres o menos
19       nombreCurso = nombre; // almacena el nombre del curso en el objeto
20
21    if ( nombre.size() > 25 ) // si nombre tiene más de 25 caracteres
22    {
23       // establece nombreCurso a los primeros 25 caracteres del parámetro nombre
24       nombreCurso = nombre.substr( 0, 25 ); // empieza en 0, longitud de 25
25
26       cerr << "El nombre \"" << nombre << "\" excede la longitud maxima (25).\n"
27          "Se limito nombreCurso a los primeros 25 caracteres.\n" << endl;
28    } // fin de if
29 } // fin de la función establecerNombreCurso
30
31 // función para obtener el nombre del curso
32 string LibroCalificaciones::obtenerNombreCurso() const
33 {
34    return nombreCurso; // devuelve el nombreCurso del objeto
35 } // fin de la función obtenerNombreCurso
36
37 // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
38 void LibroCalificaciones::mostrarMensaje() const
39 {
40    // llama a obtenerNombreCurso para obtener el nombreCurso
41    cout << "Bienvenido al libro de calificaciones para\n" << obtenerNombreCurso()
42       << "!" << endl;
43 } // fin de la función mostrarMensaje
```

Fig. 3.16 | Definiciones de las funciones miembro para la clase `LibroCalificaciones`, con una función establecer que valida la longitud del dato miembro `nombreCurso`.

La instrucción `if` en las líneas 21 a 28 se encarga del caso en el que `establecerNombreCurso` recibe un nombre de curso *inválido* (es decir, un nombre que tenga más de 25 caracteres). Aun si el parámetro `nombre` es demasiado largo, de todas formas queremos que el objeto `LibroCalificaciones` quede en un **estado consistente**; es decir, un estado en el que el dato miembro `nombreCurso` del objeto contenga

un valor válido (un `string` de 25 caracteres o menos). Por ende, truncamos (reducimos) el nombre del curso especificado y asignamos los primeros 25 caracteres de `nombre` al dato miembro `nombreCurso` (por desgracia, esto podría truncar el nombre del curso de una manera extraña). La clase `string` estándar proporciona la función miembro **`substr`** ("substring", o subcadena), la cual devuelve un nuevo objeto `string` que se crea al copiar parte de un objeto `string` existente. La llamada en la línea 24 (es decir, `nombre.substr(0,25)`) pasa dos enteros (`0` y `25`) a la función miembro `substr` de `nombre`. Estos argumentos indican la porción de la cadena `nombre` que `substr` debe devolver. El primer argumento especifica la *posición inicial* en el objeto `string` original desde el que se van a copiar los caracteres; en todas las cadenas se considera que el primer carácter se encuentra en la posición 0. El segundo argumento especifica el *número de caracteres a copiar*. Por lo tanto, la llamada en la línea 24 devuelve una subcadena de 25 caracteres de `nombre`, empezando en la posición 0 (es decir, los primeros 25 caracteres en `nombre`). Por ejemplo, si `nombre` contiene el valor `"CS101 Introduccion a la programacion en C++"`, `substr` devuelve `"CS101 Introduccion a la p"`. Después de la llamada a `substr`, en la línea 24 se asigna la subcadena devuelta por `substr` al dato miembro `nombreCurso`. De esta forma, `establecerNombreCurso` asegura que a `nombreCurso` siempre se le asigne una cadena que contenga 25 caracteres o menos. Si la función miembro tiene que truncar el nombre del curso para hacerlo válido, en las líneas 26 y 27 se muestra un mensaje de advertencia mediante el uso de `cerr`, como se menciona en el capítulo 1.

La instrucción `if` en las líneas 21 a 28 contiene dos instrucciones en su cuerpo: una para establecer el `nombreCurso` a los primeros 25 caracteres del parámetro `nombre` y una para imprimir un mensaje complementario al usuario. Ambas instrucciones deben ejecutarse cuando `nombre` sea demasiado largo, por lo que las colocaremos en un par de llaves, `{ }`. En el capítulo 2 vimos que esto crea un *bloque*. En el capítulo 4 aprenderá más acerca de cómo colocar varias instrucciones en el cuerpo de una instrucción de control.

La instrucción en las líneas 26 y 27 también podría aparecer sin un operador de inserción de flujo al principio de la segunda línea de la instrucción, como en

```
cerr << "El nombre \"" << nombre << "\" excede la longitud maxima (25).\n"
   "Se limito nombreCurso a los primeros 25 caracteres.\n" << endl;
```

El compilador de C++ combina las literales de cadena adyacentes, aun si aparecen en líneas separadas de un programa. Por ende, en la instrucción anterior, el compilador de C++ combinaría las literales de cadena `"\" excede la longitud maxima (25).\n"` y `"Se limito nombreCurso a los primeros 25 caracteres.\n"` en una sola literal de cadena que produzca una salida idéntica a la de las líneas 26 y 27 de la figura 3.16. Este comportamiento nos permite imprimir cadenas extensas, al descomponerlas en varias líneas en nuestro programa, sin necesidad de incluir operaciones de inserción de flujo adicionales.

Prueba de la clase `LibroCalificaciones`

En la figura 3.17 se demuestra la versión modificada de la clase `LibroCalificaciones` (figuras 3.15 a 3.16) que incluye la validación. En la línea 12 se crea un objeto `LibroCalificaciones` llamado `libroCalificaciones1`. Recuerde que el constructor de `LibroCalificaciones` llama a `establecerNombreCurso` para inicializar el dato miembro `nombreCurso`. En versiones anteriores de la clase, el beneficio de llamar a `establecerNombreCurso` en el constructor no era evidente. Sin embargo, ahora *el constructor aprovecha la validación* que proporciona `establecerNombreCurso`. El constructor simplemente llama a `establecerNombreCurso`, *en vez de duplicar* su código de validación. Cuando en la línea 12 de la figura 3.17 se pasa un nombre inicial de `"CS101 Introduccion a la programacion en C++"` al constructor de `LibroCalificaciones`, el constructor pasa este valor a `establecerNombreCurso`, en donde ocurre la inicialización en sí. Como este nombre contiene más de 25 caracteres, se ejecuta el cuerpo de la segunda instrucción `if`, lo cual hace que `nombreCurso` se inicialice con el nombre truncado del curso de 25 caracteres de `"CS101 Introduccion a la p"` [la parte truncada se resalta con rojo (en su pantalla) en la línea 12]. La salida en la figura 3.17 contiene el mensaje de advertencia que producen las líneas 26 y 27

de la figura 3.16 en la función miembro `establecerNombreCurso`. En la línea 13 se crea otro objeto `LibroCalificaciones` llamado `libroCalificaciones2`; el nombre válido del curso que se pasa al constructor es exactamente de 25 caracteres.

```
1  // Fig. 3.17: fig03_17.cpp
2  // Crea y manipula un objeto LibroCalificaciones; ilustra la validación.
3  #include <iostream>
4  #include "LibroCalificaciones.h" // incluye la definición de la clase
                                    LibroCalificaciones
5  using namespace std;
6
7  // la función main empieza la ejecución del programa
8  int main()
9  {
10   // crea dos objetos LibroCalificaciones;
11   // el nombre inicial del curso de libroCalificaciones1 es demasiado largo
12   LibroCalificaciones libroCalificaciones1( "CS101 Introduccion a la
                                                programacion en C++" );
13   LibroCalificaciones libroCalificaciones2( "CS102 C++:Estruc de datos" );
14
15   // muestra el nombreCurso de cada LibroCalificaciones
16   cout << "el nombre inicial del curso de libroCalificaciones1 es: "
17      << libroCalificaciones1.obtenerNombreCurso()
18      << "\nel nombre inicial del curso de libroCalificaciones2 es: "
19      << libroCalificaciones2.obtenerNombreCurso() << endl;
20
21   // modifica el nombreCurso de libroCalificaciones1 (con una cadena con
        longitud válida)
22   libroCalificaciones1.establecerNombreCurso( "CS101 Programacion en C++"
23
24   // muestra el nombreCurso de cada LibroCalificaciones
25   cout << "\nel nombre del curso de libroCalificaciones1 es: "
26      << libroCalificaciones1.obtenerNombreCurso()
27      << "\nel nombre del curso de libroCalificaciones2 es: "
28      << libroCalificaciones2.obtenerNombreCurso() << endl;
29 } // fin de main
```

```
El nombre "CS101 Introduccion a la programacion en C++" excede la longitud maxima (25).
Se limito nombreCurso a los primeros 25 caracteres.

el nombre inicial del curso de libroCalificaciones1 es: CS101 Introduccion a la p
el nombre inicial del curso de libroCalificaciones2 es: CS102 C++:Estruc de datos

el nombre del curso de libroCalificaciones1 es: CS101 Programacion en C++
el nombre del curso de libroCalificaciones2 es: CS102 C++:Estruc de datos
```

Fig. 3.17 | Creación y manipulación de un objeto `LibroCalificaciones` en el que el nombre del curso está limitado a una longitud de 25 caracteres.

En las líneas 16 a 19 de la figura 3.17 se muestra el nombre del curso truncado para `libroCalificaciones1` (se resalta esto en color azul en la salida de su pantalla) y el nombre del curso para `libroCalificaciones2`. En la línea 22 se hace una llamada a la función miembro `establecerNombreCurso` de `libroCalificaciones1` directamente, para modificar el nombre del curso en el objeto `LibroCalificaciones` a un nombre más corto, que no necesite truncarse. Después, en las líneas 25 a 28 se imprimen los nombres de los cursos para los objetos `LibroCalificaciones` de nuevo.

Observaciones adicionales acerca de las funciones establecer

Una función *establecer* `public` tal como `establecerNombreCurso` debe escudriñar cuidadosamente cualquier intento por modificar el valor de un dato miembro (por ejemplo, `nombreCurso`) para asegurar que el nuevo valor sea apropiado para ese elemento de datos. Por ejemplo, un intento por *establecer* el día del mes en 37 debe rechazarse, un intento por *establecer* el peso de una persona en cero o en un valor negativo debe rechazarse, un intento por *establecer* una calificación en un examen en 185 (cuando el rango apropiado es de cero a 100) debe rechazarse, y así en lo sucesivo.

Observación de Ingeniería de Software 3.3

Hacer los datos miembros `private` *y controlar el acceso, en especial el acceso de escritura, a esos datos miembros a través de funciones miembro* `public`*, ayuda a asegurar la integridad de los datos.*

Tip para prevenir errores 3.4

Los beneficios de la integridad de datos no son automáticos sólo porque los datos miembros se hacen `private`*; debemos proporcionar una verificación de validez apropiada y reportar los errores.*

Una función *establecer* podría devolver un valor para indicar que se realizó un intento por asignar datos inválidos al objeto de una clase. Un cliente podría entonces probar el valor de retorno de la función *establecer* para determinar si el intento del cliente por modificar el objeto fue exitoso, y para realizar la acción apropiada en caso contrario. En capítulos posteriores haremos eso, después de que introduzcamos un poco más de tecnología de programación. En C++, se puede notificar a los clientes de objetos sobre los problemas a través del *mecanismo de manejo de excepciones*, que veremos ligeramente en el capítulo 7 y presentaremos de manera detallada en el capítulo 17 (en el sitio web).

3.9 Conclusión

En este capítulo aprendió a crear clases definidas por el usuario, y a crear y utilizar objetos de esas clases. Declaramos datos miembros de una clase para mantener los datos para cada objeto de la misma. También definimos funciones miembro que operan con esos datos. Aprendió que las funciones miembro que no modifican los datos de una clase deben declararse como `const`. Le mostramos cómo llamar a las funciones miembro de un objeto para solicitar los servicios que éste proporciona, y cómo pasar datos a esas funciones miembro como argumentos. Hablamos sobre la diferencia entre una variable local de una función miembro y un dato miembro de una clase. También le mostramos cómo usar un constructor y una lista inicializadora de miembros para asegurar que todos los objetos se inicialicen correctamente. Aprendió que un constructor con un solo parámetro debe declararse como `explicit`, y que un constructor no puede declararse como `const` debido a que modifica el objeto que se va a inicializar. Le demostramos cómo separar la interfaz de una clase de su implementación, para fomentar la buena ingeniería de software. Aprendió que nunca se deben colocar las directivas `using` y las declaraciones `using` en los encabezados. Presentamos un diagrama que muestra los archivos que necesitan los programadores de la implementación de la clase y los programadores del código cliente para compilar el código que escriben. Demostramos cómo se pueden utilizar las funciones *establecer* para validar los datos de un objeto y asegurar que los objetos se mantengan en un estado consistente. Además, se utilizaron diagramas de clases de UML para modelar las clases y sus constructores, las funciones miembro y los datos miembros. En el siguiente capítulo empezaremos nuestra introducción a las instrucciones de control, las cuales especifican el orden en el que se realizan las acciones de una función.

Resumen

Sección 3.2 Definición de una clase con una función miembro

- La definición de una clase (pág. 68) contiene los datos miembro y las funciones miembro que definen los atributos y comportamientos de esa clase, respectivamente.
- La definición de una clase empieza con la palabra clave `class`, seguida inmediatamente por el nombre de la clase.
- Por convención, el nombre de una clase definida por el usuario (pág. 69) empieza con una letra mayúscula, y por legibilidad, cada palabra subsiguiente en el nombre de la clase empieza con una letra mayúscula.
- El cuerpo de cada clase (pág. 68) va encerrado entre un par de llaves izquierda y derecha (`{` y `}`), y termina con un punto y coma.
- Las funciones miembro que aparecen después del especificador de acceso `public` (pág. 68) pueden ser llamadas por otras funciones en un programa, y por las funciones miembro de otras clases.
- Los especificadores de acceso siempre van seguidos de un punto y coma (`;`).
- La palabra clave `void` (pág. 69) es un tipo de valor de retorno especial, el cual indica que una función realizará una tarea, pero no devolverá datos a la función que la llamó cuando complete su tarea.
- Por convención, los nombres de las funciones (pág. 69) empiezan con la primera letra en minúscula, y todas las palabras subsiguientes en el nombre empiezan con letra mayúscula.
- Un conjunto vacío de paréntesis después del nombre de una función indica que ésta no requiere datos adicionales para realizar su tarea.
- Una función que no modifica, y no debe hacerlo, el objeto en el cual se llama debe declararse como `const`.
- Por lo general, no se puede llamar a una función miembro sino hasta que se crea un objeto de su clase.
- Cada nueva clase que creamos se convierte en un nuevo tipo en C++.
- En UML, cada clase se modela en un diagrama de clases (pág. 70) como un rectángulo con tres compartimientos, que (de arriba hacia abajo) contienen el nombre de la clase, los atributos y las operaciones, respectivamente.
- UML modela las operaciones como el nombre de la operación, seguido de paréntesis. Un signo más (+) enfrente del nombre de la operación indica que ésta es una operación `public` (es decir, una función miembro `public` en C++).

Sección 3.3 Definición de una función miembro con un parámetro

- Una función miembro puede requerir uno o más parámetros (pág. 70) para representar los datos adicionales que necesita para realizar su tarea. La llamada a una función suministra un argumento (pág. 71) para cada uno de los parámetros de esa función.
- Para llamar a una función miembro, se coloca después del nombre del objeto un operador punto (`.`), el nombre de la función y un conjunto de paréntesis que contienen los argumentos de la misma.
- Una variable de la clase `string` (pág. 72) de la biblioteca estándar de C++ representa a una cadena de caracteres. Esta clase se define en el encabezado `<string>`, y el nombre `string` pertenece al espacio de nombres `std`.
- La función `getline` (del encabezado `<string>`, pág. 72) lee caracteres de su primer argumento hasta encontrar un carácter de nueva línea, y después coloca los caracteres (sin incluir la nueva línea) en la variable `string` que se especifica como su segundo argumento. El carácter de nueva línea se descarta.
- Una lista de parámetros (pág. 73) puede contener cualquier número de parámetros, incluyendo ninguno (lo cual se representa por paréntesis vacíos) para indicar que una función no requiere parámetros.
- El número de argumentos en la llamada a una función debe coincidir con el número de parámetros en la lista de parámetros del encabezado de la función miembro a la que se llamó. Además, los tipos de los argumentos en la llamada a la función deben ser consistentes con los tipos de los parámetros correspondientes en el encabezado de la función.
- Para modelar un parámetro de una operación, UML lista el nombre del parámetro, seguido de dos puntos y del tipo del parámetro entre los paréntesis que van después del nombre de la operación.
- UML tiene sus propios tipos de datos. No todos los tipos de datos de UML tienen los mismos nombres que los tipos de C++ correspondientes. El tipo `String` de UML corresponde al tipo `string` de C++.

Sección 3.4 Datos miembros, funciones establecer *y* obtener

- Las variables que se declaran en el cuerpo de una función son variables locales (pág. 74) y sólo pueden utilizarse desde el punto en el que se declararon en la función, hasta la llave de cierre derecha (`}`) del bloque en el que se declararon.
- Una variable local debe declararse antes de poder utilizarse en una función. Una variable local no puede utilizarse fuera de la función en la que está declarada.
- Por lo general, los datos miembros (pág. 74) son `private` (pág. 76). Las variables o funciones que se declaran `private` son accesibles sólo para las funciones miembro de la clase en la que están declaradas, o para las funciones amigas de la clase.
- Cuando un programa crea (instancia) un objeto de una clase, sus datos miembros `private` se encapsulan (ocultan, pág. 76) en el objeto y sólo las funciones miembro de la clase del objeto pueden utilizarlos (o las "amigas" de la clase, como veremos en el capítulo 9).
- Cuando se hace una llamada a una función que especifica un tipo de valor de retorno distinto de `void` y ésta completa su tarea, la función devuelve un resultado a la función que la llamó.
- De manera predeterminada, el valor inicial de un objeto `string` es la cadena vacía (pág. 77); es decir, una cadena que no contenga caracteres. No aparece nada en la pantalla cuando se muestra una cadena vacía.
- A menudo, una clase proporciona funciones miembro `public` para permitir que los clientes de la clase *establezcan* u *obtengan* (pág. 78) datos miembros `private`. Los nombres de esas funciones miembro comúnmente empiezan con *establecer* u *obtener* (*set* o *get*, en inglés).
- Las funciones *establecer* y *obtener* permiten a los clientes de una clase utilizar de manera indirecta los datos ocultos. El cliente no sabe cómo realiza el objeto estas operaciones.
- Las funciones *establecer* y *obtener* de una clase deben ser utilizadas por otras funciones miembro de la clase para manipular los datos `private` de esa clase. Si la representación de los datos de la clase cambia, las funciones miembro que acceden a los datos sólo a través de las funciones *establecer* y *obtener* no requerirán modificación.
- Una función *establecer* pública debe escudriñar cuidadosamente cualquier intento por modificar el valor de un dato miembro, para asegurar que el nuevo valor sea apropiado para ese elemento de datos.
- UML representa a los datos miembros como atributos, para lo cual enlista el nombre del atributo, seguido de dos puntos y del tipo del atributo. En UML, se coloca un signo menos (-) antes de los atributos privados.
- Para indicar el tipo de valor de retorno de una operación en UML, se coloca un signo de dos puntos y el tipo de valor de retorno después de los paréntesis que siguen del nombre de la operación.
- Los diagramas de clases de UML no especifican tipos de valores de retorno para las operaciones que no devuelven valores.

Sección 3.5 Inicialización de objetos mediante constructores

- Cada clase debe proporcionar uno o más constructores (pág, 79) para inicializar un objeto de la clase cuando el objeto se crea. Un constructor se debe definir con el mismo nombre que la clase.
- Una diferencia entre los constructores y las funciones es que los constructores no pueden devolver valores, por lo que no pueden especificar un tipo de valor de retorno (ni siquiera `void`). Por lo general, los constructores se declaran como `public`.
- C++ llama automáticamente a un constructor por cada objeto que se crea, lo cual ayuda a asegurar que todo objeto se inicialice antes de usarlo en un programa.
- Un constructor sin parámetros es un constructor predeterminado (pág. 80). Si el programador no proporciona un constructor, el compilador proporciona un constructor predeterminado. También podemos definir un constructor predeterminado de manera explícita. Si define un constructor para una clase, C++ no creará un constructor predeterminado.
- Un constructor con un solo parámetro debe declararse como `explicit`.
- Un constructor usa una lista inicializadora de miembros para inicializar los datos miembros de una clase. Los inicializadores de miembros aparecen entre la lista de parámetros y la llave izquierda que comienza el cuerpo del constructor. La lista inicializadora de miembros se separa de la lista de parámetros con un signo de dos puntos (`:`). Un inicializador de miembro consiste en el nombre de variable de un miembro de datos seguido de paréntesis que contienen el valor inicial del miembro. Podemos realizar la inicialización en el cuerpo del constructor, pero

más adelante en el libro veremos que es más eficiente hacerlo con inicializadores de miembros; además algunos tipos de datos miembros deben inicializarse de esta forma.

- UML modela a los constructores como operaciones en el tercer compartimiento de un diagrama de clases, con la palabra "constructor" entre los signos « y » antes del nombre del constructor.

Sección 3.6 Colocar una clase en un archivo separado para fines de reutilización

- Cuando las definiciones de clases se empaquetan en forma apropiada, los programadores de todo el mundo pueden reutilizarlas.
- Es costumbre definir una clase en un encabezado (pág. 83) que tenga una extensión de nombre de archivo `.h`.

Sección 3.7 Separar la interfaz de la implementación

- Si cambia la implementación de la clase, los clientes de ésta no tienen que cambiar.
- Las interfaces definen y estandarizan las formas en que deben interactuar las cosas como las personas y los sistemas.
- La interfaz `public` de una clase (pág. 87) describe las funciones miembro `public` que están disponibles para los clientes de la clase. La interfaz describe *qué* servicios (pág. 87) pueden usar los clientes y cómo *reutilizar* esos servicios, pero no especifica *cómo* es que la clase lleva a cabo los servicios.
- Al separar la interfaz de la implementación (pág. 87) se facilita la modificación de los programas. Los cambios en la implementación de la clase no afectan al cliente, mientras que la interfaz de la clase permanezca sin cambios.
- Nunca se deben colocar directivas `using` ni declaraciones `using` en los encabezados.
- El prototipo de una función (pág. 87) contiene el nombre de una función, su tipo de valor de retorno y el número, tipos y orden de los parámetros que la función espera recibir.
- Una vez que se define una clase y se declaran sus funciones miembro (a través de los prototipos de función), las funciones miembro deben definirse en un archivo de código fuente separado.
- Para cada función miembro definida fuera de su correspondiente definición de clase, hay que anteponer al nombre de la función el nombre de la clase y el operador de resolución de ámbito (`::`, pág. 89).

Sección 3.8 Validación de datos mediante funciones establecer

- La función miembro `size` de la clase `string` (pág. 93) devuelve el número de caracteres en un objeto `string`.
- La función miembro `substr` (pág. 95) de la clase `string` devuelve un nuevo objeto `string` que contiene una copia de parte de un objeto `string` existente. El primer argumento especifica la posición inicial en el objeto `string` original. El segundo argumento especifica el número de caracteres a copiar.

Ejercicios de autoevaluación

3.1 Complete las siguientes oraciones:

a) Cada declaración de clase contiene la palabra clave _____, seguida inmediatamente por el nombre de la clase.

b) Por lo general, la definición de una clase se almacena en un archivo con la extensión de archivo _____.

c) Cada parámetro en un encabezado de función debe especificar un(a) _____ y un(a) _____.

d) Cuando cada objeto de una clase mantiene su propia versión de un atributo, la variable que representa a este atributo se conoce también como _____.

e) La palabra clave `public` es un(a) _____.

f) El tipo de valor de retorno _____ indica que una función realizará una tarea, pero no devolverá información cuando complete su tarea.

g) La función _____ de la biblioteca `<string>` lee caracteres hasta encontrar una nueva línea, y después copia esos caracteres en el objeto `string` especificado.

h) Cuando se define una función miembro fuera de la definición de una clase, el encabezado de la función debe incluir el nombre de la clase y el _____, seguido del nombre de la función para "enlazar" la función miembro con la definición de la clase.

i) El archivo de código fuente, y cualquier otro archivo que utilice una clase, pueden incluir el archivo de encabezado de la clase mediante una directiva del preprocesador _____.

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

a) Por convención, los nombres de las funciones empiezan con la primera letra en mayúscula y todas las palabras subsiguientes en el nombre empiezan con la primera letra en mayúscula.
b) Los paréntesis vacíos que van después del nombre de una función en un prototipo de función indican que ésta no requiere parámetros para realizar su tarea.
c) Los datos miembros o las funciones miembro que se declaran con el modificador de acceso `private` son accesibles para las funciones miembro de la clase en la que se declaran.
d) Las variables que se declaran en el cuerpo de una función miembro específica se conocen como datos miembros, y pueden utilizarse en todas las funciones miembro de la clase.
e) El cuerpo de toda función está delimitado por las llaves izquierda y derecha (`{` y `}`).
f) Cualquier archivo de código fuente que contenga `int main()` puede usarse para ejecutar un programa.
g) Los tipos de los argumentos en la llamada a una función deben ser consistentes con los tipos de los parámetros correspondientes en la lista de parámetros del prototipo de la función.

3.3 ¿Cuál es la diferencia entre una variable local y un dato miembro?

3.4 Explique el propósito de un parámetro de una función. ¿Cuál es la diferencia entre un parámetro y un argumento?

Respuestas a los ejercicios de autoevaluación

3.1 a) `class`. b) `.h`. c) tipo, nombre. d) miembro de datos. e) especificador de acceso. f) `void`. g) `getline`. h) operador de resolución de ámbito (`::`). i) `#include`.

3.2 a) Falso. Los nombres de las funciones empiezan con una primera letra en minúscula y todas las palabras subsiguientes en el nombre empiezan con una letra en mayúscula. b) Verdadero. c) Verdadero. d) Falso. Dichas variables son variables locales y sólo pueden usarse en la función miembro en la que están declaradas. e) Verdadero. f) Verdadero. g) Verdadero.

3.3 Una variable local se declara en el cuerpo de una función, y sólo puede utilizarse desde el punto en el que se declaró, hasta la llave de cierre del bloque correspondiente. Un miembro de datos se declara en una clase, pero no en el cuerpo de alguna de las funciones miembro de la clase. Cada objeto de una clase tiene una copia separada de los datos miembros de la clase. Los datos miembros están accesibles para todas las funciones miembro de la clase.

3.4 Un parámetro representa la información adicional que requiere una función para realizar su tarea. Cada parámetro requerido por una función está especificado en el encabezado de la función. Un argumento es el valor que se suministra en la llamada a la función. Cuando se llama a la función, el valor del argumento se pasa al parámetro de la función, para que ésta pueda realizar su tarea.

Ejercicios

3.5 ***(Prototipos y definiciones de funciones)*** Explique la diferencia entre un prototipo de función y la definición de una función.

3.6 ***(Constructor predeterminado)*** ¿Qué es un constructor predeterminado? ¿Cómo se inicializan los datos miembros de un objeto, si una clase sólo tiene un constructor predeterminado definido en forma implícita?

3.7 ***(Datos miembros)*** Explique el propósito de un dato miembro.

3.8 ***(Encabezado y archivos de código fuente)*** ¿Qué es un encabezado? ¿Qué es un archivo de código fuente? Hable sobre el propósito de cada uno.

3.9 ***(Uso de una clase sin una directiva `using`)*** Explique cómo puede usar un programa una clase `string` sin insertar una directiva `using`.

3.10 *(Funciones* **establecer** *y* **obtener***)* Explique por qué una clase podría proporcionar una función *establecer* y una función *obtener* para un dato miembro.

3.11 *(Modificación de la clase* `LibroCalificaciones`*)* Modifique la clase `LibroCalificaciones` (figuras 3.11 a 3.12) de la siguiente manera:

a) Incluya un segundo miembro de datos `string`, que represente el nombre del instructor del curso.
b) Proporcione una función *establecer* para modificar el nombre del instructor, y una función *obtener* para obtenerlo.
c) Modifique el constructor para especificar dos parámetros: uno para el nombre del curso y otro para el nombre del instructor.
d) Modifique la función `mostrarMensaje`, de tal forma que primero imprima el mensaje de bienvenida y el nombre del curso, y que después imprima `"Este curso es presentado por: "`, seguido del nombre del instructor.

Use su clase modificada en un programa de prueba que demuestre las nuevas capacidades de la clase.

3.12 *(Clase* `Cuenta`*)* Cree una clase llamada `Cuenta` que podría ser utilizada por un banco para representar las cuentas bancarias de sus clientes. Incluya un miembro de datos de tipo `int` para representar el saldo de la cuenta. [*Nota:* en los siguientes capítulos, utilizaremos números que contienen puntos decimales (por ejemplo, 2.75), a los cuales se les conoce como valores de punto flotante, para representar montos en dólares]. Proporcione un constructor que reciba un saldo inicial y lo utilice para inicializar el dato miembro. El constructor debe validar el saldo inicial para asegurar que sea mayor o igual que 0. De no ser así, establezca el saldo en 0 y muestre un mensaje de error, indicando que el saldo inicial era inválido. Proporcione tres funciones miembro. La función miembro `abonar` debe agregar un monto al saldo actual. La función miembro `cargar` deberá retirar dinero del objeto `Cuenta` y asegurarse que el monto a cargar no exceda el saldo de `Cuenta`. Si lo hace, el saldo debe permanecer sin cambio y la función debe imprimir un mensaje que indique `"El monto a cargar excede el saldo de la cuenta."` La función miembro `obtenerSaldo` debe devolver el saldo actual. Cree un programa que cree dos objetos `Cuenta` y evalúe las funciones miembro de la clase `Cuenta`.

3.13 *(Clase* `Factura`*)* Cree una clase llamada `Factura`, que una ferretería podría utilizar para representar una factura por un artículo vendido en la tienda. Una `Factura` debe incluir cuatro datos miembros: un número de pieza (tipo `string`), la descripción de la pieza (tipo `string`), la cantidad de artículos de ese tipo que se van a comprar (tipo `int`) y el precio por artículo (tipo `int`). [*Nota:* en los siguientes capítulos, utilizaremos números que contienen puntos decimales (por ejemplo, 2.75), a los cuales se les conoce como valores de punto flotante, para representar montos en dólares]. Su clase debe tener un constructor que inicialice los cuatro datos miembros. Un constructor que recibe múltiples argumentos se define mediante la siguiente forma:

nombreClase(*nombreTipo1 nombreParámetro1, nombreTipo2 nombreParámetro2,* …)

Proporcione una función *establecer* y una función *obtener* para cada miembro de datos. Además, proporcione una función miembro llamada `obtenerMontoFactura`, que calcule el monto de la factura (es decir, que multiplique la cantidad por el precio por artículo) y después devuelva ese monto como un valor `int`. Si la cantidad no es positiva, debe establecerse en `0`. Si el precio por artículo no es positivo, debe establecerse en `0`. Escriba un programa de prueba que demuestre las capacidades de la clase `Factura`.

3.14 *(Clase* `Empleado`*)* Cree una clase llamada `Empleado`, que incluya tres piezas de información como datos miembros: un primer nombre (tipo `string`), un apellido paterno (tipo `string`) y un salario mensual (tipo `int`). [*Nota:* en los siguientes capítulos, utilizaremos números que contienen puntos decimales (por ejemplo, 2.75), a los cuales se les conoce como valores de punto flotante, para representar montos en dólares]. Su clase debe tener un constructor que inicialice los tres datos miembros. Proporcione una función *establecer* y una función *obtener* para cada dato miembro. Si el salario mensual no es positivo, establézcalo en `0`. Escriba un programa de prueba que demuestre las capacidades de la clase `Empleado`. Cree dos objetos `Empleado` y muestre el salario *anual* de cada objeto. Después, proporcione a cada `Empleado` un aumento del 10% y muestre el salario anual de cada `Empleado` otra vez.

3.15 *(Clase* `Fecha`*)* Cree una clase llamada `Fecha`, que incluya tres piezas de información como datos miembros: un mes (tipo `int`), un día (tipo `int`) y un año (tipo `int`). Su clase debe tener un constructor con tres parámetros, los cuales debe utilizar para inicializar los tres datos miembros. Para los fines de este ejercicio, suponga que los valores que se proporcionan para el año y el día son correctos, pero asegúrese que el valor del mes se encuentre en el rango de 1 a 12; de no ser así, establezca el mes en `1`. Proporcione una función *establecer* y una función *obtener* para

cada miembro de datos. Proporcione una función miembro `mostrarFecha`, que muestre el mes, día y año, separados por barras diagonales (/). Escriba un programa de prueba que demuestre las capacidades de la clase `Fecha`.

Hacer la diferencia

3.16 ***(Calculadora de la frecuencia cardiaca esperada)*** Mientras se ejercita, puede usar un monitor de frecuencia cardiaca para ver que su corazón permanezca dentro de un rango seguro sugerido por sus entrenadores y doctores. De acuerdo con la Asociación Estadounidense del Corazón (AHA) (`www.americanheart.org/presenter.jhtml?identifier=4736`), la fórmula para calcular su *frecuencia cardiaca máxima* en pulsos por minuto es 220 menos su edad en años. Su *frecuencia cardiaca esperada* es un rango que está entre el 50 y el 85% de su frecuencia cardiaca máxima. [*Nota: estas fórmulas son estimaciones proporcionadas por la AHA. Las frecuencias cardiacas máxima y esperada pueden variar con base en la salud, condición física y sexo del individuo. Siempre debe consultar un médico o a un profesional de la salud antes de empezar o modificar un programa de ejercicios*]. Cree una clase llamada `FrecuenciasCardiacas`. Los atributos de la clase deben incluir el primer nombre de la persona, su apellido y fecha de nacimiento (la cual debe consistir de atributos separados para el mes, día y año de nacimiento). Su clase debe tener un constructor que reciba estos datos como parámetros. Para cada atributo debe proveer funciones *establecer* y *obtener*. La clase también debe incluir una función `obtenerEdad` que calcule y devuelva la edad de la persona (en años), una función `obtenerFrecuenciaCardiacaMaxima` que calcule y devuelva la frecuencia cardiaca máxima de esa persona, y una función `obtenerFrecuenciaCardiacaEsperada` que calcule y devuelva la frecuencia cardiaca esperada de la persona. Puesto que todavía no sabe cómo obtener la fecha actual de la computadora, la función `obtenerEdad` debe pedir al usuario que introduzca el mes, día y año actual antes de calcular la edad de la persona. Escriba una aplicación que pida la información de la persona, cree una instancia de un objeto de la clase `FrecuenciasCardiacas` e imprima la información a partir de ese objeto (incluyendo el primer nombre de la persona, su apellido y fecha de nacimiento), y que después calcule e imprima la edad de la persona en (años), frecuencia cardiaca máxima y rango de frecuencia cardiaca esperada.

3.17 ***(Computarización de los registros médicos)*** Un problema relacionado con la salud que ha estado últimamente en las noticias es la computarización de los registros médicos. Esta posibilidad se está tratando con mucho cuidado, debido a las delicadas cuestiones de privacidad y seguridad, entre otras cosas. [Trataremos esas cuestiones en ejercicios posteriores]. La computarización de los registros médicos puede facilitar a los pacientes el proceso de compartir sus perfiles e historiales médicos con los diversos profesionales de la salud que consulten. Esto podría mejorar la calidad del servicio médico, ayudar a evitar conflictos de fármacos y prescripciones erróneas, reducir los costos y, en emergencias, podría ayudar a salvar vidas. En este ejercicio usted diseñará una clase "inicial" llamada `PerfilMedico` para una persona. Los atributos de la clase deben incluir el primer nombre de la persona, su apellido, sexo, fecha de nacimiento (que debe consistir de atributos separados para el día, mes y año de nacimiento), altura (en centímetros) y peso (en kilogramos). Su clase debe tener un constructor que reciba estos datos. Para cada atributo, debe proveer las funciones *establecer* y *obtener*. La clase también debe incluir métodos que calculen y devuelvan la edad del usuario en años, la frecuencia cardiaca máxima y el rango de frecuencia cardiaca esperada (vea el ejercicio 3.16), además del índice de masa corporal (BMI; vea el ejercicio 2.30). Escriba una aplicación que pida la información de la persona, cree una instancia de un objeto de la clase `PerfilMedico` para esa persona e imprima la información de ese objeto (incluyendo el primer nombre de la persona, apellido, sexo, fecha de nacimiento, altura y peso), y que después calcule e imprima la edad de esa persona en años, junto con el BMI, la frecuencia cardiaca máxima y el rango de frecuencia cardiaca esperada. También debe mostrar la tabla de "valores del BMI" del ejercicio 2.30. Use la misma técnica que en el ejercicio 3.16 para calcular la edad de la persona.

4

Instrucciones de control, parte I: operadores de asignación, ++ y --

Desplacémonos un lugar.
—Lewis Carroll

La rueda se convirtió en un círculo completo.
—William Shakespeare

Toda la evolución que conocemos procede de lo vago a lo definido.
—Charles Sanders Peirce

Objetivos

En este capítulo aprenderá a:

- Comprender las técnicas básicas para solucionar problemas.
- Desarrollar algoritmos mediante el proceso de mejoramiento de arriba a abajo, paso a paso.
- Utilizar las estructuras de selección `if` e `if...else` para elegir entre distintas acciones alternativas.
- Utilizar la estructura de repetición `while` para ejecutar instrucciones de manera repetitiva dentro de un programa.
- Comprender la repetición controlada por un contador y la repetición controlada por un centinela.
- Utilizar los operadores de incremento, decremento y asignación.

Plan general

4.1 Introducción

Antes de escribir un programa que dé solución a un problema, es necesario tener una comprensión detallada de todo el problema, además de una metodología cuidadosamente planeada para resolverlo. Al escribir un programa, también debemos comprender los tipos de bloques de construcción disponibles, y emplear las técnicas comprobadas para construirlos. En éste y en el capítulo 5, "Instrucciones de control, parte 2: operadores lógicos", hablaremos sobre estas cuestiones cuando presentemos la teoría y los principios de la *programación estructurada*. Los conceptos aquí presentados son imprescindibles para crear clases efectivas y manipular objetos.

En este capítulo presentamos las instrucciones `if`, `if...else` y `while` de C++, tres de los bloques de construcción que permiten a los programadores especificar la lógica requerida para que las funciones miembro realicen sus tareas. Dedicamos una parte de este capítulo (y de los capítulos 5 a 7) para desarrollar más la clase `LibroCalificaciones`. En especial, agregamos a dicha clase una función miembro que utiliza instrucciones de control para calcular el promedio de un conjunto de calificaciones de estudiantes. Otro ejemplo demuestra formas adicionales de combinar instrucciones de control. Presentamos los operadores de asignación, incremento y decremento de C++. Estos operadores adicionales abrevian y simplifican muchas instrucciones de los programas.

4.2 Algoritmos

Cualquier problema de computación puede resolverse ejecutando una serie de acciones en un orden específico. Un **procedimiento** para resolver un problema en términos de

1. las **acciones** a ejecutar y
2. el **orden** en el que se ejecutan estas acciones

se conoce como un **algoritmo**. El siguiente ejemplo demuestra que es importante especificar de manera correcta el orden en el que se ejecutan las acciones.

Considere el "algoritmo para levantarse y arreglarse" que sigue un ejecutivo para levantarse de la cama e ir a trabajar: (1) levantarse, (2) quitarse la pijama, (3) bañarse, (4) vestirse, (5) desayunar, (6) transportarse al trabajo. Esta rutina logra que el ejecutivo llegue al trabajo bien preparado para tomar decisiones críticas. Suponga que los mismos pasos se realizan en un orden ligeramente distinto: (1) levantarse; (2) quitarse la pijama; (3) vestirse; (4) bañarse; (5) desayunar; (6) transportarse al trabajo. En este caso, nuestro ejecutivo llegará al trabajo todo mojado. Al proceso de especificar el orden en el que

se ejecutan las instrucciones (acciones) en un programa de computadora, se le llama **control del programa**. En este capítulo investigaremos el control de los programas mediante el uso de las **instrucciones de control** de C++.

4.3 Seudocódigo

El **seudocódigo** (o "imitación" de código) es un lenguaje artificial e informal que ayuda a los programadores a desarrollar algoritmos sin tener que preocuparse por los detalles de la sintaxis del lenguaje C++. El seudocódigo que presentaremos aquí es útil para desarrollar algoritmos que se convertirán en programas estructurados de C++. El seudocódigo es similar al lenguaje cotidiano; es conveniente y amigable con el usuario, aunque no es realmente un lenguaje de programación de computadoras.

El seudocódigo *no* se ejecuta en las computadoras. En vez de ello, ayuda al programador a "organizar" un programa antes de intentar escribirlo en un lenguaje de programación como C++.

El estilo de seudocódigo que presentaremos consiste solamente en caracteres, de manera que los programadores puedan escribir el seudocódigo convenientemente, utilizando cualquier programa editor de texto. Un programa en seudocódigo preparado de manera cuidadosa puede convertirse fácilmente en su correspondiente programa en C++. En muchos casos, esto requiere tan sólo reemplazar las instrucciones en seudocódigo con sus instrucciones equivalentes en C++.

Por lo general, el seudocódigo describe sólo las **instrucciones ejecutables**, las cuales provocan que ocurran acciones específicas después de que un programador convierte un programa de seudocódigo a C++, y el programa se compila y ejecuta en una computadora. Las declaraciones (que no tienen inicializadores, o que no implican llamadas a un constructor) *no* son instrucciones ejecutables. Por ejemplo, la declaración

```
int contador;
```

indica al compilador el tipo de la variable `contador` y lo instruye para que reserve espacio en memoria para esa variable. Esta declaración *no* hace que ocurra ninguna acción (como una operación de entrada, salida o un cálculo) cuando el programa se ejecuta. Por lo general no incluimos las declaraciones de variables en nuestro seudocódigo. Algunos programadores optan por listar las variables y mencionar sus propósitos al principio de sus programas en seudocódigo.

Veamos un ejemplo de seudocódigo que se puede escribir para ayudar a un programador a crear el programa de suma de la figura 2.5. Este seudocódigo (figura 4.1) corresponde al algoritmo que recibe como entrada dos enteros del usuario, los suma y muestra el resultado en pantalla. Aunque mostramos aquí el listado completo en seudocódigo, más adelante le mostraremos cómo *crear seudocódigo a partir del enunciado de un problema*.

Las líneas 1 y 2 corresponden a las instrucciones 13 y 14 de la figura 2.5 Observe que las instrucciones en seudocódigo son simplemente instrucciones en lenguaje cotidiano que representan la tarea que se debe realizar en C++. De igual forma, las líneas 4 y 5 corresponden a las instrucciones en las líneas 16 y 17, y las líneas 7 y 8 corresponden a las instrucciones en las líneas 19 y 21 de la figura 2.5.

```
1  Pedir al usuario que introduzca el primer entero
2  Recibir como entrada el primer entero
3
4  Pedir al usuario que introduzca el segundo entero
5  Recibir como entrada el segundo entero
6
7  Sumar el primer entero con el segundo entero, almacenar el resultado
8  Mostrar el resultado en pantalla
```

Fig. 4.1 | Seudocódigo para el programa de suma de la figura 2.5.

4.4 Estructuras de control

Por lo general, en un programa las instrucciones se ejecutan una después de otra, en el orden en que están escritas. Este proceso se conoce como **ejecución secuencial**. Varias instrucciones en C++ que pronto veremos, permiten al programador especificar que *la siguiente instrucción a ejecutarse tal vez no sea la siguiente en la secuencia*. Esto se conoce como **transferencia de control**.

Durante la década de 1960, se hizo evidente que el uso indiscriminado de las transferencias de control era el origen de muchas de las dificultades que experimentaban los grupos de desarrollo de software. A quien se señaló como culpable fue a la **instrucción `goto`**, la cual permite al programador especificar la transferencia de control a uno de los muchos posibles destinos dentro de un programa (creando lo que se conoce comúnmente como "código espagueti"). La noción de la llamada **programación estructurada** se hizo casi un sinónimo de la "**eliminación del `goto`**".

Las investigaciones de Böhm y Jacopini[1] demostraron que los programas podían escribirse *sin* instrucciones `goto`. El reto de la época para los programadores fue cambiar sus estilos a una "programación sin `goto`". No fue sino hasta la década de 1970 cuando los programadores tomaron en serio la programación estructurada. Los resultados han sido impresionantes, ya que los grupos de desarrollo de software han reportado reducciones en los tiempos de desarrollo, mayor incidencia de entregas de sistemas a tiempo y más proyectos de software finalizados sin salirse del presupuesto. La clave para estos logros es que los programas estructurados son más claros, más fáciles de depurar, probar y modificar, y hay más probabilidad de que estén libres de errores desde el principio.

El trabajo de Böhm y Jacopini demostró que todos los programas podían escribirse en términos de tres **estructuras de control** solamente: la **de secuencia**, la **de selección** y la **de repetición**. El término "estructuras de control" proviene del campo de las ciencias computacionales. Cuando presentemos las implementaciones en C++ de las estructuras de control, nos referiremos a ellas en la terminología del documento del estándar de C++ como "instrucciones de control".

Estructura de secuencia en C++

La *estructura de secuencia* está integrada en C++. A menos que se le indique lo contrario, la computadora ejecuta las instrucciones en C++ una después de otra, en el orden en que estén escritas; es decir, en secuencia. El **diagrama de actividad** en UML de la figura 4.2 ilustra una estructura de secuencia típica,

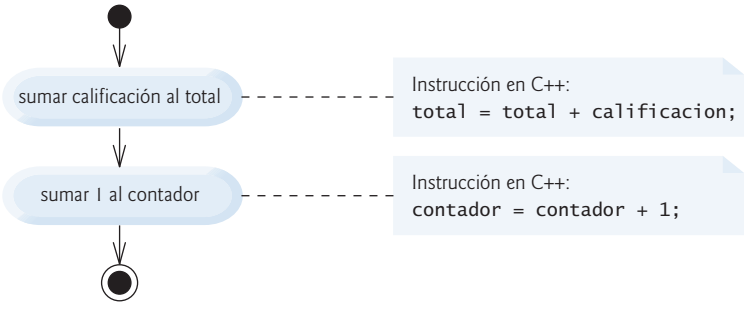


Fig. 4.2 | Diagrama de actividad de una estructura de secuencia.

1 Böhm, C. y G. Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules", *Communications of the ACM*, vol. 9, No. 5, mayo de 1996, páginas 366-371.

en la que se realizan dos cálculos en orden. C++ permite tantas acciones como deseemos en una estructura de secuencia. Como veremos pronto, en donde quiera que se coloque *una sola acción*, podrán colocarse *varias acciones en secuencia*.

En esta figura, las dos instrucciones implican sumar una calificación a una variable llamada `total`, y sumar el valor `1` a una variable llamada `contador`. Dichas instrucciones podrían aparecer en un programa que obtenga el promedio de varias calificaciones de estudiantes. Para calcular un promedio, el *total de las calificaciones* que se van a promediar se divide entre el *número de calificaciones*. Podría usarse una variable contador para llevar la cuenta del número de valores a promediar. En el programa de la sección 4.8 verá instrucciones similares.

Un diagrama de actividad modela el **flujo de trabajo** (también conocido como la **actividad**) de una parte de un sistema de software. Dichos flujos de trabajo pueden incluir una porción de un algoritmo, como la estructura de secuencia de la figura 4.2. Los diagramas de actividad están compuestos por símbolos de propósito especial, como los **símbolos de estado de acción** (un rectángulo cuyo lado izquierdo y derecho se reemplazan con arcos hacia fuera), **rombos** (**diamantes**) y **pequeños círculos**; estos símbolos se conectan mediante **flechas de transición**, que representan el flujo de la actividad.

Los diagramas de actividad muestran claramente cómo operan las estructuras de control. Considere el diagrama de actividad para la estructura de secuencia de la figura 4.2. Este diagrama contiene dos **estados de acción** que representan las acciones a realizar. Cada estado de acción contiene una **expresión de acción** (por ejemplo, "sumar calificación a total" o "sumar 1 al contador"), que especifica una acción particular a realizar. Otras acciones podrían incluir cálculos u operaciones de entrada/salida. Las flechas en el diagrama de actividad se llaman flechas de transición. Estas flechas representan **transiciones**, las cuales indican el *orden* en el que ocurren las acciones representadas por los estados de acción; el programa que implementa las actividades ilustradas por el diagrama de actividad de la figura 4.2 primero suma `calificacion` a `total`, y después suma `1` a `contador`.

El **círculo relleno** que se encuentra en la parte superior del diagrama de actividad representa el **estado inicial** de la actividad: el *inicio* del flujo de trabajo *antes* de que el programa realice las actividades modeladas. El círculo sólido rodeado por una circunferencia que aparece en la parte inferior del diagrama de actividad representa el **estado final**; es decir, el *final* del flujo de trabajo *después* de que el programa realiza sus actividades.

La figura 4.2 también incluye rectángulos que tienen la esquina superior derecha doblada. En UML, a estos rectángulos se les llama **notas**: comentarios con explicaciones que describen el propósito de los símbolos en el diagrama.

La figura 4.2 utiliza las notas de UML para mostrar el código en C++ asociado con cada uno de los estados de acción en el diagrama de actividad. Una **línea punteada** conecta cada nota con el elemento que ésta describe. Por lo general, los diagramas de actividad no muestran el código de C++ que implementa la actividad. En este libro utilizamos las notas con este propósito, para mostrar cómo se relaciona el diagrama con el código en C++. Para obtener más información sobre UML, vea nuestro caso de estudio opcional (pero ampliamente recomendado), que aparece en los capítulos 25 y 26, y visite nuestro Centro de recursos sobre UML en `www.deitel.com/UML/`.

Instrucciones de selección en C++

C++ tiene tres tipos de instrucciones de selección (las cuales se describen en este capítulo y en el siguiente). La instrucción de selección `if` realiza (selecciona) una acción si una condición es verdadera, o evita la acción si la condición es falsa. La instrucción de selección `if...else` realiza una acción si una condición es verdadera, o realiza una acción distinta si la condición es falsa. La instrucción de selección `switch` (capítulo 5) realiza una de entre *varias* acciones distintas, dependiendo del valor de una expresión entera.

La instrucción de selección `if` es una **instrucción de selección simple**, ya que selecciona o ignora una *sola acción* (o, como pronto veremos, un *solo grupo de acciones*). La instrucción `if...else` se conoce como **instrucción de selección doble**, ya que selecciona entre dos acciones distintas (o grupos de acciones). La instrucción `switch` es una **estructura de selección múltiple**, ya que selecciona entre muchas acciones distintas (o grupos de acciones).

Instrucciones de repetición en C++

C++ cuenta con tres tipos de instrucciones de repetición (también llamadas **instrucciones de ciclo** o **ciclos**) para ejecutar instrucciones en forma repetida, siempre y cuando una condición (llamada la **condición de continuación del ciclo**) siga siendo verdadera. Éstas son las instrucciones **`while`**, **`do...while`** y **`for`** (en el capítulo 5 presentaremos las instrucciones `do...while` y `for`, y en el capítulo 7 presentaremos una versión especializada de la instrucción `for`, que se utiliza con lo que se conoce como arreglos y contenedores). Las instrucciones `while` y `for` realizan la acción (o grupo de acciones) en sus cuerpos, cero o más veces; si la condición de continuación del ciclo es inicialmente falsa, *no* se ejecutará la acción (o grupo de acciones). La instrucción `do...while` realiza la acción (o grupo de acciones) en su cuerpo, *por lo menos una vez*.

Las palabras `if`, `else`, `switch`, `while`, `do` y `for` son palabras clave en C++. Las palabras clave *no pueden* usarse como identificadores, como los nombres de variables, y deben escribirse sólo en minúsculas. En la figura 4.3 aparece una lista completa de las palabras clave en C++.

Palabras clave de C++

Palabras clave comunes para los lenguajes de programación C y C++

`auto`	`break`	`case`	`char`	`const`
`continue`	`default`	`do`	`double`	`else`
`enum`	`extern`	`float`	`for`	`goto`
`if`	`int`	`long`	`register`	`return`
`short`	`signed`	`sizeof`	`static`	`struct`
`switch`	`typedef`	`union`	`unsigned`	`void`
`volatile`	`while`			

Palabras clave sólo de C++

`and`	`and_eq`	`asm`	`bitand`	`bitor`
`bool`	`catch`	`class`	`compl`	`const_cast`
`delete`	`dynamic_cast`	`explicit`	`export`	`false`
`friend`	`inline`	`mutable`	`namespace`	`new`
`not`	`not_eq`	`operator`	`or`	`or_eq`
`private`	`protected`	`public`	`reinterpret_cast`	`static_cast`
`template`	`this`	`throw`	`true`	`try`
`typeid`	`typename`	`using`	`virtual`	`wchar_t`
`xor`	`xor_eq`			

Palabras clave de C++11

`alignas`	`alignof`	`char16_t`	`char32_t`	`constexpr`
`decltype`	`noexcept`	`nullptr`	`static_assert`	`thread_local`

Fig. 4.3 | Palabras clave de C++.

Resumen de las instrucciones de control en C++

C++ sólo tiene tres tipos de estructuras de control, a las cuales nos referiremos de aquí en adelante como instrucciones de control: la instrucción de secuencia, las instrucciones de selección (tres tipos: `if`, `if...else` y `switch`) y las instrucciones de repetición (tres tipos: `while`, `for` y `do...while`). Cada programa combina tantas de estas instrucciones de control como sea apropiado para el algoritmo que implemente el programa. Podemos modelar cada una de las instrucciones de control como un diagrama de actividad, con estados iniciales y finales que representan los puntos de entrada y salida de la instrucción de control, respectivamente. Estas **instrucciones de control de una sola entrada/una sola salida** facilitan la creación de programas; las instrucciones de control están unidas entre sí mediante la conexión del punto de salida de una instrucción de control, al punto de entrada de la siguiente. Este procedimiento es similar a la manera en que un niño apila los bloques de construcción, así que a esto le llamamos **apilamiento de instrucciones de control**. En breve aprenderemos que sólo hay una manera alternativa de conectar las instrucciones de control: el **anidamiento de instrucciones de control**, en el cual una instrucción de control aparece *dentro* de otra.

Observación de Ingeniería de Software 4.1

Cualquier programa de C++ puede construirse a partir de sólo siete tipos distintos de instrucciones de control (secuencia, `if`, `if...else`, `switch`, `while`, `do...while` y `for`), combinadas en sólo dos formas (apilamiento de instrucciones de control y anidamiento de instrucciones de control).

4.5 Instrucción de selección `if`

Los programas utilizan instrucciones de selección para elegir entre los cursos alternativos de acción. Por ejemplo, suponga que la calificación para aprobar un examen es de 60. La instrucción en seudocódigo

```
Si la calificación del estudiante es mayor o igual a 60
    Imprimir "Aprobado"
```

determina si la condición "la calificación del estudiante es mayor o igual a 60" es verdadera (`true`) o falsa (`false`). Si la condición es verdadera se imprime "Aprobado", y se "ejecuta" en orden la siguiente instrucción en seudocódigo (recuerde que el seudocódigo no es un verdadero lenguaje de programación). Si la condición es falsa se ignora la instrucción para imprimir, y se ejecuta en orden la siguiente instrucción en seudocódigo. La sangría de la segunda línea es opcional, pero se recomienda ya que enfatiza la estructura inherente de los programas estructurados.

La instrucción anterior *if* en seudocódigo puede escribirse en C++ de la siguiente manera:

```
if ( calificacion >= 60 )
   cout << "Aprobado";
```

El código en C++ corresponde en gran medida con el seudocódigo. Ésta es una de las propiedades que hace del seudocódigo una herramienta de desarrollo de programas tan útil.

Es importante mencionar aquí que estamos suponiendo por casualidad que `calificacion` contiene un valor válido: un entero en el rango de 0 a 100. A lo largo del libro presentaremos muchas técnicas de validación importantes.

Tip para prevenir errores 4.1

En el código que se utiliza en la industria, siempre hay que validar todas las entradas.

La figura 4.4 muestra la instrucción `if` de selección simple. Esta figura contiene lo que quizá sea *el* símbolo más importante en un diagrama de actividad: el rombo o **símbolo de decisión**, el cual indica que se va a tomar una *decisión*. Un símbolo de decisión indica que el flujo de trabajo continuará a lo largo de una ruta determinada por las **condiciones de guardia** asociadas de ese símbolo, que pueden ser verdaderas o falsas. Cada flecha de transición que sale de un símbolo de decisión tiene una condición de guardia especificada entre *corchetes*, por encima o a un lado de la flecha de transición. Si cierta condición de guardia es verdadera, el flujo de trabajo entra al estado de acción al que apunta la flecha de transición. En la figura 4.4, si la calificación es mayor o igual a 60, el programa imprime "Aprobado" en la pantalla, y luego se dirige al estado final de esta actividad. Si la calificación es menor a 60, el programa se dirige inmediatamente al estado final sin mostrar ningún mensaje.

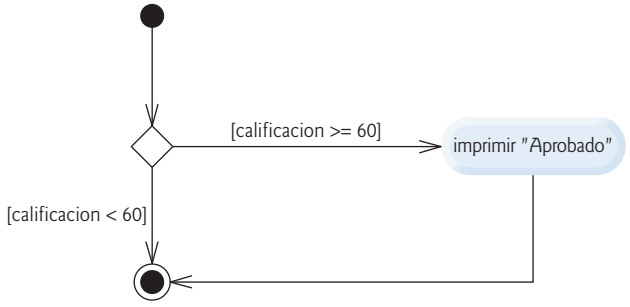


Fig. 4.4 | Diagrama de actividad de la instrucción `if` de selección simple.

En el capítulo 2 aprendimos que las decisiones se pueden basar en condiciones que contengan operadores de igualdad o relacionales. En realidad, en C++ una decisión se puede basar en *cualquier* expresión; si la expresión se evalúa como *cero*, se considera como *falsa*; si la expresión se evalúa como un *número distinto de cero*, se considera verdadera. C++ proporciona el tipo de datos `bool` para las variables que sólo pueden contener los valores `true` y `false`; cada una de éstas es una palabra clave de C++.

Tip de portabilidad 4.1

Para tener compatibilidad con versiones anteriores de C, en las que se utilizan enteros para los valores booleanos, el valor `bool true` *también se puede representar mediante cualquier valor* distinto de cero *(por lo general, los compiladores utilizan* `1`*) y el valor* `bool false` *también se puede representar como el valor cero.*

La instrucción `if` es una instrucción de *una sola entrada/una sola salida*. Pronto veremos que los diagramas de actividad para las instrucciones de control restantes también contienen estados iniciales, flechas de transición, estados de acción que indican las acciones a realizar, símbolos de decisión (con sus condiciones de guardia asociadas) que indican las decisiones a tomar, y estados finales.

Imagine siete cajones, en donde cada uno contiene diagramas de actividad de UML vacíos de uno de los siete tipos de instrucciones de control. Su tarea es ensamblar un programa a partir de los diagramas de actividad de tantas instrucciones de control de cada tipo como lo requiera el algoritmo, combinando esas instrucciones de control en sólo dos formas posibles (apilando o anidando), y después llenando los estados de acción y las decisiones con expresiones de acción y condiciones de guardia, en una manera que sea apropiada para formar una implementación estructurada para el algoritmo. Seguiremos hablando sobre la variedad de formas en que pueden escribirse las acciones y las decisiones.

4.6 Instrucción de selección doble `if...else`

La instrucción `if` de selección simple realiza una acción indicada solamente cuando la condición es verdadera (`true`); de no ser así, se evita dicha acción. La instrucción `if...else` de selección doble permite al programador especificar una acción a realizar cuando la condición es verdadera, y otra *distinta* cuando la condición es falsa (`false`). Por ejemplo, la instrucción en seudocódigo

```
Si la calificación del estudiante es mayor o igual a 60
    Imprimir "Aprobado"
De lo contrario
    Imprimir "Reprobado"
```

imprime "Aprobado" si la calificación del estudiante es mayor o igual a 60, e imprime "Reprobado" si la calificación del estudiante es menor a 60. En cualquier caso, después de que ocurre la impresión se "ejecuta" la siguiente instrucción en seudocódigo en la secuencia.

La instrucción anterior *if...else* en seudocódigo puede escribirse en C++ como

```
if ( calificacion >= 60 )
   cout << "Aprobado";
else
   cout << "Reprobado";
```

El cuerpo de la instrucción `else` también tiene sangría.

Buena práctica de programación 4.1

Si hay varios niveles de sangría, en cada nivel debe aplicarse la misma cantidad de espacio adicional para promover la legibilidad y facilidad de mantenimiento.

La figura 4.5 muestra el flujo de control en la instrucción `if...else`.

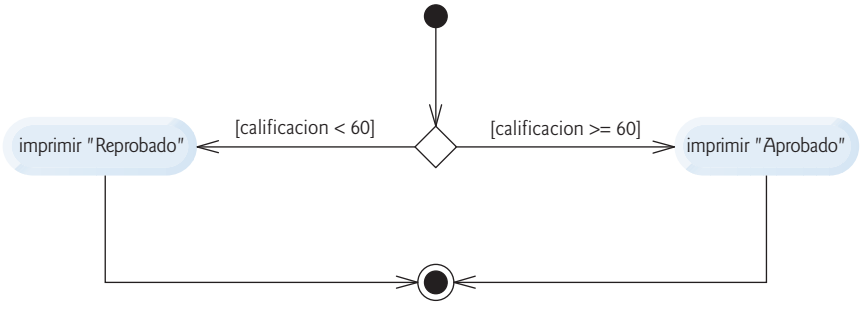


Fig. 4.5 | Diagrama de actividad de la instrucción `if...else` de selección doble.

Operador condicional (?:)

C++ cuenta con el **operador condicional** (`?:`), que está estrechamente relacionado con la instrucción `if...else`. Éste es el único **operador ternario** de C++: recibe tres operandos. En conjunto, los operan-

dos y el operador condicional forman una **expresión condicional**. El primer operando es una condición, el segundo es el valor de la expresión condicional si la condición es `true`, y el tercero es el valor de toda la expresión condicional si la condición es `false`. Por ejemplo, la instrucción de salida

```
cout << ( calificacion >= 60 ? "Aprobado" : "Reprobado" );
```

contiene una expresión condicional, `calificacion >= 60 ? "Aprobado" : "Reprobado"`, que se evalúa como la cadena `"Reprobado"` si la condición `calificacion >= 60` es `true`, pero se evalúa como la cadena `"Reprobado"` si la condición es `false`. Por lo tanto, la instrucción con el operador condicional realiza en esencia la misma función que la instrucción `if...else` anterior. Como veremos más adelante, la precedencia del operador condicional es baja, por lo cual los paréntesis en la expresión anterior son obligatorios.

Tip para prevenir errores 4.2

Para evitar problemas de precedencia (y por claridad), coloque las expresiones condicionales (que aparezcan en expresiones más grandes) entre paréntesis.

Los valores en una expresión condicional también pueden ser acciones a ejecutar. Por ejemplo, la siguiente expresión condicional también imprime `"Aprobado"` o `"Reprobado"`:

```
calificacion >= 60 ? cout << "Aprobado" : cout << "Reprobado";
```

La expresión condicional anterior se lee como: "si `calificacion` es mayor o igual que `60`, entonces `cout << "Aprobado"`; en caso contrario, `cout << "Reprobado"`". Esto también puede compararse con la instrucción `if...else` anterior. Las expresiones condicionales pueden aparecer en algunas partes de los programas en las que no se pueden utilizar instrucciones `if...else`.

Instrucciones `if...else` anidadas

Las **instrucciones `if...else` anidadas** pueden evaluar varios casos, al colocar instrucciones de selección `if...else` dentro de otras instrucciones `if...else`. Por ejemplo, la siguiente instrucción `if...else` en seudocódigo imprime `A` para las calificaciones de exámenes mayores o iguales a 90, `B` para las que están en el rango de 80 a 89, `C` en el rango de 70 a 79, `D` si están en el rango de 60 a 69 y `F` para todas las demás calificaciones:

```
Si la calificación del estudiante es mayor o igual a 90
    Imprimir "A"
de lo contrario
    Si la calificación del estudiante es mayor o igual a 80
        Imprimir "B"
    de lo contrario
        Si la calificación del estudiante es mayor o igual a 70
            Imprimir "C"
        de lo contrario
            Si la calificación del estudiante es mayor o igual a 60
                Imprimir "D"
            de lo contrario
                Imprimir "F"
```

Este seudocódigo puede escribirse en C++ como

```
if ( calificacionEstudiante >= 90 ) // 90 o más recibe una "A"
   cout << "A";
else
   if ( calificacionEstudiante >= 80 ) // 80 a 89 recibe una "B"
      cout << "B";
   else
      if ( calificacionEstudiante >= 70 ) // 70 a 79 recibe "C"
         cout << "C";
      else
         if ( calificacionEstudiante >= 60 ) // 60 a 69 recibe "D"
            cout << "D";
         else // menos de 60 recibe "F"
            cout << "F";
```

Si `calificacionEstudiante` es mayor o igual a 90, las primeras cuatro condiciones serán `true`, pero sólo se ejecutará la instrucción después de la primera prueba. Después, el programa evita la parte `else` de la instrucción `if...else` más "externa". La mayoría de los programadores escriben la instrucción `if...else` anterior así:

```
if ( calificacionEstudiante >= 90 ) // 90 o más recibe una "A"
   cout << "A";
else if ( calificacionEstudiante >= 80 ) // 80 a 89 recibe una "B"
   cout << "B";
else if ( calificacionEstudiante >= 70 ) // 70 a 79 recibe "C"
   cout << "C";
else if ( calificacionEstudiante >= 60 ) // 60 a 69 recibe "D"
   cout << "D";
else // menos de 60 recibe "F"
   cout << "F";
```

Las dos formas son idénticas, excepto por el espaciado y la sangría, que el compilador ignora. La segunda forma es más popular, ya que evita usar mucha sangría hacia la derecha en el código lo cual puede forzar a que las líneas se dividan.

Tip de rendimiento 4.1

Una instrucción `if...else` anidada puede ejecutarse con mucha más rapidez que una serie de instrucciones `if` de selección simple, debido a la posibilidad de salir antes de tiempo, una vez que se cumple una de las condiciones.

Tip de rendimiento 4.2

En una instrucción `if...else` anidada, debemos evaluar las condiciones que tengan más probabilidades de ser `true` al principio de la instrucción anidada. Esto permite que la instrucción `if...else` anidada se ejecute con más rapidez, al salir antes de lo esperado si se evaluaran primero los casos que ocurren con menos frecuencia.

Problema del `else` suelto

El compilador de C++ siempre asocia un `else` con el `if` que le *precede inmediatamente*, a menos que se le indique otra cosa mediante la colocación de llaves (`{` y `}`). Este comportamiento puede ocasionar lo que se conoce como el **problema del `else` suelto**. Por ejemplo,

```
if ( x > 5 )
   if ( y > 5 )
      cout << "x e y son > 5";
else
   cout << "x es <= 5";
```

parece indicar que si x es mayor que 5, la instrucción if anidada determina si y es también mayor que 5. De ser así, se produce como resultado la cadena "x y y son > 5". De lo contrario, parece ser que si x no es mayor que 5, la instrucción else que es parte del if...else produce como resultado la cadena "x es <= 5".

¡Cuidado! Esta instrucción if...else anidada no se ejecuta como parece ser. El compilador en realidad interpreta la instrucción así:

```
if ( x > 5 )
   if ( y > 5 )
      cout << "x y y son > 5";
   else
      cout << "x es <= 5";
```

en donde el cuerpo del primer if es un if...else anidado. La instrucción if más externa evalúa si x es mayor que 5. De ser así, la ejecución continúa evaluando si y es también mayor que 5. Si la segunda condición es verdadera, se muestra la cadena apropiada ("x y y son > 5"). No obstante, si la segunda condición es falsa se muestra la cadena "x es <= 5", aun y cuando sabemos que x es mayor que 5.

Para forzar a que la instrucción if...else anidada se ejecute como se tenía pensado originalmente, podemos escribirla de la siguiente manera:

```
if ( x > 5 )
{
   if ( y > 5 )
      cout << "x y y son > 5";
}
else
   cout << "x es <= 5";
```

Las llaves ({ }) indican al compilador que la segunda instrucción if se encuentra en el cuerpo del primer if, y que el else está asociado con el primer if. Los ejercicios 4.23 a 4.24 investigan con más detalle el problema del else suelto.

Bloques

La instrucción de selección if normalmente espera *sólo una* instrucción en su cuerpo. De manera similar, las partes if y else de una instrucción if...else esperan *sólo una* instrucción en su cuerpo. Para incluir *varias* instrucciones en el cuerpo de un if o en cualquier parte de un if...else, encierre las instrucciones entre llaves ({ y }). A un conjunto de instrucciones contenidas dentro de un par de llaves se le llama **instrucción compuesta** o **bloque**. De aquí en adelante utilizaremos el término "bloque".

Observación de Ingeniería de Software 4.2

Un bloque puede colocarse en cualquier parte de un programa en donde pueda colocarse una sola instrucción.

El siguiente ejemplo incluye un bloque en la parte else de una instrucción if...else:

```
if ( calificacionEstudiante >= 60 )
   cout << "Aprobado.\n";
else
{
   cout << "Reprobado.\n";
   cout << "Debe tomar este curso otra vez.\n";
}
```

En este caso, si `calificacionEstudiante` es menor que 60, el programa ejecuta *ambas* instrucciones en el cuerpo del `else` e imprimes

```
Reprobado.
Debe tomar este curso otra vez.
```

Observe las llaves que rodean a las dos instrucciones en la cláusula `else`. Estas llaves son importantes. Sin ellas, la instrucción

```
cout << "Debe tomar este curso otra vez.\n";
```

estaría *fuera* del cuerpo de la parte `else` de la instrucción `if` y se ejecutaría sin importar que la calificación fuera menor a 60. Éste es un error lógico.

Así como un bloque puede colocarse en cualquier parte en donde pueda colocarse una sola instrucción individual, también es posible no tener instrucción alguna; a ésta se le conoce como **instrucción nula** (o **instrucción vacía**). Para representar a la instrucción nula, se coloca un punto y coma (`;`) en donde normalmente iría una instrucción.

Error común de programación 4.1

Colocar un punto y coma después de la condición en una instrucción `if` produce un error lógico en las instrucciones `if` de selección simple, y un error de sintaxis en las instrucciones `if...else` de selección doble (cuando la parte del `if` contiene una instrucción en el cuerpo).

4.7 Instrucción de repetición `while`

Una **instrucción de repetición** especifica que un programa debe repetir una acción mientras cierta condición sea verdadera. La instrucción en seudocódigo

```
Mientras existan más artículos en mi lista de compras
    Comprar el siguiente artículo y quitarlo de mi lista
```

describe la repetición que ocurre durante una salida de compras. La *condición* "existan más artículos en mi lista de compras" puede ser verdadera o falsa. Si es verdadera, entonces se realiza la acción "Comprar el siguiente artículo y quitarlo de mi lista". Esta acción se realizará en forma repetida mientras la condición sea verdadera. La instrucción contenida en la instrucción de repetición *Mientras (while)* constituye el cuerpo de esta estructura, el cual puede ser una sola instrucción o un bloque. En algún momento, la condición se hará falsa (cuando el último artículo de la lista de compras sea adquirido y eliminado de la lista). En este punto la repetición terminará y se ejecutará la primera instrucción que esté después de la instrucción de repetición.

Como ejemplo de la instrucción de repetición `while` en C++, considere un segmento de programa diseñado para encontrar la primera potencia de 3 que sea mayor a 100. Suponga que la variable `producto` de tipo entero se inicializa en 3. Cuando la siguiente instrucción `while` termine de ejecutarse, `producto` contendrá el resultado:

```
int producto = 3;

while ( producto <= 100 )
   producto = 3 * producto;
```

Cuando esta instrucción `while` comienza a ejecutarse, el valor de `producto` es 3. Cada repetición de la instrucción `while` multiplica a `producto` por 3, por lo que `producto` toma los valores de 9, 27, 81 y 243,

sucesivamente. Cuando `producto` se vuelve 243, la condición de la instrucción `while (producto <= 100)` se torna falsa. Esto termina la repetición, por lo que el valor final de `producto` es 243. En este punto, la ejecución del programa continúa con la siguiente instrucción después de `while`.

Error común de programación 4.2

Un error lógico conocido como **ciclo infinito**, *en el que la instrucción de repetición nunca termina, ocurre si no se proporciona una acción en el cuerpo de una instrucción* `while` *que ocasione que en algún momento la condición del* `while` *se torne falsa por lo general. Esto puede hacer que un programa parezca "quedar colgado" o "congelarse" si el cuerpo del ciclo no contiene instrucciones que interactúen con el usuario.*

El diagrama de actividad de UML de la figura 4.6 muestra el flujo de control que corresponde a la instrucción `while` anterior. Una vez más, los símbolos en el diagrama (aparte del estado inicial, las flechas de transición, un estado final y tres notas) representan un estado de acción y una decisión. Este diagrama también introduce el **símbolo de fusión** de UML, el cual une dos flujos de actividad en un solo flujo de actividad. UML representa *tanto* al símbolo de fusión como al símbolo de decisión como rombos. En este diagrama, el símbolo de fusión une las transiciones del estado inicial y del estado de acción, de manera que *ambas* fluyan en la decisión que determina si el ciclo debe empezar a ejecutarse (o seguir ejecutándose). Los símbolos de decisión y de fusión pueden diferenciarse por el número de flechas de transición "entrantes" y "salientes". Un símbolo de decisión tiene una flecha de transición que apunta *hacia* el rombo y dos o más flechas que apuntan *hacia afuera* del rombo, para indicar las posibles transiciones desde ese punto. Además, cada flecha que apunta hacia fuera de un símbolo de decisión tiene una condición de guardia junto a ella. Un símbolo de fusión tiene dos o más flechas de transición que apuntan *hacia* el rombo, y sólo una que apunta *hacia fuera* del rombo, para indicar múltiples flujos de actividad que se fusionan para continuar la actividad. A diferencia del símbolo de decisión, el de fusión *no* tiene su contraparte en el código de C++.

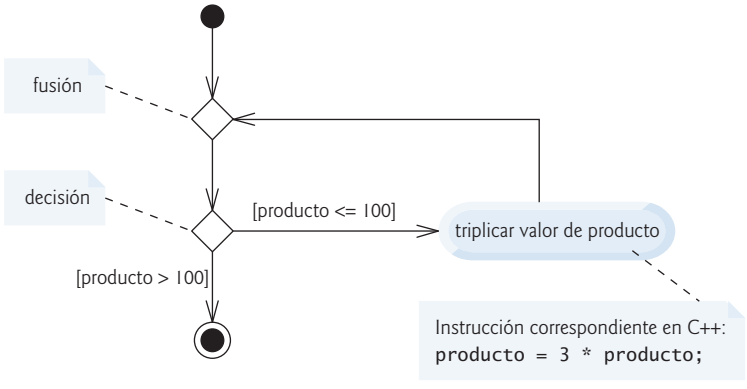


Fig. 4.6 | Diagrama de actividad de UML de la instrucción de repetición `while`.

El diagrama de la figura 4.6 muestra claramente la repetición de la instrucción `while` que vimos antes en esta sección. La flecha de transición que emerge del estado de acción apunta a la fusión, desde la cual regresa a la decisión que se evalúa en cada iteración del ciclo, hasta que la condición de guardia `producto > 100` se vuelva verdadera. Entonces, la instrucción `while` termina (llega a su estado final) y el control pasa a la siguiente instrucción en la secuencia del programa.

Tip de rendimiento 4.3

Una pequeña mejora en el rendimiento para el código que se ejecuta muchas veces en un ciclo puede producir una mejora considerable en el rendimiento en general.

4.8 Cómo formular algoritmos: repetición controlada por un contador

Para ilustrar la forma en que los programadores desarrollan los algoritmos, en esta sección y en la siguiente resolveremos dos variantes de un problema que promedia las calificaciones de una clase. Considere el siguiente enunciado del problema:

> *A una clase de diez estudiantes se les aplicó un examen. Las calificaciones (de 0 a 100) para este examen están disponibles para que usted las analice. Calcule y muestre el total de las calificaciones de todos los estudiantes y el promedio de la clase para este examen.*

El promedio de la clase es igual a la suma de las calificaciones, dividida entre el número de estudiantes. El algoritmo para resolver este problema en una computadora debe recibir como entrada cada una de las calificaciones, calcular el promedio e imprimir el resultado.

Algoritmo de seudocódigo con repetición controlada por un contador

Emplearemos seudocódigo para listar las *acciones* a ejecutar y especificar el *orden* en el que deben ocurrir. Usaremos una **repetición controlada por contador** para introducir las calificaciones, una por una. Esta técnica utiliza una variable llamada **contador** para controlar el número de veces que debe ejecutarse un conjunto de instrucciones (a lo cual se le conoce también como el número de **iteraciones** del ciclo).

A la repetición controlada por contador se le llama comúnmente **repetición definida**, ya que el número de repeticiones se conoce *antes* de que el ciclo empiece a ejecutarse. En este ejemplo, la repetición termina cuando el contador excede a 10. Esta sección presenta un algoritmo de seudocódigo completamente desarrollado (figura 4.7), y una versión de la clase `LibroCalificaciones` (figuras 4.8 y 4.9) que implementa el algoritmo en una función miembro de C++. Después presentamos una aplicación (figura 4.10) que demuestra el algoritmo en acción. En la sección 4.9 demostraremos cómo utilizar el seudocódigo para desarrollar dicho algoritmo desde cero.

Observación de Ingeniería de Software 4.3

La parte más difícil para la resolución de un problema en una computadora es desarrollar el algoritmo. El proceso de producir un programa funcional en C++ a partir de dicho algoritmo es, por lo general, relativamente sencillo.

```
1  Asignar a total el valor de cero
2  Asignar al contador de calificaciones el valor de uno
3
4  Mientras que el contador de calificaciones sea menor o igual a diez
5      Pedir al usuario que introduzca la siguiente calificación
6      Obtener como entrada la siguiente calificación
7      Sumar la calificación al total
8      Sumar uno al contador de calificaciones
9
10 Asignar al promedio de la clase el total dividido entre diez
11 Imprimir el total de las calificaciones para todos los estudiantes en la clase
12 Imprimir el promedio de la clase
```

Fig. 4.7 | Algoritmo en seudocódigo que utiliza la repetición controlada por contador para resolver el problema del promedio de una clase.

Observe las referencias en el algoritmo de seudocódigo de la figura 4.7 para un *total* y un *contador*. Un **total** es una variable que se utiliza para acumular la suma de varios valores. Un **contador** es una variable que se utiliza para contar; en este caso, el contador de calificaciones indica cuál de las 10 calificaciones está a punto de escribir el usuario. Por lo general, las variables que se utilizan para guardar totales deben inicializarse en cero antes de utilizarse en un programa; de lo contrario, la suma incluiría el valor anterior almacenado en la ubicación de memoria del total. Recuerde que en el capítulo 2 vimos que todas las variables deben inicializarse.

Mejora a la validación de `LibroCalificaciones`

Consideremos una mejora que hicimos a nuestra clase `LibroCalificaciones`. En la figura 3.16, la forma en que nuestra función miembro `establecerNombreCurso` validaría el nombre del curso sería probar primero si la longitud del mismo es menor o igual a 25 caracteres, mediante el uso de una instrucción `if`. Si esto fuera cierto, se establecería el nombre del curso. Después, a este código le seguiría otra instrucción `if` que evaluaría si la longitud del nombre del curso es mayor que 25 caracteres (en cuyo caso, el nombre del curso se reduciría). La condición de la segunda instrucción `if` es el opuesto exacto de la condición de la primera instrucción `if`. Si una condición se evalúa como `true`, la otra se debe evaluar como `false`. Dicha situación es ideal para una instrucción `if...else`, por lo que hemos modificado nuestro código, reemplazando las dos instrucciones `if` por una instrucción `if...else`, como se muestra en las líneas 18 a 25 de la figura 4.9).

Implementación de la repetición controlada por contador en la clase `LibroCalificaciones`

La clase `LibroCalificaciones` (figuras 4.8 y 4.9) contiene un constructor (declarado en la línea 10 de la figura 4.8 y definido en las líneas 9 a 12 de la figura 4.9) que asigna un valor al miembro de datos `nombreCurso` (declarado en la línea 16 de la figura 4.8) de la clase. En las líneas 16 a 26, 29 a 32 y 35 a 39 de la figura 4.9 se definen las funciones miembro `establecerNombreCurso`, `obtenerNombreCurso` y `mostrarMensaje`, respectivamente. En las líneas 42 a 64 se define la función miembro `determinarPromedioClase`, la cual implementa el algoritmo para sacar el promedio de la clase, descrito por el seudocódigo de la figura 4.7.

```
1  // Fig. 4.8: LibroCalificaciones.h
2  // Definición de la clase LibroCalificaciones que determina el promedio de una
     clase.
3  // Las funciones miembro se definen en LibroCalificaciones.cpp
4  #include <string> // el programa usa la clase string estándar de C++
5
6  // definición de la clase LibroCalificaciones
7  class LibroCalificaciones
8  {
9  public:
10    explicit LibroCalificaciones( std::string ); // inicializa el nombre del curso
11    void establecerNombreCurso( std::string ); // establece el nombre del curso
12    std::string obtenerNombreCurso() const; // obtener el nombre del curso
13    void mostrarMensaje() const; // muestra un mensaje de bienvenida
14    void determinarPromedioClase() const; // promedia las calificaciones escritas
                                             por el usuario
15 private:
16    std::string nombreCurso; // nombre del curso para este LibroCalificaciones
17 }; // fin de la clase LibroCalificaciones
```

Fig. 4.8 | Problema del promedio de una clase utilizando la repetición controlada por contador: encabezado de `LibroCalificaciones`.

```
1  // Fig. 4.9: LibroCalificaciones.cpp
2  // Definiciones de funciones miembro para la clase LibroCalificaciones que resuelve
3  // el programa del promedio de la clase con repetición controlada por contador.
4  #include <iostream>
5  #include "LibroCalificaciones.h" // incluye la definición de la clase
                                    LibroCalificaciones
6  using namespace std;
7
8  // el constructor inicializa a nombreCurso con la cadena que se suministra como
      argumento
9  LibroCalificaciones::LibroCalificaciones( string nombre )
10 {
11    establecerNombreCurso( nombre ); // valida y almacena nombreCurso
12 } // fin del constructor de LibroCalificaciones
13
14 // función para establecer el nombre del curso;
15 // asegura que el nombre del curso tenga cuando menos 25 caracteres
16 void LibroCalificaciones::establecerNombreCurso( string nombre )
17 {
18    if ( nombre.size() <= 25 ) // si nombre tiene 25 caracteres o menos
19       nombreCurso = nombre; // almacena el nombre del curso en el objeto
20    else // si nombre es mayor de 25 caracteres
21    { // establece nombreCurso a los primeros 25 caracteres del parámetro nombre
22       nombreCurso = nombre.substr( 0, 25 ); // selecciona los primeros 25
                                                 caracteres
23       cerr << "El nombre \"" << nombre << "\" excede la longitud maxima (25).\n"
24          << "Se limito nombreCurso a los primeros 25 caracteres.\n" << endl;
25    } // fin de if...else
26 } // fin de la función establecerNombreCurso
27
28 // función para obtener el nombre del curso
29 string LibroCalificaciones::obtenerNombreCurso() const
30 {
31    return nombreCurso;
32 } // fin de la función obtenerNombreCurso
33
34 // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
35 void LibroCalificaciones::mostrarMensaje() const
36 {
37    cout << "Bienvenido al libro de calificaciones para \n"
      << obtenerNombreCurso() << "!\n"
38       << endl;
39 } // fin de la función mostrarMensaje
40
41 // determina el promedio de la clase, con base en las 10 calificaciones escritas
      por el usuario
42 void LibroCalificaciones::determinarPromedioClase() const
43 {
44    // fase de inicialización
45    int total = 0; // suma de las calificaciones introducidas por el usuario
46    unsigned int contadorCalif = 1; // número de la calificación a introducir a
                                      continuación
47
48    // fase de procesamiento
49    while ( contadorCalif <= 10 ) // itera 10 veces
50    {
51       cout << "Escriba una calificacion: "; // pide la entrada
```

Fig. 4.9 | Problema del promedio de una clase utilizando la repetición controlada por contador: archivo de código fuente de `LibroCalificaciones` (parte 1 de 2).

```
52        int calificacion = 0; // valor de la calificación introducida por el usuario
53        cin >> calificacion; // recibe como entrada la siguiente calificación
54        total = total + calificacion; // suma la calificación al total
55        contadorCalif = contadorCalif + 1; // incrementa el contador por 1
56     } // fin de while
57
58     // fase de terminación
59     int promedio = total / 10; // está bien mezclar la declaración con el cálculo
60
61     // muestra el total y el promedio de las calificaciones
62     cout << "\nEl total de las 10 calificaciones es " << total << endl;
63     cout << "El promedio de la clase es " << promedio << endl;
64  } // fin de la función determinarPromedioClase
```

Fig. 4.9 | Problema del promedio de una clase utilizando la repetición controlada por contador: archivo de código fuente de `LibroCalificaciones` (parte 2 de 2).

Como la variable `contadorCalif` (figura 4.9, línea 46) se usa para contar de 1 a 10 en este programa (todos son valores positivos), declaramos la variable como **`unsigned int`**, que puede almacenar sólo valores *no negativos* (es decir, de 0 en adelante). Las variables locales `total` (figura 4.9, línea 45), `calificacion` (línea 52) y `promedio` (línea 59) son de tipo `int`. La variable `calificacion` almacena la entrada del usuario. Observe que las declaraciones anteriores aparecen en el cuerpo de la función miembro `determinarPromedioClase`. Además, la variable `calificacion` se declara en el cuerpo de la instrucción `while` debido a que se utiliza *sólo* en el ciclo; en general, hay que declarar las variables justo antes de que se utilicen. Inicializamos `calificacion` con `0` (línea 52) como una buena práctica, incluso aunque se introduzca de inmediato un nuevo valor para la calificación en la línea 53.

Buena práctica de programación 4.2

Declare cada variable en una línea separada con su propio comentario, para mejorar la legibilidad.

En las versiones de la clase `LibroCalificaciones` de este capítulo, simplemente leemos y procesamos un conjunto de calificaciones. El cálculo del promedio se realiza en la función miembro `determinarPromedioClase`, usando *variables locales*; no preservamos información acerca de las calificaciones de los estudiantes en los miembros de datos de la clase. En el capítulo 7, modificaremos la clase `LibroCalificaciones` para mantener las calificaciones en memoria, utilizando un miembro de datos que hace referencia a una estructura de datos conocida como *arreglo*. Esto permite que un objeto `LibroCalificaciones` realice varios cálculos sobre un conjunto de calificaciones, sin requerir que el usuario escriba las calificaciones varias veces.

En las líneas 45 y 46 se inicializan `total` a `0` y `contadorCalif` a `1` antes de usarse en los cálculos. Por lo general, las variables contador se inicializan con cero o uno, dependiendo de su uso en un algoritmo. Una variable no inicializada contiene un **valor "basura"** (también conocido como **valor indefinido**): el último valor almacenado en la ubicación de memoria reservada para esa variable.

Tip para prevenir errores 4.3

Inicialice siempre las variables al momento de declararlas. Esto le ayudará a evitar los errores lógicos que ocurren al realizar cálculos con variables sin inicializar.

Tip para prevenir errores 4.4

En algunos casos, los compiladores emiten una advertencia *si usted intenta usar el valor de una variable sin inicializar. Siempre hay que obtener una* compilación limpia, *al resolver* todos *los errores y advertencias.*

La línea 49 indica que la instrucción `while` debe continuar ejecutando el ciclo (lo que también se conoce como **iterar**), siempre y cuando el valor de `contadorCalif` sea menor o igual a `10`. Mientras esta condición sea verdadera, la instrucción `while` ejecutará en forma repetida las instrucciones entre las llaves que delimitan su cuerpo (líneas 49 a 56).

En la línea 51 se muestra el indicador `"Escriba una calificacion:"`. Esta línea corresponde a la instrucción en seudocódigo "*Pedir al usuario que introduzca la siguiente calificación*". En la línea 53 se lee la calificación escrita por el usuario y se asigna a la variable `calificacion`. Esta línea corresponde a la instrucción en seudocódigo "*Obtener como entrada la siguiente calificación*". En la línea 54 se suma la nueva `calificacion` escrita por el usuario al `total`, y se asigna el resultado a `total`, que *sustituye* su valor anterior.

En la línea 55 se suma `1` a `contadorCalif` para indicar que el programa ha procesado la calificación actual y está listo para recibir la siguiente calificación del usuario. Al incrementar a `contadorCalif` en cada iteración, en un momento dado su valor excederá a `10`. En ese momento, el ciclo `while` termina debido a que su condición (línea 49) se vuelve falsa.

Cuando el ciclo termina, en la línea 59 se realiza el cálculo del promedio y se asigna su resultado a la variable `promedio`. En la línea 62 se muestra el texto `"El total de las 10 calificaciones es "`, seguido del valor de la variable `total`. Después, en la línea 63 se muestra el texto `"El promedio de la clase es "`, seguido del valor de la variable `promedio`. A continuación, la función miembro `determinarPromedioClase` devuelve el control a la función que hizo la llamada (es decir, a `main` en la figura 4.10).

Demostración de la clase `LibroCalificaciones`

La figura 4.10 contiene la función `main` de esta aplicación, la cual crea un objeto de la clase `LibroCalificaciones` y demuestra sus capacidades. En la línea 9 de la figura 4.10 se crea un nuevo objeto `LibroCalificaciones` llamado `miLibroCalificaciones`. La cadena en la línea 9 se pasa al constructor de `LibroCalificaciones` (líneas 9 a 12 de la figura 4.9). En la línea 11 de la figura 4.10 se hace una llamada a la función miembro `mostrarMensaje` de `miLibroCalificaciones` para mostrar un mensaje de bienvenida al usuario. Después, en la línea 12 se hace una llamada a la función miembro `determinarPromedioClase` de `miLibroCalificaciones` para permitir que el usuario introduzca 10 calificaciones, para las cuales la función miembro posteriormente calcula e imprime el promedio; la función miembro ejecuta el algoritmo que se muestra en el seudocódigo de la figura 4.7.

```
1  // Fig. 4.10: fig04_10.cpp
2  // Crea un objeto LibroCalificaciones e invoca a su función
      determinarPromedioClase.
3  #include "LibroCalificaciones.h" // incluye la definición de la clase
                                    LibroCalificaciones
4
5  int main()
6  {
7     // crea un objeto LibroCalificaciones llamado miLibroCalificaciones y
8     // pasa el nombre del curso al constructor
9     LibroCalificaciones miLibroCalificaciones( "CS101 Programacion en C++" );
10
11    miLibroCalificaciones.mostrarMensaje(); // muestra el mensaje de bienvenida
12    miLibroCalificaciones.determinarPromedioClase();
      // busca el promedio de 10 calificaciones
13 } // fin de main
```

```
Bienvenido al libro de calificaciones para
CS101 Programacion en C++
```

Fig. 4.10 | Problema del promedio de una clase utilizando la repetición controlada por contador: creación de un objeto de la clase `LibroCalificaciones` (figuras 4.8 y 4.9) e invocación de su función miembro `determinarPromedioClase` (parte 1 de 2).

```
Escriba una calificacion: 67
Escriba una calificacion: 78
Escriba una calificacion: 89
Escriba una calificacion: 67
Escriba una calificacion: 87
Escriba una calificacion: 98
Escriba una calificacion: 93
Escriba una calificacion: 85
Escriba una calificacion: 82
Escriba una calificacion: 100

El total de las 10 calificaciones es 846
El promedio de la clase es 84
```

Fig. 4.10 | Problema del promedio de una clase utilizando la repetición controlada por contador: creación de un objeto de la clase `LibroCalificaciones` (figuras 4.8 y 4.9) e invocación de su función miembro `determinarPromedioClase` (parte 2 de 2).

Observaciones acerca de la división de enteros y el truncamiento

El cálculo del promedio realizado en respuesta a la llamada a la función en la línea 12 de la figura 4.10, produce un resultado entero. La ejecución de ejemplo indica que la suma de los valores de las calificaciones es 846, que al dividirse entre 10, debe producir 84.6; un número con un punto decimal. Sin embargo, el resultado del cálculo `total / 10` (línea 59 de la figura 4.9) es el entero 84, ya que `total` y 10 son enteros. Al dividir dos enteros se produce una división entera: se **trunca** cualquier parte fraccionaria del cálculo (es decir, se descarta). En la siguiente sección veremos cómo obtener un resultado que incluye un punto decimal a partir del cálculo del promedio.

Error común de programación 4.3

Asumir que la división entera redondea (en vez de truncar) puede producir resultados erróneos. Por ejemplo, 7 ÷ 4 produce 1.75 en la aritmética convencional, pero trunca la parte de punto flotante (.75) en la aritmética entera. Por lo tanto, el resultado es 1. De manera similar, –7 ÷ 4 produce –1.

En la figura 4.9, si en la línea 59 se utilizará `contadorCalif` en vez de 10, el resultado para este programa mostraría un valor incorrecto, 76. Esto ocurriría debido a que en la iteración final de la instrucción `while`, `contadorCalif` se incrementó al valor 11 en la línea 55.

Error común de programación 4.4

*El uso de una variable de control tipo contador de un ciclo en un cálculo después del ciclo produce un error lógico, conocido como **error de desplazamiento en 1**. En un ciclo controlado por contador que cuenta en uno cada vez que recorre el ciclo, éste termina cuando el valor del contador es uno más que su último valor legítimo (es decir, 11 en el caso de contar del 1 al 10).*

Observaciones sobre el desbordamiento aritmético

En la figura 4.9, la línea 54

```
total = total + calificacion; // suma la calificación al total
```

sumó cada `calificacion` introducida por el usuario al `total`. Incluso esta instrucción simple tiene un problema *potencial*: sumar los enteros podría producir un valor que sea *demasiado grande* como para almacenarlo en una variable `int`. Esto se conoce como **desbordamiento aritmético** y provoca un

comportamiento indefinido, que puede producir resultados inesperados (`en.wikipedia.org/wiki/Integer_overflow#Secutiry-ramifications`). El programa de suma de la figura 2.5 tiene el mismo problema en la línea 19, que calcula la suma de dos valores `int` introducidos por el usuario:

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

Los valores máximo y mínimo que pueden almacenarse en una variable `int` se representan mediante las constantes `INT_MAX` e `INT_MIN`, respectivamente, que se definen en el encabezado `<climits>`. Hay constantes similares para los otros tipos enteros y para los tipos de punto flotante. Puede ver los valores de su plataforma para estas constantes si abre los encabezados `<climits>` y `<cfloat>` en un editor de texto (puede buscar estos archivos en su sistema de archivos).

Se considera una buena práctica asegurarse de que, *antes* de realizar cálculos aritméticos como los de la línea 54 de la figura 4.9 y la línea 19 de la figura 2.5, *no* se desborden. El código para esto se muestra en el sitio Web de CERT `www.securecoding.cert.org`; sólo tiene que buscar el lineamiento "INT32-CPP". El código utiliza los operadores `&&` (AND lógico) y `||` (OR lógico), que se introducen en el capítulo 5. En el código de calidad industrial, hay que realizar revisiones como éstas para *todos* los cálculos.

Un análisis más detallado del proceso de recibir los datos de entrada del usuario

Cada vez que un programa recibe la entrada del usuario, podrían ocurrir varios problemas. Por ejemplo, en la línea 53 de la figura 4.9

```
cin >> calificacion; // recibe como entrada la siguiente calificacion
```

asumimos que el usuario introducirá una calificación entera en el rango de 0 a 100. Sin embargo, la persona que introduce una calificación podría introducir un entero menor a 0, un entero mayor a 100, un entero fuera del rango de valores que pueden almacenarse en una variable `int`, un número que contenga un punto decimal o un valor que contenga letras o símbolos especiales que ni siquiera sea un entero.

Para asegurar que la entrada del usuario sea válida, los programas de calidad industrial deben probar todos los casos erróneos posibles. A medida que avance por el libro, aprenderá diversas técnicas para lidiar con el amplio rango de posibles problemas de entrada.

4.9 Cómo formular algoritmos: repetición controlada por un centinela

Generalicemos el problema para los promedios de una clase. Considere el siguiente problema:

> *Desarrollar un programa que calcule el promedio de una clase y que procese las calificaciones para un número* arbitrario *de estudiantes cada vez que se ejecute.*

En el ejemplo anterior, el enunciado del problema especificó el número de estudiantes, por lo que se conocía el número de calificaciones (10) por adelantado. En este ejemplo no se indica cuántas calificaciones va a introducir el usuario durante la ejecución del programa. El programa debe procesar un número *arbitrario* de calificaciones. ¿Cómo puede el programa determinar cuándo terminar de introducir calificaciones? ¿Cómo sabrá cuándo calcular e imprimir el promedio de la clase?

Para resolver este problema podemos utilizar un valor especial denominado **valor centinela** (también llamado **valor de señal**, **valor de prueba** o **valor de bandera**) para indicar el "fin de la introducción de datos". Después de escribir las calificaciones que desea, el usuario escribe el valor centinela para indicar que se introdujo la última calificación. A la repetición controlada por centinela a menudo se le llama **repetición indefinida**, ya que el número de repeticiones *no* se conoce antes de que comience la ejecución del ciclo.

Debe elegirse un valor centinela de tal forma que no pueda confundirse con un valor de entrada permitido. Por lo general, las calificaciones de un examen son enteros positivos, por lo que –1 es un valor centinela aceptable para este problema. Por lo tanto, una ejecución del programa podría procesar una cadena de entradas como 95, 96, 75, 74, 89 y –1. El programa entonces calcularía e imprimiría el promedio de la clase para las calificaciones 95, 96, 75, 74 y 89. Como –1 es el valor centinela, no debe entrar en el cálculo del promedio.

Desarrollo del algoritmo en seudocódigo con el método de mejoramiento de arriba a abajo, paso a paso: la primera mejora (cima)

Vamos a desarrollar el programa para promediar clases con una técnica llamada **mejoramiento de arriba a abajo, paso a paso**, la cual es esencial para el desarrollo de programas bien estructurados. Comenzamos con una representación en seudocódigo de la **cima**, una sola instrucción que transmite la función del programa en general:

Determinar el promedio de la clase para el examen, para un número arbitrario de estudiantes

La cima es, en efecto, la representación *completa* de un programa. Desafortunadamente, la cima (como en este caso) pocas veces transmite los detalles suficientes como para escribir un programa. Por lo tanto, ahora comenzaremos el proceso de mejora. Dividiremos la cima en una serie de tareas más pequeñas y las enlistaremos en el orden en el que se van a realizar. Esto arroja como resultado la siguiente **primera mejora**:

Inicializar variables
Introducir, sumar y contar las calificaciones del examen
Calcular e imprimir el total de las calificaciones de todos los estudiantes y el promedio de la clase

Esta mejora utiliza sólo la estructura de secuencia; estos pasos se ejecutan en orden.

Observación de Ingeniería de Software 4.4

Cada mejora, así como la cima en sí, es una especificación completa del algoritmo; sólo varía el nivel del detalle.

Observación de Ingeniería de Software 4.5

Muchos programas pueden dividirse lógicamente en tres fases: una fase de inicialización *en la que se inicializan las variables del programa; una* fase de procesamiento *en la que se introducen los valores de los datos y se ajustan las variables del programa (como contadores y totales) según sea necesario; y una* fase de terminación*, que calcula y produce los resultados finales.*

Cómo proceder a la segunda mejora

La anterior *Observación de Ingeniería de Software* es a menudo todo lo que usted necesita para la primera mejora en el proceso de arriba a abajo. En la **segunda mejora**, nos comprometemos a usar variables específicas. En este ejemplo necesitamos el total actual de los números, una cuenta de cuántos números se han procesado, una variable para recibir el valor de cada calificación, a medida que el usuario las vaya introduciendo, y una variable para almacenar el promedio calculado. La instrucción en seudocódigo

Inicializar las variables

puede mejorarse como sigue:

Inicializar total en cero
Inicializar contador en cero

La instrucción en seudocódigo

```
Introducir, sumar y contar las calificaciones del examen
```

requiere una estructura de repetición (es decir, un ciclo) que introduzca cada calificación en forma sucesiva. No sabemos de antemano cuántas calificaciones van a procesarse, por lo que utilizaremos la **repetición controlada por centinela**. El usuario introduce las calificaciones una por una. Después de introducir la última calificación, introduce el valor centinela. El programa evalúa el valor centinela después de la introducción de cada calificación, y termina el ciclo cuando el usuario introduce el valor centinela. Entonces, la segunda mejora de la instrucción anterior en seudocódigo sería

```
Pedir al usuario que introduzca la primera calificación
Recibir como entrada la primera calificación (puede ser el centinela)

While (mientras) el usuario no haya introducido aún el centinelal
    Sumar esta calificación al total actual
    Sumar uno al contador de calificaciones
    Pedir al usuario que introduzca la siguiente calificación
    Recibir como entrada la siguiente calificación (puede ser el centinela)
```

En seudocódigo *no* utilizamos llaves alrededor de las instrucciones que forman el cuerpo de la estructura *While*. Simplemente aplicamos *sangría* a las instrucciones bajo el *Mientras* para mostrar que pertenecen a esta instrucción. De nuevo, el seudocódigo es solamente una herramienta informal para desarrollar programas.

La instrucción en seudocódigo

```
Calcular e imprimir el total de las calificaciones de todos los estudiantes y el promedio de la clase
```

puede redefinirse de la siguiente manera:

```
Si el contador no es igual a cero
    Asignar al promedio el total dividido entre el contador
    Imprimir el total de las calificaciones para todos los estudiantes en la clase
    Imprimir el promedio de la clase
de lo contrario
    Imprimir "No se introdujeron calificaciones"
```

Evaluamos la posibilidad de una *división entre cero*; por lo general esto es un **error lógico fatal** que, si no se detecta, haría que el programa fallara (a lo que a menudo se le conoce como "**crashing**"). La segunda mejora completa del seudocódigo para el problema del promedio de una clase se muestra en la figura 4.11.

Error común de programación 4.5

Un intento de dividir entre cero produce un comportamiento indefinido y generalmente un error fatal en tiempo de ejecución.

```
1  Inicializar total en cero
2  Inicializar contador en cero
3
4  Pedir al usuario que introduzca la primera calificación
5  Recibir como entrada la primera calificación (puede ser el centinela)
```

Fig. 4.11 | Algoritmo en seudocódigo del problema para promediar una clase, con una repetición controlada por centinela (parte 1 de 2).

```
6
7   Mientras el usuario no haya introducido aún el centinela
8       Sumar esta calificación al total actual
9       Sumar uno al contador de calificaciones
10      Pedir al usuario que introduzca la siguiente calificación
11      Recibir como entrada la siguiente calificación (puede ser el centinela)
12
13  Si el contador no es igual a cero
14      Asignar al promedio el total dividido entre el contador
15      Imprimir el total de las calificaciones para todos los estudiantes de la clase
16      Imprimir el promedio de la clase
17  de lo contrario
18      Imprimir "No se introdujeron calificaciones"
```

Fig. 4.11 | Algoritmo en seudocódigo del problema para promediar una clase, con una repetición controlada por centinela (parte 2 de 2).

Tip para prevenir errores 4.5

Al realizar una división entre una expresión cuyo valor pudiera ser cero, debemos evaluar explícitamente esta posibilidad y manejarla de manera apropiada en el programa (como imprimir un mensaje de error), en vez de permitir que ocurra el error fatal. Hablaremos más sobre cómo lidiar con estos tipos de errores cuando veamos el manejo de excepciones (capítulos 7, 9 y 17 este último en el sitio web).

El seudocódigo en la figura 4.11 resuelve el problema más general para promediar una clase. Este algoritmo sólo requirió dos niveles de mejoramiento. En ocasiones, se requieren más niveles de mejoramiento.

Observación de Ingeniería de Software 4.6

Termine el proceso de mejoramiento de arriba a abajo, paso a paso, cuando haya especificado el algoritmo en seudocódigo con el detalle suficiente como para poder convertir el seudocódigo en C++. Por lo general, la implementación del programa en C++ después de esto es mucho más sencilla.

Observación de Ingeniería de Software 4.7

Muchos programadores experimentados escriben programas sin utilizar herramientas de desarrollo de programas como el seudocódigo. Estos programadores sienten que su meta final es resolver el problema en una computadora y que usar herramientas de desarrollo de programas como el seudocódigo simplemente retarda la producción de los resultados finales. Aunque este método pudiera funcionar para problemas sencillos y conocidos, tiende a ocasionar graves errores y retrasos en proyectos grandes y complejos.

Implementación de la repetición controlada por centinela en la clase `LibroCalificaciones`

En las figuras 4.12 y 4.13 se muestra la clase `LibroCalificaciones` que contiene la función miembro `determinarPromedioClase`, la cual implementa el algoritmo en seudocódigo de la figura 4.11 (esta clase se demuestra en la figura 4.14). Aunque cada calificación introducida es un valor entero, existe la probabilidad de que el cálculo del promedio produzca un número con un punto decimal; en otras palabras, un número real o **número de punto flotante** (por ejemplo, 7.33, 0.0975 o 1000.12345). El tipo `int` no puede representar un número de este tipo, por lo que esta clase debe usar otro tipo para hacerlo. C++ proporciona varios tipos de datos para almacenar números de punto flotante en la memoria, incluyendo `float` y `double`. La principal diferencia entre estos tipos es que, en comparación con las variables `float`, las variables `double` pueden almacenar comúnmente números con una mayor magnitud y un detalle más fino (es decir, más dígitos a la derecha del punto decimal; a esto se le conoce también como la **precisión** del número). Este programa introduce un operador especial llamado **operador de conversión**, para *forzar* a que el cálculo del promedio produzca un resultado numérico de punto flotante.

```
1  // Fig. 4.12: LibroCalificaciones.h
2  // Definición de la clase LibroCalificaciones que determina el promedio de una
      clase.
3  // Las funciones miembro se definen en LibroCalificaciones.cpp
4  #include <string> // el programa usa la clase string estándar de C++
5
6  // definición de la clase LibroCalificaciones
7  class LibroCalificaciones
8  {
9  public:
10    explicit LibroCalificaciones( std::string ); // inicializa el nombre del curso
11    void establecerNombreCurso( std::string ); // establece el nombre del curso
12    std::string obtenerNombreCurso() const; // obtiene el nombre del curso
13    void mostrarMensaje() const; // muestra un mensaje de bienvenida
14    void determinarPromedioClase() const; // promedia las calificaciones
                                             escritas por el usuarios
15 private:
16    std::string nombreCurso; // nombre del curso para este LibroCalificaciones
17 }; // fin de la clase LibroCalificaciones
```

Fig. 4.12 | Problema del promedio de una clase utilizando la repetición controlada por centinela: encabezado de `LibroCalificaciones`.

```
1  // Fig. 4.13: LibroCalificaciones.cpp
2  // Definiciones de funciones miembro para la clase LibroCalificaciones que resuelve
3  // el programa del promedio de la clase con repetición controlada por centinela.
4  #include <iostream>
5  #include <iomanip> // manipuladores de flujo parametrizados
6  #include "LibroCalificaciones.h" // incluye la definición de la clase
                                    LibroCalificaciones
7  using namespace std;
8
9  // el constructor inicializa a nombreCurso con la cadena que se suministra como
      argumento
10 LibroCalificaciones::LibroCalificaciones( string nombre )
11 {
12    establecerNombreCurso( nombre ); // valida y almacena nombreCurso
13 } // fin del constructor de LibroCalificaciones
14
15 // función para establecer el nombre del curso;
16 // asegura que el nombre del curso tenga cuando mucho 25 caracteres
17 void LibroCalificaciones::establecerNombreCurso( string nombre )
18 {
19    if ( nombre.size() <= 25 ) // si el nombre tiene 25 caracteres o menos
20       nombreCurso = nombre; // almacena el nombre del curso en el objeto
21    else // si el nombre es mayor de 25 caracteres
22    { // establece nombreCurso a los primeros 25 caracteres del parámetro nombre
23       nombreCurso = nombre.substr( 0, 25 ); // selecciona los primeros 25
                                                 caracteres
24       cerr << "El nombre \"" << nombre << "\" excede la longitud maxima (25).\n"
25          << "Se limito nombreCurso a los primeros 25 caracteres.\n" << endl;
26    } // fin de if...else
27 } // fin de la función establecerNombreCurso
```

Fig. 4.13 | Problema del promedio de una clase utilizando la repetición controlada por centinela: archivo de código fuente de `LibroCalificaciones` (parte 1 de 3).

```
28
29  // función para obtener el nombre del curso
30  string LibroCalificaciones::obtenerNombreCurso() const
31  {
32     return nombreCurso;
33  } // fin de la función obtenerNombreCurso
34
35  // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
36  void LibroCalificaciones::mostrarMensaje() const
37  {
38     cout << "Bienvenido al libro de calificaciones para\n"
       << obtenerNombreCurso() << "!\n"
39        << endl;
40  } // fin de la función mostrarMensaje
41
42  // determina el promedio de la clase con base en las 10 calificaciones escritas
       por el usuario
43  void LibroCalificaciones::determinarPromedioClase() const
44  {
45     // fase de inicialización
46     int total = 0; // suma de las calificaciones introducidas por el usuario
47     unsigned int contadorCalif = 0; // número de calificaciones introducidas
48
49     // fase de procesamiento
50     // pide la entrada y lee la calificación del usuario
51     cout << "Escriba la calificacion o -1 para salir: ";
52     int calificacion = 0; // valor de la calificación
53     cin >> calificacion; // recibe como entrada la calificacion o el valor
                            centinela
54
55     // itera hasta leer el valor centinela del usuario
56     while ( calificacion != -1 ) // mientras calificacion no sea -1
57     {
58        total = total + calificacion; // suma la calificacion al total
59        contadorCalif = contadorCalif + 1; // incrementa el contador
60
61        // pide la entrada y lee la siguiente calificación del usuario
62        cout << "Escriba la calificacion o -1 para salir: ";
63        cin >> calificacion; // recibe como entrada la calificacion o el valor
                               centinela
64     } // fin de while
65
66     // fase de terminación
67     if ( contadorCalif != 0 ) // si el usuario introdujo al menos una
                                    calificacion...
68     {
69        // calcula el promedio de todas las calificaciones introducidas
70        double promedio = static_cast< double >( total ) / contadorCalif;
71
72        // muestra el total y el promedio (con dos dígitos de precisión)
73        cout << "\nEl total de las " << contadorCalif << " calificaciones
          introducidas es "
74           << total << endl;
75        cout << setprecision( 2 ) << fixed;
76        cout << "El promedio de la clase es " << promedio << endl;
77     } // fin de if
```

Fig. 4.13 | Problema del promedio de una clase utilizando la repetición controlada por centinela: archivo de código fuente de `LibroCalificaciones` (parte 2 de 3).

```
78     else // no se introdujeron calificaciones, por lo que imprime el mensaje
              apropiado
79        cout << "No se introdujeron calificaciones" << endl;
80  } // fin de la función determinarPromedioClase
```

Fig. 4.13 | Problema del promedio de una clase utilizando la repetición controlada por centinela: archivo de código fuente de `LibroCalificaciones` (parte 3 de 3).

```
1   // Fig. 4.14: fig04_14.cpp
2   // Crea un objeto LibroCalificaciones e invoca a su función
       determinarPromedioClase.
3   #include "LibroCalificaciones.h" // incluye la definición de la clase
                                        LibroCalificaciones
4
5   int main()
6   {
7      // crea el objeto LibroCalificaciones llamado miLibroCalificaciones y
8      // pasa el nombre del curso al constructor
9      LibroCalificaciones miLibroCalificaciones( "CS101 Programacion en C++" );
10
11     miLibroCalificaciones.mostrarMensaje(); // muestra el mensaje de bienvenida
12     miLibroCalificaciones.determinarPromedioClase(); // busca el promedio de 10
                                                           calificaciones
13  } // fin de main
```

```
Bienvenido al libro de calificaciones para
CS101 Programacion en C++!

Escriba la calificacion o -1 para salir: 97
Escriba la calificacion o -1 para salir: 88
Escriba la calificacion o -1 para salir: 72
Escriba la calificacion o -1 para salir: -1

El total de las 3 calificaciones introducidas es 257
El promedio de la clase es 85.67
```

Fig. 4.14 | Problema del promedio de una clase utilizando la repetición controlada por centinela: creación de un objeto de la clase `LibroCalificaciones` e invocación de su función miembro determinar`PromedioClase`.

En este ejemplo vemos que las estructuras de control pueden apilarse una encima de otra; la instrucción `while` (líneas 56 a 64 de la figura 4.13) va seguida por una instrucción `if...else` (líneas 67 a 79) en secuencia. La mayor parte del código en este programa es idéntico al código de la figura 4.9, por lo que nos concentraremos en las nuevas características y conceptos.

Las líneas 46 y 47 inicializan las variables `total` y `contadorCalif` en `0`, ya que no se han introducido calificaciones todavía. Recuerde que este programa utiliza la repetición controlada por centinela. Para mantener un registro preciso del número de calificaciones introducidas, el programa incrementa `contadorCalif` *sólo* cuando el usuario introduce un valor para la calificación que *no* sea el valor centinela y el programa completa el procesamiento de la calificación. Declaramos e inicializamos las variables `calificacion` (línea 52) y `promedio` (línea 70) en donde se utilizan. Cabe mencionar que la línea 70 declara la variable `promedio` como de tipo `double`. Recuerde que usamos una variable `int` en el ejemplo anterior para almacenar el promedio de la clase. Al usar el tipo `double` en el ejemplo anterior podemos almacenar el resultado del cálculo del promedio de la clase como un número de punto flotante. Por último, observe que antes de ambas instrucciones (líneas 53 y 63) se coloca una instrucción de salida que pide al usuario los datos de entrada.

Buena práctica de programación 4.3

Pida al usuario cada dato de entrada del teclado. El indicador debe indicar la forma de la entrada y cualquier valor de entrada especial. En un ciclo controlado por centinela, los indicadores que solicitan datos de entrada deben recordar explícitamente al usuario cuál es el valor centinela.

Comparación entre la lógica del programa para la repetición controlada por centinela y la repetición controlada por contador

Compare la lógica del programa para la repetición controlada por centinela con la repetición controlada por contador en la figura 4.9. En la repetición controlada por contador, cada iteración de la instrucción `while` (líneas 49 a 56 de la figura 4.9) lee un valor del usuario, para el número especificado de iteraciones. En la repetición controlada por centinela, el programa lee el primer valor (líneas 51 a 53 de la figura 4.13) antes de llegar al `while`. Este valor determina si el flujo de control del programa debe entrar al cuerpo del `while`. Si la condición es falsa, el usuario introdujo el valor centinela, por lo que el cuerpo no se ejecuta (es decir, no se introdujeron calificaciones). Si, por otro lado, la condición es verdadera, el cuerpo comienza a ejecutarse y el ciclo suma el valor de `calificacion al total` (línea 58) e incrementa `contadorCalif` (línea 59). Después, en las líneas 62 y 63 en el cuerpo del ciclo se pide y recibe el siguiente valor del usuario. A continuación, el control del programa se acerca a la llave derecha de terminación (`}`) del cuerpo del `while` en la línea 64, por lo que la ejecución continúa con la evaluación de la condición del `while` (línea 56). La condición utiliza el valor más reciente de `calificacion` que acaba de introducir el usuario, para determinar si el cuerpo de la instrucción debe ejecutarse otra vez. El valor de la variable `calificacion` siempre lo introduce el usuario inmediatamente antes de que el programa evalúe la condición del `while`. Esto permite al programa determinar si el valor que *acaba de introducir* el usuario es el valor centinela, *antes* de que el programa procese ese valor (es decir, que lo sume al `total` e incremente `contadorCalif`). Si el valor introducido es el valor centinela, el ciclo termina y el programa no suma el valor `-1` al `total`.

Una vez que termina el ciclo, se ejecuta la instrucción `if...else` (líneas 67 a 79). La condición en la línea 67 determina si se introdujeron calificaciones o no. Si no se introdujo ninguna, se ejecuta la parte del `else` (líneas 78 y 89) de la instrucción `if...else` y muestra el mensaje "`No se introdujeron calificaciones`", y la función miembro devuelve el control a la función que la llamó.

Observe el bloque de la instrucción `while` en la figura 4.13. Sin las llaves, las últimas tres instrucciones en el cuerpo del ciclo quedarían fuera de éste, ocasionando que la computadora interpretara el código incorrectamente, como se muestra a continuación:

```
// itera hasta leer el valor centinela del usuario
while ( calificacion != -1 )
   total = total + calificacion; // suma calificacion al total
contadorCalif = contadorCalif + 1; // incrementa el contador

// pide la entrada y lee la siguiente calificación del usuario
cout << "Escriba calificacion o -1 para salir: ";
cin >> calificacion;
```

El código anterior ocasionaría un ciclo infinito en el programa si el usuario no introduce el centinela `-1` para la primera calificación (en línea 53).

Error común de programación 4.6

Omitir las llaves que delimitan a un bloque puede provocar errores lógicos, como ciclos infinitos. Para prevenir este problema, algunos programadores encierran el cuerpo de todas las instrucciones de control con llaves, aun si el cuerpo sólo contiene una instrucción.

Precisión de los números de punto flotante y requerimientos de memoria

Las variables de tipo `float` representan **números de punto flotante con precisión simple** y tienen alrededor de siete dígitos significativos en la mayoría de los sistemas actuales. Las variables de tipo `double` representan **números de punto flotante con precisión doble**. Estos requieren el doble de memoria que las variables `float` y pueden proporcionar alrededor de 15 dígitos significativos en la mayoría de los sistemas actuales; aproximadamente el doble de precisión que las variables `float`. La mayoría de los programadores representan a los números de punto flotante con el tipo `double`. De hecho, C++ considera a todos los números de punto flotante que escribimos en el código fuente de un programa (como 7.33 y 0.0975) como valores `double` de manera predeterminada. Dichos valores en el código fuente se conocen como **constantes de punto flotante**. En el apéndice C, Tipos fundamentales, podrá consultar los rangos de valores para los números `float` y `double`.

En la aritmética convencional, los números de punto flotante surgen con frecuencia como resultado de la división; cuando dividimos 10 entre 3, el resultado es 3.3333333..., donde la secuencia de números 3 se repite en forma infinita. La computadora asigna sólo una cantidad *fija* de espacio para guardar ese valor, por lo que evidentemente el valor de punto flotante guardado almacenado sólo puede ser una *aproximación*.

Error común de programación 4.7

Usar números de punto flotante de una manera que suponga que se representan con precisión de manera exacta (por ejemplo, usándolos en las comparaciones de igualdad) puede producir resultados imprecisos. Los números de punto flotante se representan sólo en forma aproximada.

Aunque los números de punto flotante no siempre son 100 por ciento precisos, tienen numerosas aplicaciones. Por ejemplo, cuando hablamos de una temperatura corporal "normal" de 36.7 grados centígrados, no necesitamos tener una precisión con una gran cantidad de dígitos. Cuando leemos la temperatura en un termómetro como 36.7, en realidad podría ser 36.6999473210643. Considerar a este número simplemente como 36.7 está bien para la mayoría de las aplicaciones en las que se utilizan temperaturas corporales. Debido a la naturaleza imprecisa de los números de punto flotante, se prefiere el tipo `double` al tipo `float`, ya que las variables `double` pueden representar números de punto flotante con más precisión. Por esta razón, usamos el tipo `double` en el resto del libro.

Conversión explícita e implícita entre los tipos fundamentales

La variable `promedio` se declara como de tipo `double` (línea 70 de la figura 4.13) para capturar el resultado fraccionario de nuestro cálculo. Sin embargo, `total` y `contadorCalif` son variables enteras. Recuerde que al dividir dos enteros se produce una división entera, en la cual cualquier parte fraccionaria del cálculo se trunca. En la siguiente instrucción:

```
double promedio = total / contadorCalif;
```

primero se realiza el cálculo de la división, por lo que se pierde la parte fraccionaria del resultado *antes* de asignarlo a `promedio`. Para realizar un cálculo de punto flotante con valores enteros, debemos crear valores *temporales* de punto flotante. C++ cuenta con el **operador `static_cast`** para llevar a cabo esta tarea. En la línea 70 se utiliza el operador de conversión de tipo `static_cast<double>(total)` para crear una copia de punto flotante *temporal* de su operando entre paréntesis: `total`. Utilizar un operador de conversión de tipo de esta forma es un proceso que se denomina **conversión explícita**. El valor almacenado en `total` sigue siendo un entero.

El cálculo ahora consiste de un valor de punto flotante (la versión temporal `double` de `total`) dividido entre el entero `contadorCalif`. El compilador sabe cómo evaluar *sólo* expresiones en las que los tipos de datos de los operandos sean *idénticos*. Para asegurar que los operandos sean del mismo tipo, el

compilador realiza una operación llamada **promoción** (o **conversión implícita)** en los operandos seleccionados. Por ejemplo, en una expresión que contenga valores de los tipos de datos `int` y `double`, C++ **promueve** los operandos `int` a valores `double`. En nuestro ejemplo, tratamos a `total` como `double` (mediante el operador `static_cast`), por lo que el compilador promueve el valor de `contadorCalif` al tipo `double`, con lo cual permite realizar el cálculo; el resultado de la división de punto flotante se asigna a `promedio`. En el capítulo 6, Funciones y una introducción a la recursividad, hablaremos sobre todos los tipos de datos fundamentales y su orden de promoción.

Los operadores de conversión de tipo están disponibles para usarse con cualquier tipo de datos, además de los tipos de clases. El operador `static_cast` se forma colocando la palabra clave `static_cast` entre los signos `<` y `>`, alrededor del nombre de un tipo de datos. El operador `static_cast` es un **operador unario**; un operador que toma sólo un operando. En el capítulo 2 estudiamos los operadores aritméticos binarios. C++ también soporta las versiones unarias de los operadores de suma (`+`) y resta (`-`), por lo que el programador puede escribir expresiones como `-7` o `+5`. Los operadores de conversión de tipo tienen mayor precedencia que los demás operadores unarios, como el `+` unario y el `-` unario. Esta precedencia es un nivel mayor que la de los **operadores de multiplicación** `*`, `/` y `%`, y menor que la de los paréntesis. En nuestras tablas de precedencia, indicamos el operador de conversión de tipos con la notación `static_cast<`*`tipo`*`>()`.

Formato para los números de punto flotante

Aquí veremos brevemente las herramientas de formato en la figura 4.13, y las explicaremos con detalle en el capítulo 13, Entrada/salida de flujos: un análisis detallado. La llamada a **`setprecision`** en la línea 75 (con un argumento de `2`) indica que la variable `double` llamada `promedio` debe imprimirse con *dos* dígitos de **precisión** a la derecha del punto decimal (por ejemplo, 92.37). A esta llamada se le conoce como **manipulador de flujo parametrizado** (debido al `2` entre paréntesis). Los programas que utilizan estas llamadas deben contener la directiva del preprocesador (línea 5):

```
#include <iomanip>
```

El manipulador `endl` es un **manipulador de flujos no parametrizado** (ya que no va seguido de un valor o expresión entre paréntesis), por lo cual *no* requiere el encabezado `<iomanip>`. Si no se especifica la precisión, los valores de punto flotante se imprimen generalmente con *seis* dígitos de precisión (es decir, la **precisión predeterminada** en la mayoría de los sistemas actuales), aunque en un momento veremos una excepción a esto.

El manipulador de flujo **`fixed`** (línea 75) indica que los valores de punto flotante deben imprimirse en lo que se denomina **formato de punto fijo**, en oposición a la **notación científica**. La notación científica es una forma de mostrar un número como valor de punto flotante entre los valores de 1.0 y 10.0, multiplicado por una potencia de 10. Por ejemplo, el valor 3 100.0 se mostraría en notación científica como 3.1×10^3. La notación científica es útil cuando se muestran valores muy grandes o muy pequeños. En el capítulo 13 hablaremos sobre el formato mediante el uso de la notación científica. Por otra parte, el formato de punto fijo se utiliza para forzar a que un número de punto flotante muestre un número específico de dígitos. Al especificar el formato de punto fijo también forzamos a que se imprima el punto decimal y los ceros a la derecha, aun si el valor es una cantidad entera, como 88.00. Sin la opción de formato de punto fijo, dicho valor se imprime en C++ como 88, *sin* los ceros a la derecha *ni* el punto decimal. Al utilizar los manipuladores de flujos `fixed` y `setprecision` en un programa, el valor *impreso* se **redondea** al número de posiciones decimales indicado por el valor que se pasa a `setprecision` (por ejemplo, el valor `2` en la línea 75), aunque el valor en memoria permanece sin cambios. Por ejemplo, los valores 87.946 y 67.543 se imprimen como 87.95 y 67.54, respectivamente. También es posible *forzar* a que aparezca un punto decimal mediante el uso del manipulador de flujos **`showpoint`**. Si se especifica `showpoint` sin `fixed`, entonces no se imprimirán ceros a la derecha. Al igual que `endl`, los manipuladores `fixed` y `showpoint` no usan parámetros y no requieren el encabezado `<iomanip>`. Ambos se encuentran en el encabezado `<iostream>`.

En las líneas 75 y 76 de la figura 4.13 se imprime el promedio de la clase *redondeado* a la centésima más cercana, y lo imprimimos con *sólo* dos dígitos a la derecha del punto decimal. El manipulador de flujos parametrizado (línea 75) indica que el valor de la variable `promedio` se debe mostrar con *dos* dígitos de precisión a la derecha del punto decimal; esto se indica mediante `setprecision(2)`. Las tres calificaciones introducidas durante la ejecución de ejemplo del programa de la figura 4.14 dan un total de 257, que a su vez produce el promedio 85.666… y se imprime con redondeo como 85.67.

Una nota sobre los enteros sin signo

En la línea 46 de la figura 4.9, se declaró la variable `contadorCalif` como `unsigned int` debido a que sólo puede asumir los valores de 1 a 11 (el 11 termina el ciclo), que son todos valores positivos. En general, los contadores que deben almacenar sólo valores no negativos deberían declararse con tipos `unsigned`. Las variables de tipos enteros `unsigned` pueden representar valores desde 0 hasta aproximadamente el *doble del rango positivo* de los tipos enteros con signo correspondientes. Puede determinar el valor `unsigned int` máximo de su plataforma con la constante `UINT_MAX` de `<climits>`.

La figura 4.9 podría haber declarado también como `unsigned int` las variables `calificacion`, `total` y `promedio`. Por lo general, las calificaciones son valores de 0 a 100, por lo que `total` y `promedio` deberían ser cada uno mayor o igual a 0. Declaramos esas variables como `int` debido a que no podemos controlar lo que el usuario introduzca en realidad; incluso podría introducir valores *negativos*. Peor aún, el usuario podría introducir un valor que ni siquiera sea un número (más adelante en el libro le mostraremos cómo lidiar con dichas entradas erróneas).

Algunas veces los ciclos controlados por centinela usan valores inválidos de manera *intencional* para terminar un ciclo. Por ejemplo, en la línea 56 de la figura 4.13 terminamos el ciclo cuando el usuario introduce el valor centinela `-1` (una calificación inválida), por lo que sería inapropiado declarar la variable `calificacion` como `unsigned int`. Como veremos más adelante, el indicador de fin de archivo (`EOF`) (que presentaremos en el siguiente capítulo y que se utiliza a menudo para terminar ciclos controlados por centinela) también se implementa de manera interna en el compilador como un número negativo.

4.10 Cómo formular algoritmos: instrucciones de control anidadas

En el siguiente ejemplo formularemos una vez más un algoritmo utilizando seudocódigo y el mejoramiento de arriba a abajo, paso a paso, y después escribiremos el correspondiente programa en C++. Hemos visto que las instrucciones de control pueden *apilarse* una encima de otra (en secuencia). Aquí examinaremos la otra forma en la que pueden conectarse las instrucciones de control, a saber, mediante el **anidamiento** de una instrucción de control dentro de otra. Considere el siguiente enunciado de un problema:

> *Una universidad ofrece un curso que prepara a los estudiantes para el examen estatal de certificación del estado como corredores de bienes raíces. El año pasado, diez de los estudiantes que completaron este curso tomaron el examen. La universidad desea saber qué tan bien se desempeñaron sus estudiantes en el examen. A usted se le ha pedido que escriba un programa para sintetizar los resultados. Se le dio una lista de estos 10 estudiantes. Junto a cada nombre hay un 1 escrito, si el estudiante aprobó el examen, o un 2 si lo reprobó.*
>
> *Su programa debe analizar los resultados del examen de la siguiente manera:*
>
> 1. *Introducir cada resultado de la prueba (es decir, un 1 o un 2). Mostrar el mensaje "Escriba el resultado" en la pantalla, cada vez que el programa solicite otro resultado de la prueba.*
> 2. *Contar el número de resultados de la prueba, de cada tipo.*
> 3. *Mostrar un resumen de los resultados de la prueba, indicando el número de estudiantes que aprobaron y el número de estudiantes que reprobaron.*
> 4. *Si más de ocho estudiantes aprobaron el examen, imprimir el mensaje "Bono para el instructor".*

Después de leer cuidadosamente el enunciado del programa, hacemos las siguientes observaciones:

1. El programa debe procesar los resultados de la prueba para 10 estudiantes. Puede usarse un *ciclo controlado por contador*, ya que el número de resultados de la prueba se conoce de antemano.
2. Cada resultado de la prueba es un número; ya sea 1 o 2. Cada vez que el programa lee un resultado de la prueba, debe determinar si el número es 1 o 2. Para fines de simplificación, nosotros sólo evaluamos un 1 en nuestro algoritmo. Si el número no es 1, suponemos que es un 2 (asegúrese de hacer el ejercicio 4.20, en donde se consideran las consecuencias de esta suposición).
3. Se utilizan dos contadores para llevar el registro de los resultados del examen: uno para contar el número de estudiantes que aprobaron el examen y uno para contar el número de estudiantes que reprobaron el examen.
4. Una vez que el programa ha procesado todos los resultados, debe decidir si más de ocho estudiantes aprobaron el examen.

Veamos ahora el mejoramiento de arriba a abajo, paso a paso. Comencemos con la representación del seudocódigo de la cima:

```
Analizar los resultados del examen y decidir si hay que pagar un bono o no
```

Una vez más, es importante enfatizar que la cima es una representación *completa* del programa, pero es probable que se necesiten varias mejoras antes de que el seudocódigo pueda evolucionar de manera natural en un programa en C++.

Nuestra primera mejora es

```
Inicializar variables
Introducir las 10 calificaciones del examen, y contar los aprobados y reprobados
Imprimir un resumen de los resultados del examen y decidir si debe pagarse un bono
```

Aquí también, aun y cuando tenemos una representación *completa* de todo el programa, es necesario mejorarla. Ahora nos comprometemos con variables específicas. Se necesitan contadores para registrar los aprobados y reprobados; utilizaremos un contador para controlar el proceso de los ciclos y necesitaremos una variable para guardar la entrada del usuario.

La instrucción en seudocódigo

```
Inicializar variables
```

puede mejorarse de la siguiente manera:

```
Inicializar aprobados en cero
Inicializar reprobados en cero
Inicializar contador de estudiantes en uno
```

Observe que sólo se inicializan los contadores al principio del algoritmo.

La instrucción en seudocódigo

```
Introducir las 10 calificaciones del examen, y contar los aprobados y reprobados
```

requiere un ciclo en el que se introduzca sucesivamente el resultado de cada examen. Sabemos de antemano que hay precisamente 10 resultados del examen, por lo que es apropiado utilizar un ciclo controlado por contador. Dentro del ciclo (es decir, **anidado** dentro del ciclo), una instrucción `if...else` determinará si cada resultado del examen es aprobado o reprobado, e incrementará el contador apropiado. Entonces, la mejora al seudocódigo anterior es

```
Mientras el contador de estudiantes sea menor o igual a 10
    Pedir al usuario que introduzca el siguiente resultado del examen
    Recibir como entrada el siguiente resultado del examen

    Si el estudiante aprobó
        Sumar uno a aprobados
    De lo contrario
        Sumar uno a reprobados

    Sumar uno al contador de estudiantes
```

Nosotros utilizamos líneas en blanco para aislar la estructura de control *Si...De lo contrario*, lo cual mejora la legibilidad.

La instrucción en seudocódigo

```
Imprimir un resumen de los resultados de los exámenes y decidir si debe pagarse un bono
```

puede mejorarse de la siguiente manera:

```
Imprimir el número de aprobados
Imprimir el número de reprobados

Si más de ocho estudiantes aprobaron
    Imprimir "Bono para el instructor"
```

La segunda mejora completa aparece en la figura 4.15. Se utilizan líneas en blanco para separar la estructura *Mientras* y mejorar la legibilidad del programa. Este seudocódigo está ahora lo suficientemente mejorado para su conversión a C++.

```
1  Inicializar aprobados en cero
2  Inicializar reprobados en cero
3  Inicializar contador de estudiantes en uno
4
5  Mientras el contador de estudiantes sea menor o igual a 10
6      Pedir al usuario que introduzca el siguiente resultado del examen
7      Recibir como entrada el siguiente resultado del examen
8
9      Si el estudiante aprobó
10         Sumar uno a aprobados
11     De lo contrario
12         Sumar uno a reprobados
13
14     Sumar uno al contador de estudiantes
15
16 Imprimir el número de aprobados
17 Imprimir el número de reprobados
```

Fig. 4.15 | El seudocódigo para el problema de los resultados del examen (parte 1 de 2).

```
18
19  Si más de ocho estudiantes aprobaron
20      Imprimir "Bono para el instructor"
```

Fig. 4.15 | El seudocódigo para el problema de los resultados del examen (parte 2 de 2).

Análisis de conversión a la clase

El programa que implementa el algoritmo en seudocódigo se muestra en la figura 4.16. Este ejemplo no contiene una clase; sólo contiene un archivo de código fuente en donde la función `main` realiza todo el trabajo de la aplicación. En éste y en el capítulo 3, hemos visto ejemplos que consisten en una clase (incluyendo los archivos de encabezado y código fuente para esta clase), así como otro archivo de código fuente para probar la clase. Este archivo de código fuente contenía la función `main`, que creaba un objeto de la clase y llamaba a sus funciones miembro. En ocasiones, cuando no tenga sentido tratar de crear una clase *reutilizable* para demostrar un concepto, usaremos un ejemplo contenido totalmente dentro de la función `main` de un *sólo* archivo de código fuente.

En las líneas 9 a 11 y 18 se declaran e inicializan las variables que se utilizan para procesar los resultados del examen. Algunas veces los programas de iteración requieren la inicialización al principio de *cada* repetición; dicha reinicialización se realiza mediante instrucciones de asignación en vez de declaraciones, o se pueden mover las declaraciones adentro de los cuerpos de los ciclos.

```
1  // Fig. 4.16: fig04_16.cpp
2  // Problema de los resultados del examen: instrucciones de control anidadas.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     // inicialización de las variables en las declaraciones
9     unsigned int aprobados = 0; // número de aprobados
10    unsigned int reprobados = 0; // número de reprobados
11    unsigned int contadorEstudiantes = 1; // contador de estudiantes
12
13    // procesa 10 estudiantes usando el ciclo controlado por contador
14    while ( contadorEstudiantes <= 10 )
15    {
16       // pide datos de entrada y obtiene el valor del usuario
17       cout << "Escriba el resultado (1 = aprobado, 2 = reprobado): ";
18       int resultado = 0; // resultado de un examen (1 = aprobado, 2 = reprobado)
19       cin >> resultado; // recibe como entrada el resultado
20
21       // if...else anidado en la instrucción while
22       if ( resultado == 1 )           // si resultado es 1,
23          aprobados = aprobados + 1;   // incrementa aprobados;
24       else                            // else resultado no es 1, por lo que
25          reprobados = reprobados + 1; // incrementa reprobados
26
27       // incrementa contadorEstudiantes para que el ciclo termine en cierto
            momento
28       contadorEstudiantes = contadorEstudiantes + 1;
29    } // fin de while
```

Fig. 4.16 | Problema de los resultados de un examen: instrucciones de control anidadas (parte 1 de 2).

```
30
31     // fase de terminación; muestra el número de aprobados y reprobados
32     cout << "Aprobados " << aprobados << "\nReprobados "  << reprobados << endl;
33
34     // determina si aprobaron más de ocho estudiantes
35     if ( aprobados > 8 )
36        cout << "Bono para el instructor" << endl;
37  } // fin de main
```

```
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Aprobados 6
Reprobados 4
```

```
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 2
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Escriba el resultado (1 = aprobado, 2 = reprobado): 1
Aprobados 9
Reprobados 1
Bono para el instructor
```

Fig. 4.16 | Problema de los resultados de un examen: instrucciones de control anidadas (parte 2 de 2).

La instrucción `while` (líneas 14 a 29) itera 10 veces. Durante cada iteración, el ciclo recibe y procesa un resultado del examen. La instrucción `if...else` (líneas 22 a 25) para procesar cada resultado se anida en la instrucción `while`. Si `resultado` es `1`, la instrucción `if...else` incrementa a `aprobados`; en caso contrario, asume que `resultado` es `2` e incrementa `reprobados`. La línea 28 incrementa `contadorEstudiantes` antes de que se evalúe otra vez la condición del ciclo, en la línea 15. Después de introducir 10 valores, el ciclo termina y en la línea 32 se muestra el número de `aprobados` y de `reprobados`. La instrucción `if` de las líneas 35 a 36 determina si más de ocho estudiantes aprobaron el examen y, de ser así, imprime el mensaje `"Bono para el instructor"`.

En la figura 4.16 se muestra la entrada y la salida de dos ejecuciones de ejemplo del programa. Al final de la segunda ejecución de ejemplo, la condición en la línea 35 es verdadera: más de ocho estudiantes aprobaron el examen, por lo que el programa imprime un mensaje en pantalla indicando que el instructor debe recibir un bono.

Inicialización de listas en C++11
C++11 introduce una nueva sintaxis de inicialización de variables. La **inicialización de listas** (también conocida como inicialización uniforme) nos permite usar una sintaxis para inicializar una variable de *cualquier* tipo. Considere la línea 11 de la figura 4.16:

```
unsigned int contadorEstudiantes = 1;
```

En C++11 podemos escribir esto como

```
unsigned int contadorEstudiantes = { 1 };
```

o

```
unsigned int contadorEstudiantes{ 1 };
```

Las llaves ({ y }) representan el *inicializador de listas*. Para una variable de tipo fundamental, sólo se coloca un valor en el inicializador de listas. Para un objeto, el inicializador de listas puede ser una *lista* de valores *separada por comas* que se pasan al constructor del objeto. Por ejemplo, el ejercicio 3.14 le pidió crear una clase `Empleado` que representara el primer nombre, apellido y salario de un empleado. Suponiendo que la clase define un constructor que recibe objetos `string` para el primer nombre y el apellido, y un `double` para el salario, podría inicializar el objeto `Empleado` de la siguiente forma:

```
Empleado empleado1{ "Bob", "Blue", 1234.56 };
Empleado empleado2 = { "Sue", "Green", 2143.65 };
```

Para las variables de tipos fundamentales, la sintaxis de inicialización de listas también *evita* lo que se conoce como **conversiones de reducción (narrowing)**, que podrían provocar *pérdida de datos*. Por ejemplo, antes era posible escribir la siguiente instrucción:

```
int x = 12.7;
```

la cual intenta asignar el valor `double 12.7` a la variable `int x`. Para convertir un valor `double` en `int`, se *trunca* la parte de punto flotante (`.7`) y se produce una *pérdida* de información: una *conversión de reducción*. El valor real asignado a `x` es `12`. Muchos compiladores generan una *advertencia* para esta instrucción, pero de todas formas permiten que se compile. Pero si utilizamos la inicialización de listas, como en

```
int x = { 12.7 };
```

o

```
int x{ 12.7 };
```

se produce un *error de compilación*, el cual le ayuda a evitar un error lógico potencialmente sutil. Por ejemplo, el compilador Xcode LLVM de Apple proporciona el error

```
Type 'double' cannot be narrowed to 'int' in initializer list
```

En capítulos posteriores hablaremos sobre características adicionales de los inicializadores de listas.

4.11 Operadores de asignación

C++ cuenta con varios **operadores de asignación** para abreviar las expresiones de asignación. Por ejemplo, la instrucción

```
c = c + 3;
```

puede abreviarse mediante el **operador de asignación de suma**, `+=`, de la siguiente manera:

```
c += 3;
```

este operador suma el valor de la expresión que está a la derecha del operador, al valor de la variable que está a la izquierda del operador, y almacena el resultado en la variable que está a la izquierda del operador. Cualquier instrucción de la forma

variable = *variable operador expresión*;

en donde la misma *variable* aparece en ambos lados del operador de asignación y *operador* es uno de los operadores binarios +, -, *, / o % (u otros que veremos más adelante en el libro), puede escribirse de la siguiente forma:

variable operador= *expresión*;

Por lo tanto, la expresión de asignación `c += 3` suma 3 a `c`. La figura 4.17 muestra los operadores de asignación aritméticos, algunas expresiones de ejemplo en las que se utilizan los operadores y las explicaciones de lo que estos operadores hacen.

Operador de asignación	Expresión de ejemplo	Explicación	Asigna
Suponer que: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	`10 a c`
`-=`	`d -= 4`	`d = d - 4`	`1 a d`
`*=`	`e *= 5`	`e = e * 5`	`20 a e`
`/=`	`f /= 3`	`f = f / 3`	`2 a f`
`%=`	`g %= 9`	`g = g % 9`	`3 a g`

Fig. 4.17 | Operadores de asignación aritméticos.

4.12 Operadores de incremento y decremento

Además de los operadores de asignación aritméticos, C++ proporciona dos operadores unarios para sumar 1, o restar 1, al valor de una variable numérica. Estos operadores son el **operador de incremento** unario, `++`, y el **operador de decremento** unario, `--`, los cuales se sintetizan en la figura 4.18. Un programa puede incrementar en 1 el valor de una variable llamada `c`, utilizando el operador de incremento, `++`, en lugar de usar la expresión `c = c + 1` o `c +=1`. A un operador de incremento o decremento que se coloca *antes* de una variable se le llama **operador de preincremento** o **predecremento**, respectivamente. A un operador de incremento o decremento que se coloca *después* de una variable se le llama **operador de postincremento** o **postdecremento**, respectivamente.

Operador	Llamado	Expresión de ejemplo	Explicación
`++`	preincremento	`++a`	Incrementar a en 1, después utilizar el nuevo valor de a en la expresión en que esta variable reside.
`++`	postincremento	`a++`	Usar el valor actual de a en la expresión en la que esta variable reside, después incrementar a en 1.

Fig. 4.18 | Los operadores de incremento y decremento (parte 1 de 2).

Operador	Llamado	Expresión de ejemplo	Explicación
`--`	predecremento	`--b`	Decrementar `b` en `1`, después utilizar el nuevo valor de `b` en la expresión en que esta variable reside.
`--`	postdecremento	`b--`	Usar el valor actual de `b` en la expresión en la que esta variable reside, después decrementar `b` en `1`.

Fig. 4.18 | Los operadores de incremento y decremento (parte 2 de 2).

Al proceso de utilizar el operador de preincremento (o postdecremento) para sumar (o restar) 1 a una variable, se le conoce como **preincrementar** (o **predecrementar**) la variable. Al preincrementar (o predecrementar) una variable, ésta se incrementa (o decrementa) en 1, y después el nuevo valor de la variable se utiliza en la expresión en la que aparece. Al proceso de utilizar el operador de postincremento (o postdecremento) para sumar (o restar) 1 a una variable, se le conoce como **postincrementar** (o **postdecrementar**) la variable. Al postincrementar (o postdecrementar) una variable, el valor *actual* de la variable se utiliza en la expresión en la que aparece y después el valor de la variable se incrementa (o decrementa) en 1.

Buena práctica de programación 4.4

A diferencia de los operadores binarios, los operadores unarios de incremento y decremento deben colocarse enseguida de sus operandos, sin espacios entre ellos.

En la figura 4.19 se demuestra la diferencia entre la versión de preincremento y la versión de predecremento del operador de incremento ++. El operador de decremento (--) funciona de manera similar.

```
1  // Fig. 4.19: fig04_19.cpp
2  // Preincremento y postincremento.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     // demuestra el postincremento
9     int c = 5; // asigna 5 a c
10    cout << c << endl; // imprime 5
11    cout << c++ << endl; // imprime 5 y después postincrementa
12    cout << c << endl; // imprime 6
13
14    cout << endl; // salta una línea
15
16    // demuestra el preincremento
17    c = 5; // asigna 5 a c
18    cout << c << endl; // imprime 5
19    cout << ++c << endl; // preincrementa y después imprime 6
20    cout << c << endl; // imprime 6
21 } // fin de main
```

Fig. 4.19 | Preincrementar y postdecrementar (parte 1 de 2).

```
5
5
6

5
6
6
```

Fig. 4.19 | Preincrementar y postdecrementar (parte 2 de 2).

En la línea 9 se inicializa c con 5, y en la línea 10 se imprime el valor inicial de c. En la línea 11 se imprime el valor de la expresión c++. Esta expresión postincrementa la variable c, por lo que se imprime su valor original (5) y después se incrementa su valor. Así, en la línea 11 se imprime el valor inicial de c (5) otra vez. En la línea 12 se imprime el nuevo valor de c (6) para mostrar que se incrementó el valor de la variable en la línea 11.

En la línea 17 se restablece el valor de c a 5 y en la línea 18 se imprime ese valor. En la línea 19 se imprime el valor de la expresión ++c. Esta expresión preincrementa a c, por lo que se incrementa su valor y luego se imprime el nuevo valor (6). En la línea 20 se imprime el valor de c otra vez para mostrar que sigue siendo 6 después de que se ejecuta la línea 19.

Los operadores de asignación aritméticos y los operadores de incremento y decremento pueden utilizarse para simplificar las instrucciones de los programas. Las tres instrucciones de asignación de la figura 4.16:

```
aprobados = aprobados + 1;
reprobados = reprobados + 1;
contadorEstudiantes = contadorEstudiantes + 1;
```

se pueden escribir en forma más concisa con operadores de asignación, de la siguiente manera:

```
aprobados += 1;
reprobados += 1;
contadorEstudiantes += 1;
```

con operadores de preincremento de la siguiente forma:

```
++aprobados;
++reprobados;
++contadorEstudiantes;
```

o con operadores de postincremento de la siguiente forma:

```
aprobados++;
reprobados++;
contadorEstudiantes++;
```

Al incrementar (++) o decrementar (--) una variable entera que se encuentre en una instrucción por sí sola, las formas preincremento y postincremento tienen el mismo efecto lógico, y las formas predecremento y postdecremento tienen el mismo efecto lógico. Solamente cuando una variable aparece en el contexto de una expresión más grande es cuando los operadores preincremento y postincremento tienen distintos efectos (y lo mismo se aplica a los operadores de predecremento y postdecremento).

Error común de programación 4.8

Tratar de usar el operador de incremento o decremento en una expresión que no sea un nombre de variable que pueda modificarse [por ejemplo, escribir ++(x + 1)] es un error de sintaxis.

En la figura 4.20 se muestra la precedencia y la asociatividad de los operadores que se han presentado hasta este punto. Los operadores se muestran de arriba a abajo, en orden descendente de precedencia. La segunda columna indica la asociatividad de los operadores en cada nivel de precedencia. Cabe mencionar que el operador condicional (`?:`), los operadores unarios de preincremento (`++`), predecremento (`--`), suma (`+`) y resta (`-`), y los operadores de asignación `=`, `+=`, `-=`, `*=`, `/=` y `%=` se asocian de *derecha a izquierda*. Todos los demás operadores en la figura 4.20 se asocian de *izquierda a derecha*. La tercera columna enlista los diversos grupos de operadores.

Operadores	Asociatividad	Tipo
`::` `()`	izquierda a derecha *[Vea la precaución en la figura 2.10 con respecto al agrupamiento de paréntesis]*	primario
`++` `--` `static_cast<type>()`	izquierda a derecha	postfijo
`++` `--` `+` `-`	derecha a izquierda	unario (prefijo)
`*` `/` `%`	izquierda a derecha	multiplicativo
`+` `-`	izquierda a derecha	aditivo
`<<` `>>`	izquierda a derecha	inserción/extracción
`<` `<=` `>` `>=`	izquierda a derecha	relacional
`==` `!=`	izquierda a derecha	igualdad
`?:`	derecha a izquierda	condicional
`=` `+=` `-=` `*=` `/=` `%=`	derecha a izquierda	asignación

Fig. 4.20 | Precedencia de los operadores vistos hasta ahora en el libro.

4.13 Conclusión

Este capítulo presentó las técnicas básicas de solución de problemas que los programadores utilizan para crear clases y desarrollar funciones miembro para estas clases. Demostramos cómo construir un algoritmo (es decir, una metodología para resolver un problema) en seudocódigo, y después cómo refinar el algoritmo a través de diversas fases de desarrollo de seudocódigo, lo cual produce código en C++ que puede ejecutarse como parte de una función. En este capítulo aprendió a utilizar el método de mejoramiento de arriba a abajo, paso a paso, para planear las acciones específicas que debe realizar una función, y el orden en el que debe realizar estas acciones.

Aprendió que sólo hay tres tipos de estructuras de control (secuencia, selección y repetición) necesarias para desarrollar cualquier algoritmo. Demostramos dos de las instrucciones de selección de C++: la instrucción de selección simple `if` y la instrucción de selección doble `if...else`. La instrucción `if` se utiliza para ejecutar un conjunto de instrucciones basadas en una condición; si la condición es verdadera, se ejecutan las instrucciones; si no, se omiten. La instrucción de selección doble `if...else` se utiliza para ejecutar un conjunto de instrucciones si se cumple una condición, y otro conjunto de instrucciones si la condición es falsa. Después vimos la instrucción de repetición `while`, en donde un conjunto de instrucciones se ejecutan de manera repetida, mientras que una condición sea verdadera. Utilizamos el apilamiento de instrucciones de control para calcular el total y el promedio de un conjunto de calificaciones de estudiantes, mediante la repetición controlada por un contador y controlada por un centinela, y utilizamos el anidamiento de instrucciones de control para analizar y tomar decisiones con base en un conjunto de resultados de un examen. Vimos una introducción a los operadores de asignación, que pueden utilizarse para abreviar instrucciones. Presentamos los operadores de incremento y decremento, los cuales se pueden utilizar para sumar o restar el valor 1 de una variable. En el siguiente capítulo continuaremos nuestra discusión acerca de las instrucciones de control, en donde presentaremos las instrucciones `for`, `do...while` y `switch`.

Resumen

Sección 4.2 Algoritmos

- Un algoritmo (pág. 105) es un procedimiento para resolver un problema, en términos de las acciones a ejecutar y el orden en el que se ejecutan.
- El proceso de especificar el orden en el que se ejecutan las instrucciones en un programa se denomina control del programa (pág. 106).

Sección 4.3 Seudocódigo

- El seudocódigo (pág. 106) ayuda al programador a idear un programa antes de intentar escribirlo en un lenguaje de programación.

Sección 4.4 Estructuras de control

- Un diagrama de actividad modela el flujo de trabajo (también conocido como la actividad; pág. 108) de un sistema de software.
- Los diagramas de actividad (pág. 107) se componen de símbolos tales como los símbolos de estados de acción, rombos y pequeños círculos, que se conectan mediante flechas de transición, las cuales representan el flujo de la actividad.
- Al igual que el seudocódigo, los diagramas nos ayudan a desarrollar y representar algoritmos.
- Un estado de acción se representa como un rectángulo en el que sus lados izquierdo y derecho se sustituyen con arcos que se curvean hacia fuera. La expresión de acción (pág. 108) aparece dentro del estado de acción.
- Las flechas en un diagrama de actividad representan las transiciones (pág. 108), que indican el orden en el que ocurren las acciones representadas por los estados de acción.
- El círculo relleno en un diagrama de actividad representa el estado inicial (pág. 108): el comienzo del flujo de trabajo antes de que el programa realice las acciones modeladas.
- El círculo sólido rodeado por una circunferencia, que aparece en la parte inferior del diagrama de actividad, representa el estado final (pág. 108): el término del flujo de trabajo después de que el programa realiza sus acciones.
- Los rectángulos con las esquinas superiores derechas dobladas se llaman notas (pág. 108) en UML. Una línea punteada (pág. 108) conecta a cada nota con el elemento que ésta describe.
- Existen tres tipos de estructuras de control (pág. 107): secuencia, selección y repetición.
- La estructura de secuencia está integrada en C++; de manera predeterminada, las instrucciones se ejecutan en el orden en el que aparecen.
- Una estructura de selección elige uno de varios cursos alternativos de acción.

Sección 4.5 Instrucción de selección `if`

- La instrucción `if` de selección simple (pág. 110) ejecuta (selecciona) una acción si una condición es verdadera, o la ignora si la condición es falsa.
- Un símbolo de decisión (pág. 111) en un diagrama de actividad indica que se debe tomar una decisión. El flujo de trabajo sigue una ruta determinada por las condiciones de guardia asociadas. Cada flecha de transmisión que sale de un símbolo de decisión tiene una condición de guardia. Si una condición de guardia es verdadera, el flujo de trabajo entra al estado de acción al que apunta la flecha de transición.

Sección 4.6 Instrucción de selección doble `if...else`

- La instrucción `if...else` de selección doble (pág. 112) ejecuta (selecciona) una acción cuando la condición es verdadera, y otra acción distinta cuando la condición es falsa.
- Para incluir varias instrucciones en el cuerpo del `if` (o en el cuerpo del `else` para una instrucción `if...else`), encierre las instrucciones entre llaves (`{` y `}`). A un conjunto de instrucciones contenidas dentro de un par de llaves se le llama bloque (pág. 115). Un bloque puede colocarse en cualquier parte de un programa en donde se pueda colocar una sola instrucción.
- Una instrucción nula (pág. 116), la cual implica que no puede realizarse ninguna acción, se indica mediante un punto y coma (;).

Sección 4.7 Instrucción de repetición `while`

- Una instrucción de repetición (pág. 116) repite una acción mientras cierta condición sea verdadera.
- En un símbolo de fusión de UML (pág. 117) hay dos o más flechas de transición que apuntan al rombo y sólo una que sale de él, para indicar que se fusionan múltiples flujos de actividad para continuar con la actividad.

Sección 4.8 Cómo formular algoritmos: repetición controlada por un contador

- La repetición controlada por un contador (pág. 118) se utiliza cuando se conoce el número de repeticiones antes de que un ciclo empiece a ejecutarse; es decir, cuando hay una repetición definida.
- Una suma de enteros puede producir un valor que sea demasiado grande como para almacenarlo en una variable `int`. A esto se le conoce como desbordamiento aritmético y provoca un comportamiento impredecible en tiempo de ejecución.
- Los valores máximo y mínimo que pueden almacenarse en una variable `int` se representan mediante las constantes `INT_MAX` e `INT_MIN`, respectivamente, del encabezado `<climits>`.
- Se considera una buena práctica asegurar que los cálculos aritméticos no se desborden antes de realizarlos. En código de uso industrial, hay que realizar revisiones para todos los cálculos que puedan producir un desbordamiento o subdesbordamiento.

Sección 4.9 Cómo formular algoritmos: repetición controlada por un centinela

- El proceso de mejoramiento de arriba a abajo, paso a paso (pág. 125) es un proceso para refinar el seudocódigo, manteniendo una representación completa del programa durante cada mejoramiento.
- La repetición controlada por un centinela (pág. 126) se utiliza cuando no se conoce el número de repeticiones antes de que un ciclo se empiece a ejecutar; es decir, cuando hay repetición indefinida.
- Un valor que contiene una parte fraccionaria se conoce como número de punto flotante y se representa de manera aproximada mediante tipos de datos como `float` y `double` (pág. 127).
- El operador de conversión de tipos `static_cast<double>` (pág. 132) se puede utilizar para crear una copia temporal de punto flotante del operando.
- Los operadores unarios (pág. 133) sólo reciben un operando; los operadores binarios reciben dos.
- El manipulador de flujo parametrizado `setprecision` (pág. 133) indica el número de dígitos de precisión que deben mostrarse a la derecha del punto decimal.
- El manipulador de flujo `fixed` (pág. 133) indica que los valores de punto flotante se deben imprimir en lo que se denomina formato de punto fijo, en oposición a la notación científica.
- En general, cualquier variable entera que deba almacenar sólo valores no negativos debe declararse con `unsigned` antes del tipo entero. Las variables de tipos `unsigned` pueden representar valores de 0 hasta aproximadamente el doble del rango positivo del tipo entero con signo correspondiente.
- Puede determinar el valor `unsigned int` máximo de su plataforma con la constante `UINT_MAX` de `<climits>`.

Sección 4.10 Cómo formular algoritmos: instrucciones de control anidadas

- Una instrucción de control anidada (pág. 134) aparece en el cuerpo de otra instrucción de control.
- C++11 introduce la nueva inicialización de listas para inicializar variables en sus declaraciones, como en:

```
int contadorEstudiantes = { 1 };
```

o

```
int contadorEstudiantes{ 1 };
```

- Las llaves (`{` y `}`) representan el inicializador de listas. Para una variable de tipo fundamental, sólo se coloca un valor en el inicializador de listas. Para un objeto, el inicializador de listas puede ser una lista de valores separada por comas, que se pasan al constructor del objeto.
- Para las variables de tipo fundamental, la sintaxis de inicialización de listas también previene lo que se conoce como conversiones de reducción, que podrían provocar pérdidas de datos.

Sección 4.11 Operadores de asignación

- Los operadores aritméticos `+=`, `-=`, `*=`, `/=` y `%=` abrevian las expresiones de asignación (pág. 140).

Sección 4.12 Operadores de incremento y decremento

- Los operadores de incremento (`++`) y de decremento (`--`) (pág. 140) incrementan o decrementan una variable en 1, respectivamente. Si el operador se coloca antes de la variable, ésta se incrementa o decrementa en 1 primero, y después su nuevo valor se utiliza en la expresión en la que aparece. Si el operador se coloca después de la variable, ésta se utiliza primero en la expresión en la que aparece, y después su valor se incrementa o decrementa en 1.

Ejercicios de autoevaluación

4.1 Complete los siguientes enunciados:

a) Todos los programas pueden escribirse en términos de tres tipos de estructuras de control: _____, _____ y _____.

b) La instrucción de selección _____ se utiliza para ejecutar una acción cuando una condición es verdadera, y otra acción cuando esa condición es `falsa`.

c) Al proceso de repetir un conjunto de instrucciones un número específico de veces se le llama repetición _____.

d) Cuando no se sabe de antemano cuántas veces se repetirá un conjunto de instrucciones, se puede usar un valor _____ para terminar la repetición.

4.2 Escriba cuatro instrucciones distintas en C++, en donde cada una sume `1` a la variable entera `x`.

4.3 Escriba instrucciones en C++ para realizar cada una de las siguientes tareas:

a) En una instrucción, asignar la suma del valor actual de `x` y `y` a `z`, y postincrementar el valor de `x`.

b) Determinar si el valor de la variable `cuenta` es mayor que `10`. De ser así, imprimir `"Cuenta es mayor que 10"`.

c) Predecrementar la variable `x` en `1`, luego restarla a la variable `total`.

d) Calcular el residuo después de dividir `q` entre `divisor`, y asignar el resultado a `q`. Escriba esta instrucción de dos maneras distintas.

4.4 Escriba instrucciones en C++ para realizar cada una de las siguientes tareas:

a) Declarar la variable `suma` como de tipo `unsigned int` e inicializarla con `0`.

b) Declarar la variable `x` como de tipo `unsigned int` e inicializarla con `1`.

c) Sumar la variable `x` a `suma` y asignar el resultado a la variable `suma`.

d) Imprimir la cadena `"La suma es: "`, seguida del valor de la variable `suma`.

4.5 Combine las instrucciones que escribió en el ejercicio 4.4 para formar un programa que calcule e imprima la suma de los enteros del 1 al 10. Use una instrucción `while` para iterar a través de las instrucciones de cálculo e incremento. El ciclo debe terminar cuando el valor de `x` se vuelva 11.

4.6 Determine el valor de *cada* una de estas variables `unsigned int` después de realizar el cálculo. Suponga que, cuando se empieza a ejecutar cada una de las instrucciones, todas las variables tienen el valor 5 entero.

a) `producto *= x++;`

b) `cociente /= ++x;`

4.7 Escriba instrucciones individuales de C++ o porciones de instrucciones que realicen lo siguiente:

a) Recibir como entrada la variable `unsigned int x` con `cin` y `>>`.

b) Recibir como entrada la variable `unsigned int y` con `cin` y `>>`.

c) Declarar la variable `unsigned int i` e inicializarla con `1`.

d) Declarar la variable entera `potencia` e inicializarla con `1`.

e) Multiplicar la variable `potencia` por `x` y asignar el resultado a `potencia`.

f) Preincrementar la variable `i` en `1`.

g) Determinar si `i` es menor o igual a `y`.

h) Imprimir la variable entera `potencia` con `cout` y `<<`.

4.8 Escriba un programa en C++ que utilice las instrucciones del ejercicio 4.7 para calcular `x` elevada a la `y` potencia. El programa debe tener una instrucción de repetición `while`.

4.9 Identifique y corrija los errores en cada uno de los siguientes fragmentos de código:

a)
```
while ( c <= 5 )
{
   product *= c;
   ++c;
```

b)
```
cin << valor;
```

c)
```
if ( genero == 1 )
   cout << "Mujer" << endl;
else;
   cout << "Hombre" << endl;
```

4.10 ¿Qué está mal en la siguiente instrucción de repetición `while`?

```
while ( z >= 0 )
  suma += z;
```

Respuestas a los ejercicios de autoevaluación

4.1 a) Secuencia, selección y repetición. b) `if...else` c) Controlada por contador o definida. d) Centinela, de señal, de bandera o de prueba.

4.2
```
x = x + 1;
x += 1;
++x;
x++;
```

4.3 a)
```
z = x++ + y;
```

b)
```
if ( cuenta > 10 )
   cout << "Cuenta es mayor que 10" << endl;
```

c)
```
total -= --x;
```

d)
```
q %= divisor;
q = q % divisor;
```

4.4 a)
```
unsigned int suma = 0;
```

b)
```
unsigned int x = 1;
```

c)
```
suma += x;
```
o
```
suma = suma + x;
```

d)
```
cout << "La suma es: " << suma << endl;
```

4.5 Vea el siguiente código:

```
1  // Solución al ejercicio 4.5: ej04_05.cpp
2  // Calcula la suma de los enteros del 1 al 10.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     unsigned int suma = 0; // almacena la suma de los enteros del 1 al 10
9     unsigned int x = 1; // contador
10
11    while ( x <= 10 ) // itera 10 veces
12    {
13       suma += x; // suma x a suma
14       ++x; // incrementa x
15    } // fin de while
16
17    cout << "La suma es: " << suma << endl;
18 } // fin de main
```

```
La suma es: 55
```

4.6 a) `producto = 25, x = 6;`
b) `quotient = 0, x = 6;`

4.7 a) `cin >> x;`
b) `cin >> y;`
c) `unsigned int i = 1;`
d) `unsigned int potencia = 1;`
e) `potencia *= x;`
o
`potencia = potencia * x;`
f) `++i;`
g) `if ( i <= y )`
h) `cout << potencia << endl;`

4.8 Vea el siguiente código:

```
1  // Solución al ejercicio 4.8: ej04_08.cpp
2  // Eleva x a la y potencia.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     unsigned int i = 1; // inicializa i para comenzar a contar desde 1
9     unsigned int potencia = 1; // inicializa potencia
10
11    cout << "Escriba la base como un entero: ";  // pide la base
12    unsigned int x; // base
13    cin >> x; // recibe la base como entrada
14
15    cout << "Escriba el exponente como un entero: "; // pide el exponente
16    unsigned int y; // exponente
17    cin >> y; // recibe el exponente como entrada
18
19    // cuenta de 1 a y y multiplica potencia por x cada vez
20    while ( i <= y )
21    {
22       potencia *= x;
23       ++i;
24    } // fin de while
25
26    cout << potencia << endl; // muestra el resultado
27 } // fin de main
```

```
Escriba la base como un entero: 2
Escriba el exponente como un entero: 3
8
```

4.9 a) *Error:* falta la llave derecha de cierre del cuerpo de la instrucción `while`.
Corrección: Agregar una llave derecha de cierre después de la instrucción `++c;`.
b) *Error:* Se utiliza inserción de flujo, en vez de extracción de flujo.
Corrección: cambie << a >>.
c) *Error:* El punto y coma después de `else` produce un error lógico. La segunda instrucción de salida siempre se ejecutará.
Corrección: Quitar el punto y coma después de `else`.

4.10 El valor de la variable z nunca se cambia en la instrucción `while`. Por lo tanto, si la condición de continuación de ciclo (`z >= 0`) es en un principio verdadera, se crea un ciclo infinito. Para evitar que ocurra un ciclo infinito, z debe decrementarse de manera que eventualmente se vuelva menor que 0.

Ejercicios

4.11 *(Corrija los errores de código)* Identifique y corrija los errores en cada uno de los siguientes fragmentos de código:

a)
```
if ( edad >= 65 );
   cout << "Edad es mayor o igual que 65" << endl;
else
   cout << "Edad es menor que 65 << endl";
```

b)
```
if ( edad >= 65 )
   cout << "Edad es mayor o igual que 65" << endl;
else;
   cout << "Edad es menor que 65 << endl";
```

c)
```
unsigned int x = 1;
unsigned int total;

while ( x <= 10 )
{
   total += x;
   ++x;
}
```

d)
```
While ( x <= 100 )
   total += x;
   ++x;
```

e)
```
while ( y > 0 )
{
   cout << y << endl;
   ++y;
}
```

4.12 *(¿Qué hace este programa?)* ¿Qué es lo que imprime el siguiente programa?

```
1  // Ejercicio 4.12: ej04_12.cpp
2  // ¿Qué imprime este programa?
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     unsigned int y = 0; // declara e inicializa y
9     unsigned int x = 1; // declara e inicializa x
10    unsigned int total = 0; // declara e inicializa el total
11
12    while ( x <= 10 ) // itera 10 veces
13    {
14       y = x * x; // realiza el cálculo
15       cout << y << endl; // imprime el resultado
16       total += y; // suma y al total
17       ++x; // incrementa el contador x
18    } // fin de while
19
20    cout << "El total es " << total << endl; // muestra el resultado
21 } // fin de main
```

Para los ejercicios 4.13 a 4.16, realice cada uno de los siguientes pasos:

a) Lea el enunciado del problema.
b) Formule el algoritmo utilizando seudocódigo y el proceso de mejoramiento de arriba a abajo, paso a paso.
c) Escriba un programa en C++.
d) Pruebe, depure y ejecute el programa en C++.

4.13 ***(Kilometraje de gasolina)*** Los conductores se preocupan acerca del kilometraje de sus automóviles. Un conductor ha llevado el registro de varios viajes, anotando los kilómetros conducidos y los litros usados en cada viaje. Desarrolle un programa en C++ que utilice una instrucción `while` para recibir como entrada los kilómetros conducidos y los litros usados por cada viaje, y que imprima el total de kilómetros por litro obtenidos en todos los reabastecimientos hasta este punto.

```
Escriba los kilometros usados (-1 para salir): 287
Escriba los litros: 13
KPL en este reabastecimiento: 22.076923
Total KPL: 22.076923

Escriba los kilometros usados (-1 para salir): 200
Escriba los litros: 10
KPL en este reabastecimiento: 20.000000
Total KPL: 21.173913

Escriba los kilometros usados (-1 para salir): 120
Escriba los litros: 5
KPL en este reabastecimiento: 24.000000
Total KPL: 21.678571

Escriba los kilometros usados (-1 para salir): -1
```

4.14 ***(Límites de crédito)*** Desarrolle una aplicación en C++ que determine si alguno de los clientes de una tienda de departamentos se ha excedido del límite de crédito en una cuenta. Para cada cliente se tienen los siguientes datos:

a) Número de cuenta (un entero)
b) Saldo al inicio del mes
c) Total de todos los artículos cargados por el cliente en el mes
d) Total de todos los créditos aplicados a la cuenta del cliente en el mes
e) Límite de crédito permitido.

El programa debe usar una instrucción `while` para recibir como entrada cada uno de estos datos, debe calcular el nuevo saldo (= saldo inicial + cargos – créditos) y determinar si éste excede el límite de crédito del cliente. Para los clientes cuyo límite de crédito se ha excedido, el programa debe mostrar el número de cuenta del cliente, su límite de crédito, el nuevo saldo y el mensaje "Se excedio el limite de su credito".

```
Introduzca el numero de cuenta (o -1 para salir): 100
Introduzca el saldo inicial: 5394.78
Introduzca los cargos totales: 1000.00
Introduzca los creditos totales: 500.00
Introduzca el limite de credito: 5500.00
El nuevo saldo es 5894.78
Cuenta:           100
Limite de credito: 5500.00
Saldo:            5894.78
Se excedio el limite de su credito.

Introduzca el numero de cuenta (o -1 para salir): 200
Introduzca el saldo inicial: 1000.00
Introduzca los cargos totales: 123.45
Introduzca los creditos totales: 321.00
Introduzca el limite de credito: 1500.00
El nuevo saldo es 802.45

Introduzca el numero de cuenta (o -1 para salir): -1
```

4.15 ***(Calculadora de comisiones de ventas)*** Una empresa grande paga a sus vendedores mediante comisiones. Los vendedores reciben $200 por semana, más el 9% de sus ventas brutas durante esa semana. Por ejemplo, un vendedor que vende $5000 de productos químicos en una semana, recibe $200 más el 9% de $5000, o un total de $650. Desarrolle un programa en C++ que utilice una instrucción `while` para recibir como entrada las ventas

brutas de cada vendedor de la semana anterior, y que calcule y muestre los ingresos de ese vendedor. Procese las cifras de un vendedor a la vez.

```
Introduzca las ventas en dolares (-1 para salir): 5000.00
El salario es: $650.00

Introduzca las ventas en dolares (-1 para salir): 6000.00
El salario es: $740.00

Introduzca las ventas en dolares (-1 para salir): 7000.00
El salario es: $830.00

Introduzca las ventas en dolares (-1 para salir): -1
```

4.16 ***(Calculadora de salario)*** Desarrolle un programa en C++ que utilice una instrucción `while` para determinar el sueldo bruto para cada uno de varios empleados. La empresa paga la "cuota normal" en las primeras 40 horas de trabajo de cada empleado, y paga "cuota y media" en todas las horas trabajadas que excedan de 40. Usted recibe una lista de los empleados de la empresa, el número de horas que trabajó cada empleado la semana pasada y la tarifa por horas de cada empleado. Su programa debe recibir como entrada esta información para cada empleado, debe determinar y mostrar el sueldo bruto de cada empleado.

```
Introduzca las horas trabajadas (-1 para salir): 39
Introduzca la tarifa por horas del empleado ($00.00): 10.00
El salario es $390.00

Introduzca las horas trabajadas (-1 para salir): 40
Introduzca la tarifa por horas del empleado ($00.00): 10.00
El salario es $400.00

Introduzca las horas trabajadas (-1 para salir): 41
Introduzca la tarifa por horas del empleado ($00.00): 10.00
El salario es $415.00

Introduzca las horas trabajadas (-1 para salir): -1
```

4.17 ***(Encontrar el más grande)*** El proceso de encontrar el número más grande (es decir, el máximo de un grupo de números) se utiliza frecuentemente en aplicaciones de computadora. Por ejemplo, un programa para determinar el ganador de un concurso de ventas recibe como entrada el número de unidades vendidas por cada vendedor. El vendedor que haya vendido más unidades es el que gana el concurso. Escriba un programa en seudocódigo y después una aplicación en C++ que utilice una instrucción `while` para determinar e imprimir el mayor número de una serie de 10 números introducidos por el usuario. Su programa debe utilizar tres variables, como se muestra a continuación:

`contador:`	Un contador para contar hasta 10 (es decir, para llevar el registro de cuántos números se han introducido, y para detectar cuando se hayan procesado los 10 números).
`numero:`	El número actual que se introduce al programa.
`mayor:`	El número más grande encontrado hasta ahora.

4.18 ***(Salida tabular)*** Escriba un programa en C++ que utilice una instrucción `while` y la secuencia de escape de tabulación `\t` para imprimir la siguiente tabla de valores:

```
N       10*N    100*N   1000*N

1       10      100     1000
2       20      200     2000
3       30      300     3000
4       40      400     4000
5       50      500     5000
```

4.19 ***(Encontrar los dos números más grandes)*** Utilizando una metodología similar a la del ejercicio 4.17, encuentre los *dos* valores más grandes de los 10 que se introdujeron. [*Nota:* debe introducir cada número sólo una vez].

4.20 ***(Validar la entrada del usuario)*** El programa de resultados de un examen de la figura 4.16 asume que cualquier valor introducido por el usuario que no sea un 1 debe ser un 2. Modifique la aplicación para validar sus entradas. Para cualquier entrada, si el valor introducido es distinto de 1 o 2, debe seguir iterando hasta que el usuario introduzca un valor correcto.

4.21 ***(¿Qué hace este programa?)*** ¿Qué es lo que imprime el siguiente programa?

```
// Ejercicio 4.21: ej04_21.cpp
// ¿Qué es lo que imprime este programa?
#include <iostream>
using namespace std;

int main()
{
   unsigned int cuenta = 1; // inicializa cuenta

   while ( cuenta <= 10 ) // itera 10 veces
   {
      // imprime una línea de texto
      cout << ( cuenta % 2 ? "****" : "++++++++" ) << endl;
      ++cuenta; // incrementa cuenta
   } // fin de while
} // fin de main
```

4.22 ***(¿Qué hace este programa?)*** ¿Qué es lo que imprime el siguiente programa?

```
// Ejercicio 4.22: ej04_22.cpp
// ¿Qué es lo que imprime este programa?
#include <iostream>
using namespace std;

int main()
{
   unsigned int fila = 10; // inicializa fila

   while ( fila >= 1 ) // itera hasta que fila < 1
   {
      unsigned int columna = 1; // establece columna a 1 cuando empieza la iteración

      while ( columna <= 10 ) // itera 10 vecess
      {
         cout << ( fila % 2 ? "<" : ">" ); // salida
         ++columna; // incrementa columna
      } // fin de while interior

      --fila; // decrementa fila
      cout << endl; // empieza nueva línea de salida
   } // fin de while exterior
} // fin de main
```

4.23 ***(Problema del `else` suelto)*** Determine la salida de cada uno de los siguientes conjuntos de código, cuando `x` es `11` y `y` es `9`. El compilador ignora la sangría en un programa en C++. El compilador de C++ siempre asocia un `else` con el `if` anterior, a menos que se le indique de otra forma mediante la colocación de llaves (`{}`). A primera vista, el programador tal vez no esté seguro de cuál `if` corresponde a cuál `else`; esta situación se conoce como el

"problema del `else` suelto". Hemos eliminado la sangría del siguiente código para hacer el problema más retador. [*Sugerencia:* aplique las convenciones de sangría que aprendió].

a)
```
if ( x < 10 )
if ( y > 10 )
cout << "*****" << endl;
else
cout << "#####" << endl;
cout << "$$$$$" << endl;
```

b)
```
if ( x < 10 )
{
if ( y > 10 )
cout << "*****" << endl;
}
else
{
cout << "#####" << endl;
cout << "$$$$$" << endl;
}
```

4.24 ***(Otro problema de `else` suelto)*** Modifique el siguiente código para producir la salida que se muestra. Utilice las técnicas de sangría apropiadas. No debe hacer modificaciones en el código, sólo insertar llaves. El compilador ignora la sangría en un programa en C++. Hemos eliminado la sangría en el código dado, para hacer el problema más retador. [*Nota:* es posible que no se requieran modificaciones].

```
if ( y == 8 )
if ( x == 5 )
cout << "@@@@@" << endl;
else
cout << "#####" << endl;
cout << "$$$$$" << endl;
cout << "&&&&&" << endl;
```

a) Suponiendo que `x = 5` y `y = 8`, se produce la siguiente salida:

```
@@@@@
$$$$$
&&&&&
```

b) Suponiendo que `x = 5` y `y = 8`, se produce la siguiente salida:

```
@@@@@
```

c) Suponiendo que `x = 5` y `y = 8`, se produce la siguiente salida:

```
@@@@@
&&&&&
```

d) Suponiendo que `x = 5` y `y = 7`, se produce la siguiente salida. [*Nota:* las tres últimas instrucciones de salida después del `else` forman parte de un bloque].

```
#####
$$$$$
&&&&&
```

4.25 ***(Cuadrado de asteriscos)*** Escriba un programa que pida al usuario que introduzca el tamaño del lado de un cuadrado y que muestre un cuadrado hueco de ese tamaño, compuesto de asteriscos y espacios en blanco. Su programa debe funcionar con cuadrados que tengan lados de todas las longitudes entre 1 y 20. Por ejemplo, si su programa lee un tamaño de 5, debe imprimir

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

4.26 ***(Palíndromos)*** Un palíndromo es un número o una frase de texto que se lee igual al derecho y al revés. Por ejemplo, cada uno de los siguientes enteros de cinco dígitos es un palíndromo: 12321, 55555, 45554 y 11611. Escriba una aplicación que lea un entero de cinco dígitos y determine si es un palíndromo. [*Sugerencia:* use los operadores de división y módulo para separar el número en sus dígitos individuales].

4.27 ***(Imprimir el equivalente decimal de un número binario)*** Escriba un programa que reciba como entrada un entero que contenga sólo ceros y unos (es decir, un entero "binario"), y que imprima su equivalente decimal. Use los operadores módulo y división para elegir los dígitos del número "binario" uno a la vez, de derecha a izquierda. En forma parecida al sistema numérico decimal, en donde el dígito más a la derecha tiene un valor posicional de 1 y el siguiente dígito a la izquierda tiene un valor posicional de 10, después 100, después 1000, etcétera, en el sistema numérico binario, el dígito más a la derecha tiene un valor posicional de 1, el siguiente dígito a la izquierda tiene un valor posicional de 2, luego 4, luego 8, etcétera. Así, el número decimal 234 se puede interpretar como 2 * 100 + 3 * 10 + 4 * 1. El equivalente decimal del número binario 1101 es 1 * 1 + 0 * 2 + 1 * 4 + 1 * 8, o 1 + 0 + 4 + 8, o 13. [*Nota:* para aprender más acerca de los números binarios, consulte el apéndice D].

4.28 ***(Patrón de ajedrez de asteriscos)*** Escriba un programa que muestre el siguiente patrón de tablero de damas. Su programa debe utilizar sólo tres instrucciones de salida, una para cada una de las siguientes formas:

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

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

4.29 ***(Múltiplos de 2 con un ciclo infinito)*** Escriba un programa que imprima las potencias del entero 2; a saber, 2, 4, 8, 16, 32, 64, etcétera. Su ciclo `while` no debe terminar (es decir, debe crear un ciclo infinito). Para ello, simplemente use la palabra clave `true` como la expresión para la instrucción `while`. ¿Qué ocurre cuando ejecuta este programa?

4.30 ***(Calcular el diámetro, la circunferencia y el área de un círculo)*** Escriba un programa que lea el radio de un círculo (como un valor `double`), calcule e imprima el diámetro, la circunferencia y el área. Use el valor 3.14159 para π.

4.31 ¿Qué está mal con la siguiente instrucción? Proporcione la instrucción correcta para realizar lo que probablemente el programador trataba de hacer.

```
cout << ++( x + y );
```

4.32 ***(Lados de un triángulo)*** Escriba un programa que lea tres valores `double` distintos de cero, y que determine e imprima si podrían representar los lados de un triángulo.

4.33 ***(Lados de un triángulo recto)*** Escriba un programa que lea tres enteros distintos de cero, y que determine e imprima si podrían ser los lados de un triángulo recto.

4.34 ***(Factorial)*** El factorial de un entero n no negativo se escribe como $n!$ (n factorial) y se define de la siguiente manera:

$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$ (para valores de n mayores o iguales a 1)

y

$n! = 1$ (para $n = 0$ o $n = 1$).

Por ejemplo, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, que es 120. Use instrucciones `while` en cada uno de los siguientes casos:

a) Escriba un programa que lea un entero no negativo, que calcule e imprima su factorial.

b) Escriba un programa que estime el valor de la constante matemática e, utilizando la fórmula:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

Pida al usuario la precisión deseada de e (es decir, el número de términos en la suma).

c) Escriba una aplicación que calcule el valor de e^x, utilizando la fórmula

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

Pida al usuario la precisión deseada de e (es decir, el número de términos en la suma).

4.35 ***(Inicializadores de listas de C++)*** Escriba instrucciones que usen la inicialización de listas de C++ para realizar cada una de las siguientes tareas:

a) Inicializar la variable `unsigned int contadorEstudiantes` con `0`.

b) Inicializar la variable `double saldoInicial` con `1000.0`

c) Inicializar un objeto de la clase `Cuenta` que proporcione un constructor que reciba un `unsigned int`, dos `string` y un `double` para inicializar los miembros de datos `numeroCuenta`, `primerNombre`, `apellido` y `saldo`.

Hacer la diferencia

4.36 ***(Implementar la privacidad con la criptografía)*** El crecimiento explosivo de las comunicaciones de Internet y el almacenamiento de datos en computadoras conectadas a Internet, ha incrementado de manera considerable los problemas de privacidad. El campo de la criptografía se dedica a la codificación de datos para dificultar (y, mediante los esquemas más avanzados, tratar de imposibilitar) su lectura a los usuarios no autorizados. En este ejercicio, usted investigará un esquema simple para cifrar y descifrar datos. Una compañía que desea enviar datos por Internet le pidió que escribiera un programa que los cifre, de modo que se puedan transmitir con más seguridad. Todos los datos se transmiten como enteros de cuatro dígitos. Su aplicación debe leer un entero de cuatro dígitos introducido por el usuario, y cifrarlo de la siguiente manera: Reemplace cada dígito con el resultado de sumarle 7 y obtenga el residuo después de dividir el nuevo valor entre 10. Después intercambie el primer dígito con el tercero, y el segundo dígito con el cuarto. Luego imprima el entero cifrado. Escriba una aplicación separada que reciba como entrada el número entero de cuatro dígitos cifrado y lo descifre (invirtiendo el esquema de cifrado) para formar el número original. [*Proyecto de lectura opcional:* investigue la "criptografía de clave pública" en general y el esquema de clave pública específico PGP (Privacidad bastante buena). Tal vez también quiera investigar el esquema RSA, que se utiliza mucho en las aplicaciones de nivel industrial].

4.37 ***(Crecimiento de la población mundial)*** La población mundial ha crecido de manera considerable a través de los siglos. El crecimiento continuo podría, en un momento dado, desafiar los límites del aire respirable, el agua potable, la tierra cultivable y otros recursos limitados. Hay evidencia de que el crecimiento se ha reducido en años

recientes, y que la población mundial podría llegar a su valor máximo en algún momento de este siglo, para luego empezar a disminuir.

Para este ejercicio, investigue en línea las cuestiones sobre el crecimiento de la población mundial. *Asegúrese de investigar varios puntos de vista*. Obtenga estimaciones de la población mundial actual y su tasa de crecimiento (el porcentaje por el cual es probable que aumente este año). Escriba un programa que calcule el crecimiento anual de la población mundial durante los siguientes 75 años, *utilizando la suposición simplificada de que la tasa de crecimiento actual permanecerá constante*. Imprima los resultados en una tabla. La primera columna debe mostrar el año, desde el año 1 hasta el año 75. La segunda columna debe mostrar la población mundial anticipada al final de ese año. La tercera columna deberá mostrar el aumento numérico en la población mundial que ocurriría ese año. Use sus resultados para determinar el año en el que el tamaño de la población será del doble del actual, si fuera a persistir la tasa de crecimiento de este año.

Instrucciones de control, parte 2: operadores lógicos

5

¿Quién puede controlar su destino?
—William Shakespeare

La llave usada siempre brilla.
—Benjamín Franklin

Objetivos

En este capítulo aprenderá a:

- Conocer los fundamentos de la repetición controlada por un contador.
- Usar las instrucciones de repetición `for` y `do...while` para ejecutar instrucciones repetidas veces en un programa.
- Implementar la selección múltiple mediante el uso de la instrucción de selección `switch`.
- Usar `break` y `continue` para alterar el flujo de control.
- Usar los operadores lógicos para formar expresiones condicionales complejas en las instrucciones de control.
- Evitar las consecuencias de confundir los operadores de igualdad y de asignación.

Plan general

5.1 Introducción

En este capítulo, continuaremos nuestra presentación de la programación estructurada, presentando el resto de las instrucciones de control en C++. Las instrucciones de control que estudiaremos aquí y las que vimos en el capítulo 4 son útiles para crear y manipular objetos. Continuaremos con nuestro énfasis anticipado sobre la programación orientada a objetos, que empezó con una discusión de los conceptos básicos en el capítulo 1, además de muchos ejemplos y ejercicios de código orientado a objetos en los capítulos 3 y 4.

En este capítulo demostraremos las instrucciones `for`, `do...while` y `switch`. A través de una serie de ejemplos cortos en los que utilizaremos las instrucciones `while` y `for`, exploraremos la repetición controlada por contador. Expandiremos la clase `LibroCalificaciones` que utiliza una instrucción `switch` para contar el número de calificaciones equivalentes de A, B, C, D y F, en un conjunto de calificaciones numéricas introducidas por el usuario. Presentaremos las instrucciones de control de programa `break` y `continue`. Hablaremos sobre los operadores lógicos, que nos permiten utilizar expresiones condicionales más complejas. También examinaremos el error común de confundir los operadores de igualdad (==) y desigualdad (=), y cómo evitarlo.

5.2 Fundamentos de la repetición controlada por un contador

Esta sección utiliza la instrucción de repetición `while`, presentada en el capítulo 4, para formalizar los elementos requeridos para llevar a cabo la repetición controlada por contador:

1. el **nombre de una variable de control** (o contador de ciclo)
2. el **valor inicial** de la variable de control
3. la **condición de continuación de ciclo**, que evalúa el **valor final** de la variable de control (es decir, determina si el ciclo debe continuar o no)
4. el **incremento** (o **decremento**) con el que se modifica la variable de control cada vez que pasa por el ciclo.

El programa simple de la figura 5.1 imprime los números del 1 al 10. La declaración en la línea 8 *nombra* a la variable de control (`contador`), la declara como `unsigned int`, reserva espacio para ella en memoria y la establece a un *valor inicial* de `1`. Las declaraciones que requieren inicialización son instrucciones *ejecutables*. En C++, es más preciso llamar a una declaración de variable que también reserva memoria a una **definición**. Como las definiciones también son declaraciones, utilizaremos el término "declaración" excepto cuando la distinción sea importante

En la línea 13 se *incrementa* el contador del ciclo en 1 cada vez que se ejecuta el cuerpo del ciclo. La condición de continuación de ciclo (línea 10) en la instrucción `while` determina si el valor de la variable

```
1  // Fig. 5.1: fig05_01.cpp
2  // Repetición controlada por un contador.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     unsigned int contador = 1; // declara e inicializa la variable de control
9
10    while ( contador <= 10 ) // condición de continuación de ciclo
11    {
12       cout << contador << " ";
13       ++contador; // incrementa la variable de control en 1
14    } // fin de while
15
16    cout << endl; // imprime una nueva línea
17 } // fin de main
```

```
1 2 3 4 5 6 7 8 9 10
```

Fig. 5.1 | Repetición controlada por contador.

de control es menor o igual que 10 (el valor final para el que la condición es `true`). El cuerpo de este `while` se ejecuta, aún y cuando la variable de control sea 10. El ciclo termina cuando la variable de control es mayor a 10 (es decir, cuando `contador` se convierte en 11).

La figura 5.1 se puede hacer más concisa si se inicializa `contador` con 0 y se sustituye la instrucción `while` con:

```
contador = 0;
while ( ++contador <= 10 ) // condición de continuación de ciclo
   cout << contador << " ";
```

Este código ahorra una instrucción, ya que el incremento se realiza de manera directa en la condición del `while`, *antes* de evaluarla. Además, el código elimina las llaves alrededor del cuerpo del `while`, ya que éste ahora sólo contiene *una* instrucción. La codificación de tal forma condensada puede producir programas que sean más difíciles de leer, depurar, modificar y mantener.

Tip para prevenir errores 5.1

Los valores de punto flotante son aproximados, por lo que controlar los ciclos de conteo con variables de punto flotante puede producir valores de contador imprecisos y pruebas incorrectas para la terminación. Controle los ciclos de conteo con valores enteros. Por separado, ++ y -- pueden usarse sólo con operandos enteros.

5.3 Instrucción de repetición for

Además de `while`, C++ cuenta con la **instrucción de repetición for**, la cual especifica los detalles de la repetición controlada por contador en una sola línea de código. Para ilustrar el poder del `for`, vamos a modificar el programa de la figura 5.1. El resultado se muestra en la figura 5.2.

Cuando la instrucción `for` (líneas 10 y 11) se empieza a ejecutar, la variable de control `contador` se declara e inicializa en 1. A continuación, el programa verifica la condición de continuación de ciclo (línea 10 entre los signos de punto y coma) `contador <= 10`. Como el valor inicial de `contador` es 1, la condición se satisface y la instrucción del cuerpo (línea 11) imprime el valor de `contador`, que es 1.

```
1  // Fig. 5.2: fig05_02.cpp
2  // Repetición controlada por contador con la instrucción for.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     // el encabezado de la instrucción for incluye la inicialización
9     // la condición de continuación del ciclo y el incremento.
10    for ( unsigned int contador = 1; contador <= 10; ++contador )
11       cout << contador << " ";
12
13    cout << endl; // imprime una nueva línea
14 } // fin de main
```

```
1 2 3 4 5 6 7 8 9 10
```

Fig. 5.2 | Repetición controlada por contador con la instrucción `for`.

Después, la expresión `++contador` incrementa la variable de control `contador` y el ciclo empieza de nuevo, con la prueba de continuación de ciclo. Ahora la variable de control es igual a 2, por lo que no se excede del valor final y el programa ejecuta la instrucción del cuerpo otra vez. Este proceso continúa hasta que el cuerpo del ciclo se haya ejecutado 10 veces y la variable de control `contador` se incremente a 11, con lo cual falla la prueba de continuación de ciclo y termina la repetición. El programa continúa, ejecutando la primera instrucción después de la instrucción `for` (en este caso, la instrucción de salida en la línea 13).

Componentes del encabezado de la instrucción `for`

La figura 5.3 muestra un análisis más detallado del encabezado de la instrucción `for` (línea 10) de la figura 5.2. Observe que el encabezado de la instrucción `for` "se encarga de todo": especifica cada uno de los elementos necesarios para la repetición controlada por contador con una variable de control. Si hay más de una instrucción en el cuerpo del `for`, se requieren llaves para encerrar el cuerpo del ciclo. Por lo general, las instrucciones `for` se utilizan para la repetición controlada por contador y las instrucciones `while` se usan para la repetición controlada por centinela.

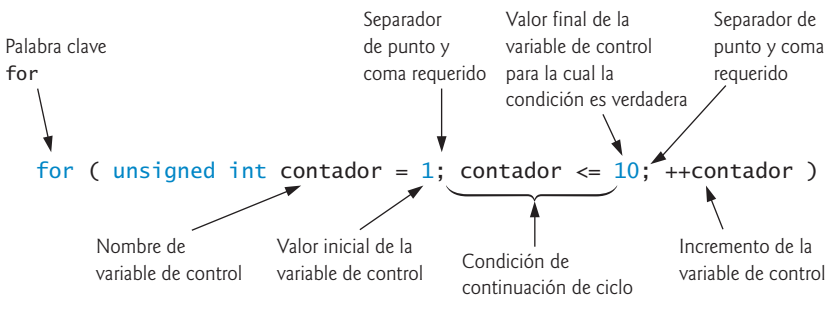


Fig. 5.3 | Componentes del encabezado de la instrucción `for`.

Errores por desplazamiento en uno

Si usted especificara por error `contador < 10` como la condición de continuación de ciclo en la figura 5.2, el ciclo sólo se ejecutaría 9 veces. A este error lógico común se le conoce como **error por desplazamiento en 1**.

Error común de programación 5.1

Utilizar un operador relacional incorrecto o un valor final incorrecto de un contador de ciclo en la condición de continuación de ciclo de una instrucción `while` *o* `for` *puede producir errores por desplazamiento en 1.*

Buena práctica de programación 5.1

Utilizar el valor final en la condición de una instrucción `while` *o* `for` *con el operador relacional* `<=` *nos ayuda a evitar los errores por desplazamiento en 1. Por ejemplo, para un ciclo que imprime los valores del 1 al 10, la condición de continuación de ciclo debe ser* `contador <= 10`*, en vez de* `contador < 10` *(lo cual produce un error por desplazamiento en uno) o* `contador < 11` *(que es correcto). Muchos programadores prefieren el llamado* ***conteo con base cero****, en el cual para contar 10 veces,* `contador` *se inicializaría a cero y la prueba de continuación de ciclo sería* `contador < 10`*.*

Formato general de una instrucción `for`

El formato general de la instrucción `for` es

```
for ( inicialización; condiciónDeContinuaciónDeCiclo; incremento )
   instrucción
```

en donde la expresión *inicialización* inicializa la variable de control del ciclo, la *condiciónDeContinuaciónDeCiclo* determina si el ciclo debe seguir ejecutándose y el *incremento* aumenta el valor de la variable de control. En la mayoría de los casos, la instrucción `for` se puede representar mediante una instrucción `while` equivalente, como se muestra a continuación:

```
inicialización;

while ( condiciónDeContinuaciónDeCiclo )
{
   instrucción
   incremento;
}
```

Hay una excepción a esta regla, que veremos en la sección 5.7.

Si la expresión de *inicialización* declara la variable de control (es decir, si el tipo de la variable de control se especifica antes del nombre de la variable), ésta puede utilizarse *sólo* en el cuerpo de esa instrucción `for`; ya que sería desconocida *fuera* de la instrucción `for`. Este uso restringido del nombre de la variable de control se conoce como el **alcance** de la variable. El alcance de una variable especifica en *dónde* puede utilizarse en un programa. En el capítulo 6 veremos con detalle el concepto de alcance.

Listas de expresiones separadas por comas

Las expresiones *inicialización* e *incremento* pueden ser listas de expresiones separadas por comas. Las comas, según el uso que se les da en estas expresiones, son **operadores coma**, los cuales garantizan que las listas de expresiones se evalúen de izquierda a derecha. El operador coma tiene la menor precedencia de todos los operadores de C++. El *valor y tipo de una lista de expresiones separadas por comas es el valor y tipo de la expresión que está más a la derecha*. El operador coma se utiliza con frecuencia en las instrucciones `for`. Su principal aplicación es permitir al programador utilizar *varias expresiones de inicialización*

y/o *varias expresiones de incremento*. Por ejemplo, puede haber distintas variables de control en una sola instrucción `for` que deban inicializarse e incrementarse.

Buena práctica de programación 5.2

Coloque sólo expresiones que involucren a las variables de control en las secciones de inicialización e incremento de una instrucción `for`.

Las expresiones en el encabezado de la instrucción `for` son opcionales

Las tres expresiones en el encabezado de la instrucción `for` son opcionales (pero los dos separadores de punto y coma son *obligatorios*). Si se omite la *condiciónDeContinuaciónDeCiclo*, C++ asume que esta condición es verdadera, con lo cual se crea un *ciclo infinito*. Podríamos omitir la expresión de *inicialización* si el programa inicializa la variable de control antes del ciclo. Podríamos omitir la expresión de *incremento* si el programa calcula el incremento mediante instrucciones dentro del cuerpo del `for`, o si no se necesita un incremento.

La expresión de incremento actúa como una instrucción independiente

La expresión de incremento en una instrucción `for` actúa como si fuera una instrucción independiente al final del cuerpo de la instrucción `for`. Por lo tanto, para contadores enteros, las expresiones

```
contador = contador + 1
contadorr += 1
++contador
contador++
```

son todas equivalentes en la expresión de *incremento* (cuando no aparece ningún otro código ahí). La variable que se incrementa aquí no aparece en una expresión más grande, por lo que los operadores de preincremento y postdecremento tienen en realidad el *mismo* efecto.

Error común de programación 5.2

Al colocar un punto y coma justo a la derecha del paréntesis derecho del encabezado de un `for`, el cuerpo de esa instrucción `for` se convierte en una instrucción vacía. Por lo general, esto es un error lógico.

Instrucción `for`: notas y observaciones

Las expresiones de inicialización, condición de continuación de ciclo e incremento de una instrucción `for` pueden contener expresiones aritméticas. Por ejemplo, si `x = 2` y `y = 10`, y además, `x` y `y` no se modifican en el cuerpo del ciclo, el siguiente encabezado de `for`:

```
for ( unsigned int j = x; j <= 4 * x * y; j += y / x )
```

es equivalente a la instrucción

```
for ( unsigned int j = 2; j <= 80; j += 5 )
```

El "incremento" de una instrucción `for` también puede ser negativo, en cuyo caso sería realmente un *decremento* y el ciclo contaría en orden *descendente* (como se muestra en la sección 5.4).

Si la condición de continuación de ciclo es *inicialmente falsa*, el programa no ejecutará el cuerpo de la instrucción `for`, sino que la ejecución continuará con la instrucción que siga inmediatamente después del `for`.

Con frecuencia, la variable de control se imprime o utiliza en cálculos dentro del cuerpo de una instrucción `for`, pero este uso no es obligatorio. Por lo general, la variable de control se utiliza para controlar la repetición sin que se le mencione dentro del cuerpo de la instrucción `for`.

Tip para prevenir errores 5.2

Aunque el valor de la variable de control puede cambiarse en el cuerpo de una instrucción `for`*, evite hacerlo, ya que esta práctica puede llevarlo a cometer errores sutiles.*

Diagrama de actividad de UML de la instrucción `for`

El diagrama de actividad de UML de la instrucción `for` es similar al de la instrucción `while` (figura 4.6). La figura 5.4 muestra el diagrama de actividad de la instrucción `for` de la figura 5.2. El diagrama hace evidente que la inicialización ocurre sólo una vez *antes* de evaluar la condición de continuación de ciclo por primera vez, y que el incremento ocurre *cada vez* que se realiza una iteración, *después* de que se ejecuta la instrucción del cuerpo. Observe que (además de un estado inicial, flechas de transición, una fusión, un estado final y varias notas) el diagrama sólo contiene *estados de acción* y una *decisión*.

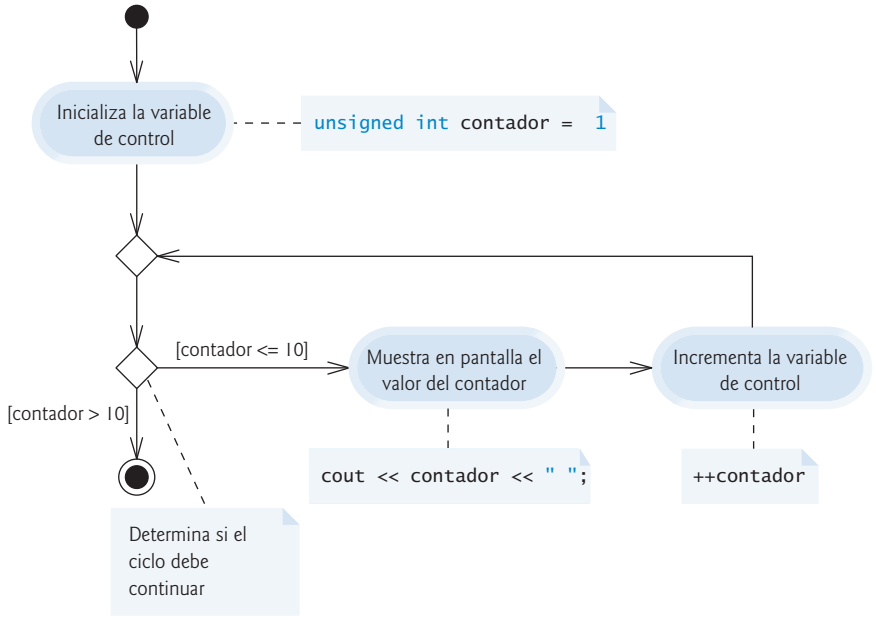


Fig. 5.4 | Diagrama de actividad de UML para la instrucción `for` de la figura 5.2.

5.4 Ejemplos acerca del uso de la instrucción for

Los siguientes ejemplos muestran métodos para modificar la variable de control en una instrucción `for`. En cada caso, escribimos el encabezado `for` apropiado. Observe el cambio en el operador relacional para los ciclos que *decrementan* la variable de control.

a) Modificar la variable de control de `1` a `100` en incrementos de `1`.

```
for ( unsigned int i = 1; i <= 100; ++i )
```

b) Modificar la variable de control de `100` a `0` en decrementos de `1`. Cabe mencionar que usamos el tipo `int` para la variable de control en este encabezado de `for`. La condición no se vuelve

falsa sino hasta que la variable de control contenga -1, por lo que ésta debe ser capaz de almacenar números tanto positivos como negativos.

```
for ( int i = 100; i >= 0; --i )
```

c) Modificar la variable de control de 7 a 77 en incrementos de 7.

```
for ( unsigned int i = 7; i <= 77; i += 7 )
```

d) Modificar la variable de control de 20 a 2 en incrementos de -2.

```
for ( unsigned int i = 20; i >= 2; i -= 2 )
```

e) Modificar la variable de control con la siguiente secuencia de valores: 2, 5, 8, 11, 14, 17.

```
for ( unsigned int i = 2; i <= 17; i += 3 )
```

f) Modificar la variable de control con la siguiente secuencia de valores: 99, 88, 77, 66, 55.

```
for ( unsigned int i = 99; i >= 55; i -= 11 )
```

Error común de programación 5.3

No utilizar el operador relacional apropiado en la condición de continuación de un ciclo que cuente en forma regresiva (como usar incorrectamente `i <= 1` en vez de `i >= 1` en un ciclo que cuente en forma regresiva hasta llegar a 1) es generalmente un error lógico que produce resultados incorrectos al momento de ejecutar el programa.

Error común de programación 5.4

No use operadores de igualdad (`!=` o `==`) en una condición de continuación de ciclo si la variable de control del ciclo se incrementa o decrementa por más de 1. Por ejemplo, considere el encabezado de la instrucción `for ( unsigned int contador = 1; contador != 10; contador += 2 )`. La prueba de continuación de ciclo `contador != 10;` nunca se volverá falsa (lo que produce un ciclo infinito*) debido a que contador se incrementa por 2 después de cada iteración.*

Aplicación: sumar los enteros pares del 2 al 20

El programa de la figura 5.5 utiliza una instrucción `for` para sumar los enteros pares del 2 al 20. Cada iteración del ciclo (líneas 11 y 12) suma el valor de la variable de control `numero` a la variable `total`.

```
1  // Fig. 5.5: fig05_05.cpp
2  // Suma de enteros con la instrucción for.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     unsigned int total = 0; // inicializa el total
9
10    // obtiene el total de los enteros pares del 2 al 20
11    for ( unsigned int numero = 2; numero <= 20; numero += 2 )
12       total += numero;
13
14    cout << "La suma es " << total << endl; // muestra los resultados
15 } // fin de main
```

Fig. 5.5 | Suma de enteros con la instrucción `for` (parte 1 de 2).

```
La suma es 110
```

Fig. 5.5 | Suma de enteros con la instrucción `for` (parte 2 de 2).

El cuerpo de la instrucción `for` de la figura 5.5 podría mezclarse con la porción del incremento del encabezado `for` mediante el uso del *operador coma*, como se muestra a continuación:

```
for ( unsigned int numero = 2; // inicialización
      numero <= 20; // condición de continuación de ciclo
      total += numero, numero += 2 ) // calcula el total e incrementa
   ; // cuerpo vacío
```

Buena práctica de programación 5.3

Aunque las instrucciones antes de un `for` y las instrucciones en el cuerpo de un `for` comúnmente se pueden fusionar en el encabezado del `for`, esto podría hacer que el programa fuera más difícil de leer, mantener, modificar y depurar.

Aplicación: cálculo del interés compuesto

Considere el siguiente enunciado del problema:

> *Una persona invierte $1000.00 en una cuenta de ahorro que produce el 5 por ciento de interés. Suponiendo que todo el interés se deposita en la cuenta, calcule e imprima el monto de dinero en la cuenta al final de cada año, durante 10 años. Use la siguiente fórmula para determinar los montos:*
>
> $c = p\,(1 + r)^n$
>
> *en donde*
>
> p *es el monto que se invirtió originalmente (es decir, el monto principal)*
> t *es la tasa de interés anual*
> n *es el número de años y*
> c *es la cantidad depositada al final del* $n^{\text{ésimo}}$ año.

La instrucción `for` (figura 5.6, líneas 21 a 28) ejecuta el cálculo indicado por cada uno de los 10 años que el dinero permanece en depósito, variando una variable de control de `1` a `10`, en incrementos de `1`. C++ *no* incluye un operador de exponenciación, por lo que utilizamos la **función pow de la biblioteca estándar** (línea 24). La función `pow(x, y)` calcula el valor de `x` elevado a la $y^{\text{ésima}}$ potencia. En este ejemplo, la expresión algebraica $(1 + r)^n$ se escribe como `pow(1.0 + tasa, anio)`, en donde la variable `tasa` representa a r y la variable `anio` representa a n. La función `pow` recibe dos argumentos de tipo `double` y devuelve un valor `double`.

```
1  // Fig. 5.6: fig05_06.cpp
2  // Cálculo del interés compuesto con for.
3  #include <iostream>
4  #include <iomanip>
5  #include <cmath> // biblioteca de matemáticas estándar
6  using namespace std;
7
8  int main()
9  {
```

Fig. 5.6 | Cálculo del interés compuesto con `for` (parte 1 de 2).

```
10     double monto; // monto a depositar al final de cada año
11     double principal = 1000.0; // monto inicial antes del interés
12     double tasa = .05; // atasa de interés anual
13
14     // muestra los encabezados
15     cout << "Anio" << setw( 21 ) << "Monto en deposito" << endl;
16
17     // establece el formato de número de punto flotante
18     cout << fixed << setprecision( 2 );
19
20     // calcula el monto en depósito para cada uno de los diez años
21     for ( unsigned int anio = 1; anio <= 10; ++anio )
22     {
23        // calcula el nuevo monto para el año especificado
24        monto = principal * pow( 1.0 + tasa, anio );
25
26        // muestra el año y el monto
27        cout << setw( 4 ) << anio << setw( 21 ) << monto << endl;
28     } // fin de for
29  } // fin de main
```

```
Anio   Monto en deposito
   1              1050.00
   2              1102.50
   3              1157.63
   4              1215.51
   5              1276.28
   6              1340.10
   7              1407.10
   8              1477.46
   9              1551.33
  10              1628.89
```

Fig. 5.6 | Cálculo del interés compuesto con `for` (parte 2 de 2).

Este programa no se compilará si no se incluye el encabezado <cmath> (línea 5). La función `pow` requiere dos argumentos `double`. La variable `anio` es un entero. El encabezado <cmath> incluye información que indica al compilador cómo convertir el valor de `anio` en una representación `double` temporal antes de llamar a la función. Esta información se incluye en el prototipo de la función `pow`. En el capítulo 6 veremos un resumen de las demás funciones matemáticas de la biblioteca.

Error común de programación 5.5

Olvidar incluir el encabezado apropiado al utilizar funciones de la biblioteca estándar (por ejemplo, <cmath> en un programa que utilice funciones matemáticas de la biblioteca) es un error de compilación.

Una advertencia en relación con el uso de los tipos `float` o `double` para cantidades monetarias

En las líneas 10 a 12 se declaran las variables `double` llamadas `monto`, `principal` y `tasa`. Hicimos esto para simplificar, ya que estamos tratando con partes fraccionarias de dólares, y necesitamos un tipo que permita puntos decimales en sus valores. Por desgracia, esto puede ocasionar problemas. He aquí una explicación simple de lo que puede salir mal al utilizar `float` o `double` para representar montos en dólares (asumiendo que se utiliza `setprecision(2)` para especificar dos dígitos de

precisión a la hora de imprimir): dos montos en dólares almacenados en el equipo podrían ser 14.234 (que se imprime como 14.23) y 18.673 (que se imprime como 18.67). Al sumar estos montos, producen la suma interna 32.907, la cual se imprime como 32.91. Por ende, el resultado podría aparecer como

```
  14.23
+ 18.67
-------
  32.91
```

pero ¡alguien que sumara los números individuales que aparecen impresos esperaría la suma de 32.90! ¡Ya ha sido advertido! En los ejercicios exploramos el uso de enteros para realizar cálculos monetarios. [*Nota:* algunos distribuidores independientes venden bibliotecas de clases de C++ que realizan cálculos monetarios precisos].

Uso de manipuladores de flujo para dar formato a los resultados numéricos

La instrucción de salida en la línea 18 antes del ciclo `for`, y la instrucción de salida en la línea 27 en el ciclo `for` se combinan para imprimir los valores de las variables `anio` y `monto`, con el formato especificado por los manipuladores de flujo parametrizados `setprecision` y **`setw`**, y por el manipulador de flujo no parametrizado `fixed`. El manipulador de flujo `setw(4)` especifica que el siguiente valor a imprimir debe aparecer en una **anchura de campo** de 4; por ejemplo, `cout` imprime el valor con *al menos* 4 posiciones de caracteres. Si el valor a imprimir es *menor* que 4 caracteres de ancho, de manera predeterminada se imprime **justificado a la derecha**. Si el valor a imprimir es *mayor* que 4 caracteres, la anchura de campo se extiende *a la derecha* para dar cabida a todo el valor. Para indicar que los valores deben imprimirse **justificados a la izquierda**, simplemente imprima el manipulador de flujo no parametrizado **`left`** (que se encuentra en el encabezado `<iostream>`). La justificación a la derecha se puede restaurar al imprimir el manipulador de flujo no parametrizado **`right`**.

El otro formato en las instrucciones de salida indica que la variable `monto` se imprime como un valor de punto fijo con un punto decimal (especificado en la línea 18 con el manipulador de flujo `fixed`), justificado a la derecha en un campo de 21 posiciones de caracteres (especificado en la línea 27 con `setw(21)`) y dos dígitos de precisión a la derecha del punto decimal (especificado en la línea 18 con el manipulador `setprecision(2)`). Aplicamos los manipuladores de flujo `fixed` y `setprecision` al flujo de salida (es decir, `cout`) antes del ciclo `for`, ya que estas opciones de formato están en vigor hasta que se modifican; a dichas opciones se les conoce como **opciones pegajosas** y *no* necesitan aplicarse durante cada iteración del ciclo. Sin embargo, la anchura de campo especificada con `setw` *sólo* se aplica al siguiente *valor* que se imprime. En el capítulo 13, Entrada/salida de flujos: un análisis detallado, hablaremos sobre las poderosas herramientas de formato de entrada/salida de C++.

El cálculo `1.0 + tasa`, que aparece como argumento para la función `pow`, está contenido en el cuerpo de la instrucción `for`. De hecho, este cálculo produce el *mismo* resultado durante cada iteración del ciclo, por lo que repetirlo es un desperdicio; debería realizarse una vez antes del ciclo.

Asegúrese de probar nuestro problema de Peter Minuit en el ejercicio 5.29. Este problema demuestra las maravillas del interés compuesto.

Tip de rendimiento 5.1

Evite colocar expresiones cuyos valores no cambien dentro de los ciclos. Si lo hace, muchos de los compiladores optimizadores sofisticados de la actualidad colocarán de manera automática dichas expresiones fuera *de los ciclos en el código de lenguaje máquina generado.*

Tip de rendimiento 5.2

Muchos compiladores contienen características de optimización que mejoran el rendimiento del código que el programador escribe, sin embargo, es mejor escribir buen código desde el principio.

5.5 Instrucción de repetición do...while

La instrucción de repetición `do...while` es similar a la instrucción `while`. En la instrucción `while`, la evaluación de la condición de continuación de ciclo ocurre al principio del ciclo, *antes* de ejecutar su cuerpo. La instrucción `do...while` evalúa la condición de continuación de ciclo *después* de ejecutar el cuerpo del ciclo; por lo tanto, *el cuerpo del ciclo siempre se ejecutará cuando menos una vez.*

La figura 5.7 utiliza una instrucción `do...while` para imprimir los números del 1 al 10. Al entrar a la instrucción `do...while`, en la línea 12 se imprime el valor de `contador` y en la línea 13 se incrementa `contador`. Después el programa evalúa la prueba de continuación de ciclo al final del mismo (línea 14). Si la condición es verdadera, el ciclo continúa a partir de la primera instrucción del cuerpo en la instrucción `do...while` (línea 12). Si la condición es falsa, el ciclo termina y el programa continúa con la siguiente instrucción después del ciclo (línea 16).

```
1  // Fig. 5.7: fig05_07.cpp
2  // La instrucción de repetición do...while.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     unsigned int contador = 1; // inicializa contador
9
10    do
11    {
12       cout << contador << " "; // muestra contador
13       ++counter; // incrementa contador
14    } while ( contador <= 10 ); // fin de do...while
15
16    cout << endl; // imprime una nueva línea
17 } // fin de main
```

```
1 2 3 4 5 6 7 8 9 10
```

Fig. 5.7 | La instrucción de repetición `do...while`.

Diagrama de actividad de UML de la instrucción `do...while`

La figura 5.8 contiene el diagrama de actividad de UML para la instrucción `do...while`, el cual hace evidente que la condición de continuación de ciclo no se evalúa sino hasta *después* que el ciclo ejecuta su cuerpo, por lo menos una vez. Compare este diagrama de actividad con el de la instrucción `while` (figura 4.6).

Llaves en una instrucción `do...while`

No es necesario utilizar llaves en la instrucción `do...while` si sólo hay una instrucción en el cuerpo; sin embargo, la mayoría de los programadores incluyen las llaves para evitar la confusión entre las instrucciones `while` y `do...while`. Por ejemplo:

```
while ( condición )
```

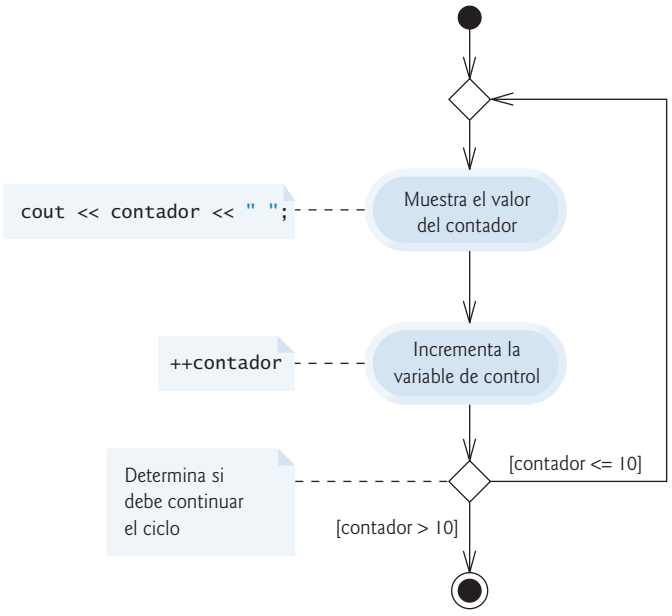


Fig. 5.8 | Diagrama de actividad de UML de la instrucción de repetición `do...while` de la figura 5.7.

generalmente se utiliza como encabezado de una instrucción `while`. Una instrucción `do...while` sin llaves, alrededor de un cuerpo con una sola instrucción, aparece así:

```
do
   instrucción
while ( condición );
```

lo cual puede ser confuso. Podríamos malinterpretar la última línea [`while(` *condición* `);`] como una instrucción `while` que contiene como cuerpo una instrucción vacía. Por ello, el `do...while` con una instrucción se escribe a menudo de la siguiente manera, para evitar confusión:

```
do
{
   instrucción
} while ( condición );
```

5.6 Instrucción de selección múltiple `switch`

C++ cuenta con la instrucción **`switch` de selección múltiple** para realizar muchas acciones distintas, con base en los posibles valores de una variable o expresión. Cada acción se asocia con un valor de una **expresión integral constante** (es decir, una combinación de constantes tipo carácter y constantes enteras que se evalúan como un valor entero constante).

La clase `LibroCalificaciones` con la instrucción `switch` para contar las calificaciones A, B, C, D y F

Esta siguiente versión de la clase `LibroCalificaciones` pide al usuario que introduzca un conjunto de calificaciones en forma de letras y después muestra un resumen del número de estudiantes que recibieron

cada calificación. La clase utiliza una instrucción `switch` para determinar si cada calificación introducida es el equivalente de A, B, C, D o F, y para incrementar el contador de la calificación apropiada. La clase `LibroCalificaciones` está definida en la figura 5.9, y las definiciones de sus funciones miembro aparecen en la figura 5.10. La figura 5.11 muestra entradas y salidas de ejemplo del programa `main` que utiliza la clase `LibroCalificaciones` para procesar un conjunto de calificaciones.

Al igual que las versiones anteriores de la definición de la clase, esta definición de `LibroCalificaciones` (figura 5.9) contiene prototipos de funciones para las funciones miembro `establecerNombreCurso` (línea 11), `obtenerNombreCurso` (línea 12) y `mostrarMensaje` (línea 13), así como el constructor de la clase (línea 10). La definición de la clase también declara el miembro de datos `private` llamado `nombreCurso` (línea 17).

Encabezado de la clase `LibroCalificaciones`

Ahora la clase `LibroCalificaciones` (figura 5.9) contiene cinco miembros de datos `private` adicionales (líneas 18 a 22): variables contador para cada categoría de calificación (A, B, C, D y F). La clase también contiene dos funciones miembro `public` adicionales: `recibirCalificaciones` y `mostrarReporteCalificaciones`. La función miembro `recibirCalificaciones` (declarada en la línea 14) lee un número arbitrario de calificaciones de letra del usuario mediante el uso de la repetición controlada por centinela, y actualiza el contador de calificaciones apropiado para cada calificación introducida. La función miembro `mostrarReporteCalificaciones` (declarada en la línea 15) imprime un reporte que contiene el número de estudiantes que recibieron cada calificación de letra.

```
1  // Fig. 5.9: LibroCalificaciones.h
2  // Definición de la clase LibroCalificaciones que cuenta calificaciones de letras.
3  // Las funciones miembro se definen en LibroCalificaciones.cpp
4  #include <string> // el programa usa la clase string estándar de C++
5
6  // definición de la clase LibroCalificaciones
7  class LibroCalificaciones
8  {
9  public:
10    explicit LibroCalificaciones( std::string ); // inicializa el nombre del curso
11    void establecerNombreCurso( std::string ); // establece el nombre del curso
12    std::string obtenerNombreCurso() const; // obtiene el nombre del curso
13    void mostrarMensaje() const; // muestra un mensaje de bienvenida
14    void recibirCalificaciones(); // recibe un número arbitrario de
                                    calificaciones del usuariouser
15    void mostrarReporteCalificaciones() const; // muestra un reporte con base en
                                                    la entrada del usuario
16 private:
17    std::string nombreCurso; // nombre del curso para este LibroCalificaciones
18    unsigned int aCuenta; // cuenta de calificaciones A
19    unsigned int bCuenta; // cuenta de calificaciones B
20    unsigned int cCuenta; // cuenta de calificaciones C
21    unsigned int dCuenta; // cuenta de calificaciones D
22    unsigned int fCuenta; // cuenta de calificaciones F
23 }; // fin de la clase LibroCalificaciones
```

Fig. 5.9 | Definición de la clase `LibroCalificaciones` que cuenta calificaciones de letras.

Archivo de código fuente de la clase `LibroCalificaciones`

El archivo de código fuente `LibroCalificaciones.cpp` (figura 5.10) contiene las definiciones de las funciones miembro para la clase `LibroCalificaciones`. Las líneas 11 a 15 en el constructor inicializan los cinco contadores de calificaciones con 0; cuando se crea un objeto `LibroCalificaciones` por primera vez, no se han introducido aún calificaciones. Estos contadores se incrementan en la función miembro `recibirCalificaciones` a medida que el usuario introduce las calificaciones. Las definicio-

nes de las funciones miembro `establecerNombreCurso`, `obtenerNombreCurso` y `mostrarMensaje` son idénticas a las de las versiones anteriores de la clase `LibroCalificaciones`.

```
1  // Fig. 5.10: LibroCalificaciones.cpp
2  // Definiciones de las funciones miembro para la clase LibroCalificaciones que
3  // utiliza una instrucción switch para contar calificaciones A, B, C, D y F.
4  #include <iostream>
5  #include "LibroCalificaciones.h" // incluye la definición de la clase
                                    LibroCalificaciones
6  using namespace std;
7
8  // el constructor inicializa nombreCurso con la cadena suministrada como
9  // argumento; inicializa los miembros de datos contadores a 0
10 LibroCalificaciones::LibroCalificaciones( string nombre )
11    : aCuenta( 0 ), // inicializa cuenta de calificaciones A con 0
12      bCuenta( 0 ), // inicializa cuenta de calificaciones B con 0
13      cCuenta( 0 ), // inicializa cuenta de calificaciones C con 0
14      dCuenta( 0 ), // inicializa cuenta de calificaciones D con 0
15      fCuenta( 0 )  // inicializa cuenta de calificaciones F con 0
16 {
17    establecerNombreCurso( nombre );
18 } // fin del constructor de LibroCalificaciones
19
20 // función para establecer el nombre del curso; limita el nombre a
      25 caracteres o menos
21 void LibroCalificaciones::establecerNombreCurso( string nombre )
22 {
23    if ( nombre.size() <= 25 ) // si nombre tiene 25 caracteres o menos
24       nombreCurso = nombre; // almacena el nombre del curso en el objeto
25    else // si el nombre es mayor que 25 caracteres
26    { // establece nombreCurso a los primeros 25 caracteres del parámetro nombre
27       nombreCurso = nombre.substr( 0, 25 ); // selecciona los primeros 25
                                                 caracteres
28       cerr << "El nombre \"" << nombre << "\" excede la longitud maxima (25).\n"
29          << "Se limito nombreCurso a los primeros 25 caracteres.\n" << endl;
30    } // fin de if...else
31 } // fin de la función establecerNombreCurso
32
33 // función para obtener el nombre del curso
34 string LibroCalificaciones::obtenerNombreCurso() const
35 {
36    return nombreCurso;
37 } // fin de la función obtenerNombreCursoe
38
39 // muestra un mensaje de bienvenida para el usuario de LibroCalificaciones
40 void LibroCalificaciones::mostrarMensaje() const
41 {
42    // esta instrucción llama a obtenerNombreCurso para obtener el
43    // nombre del curso que representa este LibroCalificaciones
44    cout << "Bienvenido al libro de calificaciones para\n" << obtenerNombreCurso()
          << "!\n"
45       << endl;
46 } // fin de la función mostrarMensaje
47
48 // recibe un número arbitrario de calificaciones del usuario; actualiza el
      contador de calificaciones
49 void LibroCalificaciones::recibirCalificaciones()
50 {
```

Fig. 5.10 | Clase `LibroCalificaciones` que usa la instrucción `switch` para contar calificaciones de letras (parte 1 de 3).

```
51    int calificacion; // calificacion introducida por el usuario
52
53    cout << "Escriba las calificaciones de letra." << endl
54       << "Escriba el caracter EOF para terminar la entrada." << endl;
55
56    // itera hasta que el usuario escriba la secuencia de fin de archivo
57    while ( ( calificacion = cin.get() ) != EOF )
58    {
59       // determina cuál calificación se introdujo
60       switch ( calificacion ) // instrucción switch anidada en el while
61       {
62          case 'A': // calificacion fue A mayúscula
63          case 'a': // o a minúscula
64             ++aCuenta; // incrementa aCuenta
65             break; // es necesario salir del switch
66
67          case 'B': // calificacion fue B mayúscula
68          case 'b': // o b minúscula
69             ++bCuenta; // incrementa bCuenta
70             break; // sale del switch
71
72          case 'C': // calificacion fue C mayúscula
73          case 'c': // o c minúscula
74             ++cCuenta; // incrementa cCuenta
75             break; // sale del switch
76
77          case 'D': // calificacion fue D mayúscula
78          case 'd': // o d minúscula
79             ++dCuenta; // incrementa dCuenta
80             break; // sale del switch
81
82          case 'F': // calificacion fue F mayúscula
83          case 'f': // o f minúscula
84             ++fCuenta; // incrementa fCuenta
85             break; // sale del switch
86
87          case '\n': // ignora caracteres de nueva línea,
88          case '\t': // tabuladores,
89          case ' ': // y espacios en la entrada
90             break; // sale del switch
91
92          default: // atrapa todos los demás caracteres
93             cout << "Se introdujo una letra de calificacion incorrecta."
94                << " Escriba una nueva calificación." << endl;
95             break; // opcional; saldrá del switch de todas formas
96       } // fin de switch
97    } // fin de while
98 } // fin de la función recibirCalificaciones
99
100 // muestra un reporte con base en las calificaciones introducidas por el usuario
101 void LibroCalificaciones::mostrarReporteCalificaciones() const
102 {
```

Fig. 5.10 | Clase `LibroCalificaciones` que usa la instrucción `switch` para contar calificaciones de letras (parte 2 de 3).

```
103    // imprime resumen de resultados
104    cout << "\n\nNumero de estudiantes que recibieron cada calificacion de letra:"
105       << "\nA: " << aCuenta // muestra el número de calificaciones A
106       << "\nB: " << bCuenta // muestra el número de calificaciones B
107       << "\nC: " << cCuenta // muestra el número de calificaciones C
108       << "\nD: " << dCuenta // muestra el número de calificaciones D
109       << "\nF: " << fCuenta // muestra el número de calificaciones F
110       << endl;
111 } // fin de la función mostrarReporteCalificaciones
```

Fig. 5.10 | Clase `LibroCalificaciones` que usa la instrucción `switch` para contar calificaciones de letras (parte 3 de 3).

Lectura de los datos de entrada tipo carácter

El usuario introduce las calificaciones de letras para un curso en la función miembro `recibirCalificaciones` (líneas 49 a 98). En el encabezado del `while`, en la línea 57, la asignación entre paréntesis `( calificacion = cin.get() )` se ejecuta primero. La función `cin.get()` lee un carácter del teclado y lo almacena en la variable entera `calificacion` (declarada en la línea 51). Por lo general, los caracteres se almacenan en las variables de tipo **`char`**; sin embargo, los caracteres se pueden almacenar en cualquier tipo de datos entero, ya que se garantiza que los tipos `short`, `int`, `long` y `long long` son por lo menos tan grandes como el tipo `char`. Por ende, podemos tratar a un carácter como entero o como carácter, dependiendo de su uso. Por ejemplo, la instrucción

```
cout << "El caracter (" << 'a' << ") tiene el valor "
   << static_cast< int > ( 'a' ) << endl;
```

imprime el carácter a y su valor entero de la siguiente manera:

```
El caracter (a) tiene el valor 97
```

El entero 97 es la representación numérica del carácter en la computadora. En el apéndice B se muestran los caracteres y sus equivalentes decimales del **conjunto de caracteres ASCII (Código estándar estadounidense para el intercambio de información)**.

En general, las instrucciones de asignación tienen el valor que se asigna a la variable en el lado izquierdo del signo =. Por ende, el valor de la expresión de asignación `calificacion = cin.get( )` es el mismo que el valor devuelto por `cin.get( )` y que se asigna a la variable `calificacion`.

El hecho de que las expresiones de asignación tengan valores puede ser útil para asignar el mismo valor a *distintas* variables. Por ejemplo,

```
a = b = c = 0;
```

evalúa primero la asignación `c = 0` (ya que el operador = asocia de derecha a izquierda). Después, a la variable `b` se le asigna el valor de `c = 0` (que es 0). Luego, a la variable `a` se le asigna el valor de `b = (c = 0)` (que también es 0). En el programa, el valor de `calificacion = cin.get()` se compara con el valor de `EOF` (un símbolo cuyo acrónimo significa "fin de archivo"). Utilizamos `EOF` (que por lo general tiene el valor –1) como el valor centinela. *Sin embargo, no debe escribir el valor –1, ni las letras EOF, como el valor centinela*. En vez de ello, debe escribir una *combinación de teclas dependiente del sistema*, que represente el "fin de archivo" para indicar que no hay más datos que introducir. `EOF` es una constante entera simbólica que se incluye en el programa mediante el encabezado `<iostream>`[1]. Si el valor asignado a `calificacion` es igual a `EOF`, el ciclo `while` (líneas 57 a 97) termina. Hemos optado por representar los caracteres introducidos en este programa como valores `int`, ya que `EOF` tiene el tipo `int`.

1 Para compilar este programa, algunos compiladores requieren el encabezado `<cstdio>` que define a `EOF`.

Introducción del indicador `EOF`

En los sistemas OS X/Linux/UNIX y muchos otros, el fin de archivo se introduce escribiendo la secuencia

```
<Ctrl> d
```

en una línea por sí sola. Esta notación significa que hay que oprimir al mismo tiempo la tecla *Ctrl* y la tecla *d*. En otros sistemas como Microsoft Windows, para introducir el fin de archivo se escribe

```
<Ctrl> z
```

[*Nota:* en algunos casos, hay que oprimir *Intro* después de escribir la secuencia de teclas anterior. Además, generalmente se muestran los caracteres `^Z` en la pantalla para representar el fin de archivo, como se muestra en la figura 5.11].

Tip de portabilidad 5.1

Las combinaciones de teclas para introducir el fin de archivo son dependientes del sistema.

Tip de portabilidad 5.2

La acción de evaluar la constante simbólica `EOF` *en vez de* `-1` *hace que los programas sean más portables. El estándar de C, del cual C++ adopta la definición de* `EOF`*, establece que* `EOF` *es un valor integral negativo, por lo que* `EOF` *podría tener distintos valores en diferentes sistemas.*

En este programa, el usuario introduce las calificaciones mediante el teclado. Cuando el usuario oprime la tecla *Intro* (o *Return*), la función `cin.get()` lee los caracteres, uno a la vez. Si el carácter introducido no es el fin de archivo, el flujo de control entra a la instrucción `switch` (figura 5.10, líneas 60 a 96), la cual incrementa el contador de calificaciones de letras apropiado.

Detalles acerca de la instrucción `switch`

La instrucción `switch` consiste en una serie de **etiquetas `case`** y un **caso `default`** opcional. Estas etiquetas se utilizan en este ejemplo para determinar qué contador incrementar, con base en una calificación. Cuando el flujo de control llega a la instrucción `switch`, el programa evalúa la expresión entre paréntesis (es decir, `calificacion`) que está después de la palabra clave `switch` (línea 60). A ésta se le conoce como **expresión de control**. La instrucción `switch` compara el valor de la expresión de control con cada etiqueta `case`. Suponga que el usuario introduce la letra `C` como una calificación. El programa compara `C` con cada `case` en la instrucción `switch`. Si ocurre una coincidencia (`case 'C' :` en la línea 72), el programa ejecuta las instrucciones para esa etiqueta `case`. Para la letra `C`, en la línea 74 se incrementa `cCuenta` en `1`. La instrucción `break` (línea 75) hace que el control del programa se reanude en la primera instrucción después del `switch`; en este programa, el control se transfiere a la línea 97. Esta línea marca el final del cuerpo del ciclo `while` que recibe las calificaciones (líneas 57 a 97), por lo que el control fluye hasta la condición del `while` (línea 57) para determinar si el ciclo debe continuar ejecutándose.

Las etiquetas `case` en nuestra instrucción `switch` evalúan explícitamente las versiones en minúscula y mayúscula de las letras A, B, C, D y F. Observe las etiquetas `case` en las líneas 62 y 63, que evalúan los valores `'A'` y `'a'` (ambos representan la calificación A). Al listar las etiquetas `case` de esta forma consecutiva, sin instrucciones entre ellas, pueden ejecutar el mismo conjunto de instrucciones; cuando la expresión de control se evalúe como `'A'` o `'a'`, se ejecutarán las instrucciones de las líneas 64 y 65. Cada etiqueta `case` puede tener varias instrucciones. La instrucción de selección `switch` no requiere llaves alrededor de varias instrucciones en cada `case`.

Sin instrucciones break, cada vez que ocurra una concordancia en la instrucción switch, se ejecutarán las instrucciones para esa etiqueta case *y* todas las etiquetas case subsiguientes, hasta llegar a una instrucción break o al final de la instrucción switch. Esta característica es perfecta para escribir un programa conciso, que muestre la canción iterativa "Los Doce Días de Navidad" en el ejercicio 5.28.

Error común de programación 5.6

Olvidar una instrucción break *cuando se necesita una en una instrucción* switch *es un error lógico.*

Error común de programación 5.7

Omitir el espacio entre la palabra case *y el valor integral que se está evaluando en una instrucción* switch *(como escribir* case3: *en vez de* case 3:*) es un error lógico. La instrucción* switch *no ejecutará las acciones apropiadas cuando su expresión de control tenga un valor de 3.*

Proporcionar un caso `default`

Si *no* ocurre una coincidencia entre el valor de la expresión de control y una etiqueta case, se ejecuta el caso default (líneas 92 a 95). Utilizamos el caso default en este ejemplo para procesar todos los valores de la expresión de control que no sean calificaciones válidas o caracteres de nueva línea, tabuladores o espacios. Si no ocurre una coincidencia, se ejecuta el caso default y las líneas 93-94 imprimen un mensaje de error indicando que se introdujo una letra de calificación incorrecta. Si no ocurre una coincidencia en una instrucción switch que no contenga un caso default, el control del programa continúa con la primera instrucción después de la instrucción switch.

Tip para prevenir errores 5.3

Proporcione un caso default *en las instrucciones* switch*. Los casos que no se evalúen en forma explícita en una instrucción* switch *sin un caso* default *deben ignorarse. Al incluir un caso* default*, nos enfocamos en la necesidad de procesar las condiciones excepcionales. Existen situaciones en las que no se necesita un procesamiento* default*. Aunque las cláusulas* case *y la cláusula del caso* default *en una instrucción* switch *pueden ocurrir en cualquier orden, es una práctica común colocar la cláusula* default *al último.*

Buena práctica de programación 5.4

El último case *en una instrucción* switch *no requiere una instrucción* break*. Algunos programadores incluyen este* break *por claridad y por simetría con otros casos.*

Ignorar los caracteres de nueva línea, tabuladores y de espacio en blanco en la entrada

Las líneas 87 a 90 en la instrucción switch de la figura 5.10 hacen que el programa omita los caracteres de nueva línea, tabuladores y espacios en blanco. Al leer los caracteres uno a la vez, pueden producirse ciertos problemas. Para hacer que el programa lea los caracteres, debemos enviarlos a la computadora, oprimiendo la tecla *Intro* en el teclado. Con esto se coloca un carácter de nueva línea en la entrada, después del carácter que deseamos procesar. A menudo, este carácter de nueva línea se debe procesar de una manera especial. Al incluir las cláusulas case anteriores en nuestra instrucción switch, evitamos que se imprima el mensaje de error en el caso default cada vez que nos encontramos un carácter de nueva línea, tabulador o espacio.

Prueba de la clase `LibroCalificaciones`

En la figura 5.11 se crea un objeto LibroCalificaciones (línea 8). En la línea 10 se invoca la función miembro mostrarMensaje del objeto para imprimir en pantalla un mensaje de bienvenida para el usuario. En la línea 11 se invoca la función miembro recibirCalificaciones del objeto para leer un conjunto de calificaciones del usuario y llevar el registro de cuántos estudiantes recibieron cada calificación. La

ventana de resultados en la figura 5.11 muestra un mensaje de error en respuesta a la acción de introducir una calificación inválida (es decir, `E`). En la línea 12 se invoca la función miembro `mostrarReporteCalificaciones` de `LibroCalificaciones` (definida en las líneas 101 a 111 de la figura 5.10), la cual imprime en pantalla un reporte con base en las calificaciones introducidas (como en los resultados de la figura 5.11).

```
1  // Fig. 5.11: fig05_11.cpp
2  // Creación de un objeto LibroCalificaciones e invocación de sus funciones
   miembro.
3  #include "LibroCalificaciones.h" // incluye la definición de la clase
                                    LibroCalificaciones
4
5  int main()
6  {
7     // crea un objeto LibroCalificaciones
8     LibroCalificaciones miLibroCalificaciones( "CS101 Programacion en C++" );
9
10    miLibroCalificaciones.mostrarMensaje(); // muestra el mensaje de bienvenida
11    miLibroCalificaciones.recibirCalificaciones(); // lee las calificaciones
                                                        del usuario
12    miLibroCalificaciones.mostrarReporteCalificaciones();
      // muestra el reporte con base en las calificaciones
13 } // fin de main
```

```
Bienvenido al libro de calificaciones para
CS101 Programacion en C++!

Escriba las calificaciones de letra.
Escriba el caracter EOF para terminar la entrada.
a
B
c
C
A
d
f
C
E
Se introdujo una letra de calificacion incorrecta. Escriba una nueva calificacion.
D
A
b
^Z


Numero de estudiantes que recibieron cada calificacion de letra:
A: 3
B: 2
C: 3
D: 2
F: 1
```

Fig. 5.11 | Creación de un objeto `LibroCalificaciones` e invocación de sus funciones miembro.

Diagrama de actividad de UML de la instrucción `switch`

La figura 5.12 muestra el diagrama de actividad de UML para la instrucción de selección múltiple `switch` general. La mayoría de las instrucciones `switch` utilizan una instrucción `break` en cada `case` para

terminar la instrucción `switch` después de procesar el `case`. La figura 5.12 enfatiza esto al incluir instrucciones `break` en el diagrama de actividad. Sin la instrucción `break`, después de procesar un `case` el control no se transferiría a la primera instrucción después de la instrucción `switch`. En vez de ello, se transferiría a las acciones del siguiente `case`.

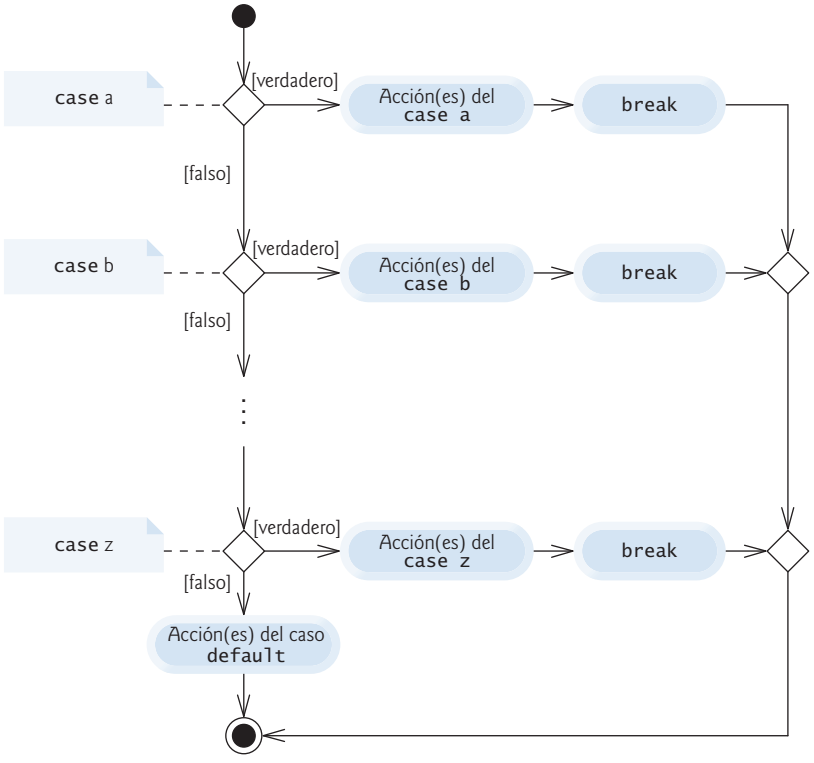


Fig. 5.12 | Diagrama de actividad de UML de la instrucción `switch` de selección múltiple con instrucciones `break`.

El diagrama hace evidente que la instrucción `break` al final de un `case` hace que el control salga de la instrucción `switch` de inmediato. De nuevo, observe que (además de un estado inicial, flechas de transición, un estado final y varias notas) el diagrama contiene estados de acción y decisiones. Además, el diagrama utiliza símbolos de fusión para fusionar las transiciones de las instrucciones `break` hacia el estado final.

Cuando utilice la instrucción `switch`, recuerde que cada `case` sólo se puede usar para evaluar una expresión integral *constante*: cualquier combinación de constantes carácter y enteras que se evalúen como un valor entero constante. Una constante de tipo carácter se representa como el carácter específico entre comillas sencillas, como `'A'`. Una constante entera es simplemente un valor entero. Además, cada etiqueta `case` puede especificar sólo una expresión integral constante.

Error común de programación 5.8

Especificar una expresión integral no constante en la etiqueta `case` de una instrucción `switch` es un error de sintaxis.

Error común de programación 5.9

Proporcionar etiquetas `case` idénticas en una instrucción `switch` es un error de compilación.

En el capítulo 12 presentaremos una manera más elegante de implementar la lógica con `switch`. Utilizaremos una técnica llamada *polimorfismo* para crear programas que sean con frecuencia más claros, concisos, fáciles de mantener y de extender que los programas que utilizan la lógica de `switch`.

Observaciones sobre los tipos de datos

C++ cuenta con *tipos de datos de tamaños flexibles* (vea el apéndice C, Tipos fundamentales). Por ejemplo, distintas aplicaciones podrían requerir enteros de distintos tamaños. C++ proporciona varios tipos de enteros. El rango de valores enteros para cada tipo de datos depende de la plataforma. Además de los tipos `int` y `char`, C++ proporciona los tipos `short` (una abreviación de `short int`), `long` (una abreviación de `long int`) y `long long` (una abreviación de `long long int`). El rango mínimo de valores para los enteros `short` es de -32 767 a 32 767. Para la amplia mayoría de los cálculos con enteros, basta con usar enteros `long`. El rango mínimo de valores para los enteros `long` es de -2 147 483 647 a 2 147 483 647. En la mayoría de las computadoras, los valores `int` son equivalentes a `short` o `long`. El rango de valores para un `int` es por lo menos el mismo que para los enteros `short`, y no más grande que para los enteros `long`. El tipo de datos `char` puede utilizarse para representar cualquiera de los caracteres en el conjunto de caracteres de la computadora. También se puede utilizar para representar enteros pequeños.

Inicializadores dentro de la clase en C++11

C++ nos permite proporcionar un valor predeterminado para un miembro de datos al declararlo en la declaración de la clase. Por ejemplo, las líneas 19 a 23 de la figura 5.9 podrían haber inicializado los miembros de datos `aCuenta`, `bCuenta`, `cCuenta`, `dCuenta` y `fCuenta` con `0`, como se muestra a continuación:

```
unsigned int aCuenta = 0; // cuenta de calificaciones A
unsigned int bCuenta = 0; // cuenta de calificaciones B
unsigned int cCuenta = 0; // cuenta de calificaciones C
unsigned int dCuenta = 0; // cuenta de calificaciones D
unsigned int fCuenta = 0; // cuenta de calificaciones F
```

en vez de inicializarlos en el constructor de la clase (figura 5.10, líneas 10 a 18). En capítulos posteriores seguiremos hablando sobre los inicializadores dentro de la clase y le mostraremos cómo nos permiten realizar ciertas inicializaciones de miembros de datos que no eran posibles en versiones anteriores de C++.

5.7 Instrucciones `break` y `continue`

C++ proporciona también las instrucciones `break` y `continue` para alterar el flujo de control. La sección anterior mostró cómo se puede utilizar `break` para terminar la ejecución de la instrucción `switch`. En esta sección veremos cómo usar `break` en una instrucción de repetición.

Instrucción `break`

Cuando la **instrucción break** se ejecuta en una instrucción `while`, `for`, `do...while`, o `switch`, ocasiona la salida inmediata de esa instrucción. La ejecución del programa continúa con la siguiente instrucción. Los usos comunes de la instrucción `break` son para escapar anticipadamente de un ciclo, o para omitir el resto de una instrucción `switch`. La figura 5.13 demuestra el uso de una instrucción `break` (línea 13) para salir de una instrucción de repetición `for`.

Cuando la instrucción `if` determina que `conteo` es `5`, se ejecuta la instrucción `break`. Esto termina la instrucción `for` y el programa continúa a la línea 18 (inmediatamente después de la instrucción `for`), la cual muestra un mensaje indicando el valor de la variable de control cuando terminó el ciclo. La instrucción `for` ejecuta su cuerpo por completo sólo cuatro veces en vez de 10. La variable de control

```
1  // Fig. 5.13: fig05_13.cpp
2  // instrucción break para salir de una instrucción for.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     unsigned int cuenta; // la variable de control también se usa después de que
                           termina el ciclo
9
10    for ( cuenta = 1; cuenta <= 10; ++cuenta ) // itera 10 veces
11    {
12       if ( cuenta == 5 )
13          break; // termina el ciclo sólo si cuenta es 5
14
15       cout << cuenta << " ";
16    } // fin de for
17
18    cout << "\nSalio del ciclo en cuenta = " << cuenta << endl;
19 } // fin de main
```

```
1 2 3 4
Salio del ciclo en cuenta = 5
```

Fig. 5.13 | Instrucción `break` para salir de una instrucción `for`.

`cuenta` se define fuera del encabezado de la instrucción `for`, por lo que podemos usar la variable de control tanto *en* el cuerpo del ciclo como *después* de que el ciclo completa su ejecución.

Instrucción `continue`

Cuando la **instrucción `continue`** se ejecuta en una instrucción `while`, `for` o `do...while`, omite las instrucciones restantes en el cuerpo de esa instrucción y continúa con la siguiente iteración del ciclo. En las instrucciones `while` y `do...while`, la prueba de continuación de ciclo se evalúa justo después de que se ejecuta la instrucción `continue`. En una instrucción `for` se ejecuta la expresión de incremento y después el programa evalúa la prueba de continuación de ciclo.

La figura 5.14 utiliza la instrucción `continue` (línea 11) en una instrucción `for` para omitir la instrucción de salida (línea 13) cuando la instrucción `if` anidada (líneas 10 y 11) determina que el valor de `cuenta` es `5`. Cuando se ejecuta la instrucción `continue`, el control del programa continúa con el incremento de la variable de control en el encabezado de la instrucción `for` (línea 8) e itera cinco veces más.

```
1  // Fig. 5.14: fig05_14.cpp
2  // instrucción continue para terminar una iteración de una instrucción for.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     for ( unsigned int cuenta = 1; cuenta <= 10; ++cuenta ) // itera 10 veces
9     {
```

Fig. 5.14 | Instrucción `continue` para terminar una iteración de una instrucción `for` (parte 1 de 2).

```
10        if ( cuenta == 5 ) // si cuenta es 5,
11           continue;       // omite el código restante en el ciclo
12
13        cout << cuenta << " ";
14     } // fin de for
15
16     cout << "\nSe uso continue para no imprimir el 5" << endl;
17  } // fin de main
```

```
1 2 3 4 6 7 8 9 10
Se uso continue para omitir imprimir 5
```

Fig. 5.14 | Instrucción `continue` para terminar una iteración de una instrucción `for` (parte 2 de 2).

En la sección 5.3 declaramos que la instrucción `while` puede utilizarse, en la mayoría de los casos, para representar a la instrucción `for`. La única excepción ocurre cuando la expresión de incremento en la instrucción `while` va después de la instrucción `continue`. En este caso, el incremento no se ejecuta antes de que el programa evalúe la condición de continuación de ciclo, por lo que el `while` no se ejecuta de la misma manera que el `for`.

Buena práctica de programación 5.5

Algunos programadores sienten que las instrucciones `break` y `continue` violan la programación estructurada. Como pronto veremos, pueden lograrse los mismos efectos de estas instrucciones con las técnicas de programación estructurada, por lo que estos programadores prefieren no utilizar instrucciones `break` o `continue`. La mayoría de los programadores consideran aceptable el uso de `break` en las instrucciones `switch`.

Observación de Ingeniería de Software 5.1

Existe una tensión entre lograr la ingeniería de software de calidad y lograr el software con mejor desempeño. A menudo, una de estas metas se logra a expensas de la otra. Para todas las situaciones excepto las que demanden el mayor rendimiento, aplique la siguiente regla empírica: primero, asegúrese de que su código sea simple y correcto; después hágalo rápido y pequeño, pero sólo si es necesario.

5.8 Operadores lógicos

Hasta ahora sólo hemos estudiado las **condiciones simples**, como `contador <= 10`, `total > 1000` y `numero != valorCentinela`. Expresamos esas condiciones en términos de los operadores relacionales >, <, >= y <=, y los operadores de igualdad == y !=. Cada decisión evaluó precisamente una condición. Para evaluar varias condiciones a la hora de tomar una decisión, realizamos estas pruebas en instrucciones separadas, o en instrucciones `if` o `if...else` anidadas.

C++ proporciona `operadores lógicos` que se utilizan para formar condiciones más complejas, al combinar condiciones simples. Los operadores lógicos son && (AND lógico), || (OR lógico) y ! (NOT lógico, también se conoce como negación lógica).

Operador AND lógico (&&)

Suponga que deseamos asegurar en cierto punto de una aplicación que dos condiciones sean *ambas* verdaderas, antes de elegir cierta ruta de ejecución. En este caso, podemos utilizar el operador **&& (AND lógico)** de la siguiente manera:

```
if ( genero == FEMENINO && edad >= 65 )
   ++mujeresMayores;
```

Esta instrucción `if` contiene dos condiciones simples. La condición `genero == FEMENINO` se utiliza aquí para determinar si una persona es de género femenino. La condición `edad >= 65` determina si una persona es un ciudadano mayor. La condición simple a la izquierda del operador `&&` se evalúa primero. Si es necesario, la condición simple a la derecha del operador `&&` se evalúa a continuación. Como veremos en breve, el lado derecho de una expresión AND lógica se evalúa *sólo* si el lado izquierdo es verdadero (`true`). Después, la instrucción `if` considera la condición combinada

```
genero == FEMENINO && edad >= 65
```

Esta condición es `true` si, y sólo si *ambas* condiciones simples son `true`. Por último, si esta condición combinada es evidentemente `true`, la instrucción en el cuerpo de la instrucción `if` incrementa la cuenta de `mujeresMayores`. Si alguna de esas condiciones simples es `false` (o ambas lo son), el programa omite el incremento y procede a la instrucción que va después del `if`. La condición combinada anterior puede hacerse más legible si se agregan paréntesis redundantes:

```
( genero == FEMENINO ) && ( edad >= 65 )
```

Error común de programación 5.10

Aunque `3 < x < 7` es una condición matemáticamente correcta, no se evalúa como podría esperarse en C++. Use `( 3 < x && x < 7 )` para obtener la evaluación apropiada en C++..

En la figura 5.15 se sintetiza el operador `&&`. Esta tabla muestra las cuatro combinaciones posibles de valores `false` y `true` para *expresión1* y *expresión2*. A dichas tablas se les conoce comúnmente como **tablas de verdad**. C++ evalúa todas las expresiones que incluyen operadores relacionales, de igualdad y/o lógicos como `true` o `false`.

expresión1	expresión2	expresión1 && expresión2
false	false	false
false	true	false
true	false	false
true	true	true

Fig. 5.15 | Tabla de verdad del operador && (AND lógico).

Operador OR lógico (||)

Ahora considere el operador `||` (**OR lógico**). Suponga que deseamos asegurar que de dos condiciones, una de ellas *o* ambas serán `true` antes de elegir cierta ruta de ejecución. En este caso, utilizamos el operador `||`, como en el siguiente segmento de un programa:

```
if ( ( promedioSemestre >= 90 ) || ( examenFinal >= 90 ) )
   cout << "La calificacion del estudiante es A" << endl;
```

Esta instrucción también contiene dos condiciones simples. La condición simple `promedioSemestre >= 90` se evalúa para determinar si el estudiante merece una "A" en el curso, debido a que tuvo un sólido rendimiento a lo largo del semestre. La condición simple `examenFinal >= 90` se evalúa para deter-

minar si el estudiante merece una "A" en el curso debido a un desempeño sobresaliente en el examen final. Después, la instrucción `if` considera la condición combinada

```
( promedioSemestre >= 90 ) || ( examenFinal >= 90 )
```

y otorga una "A" al estudiante si una o ambas de las condiciones simples son `true`. El mensaje "`La calificación del estudiante es A`" se imprime a menos que *ambas* condiciones simples sean `false`. La figura 5.16 es una tabla de verdad para el operador OR condicional (`||`).

expresión1	expresión2	expresión1 || expresión2
false	false	false
false	true	true
true	false	true
true	true	true

Fig. 5.16 | Tabla de verdad del operador `||` (OR lógico).

El operador `&&` tiene mayor precedencia que el operador `||`. Ambos operadores se asocian de izquierda a derecha. Una expresión que contiene operadores `&&` o `||` se evalúa sólo si se conoce verdad o falsedad de la expresión. Por ende, la evaluación de la expresión

```
( genero == FEMENINO ) && ( edad >= 65 )
```

se detiene de inmediato si `genero` no es igual a `FEMENINO` (es decir, toda la expresión es `false`) y continúa si `genero` es igual a `FEMENINO` (es decir, toda la expresión podría ser aún `true` si la condición `edad >= 65` es `true`). Esta característica de rendimiento para la evaluación de las expresiones con AND lógico y OR lógico se conoce como **evaluación en corto circuito**.

Tip de rendimiento 5.3

En las expresiones que utilizan el operador `&&`, si las condiciones separadas son independientes una de otra, haga que la condición que tenga más probabilidad de ser `false` sea la condición de más a la izquierda. En expresiones que utilicen el operador `||`, haga que la condición que tenga más probabilidad de ser `true` sea la condición de más a la izquierda. Este uso de la evaluación en corto circuito puede reducir el tiempo de ejecución de un programa.

Operador lógico de negación (!)

C++ cuenta con el operador `!` (**NOT lógico**, también conocido como **negación lógica**) para que un programador pueda "invertir" el significado de una condición. El operador lógico de negación unario sólo tiene una condición como operando. Este operador se coloca *antes* de una condición cuando nos interesa elegir una ruta de ejecución si la condición original (sin el operador lógico de negación) es `false`, como en el siguiente segmento de un programa:

```
if ( !( calificacion == valorCentinela ) )
   cout << "La siguiente calificacion es " << calificacion << endl;
```

Los paréntesis alrededor de la condición `calificacion == valorCentinela` son necesarios, ya que el operador lógico de negación tiene mayor precedencia que el operador de igualdad.

En la mayoría de los casos, puede evitar el operador `!` mediante el uso de un operador relacional o de igualdad apropiado. Por ejemplo, la instrucción `if` anterior también puede escribirse de la siguiente manera:

```
if ( calificacion != valorCentinela )
   cout << "La siguiente calificación es " << calificacion << endl;
```

Con frecuencia, esta flexibilidad puede ayudar a un programador a expresar una condición de una manera más "natural" o conveniente. La figura 5.17 es una tabla de verdad para el operador lógico de negación (`!`).

expresión	!expresión
false	true
true	false

Fig. 5.17 | Tabla de verdad del operador ! (negación lógica).

Ejemplo de los operadores lógicos

La figura 5.18 demuestra el uso de los operadores lógicos; para ello produce sus tablas de verdad. Los resultados muestran cada expresión que se evalúa y su resultado `boolean`. De manera predeterminada, los valores `bool true` y `false` se muestran mediante `cout` y el operador de inserción de flujo como `1` y `0`, respectivamente. Utilizamos el **manipulador de flujo `boolalpha`** (un manipulador *pegajoso*) en la línea 9 para especificar que el valor de cada expresión `bool` se debe mostrar como la palabra "true" o la palabra "false". Por ejemplo, el resultado de la expresión `false && false` en la línea 10 es `false`, así que la segunda línea de salida incluye la palabra "false". Las líneas 9 a 13 producen la tabla de verdad para el `&&`. Las líneas 16 a 20 producen la tabla de verdad para el `||`. Las líneas 23 a 25 producen la tabla de verdad para el `!`.

```
1  // Fig. 5.18: fig05_18.cpp
2  // Operadores lógicos.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     // crea la tabla de verdad para el operador && (AND lógico)
9     cout << boolalpha << "AND logico (&&)"
10       << "\nfalse && false: " << ( false && false )
11       << "\nfalse && true: " << ( false && true )
12       << "\ntrue && false: " << ( true && false )
13       << "\ntrue && true: " << ( true && true ) << "\n\n";
14
15    // crea la tabla de verdad para el operador || (OR lógico)
16    cout << "OR logico (||)"
17       << "\nfalse || false: " << ( false || false )
18       << "\nfalse || true: " << ( false || true )
19       << "\ntrue || false: " << ( true || false )
20       << "\ntrue || true: " << ( true || true ) << "\n\n";
```

Fig. 5.18 | Operadores lógicos (parte 1 de 2).

```
21
22     // crea la tabla de verdad para el operador ! (negación lógica)
23     cout << "NOT logico (!)"
24        << "\n!false: " << ( !false )
25        << "\n!true: " << ( !true ) << endl;
26  } // fin de main
```

```
AND logico (&&)
false && false: false
false && true: false
true && false: false
true && true: true

OR logico (||)
false || false: false
false || true: true
true || false: true
true || true: true

NOT logico (!)
!false: true
!true: false
```

Fig. 5.18 | Operadores lógicos (parte 2 de 2).

Resumen de precedencia y asociatividad de los operadores

La figura 5.19 agrega los operadores lógicos y de coma a la tabla de precedencia y asociatividad de los operadores. Los operadores se muestran de arriba hacia abajo, en orden descendente de precedencia.

Operadores	Asociatividad	Tipo
`::` `()`	izquierda a derecha *[Vea la precaución en la figura 2.10 con respecto al agrupamiento de paréntesis].*	primario
`++` `--` `static_cast<` *type* `>()`	izquierda a derecha	postfijo
`++` `--` `+` `-` `!`	derecha a izquierda	unario (prefijo)
`*` `/` `%`	izquierda a derecha	multiplicativo
`+` `-`	izquierda a derecha	aditivo
`<<` `>>`	izquierda a derecha	inserción/extracción
`<` `<=` `>` `>=`	izquierda a derecha	relacional
`==` `!=`	izquierda a derecha	igualdad
`&&`	izquierda a derecha	AND lógico
`||`	izquierda a derecha	OR lógico
`?:`	derecha a izquierda	condicional
`=` `+=` `-=` `*=` `/=` `%=`	derecha a izquierda	asignación
`,`	izquierda a derecha	coma

Fig. 5.19 | Precedencia/asociatividad de los operadores.

5.9 Confusión entre los operadores de igualdad (==) y de asignación (=)

Hay un tipo de error que los programadores de C++, sin importar su experiencia, tienden a cometer con tanta frecuencia que creemos requiere una sección separada. Ese error es el de intercambiar accidentalmente los operadores == (igualdad) y = (asignación). Lo que hace a estos intercambios tan peligrosos es el hecho de que por lo general *no* producen *errores sintácticos*; las instrucciones con estos errores tienden a compilarse correctamente y el programa se ejecuta hasta completarse, generando a menudo resultados incorrectos a través de *errores lógicos en tiempo de ejecución*. Algunos compiladores generan una *advertencia* cuando se utiliza = en un contexto en el que normalmente se espera ==.

Hay dos aspectos de C++ que contribuyen a estos problemas. Uno de ellos establece que *cualquier expresión que produce un valor se puede utilizar en la porción correspondiente a la decisión de cualquier instrucción de control*. Si el valor de la expresión es cero, se trata como `false`, y si el valor es distinto de cero, se trata como `true`. El segundo establece que las asignaciones producen un valor; a saber, el valor asignado a la variable del lado izquierdo del operador de asignación. Por ejemplo, suponga que tratamos de escribir

```
if ( codigoPago == 4 ) // bien
   cout << "Obtuvo un bono!" << endl;
```

pero accidentalmente escribimos

```
if ( codigoPago = 4 ) // mal
   cout << "Obtuvo un bono!" << endl;
```

La primera instrucción `if` otorga apropiadamente un bono a la persona cuyo `codigoPago` sea igual a 4. La segunda instrucción `if` (la del error) evalúa la expresión de asignación en la condición `if` a la constante 4. *Cualquier valor distinto de cero se interpreta como* `true`, por lo que esta condición siempre se evalúa como `true`, ¡y la persona *siempre* recibe un bono sin importar cuál sea el código de pago! Aún peor, ¡el código de pago se ha *modificado*, cuando se supone que sólo debía examinarse!

Error común de programación 5.11

El uso del operador == para asignación y el uso del operador = para igualdad son errores lógicos.

Tip para prevenir errores 5.4

Por lo general, los programadores escriben condiciones como `x == 7` con el nombre de la variable a la izquierda y la constante a la derecha. Al colocar la constante a la izquierda, como en `7 == x`, estará protegido por el compilador si accidentalmente sustituye el operador == con =. El compilador trata esto como un error de compilación, ya que no se puede modificar el valor de una constante. Esto evitará la potencial devastación de un error lógico en tiempo de ejecución.

lvalues *y* rvalues

Se dice que los nombres de las variables son ***lvalues*** (por "valores a la izquierda"), ya que pueden usarse del lado *izquierdo* de un operador de asignación. Se dice que las constantes son ***rvalues*** (por "valores a la derecha"), ya que sólo se pueden usar del lado *derecho* de un operador de asignación. Los *lvalues* también se pueden usar como *rvalues*, pero no al revés.

Hay otra situación que es igual de incómoda. Suponga que desea asignar un valor a una variable con una instrucción simple, como:

```
x = 1;
```

pero en vez de ello, escribe

```
x == 1;
```

Aquí podemos ver también que esto *no* es un error sintáctico. En vez de ello, el compilador simplemente evalúa la expresión condicional. Si `x` es igual a `1`, la condición es `true` y la expresión se evalúa con el valor `true`. Si `x` no es igual a `1`, la condición es `false` y la expresión se evalúa con el valor `false`. Sin importar el valor de la expresión, no hay operador de asignación, por lo que el valor sólo se pierde. El valor de `x` permanece sin alteraciones, lo que probablemente produzca un error lógico en tiempo de ejecución. ¡Por desgracia, no tenemos un truco disponible para ayudarlo con este problema!

Tip para prevenir errores 5.5

Use su editor de texto para buscar todas las ocurrencias de = en su programa, y compruebe que tenga el operador de asignación u operador lógico correcto en cada lugar.

5.10 Resumen de programación estructurada

Así como los arquitectos diseñan edificios, empleando la sabiduría colectiva de su profesión, de igual forma los programadores deben diseñar programas. Nuestro campo es mucho más joven que la arquitectura, y nuestra sabiduría colectiva es mucho más escasa. Hemos aprendido que la programación estructurada produce programas que son más fáciles de entender, probar, depurar, modificar que los programas no estructurados, e incluso probar que son correctos en sentido matemático.

La figura 5.20 utiliza diagramas de actividad para sintetizar las instrucciones de control de C++. Los estados inicial y final indican el *único punto de entrada* y el *único punto de salida* de cada instrucción de control. Si conectamos los símbolos individuales de un diagrama de actividad en forma arbitrara, existe la posibilidad de que se produzcan programas no estructurados. Por lo tanto, la profesión de la programación utiliza sólo un conjunto limitado de instrucciones de control que pueden combinarse sólo de dos formas simples, para crear programas estructurados.

Por cuestión de simpleza, sólo se utilizan *instrucciones de control de una sola entrada/una sola salida*; sólo hay una forma de entrar y una de salir de cada instrucción de control. Es sencillo conectar instrucciones de control en secuencia para formar programas estructurados: el estado final de una instrucción de control se conecta al estado inicial de la siguiente instrucción de control; es decir, las instrucciones de control se colocan una después de la otra en un programa. A esto le llamamos *apilamiento* de instrucciones de control. Las reglas para formar programas estructurados también permiten *anidar* las instrucciones de control.

La figura 5.21 muestra las reglas para formar programas estructurados. Las reglas suponen que pueden utilizarse estados de acción para indicar cualquier acción. Además, suponen que comenzamos con el *diagrama de actividad más sencillo* (figura 5.22), que consiste solamente de un estado inicial, un estado de acción, un estado final y flechas de transición.

Al aplicar las reglas de la figura 5.21, siempre se obtiene un diagrama de actividad con una agradable apariencia de bloque de construcción. Por ejemplo, si se aplica la regla 2 repetidamente al diagrama de actividad más sencillo, se obtiene un diagrama de actividad que contiene muchos estados de acción en secuencia (figura 5.23). La regla 2 genera una pila de estructuras de control, por lo que la llamaremos **regla de apilamiento**. Las líneas punteadas verticales en la figura 5.23 no son parte de UML; las utilizamos para separar los cuatro diagramas de actividad que demuestran cómo se aplica la regla 2 de la figura 5.21.

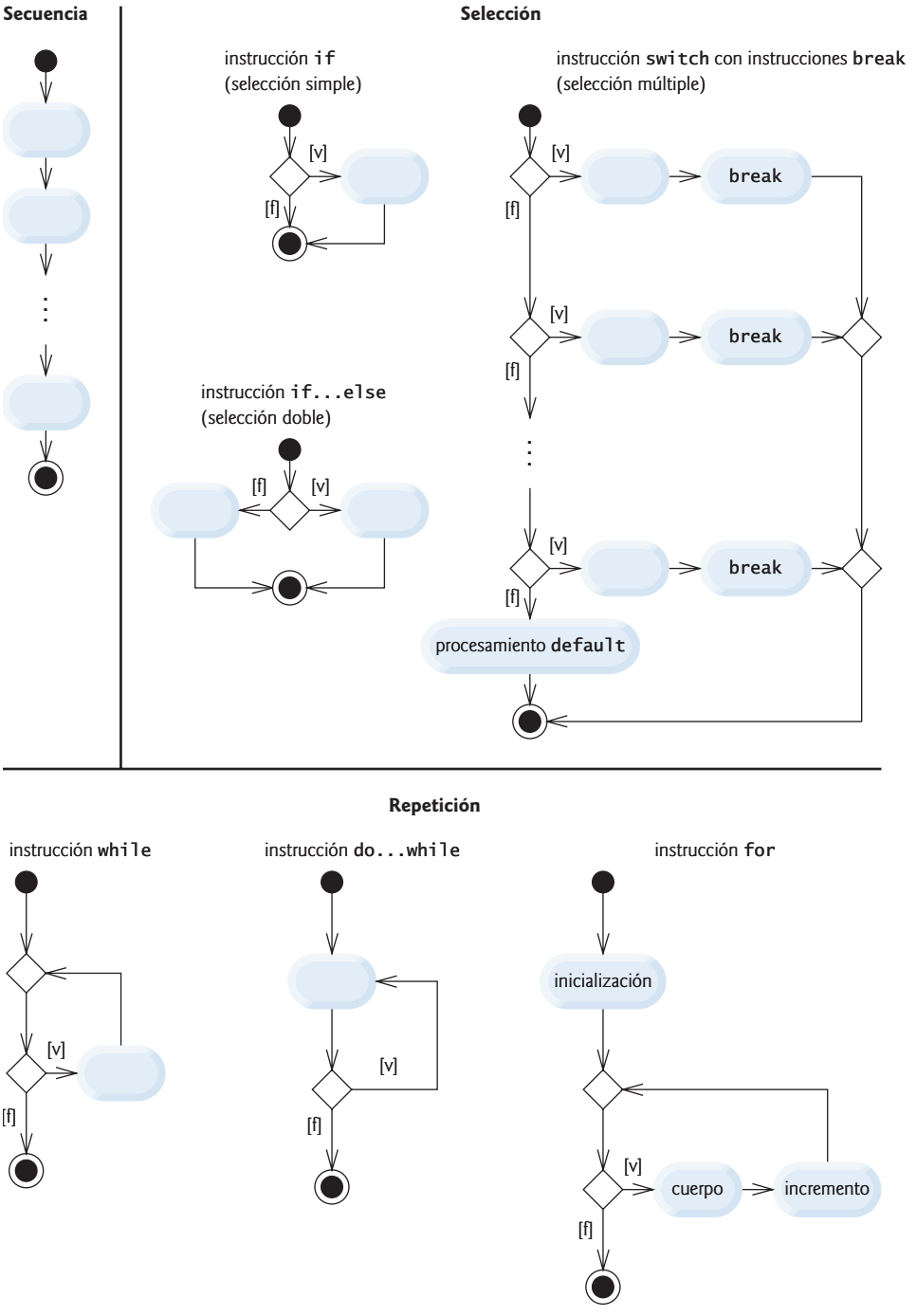


Fig. 5.20 | Instrucciones de secuencia, selección y repetición de una sola entrada/una sola salida de C++.

Reglas para formar programas estructurados

1) Comenzar con el "diagrama de actividad más sencillo" (figura 5.22).
2) Cualquier estado de acción puede reemplazarse por dos estados de acción en secuencia.
3) Cualquier estado de acción puede reemplazarse por cualquier instrucción de control (secuencia, `if`, `if...else`, `switch`, `while`, `do...while` o `for`).
4) Las reglas 2 y 3 pueden aplicarse tantas veces como se desee y en cualquier orden.

Fig. 5.21 | Reglas para formar programas estructurados.

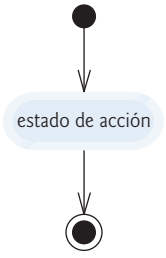


Fig. 5.22 | El diagrama de actividad más sencillo.

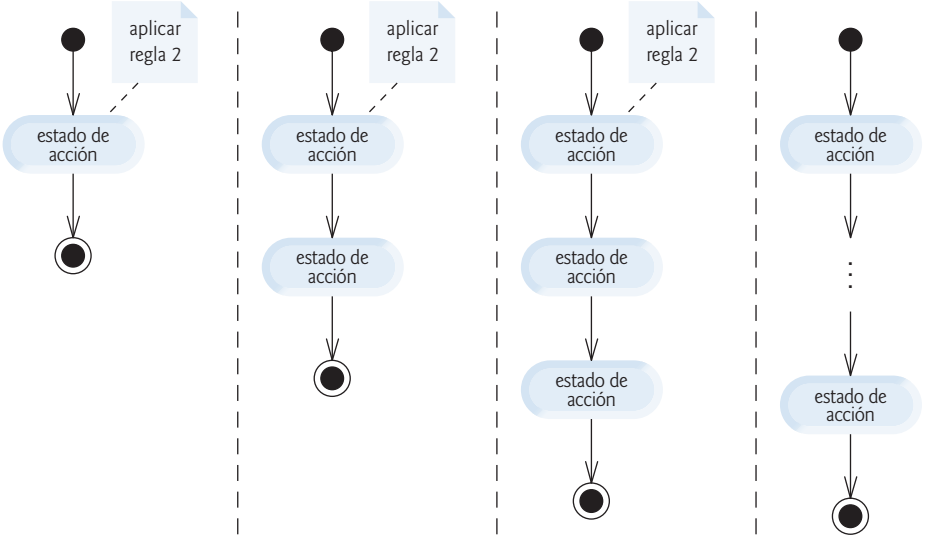


Fig. 5.23 | El resultado de aplicar la regla 2 de la figura 5.21 repetidamente al diagrama de actividad más sencillo.

La regla 3 se conoce como la **regla de anidamiento**. Al aplicarla repetidamente al diagrama de actividad más sencillo, se obtiene un diagrama de actividad con instrucciones de control perfectamente anidadas. Por ejemplo, en la figura 5.24 el estado de acción en el diagrama de actividad más sencillo se reemplaza con una instrucción de selección doble (`if...else`). Luego la regla 3 se aplica otra vez a los

estados de acción en la instrucción de selección doble, reemplazando cada uno de estos estados con una instrucción de selección doble. Los símbolos punteados de estado de acción alrededor de cada una de las instrucciones de selección doble representan el estado de acción que se reemplazó en el diagrama de actividad anterior. [*Nota:* las flechas punteadas y los símbolos punteados de estado de acción que se muestran en la figura 5.24 no son parte de UML. Aquí se utilizan como dispositivos pedagógicos para ilustrar que cualquier estado de acción puede reemplazarse con una instrucción de control].

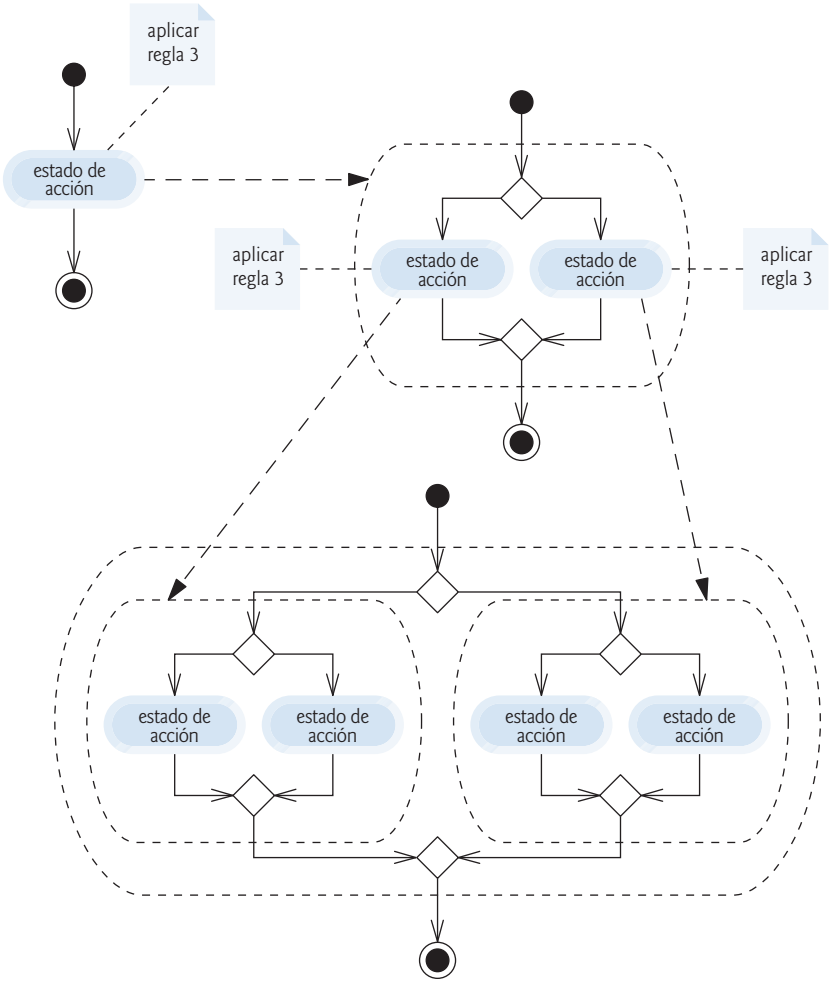


Fig. 5.24 | Aplicación de la regla 3 de la figura 5.21 varias veces al diagrama de actividad más sencillo.

La regla 4 genera instrucciones más grandes, más implicadas y más profundamente *anidadas*. Los diagramas que surgen debido a la aplicación de las reglas de la figura 5.21 constituyen el conjunto de todos los posibles diagramas de actividad estructurados y, por lo tanto, el conjunto de todos los posibles programas estructurados. La belleza de la metodología estructurada es que utilizamos solamente *siete* instrucciones de control simples de una sola entrada/una sola salida, y las ensamblamos en una de sólo *dos* formas simples.

Si se siguen las reglas de la figura 5.21, no podrá crearse un diagrama de actividad con una sintaxis ilegal (como el de la figura 5.25). Si usted no está seguro de que cierto diagrama sea estructurado, aplique las reglas de la figura 5.21 en orden inverso para tratar de reducir el diagrama al diagrama de actividad más sencillo. Si puede reducirlo, entonces el diagrama original es estructurado; de lo contrario, no es estructurado.

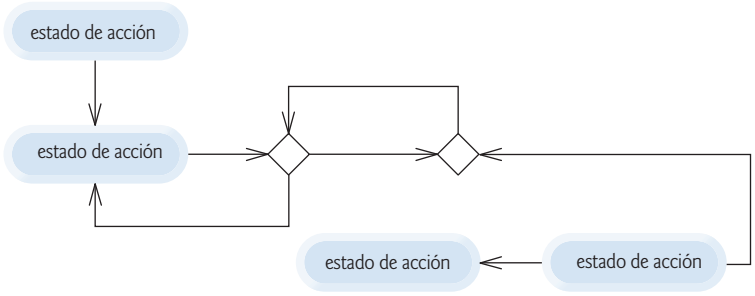


Fig. 5.25 | Diagrama de actividad con sintaxis ilegal.

La programación estructurada promueve la simpleza. Böhm y Jacopini nos han dado el resultado de que sólo se necesitan *tres* formas de control:

- Secuencia
- Selección
- Repetición

La estructura de secuencia es trivial. Simplemente enliste las instrucciones a ejecutar en el orden en el que deben ejecutarse.

La selección se implementa en una de tres formas:

- instrucción `if` (selección simple)
- instrucción `if...else` (selección doble)
- instrucción `switch` (selección múltiple)

Es sencillo demostrar que la instrucción *`if`* más simple *es suficiente* para proporcionar *cualquier* forma de selección; todo lo que pueda hacerse con las instrucciones `if...else` y `switch` puede implementarse si se combinan instrucciones `if` (aunque tal vez no con tanta claridad y eficiencia).

La repetición se implementa en una de tres maneras:

- instrucción `while`
- instrucción `do...while`
- instrucción `for`

Es sencillo demostrar que la *instrucción `while` es suficiente* para proporcionar cualquier forma de repetición. Todo lo que puede hacerse con las instrucciones `do...while` y `for`, puede hacerse también con la instrucción `while` (aunque tal vez no sea tan sencillo).

Si se combinan estos resultados, se demuestra que *cualquier* forma de control necesaria en un programa en C++ puede expresarse en términos de:

- secuencia
- instrucción `if` (selección)
- instrucción `while` (repetición)

y que estas tres instrucciones de control pueden combinarse en sólo *dos* formas: apilamiento y anidamiento. Evidentemente, la programación estructurada es la esencia de la simpleza.

5.11 Conclusión

Hemos completado nuestra introducción a las instrucciones de control, las cuales nos permiten controlar el flujo de la ejecución en los programas. El capítulo 4 trató acerca de las instrucciones de control `if`, `if...else` y `while`. En este capítulo demostramos las instrucciones de control `for`, `do...while` y `switch`. Mostramos que cualquier algoritmo puede desarrollarse mediante el uso de combinaciones de la estructura de secuencia, los tres tipos de instrucciones de selección (`if`, `if...else` y `switch`) y los tres tipos de instrucciones de repetición (`while`, `do...while` y `for`). Hablamos acerca de cómo puede combinar estos bloques de construcción para utilizar las técnicas, ya probadas, de construcción de programas y solución de problemas. Aprendió a usar las instrucciones `break` y `continue` para alterar el flujo de control de una instrucción de repetición. En este capítulo también se introdujeron los operadores lógicos, que nos permiten utilizar expresiones condicionales más complejas en las instrucciones de control. Por último, examinamos los errores comunes al confundir los operadores de igualdad y asignación, y proporcionamos sugerencias para evitar estos errores. En el capítulo 6 examinaremos las funciones con más detalle.

Resumen

Sección 5.2 Fundamentos de la repetición controlada por un contador

- En C++, es más preciso llamar definición a una declaración de variable que también reserva memoria (pág. 158).

Sección 5.3 Instrucción de repetición `for`

- La instrucción de repetición `for` (pág. 159) maneja todos los detalles de la repetición controlada por contador.
- El formato general de la instrucción `for` es

```
for ( inicialización; condiciónDeContinuacionDeCiclo; incremento )
   instrucción
```

  en donde la expresión *inicialización* inicializa la variable de control del ciclo, *condiciónDeContinuaciónDeCiclo* determina si el ciclo debe continuar su ejecución, e *incremento* incrementa o decrementa la variable de control.
- Por lo general, las instrucciones `for` se utilizan para la repetición controlada por contador y las instrucciones `while` para la repetición controlada por centinela.
- El alcance de una variable (pág. 161) especifica en qué parte de un programa se puede utilizar.
- El operador coma (pág. 161) tiene la menor precedencia de todos los operadores de C++. El valor y tipo de una lista de expresiones separadas por comas es el valor y tipo de la expresión que esté más a la derecha en la lista.
- Las expresiones de inicialización, condición de continuación de ciclo e incremento de una instrucción `for` pueden contener expresiones aritméticas. Además, el incremento de una instrucción `for` puede ser negativo.
- Si en un principio la condición de continuación de ciclo en un encabezado `for` es `false`, no se ejecuta el cuerpo de la instrucción `for`. En vez de ello, la ejecución se reanuda en la instrucción después del `for`.

Sección 5.4 Ejemplos acerca del uso de la instrucción `for`

- La función `pow(x, y)` de la biblioteca estándar (pág. 165) calcula el valor de `x` elevado a la $y^{ésima}$ potencia. La función `pow` recibe dos argumentos de tipo `double` y devuelve un valor `double`.
- El manipulador de flujo parametrizado `setw` (pág. 167) especifica la anchura de campo en la que debe aparecer el siguiente valor a imprimir, justificado a la derecha de manera predeterminada. Si el valor a imprimir es mayor que la anchura de campo, ésta se extiende para dar cabida al valor completo. El manipulador de flujo `left` (pág. 167) hace que un valor se justifique a la izquierda en un campo, y `right` (pág. 167) se puede usar para restaurar la justificación a la derecha.
- Las opciones pegajosas (pág. 167) son las opciones de formato que permanecen en vigor hasta que se modifican.

Sección 5.5 Instrucción de repetición `do...while`

- La instrucción de repetición `do...while` evalúa la condición de continuación de ciclo al final de éste, por lo que el cuerpo del ciclo se ejecutará al menos una vez. El formato para la instrucción `do...while` es

```
do
{
   instrucción
} while( condición );
```

Sección 5.6 Instrucción de selección múltiple `switch`

- La instrucción `switch` de selección múltiple (pág. 169) realiza distintas acciones, con base en el valor de su expresión de control.
- La función `cin.get()` lee un carácter del teclado. Por lo general, los caracteres se almacenan en variables de tipo `char` (pág. 173). Un carácter se puede tratar ya sea como entero o como carácter.
- Una instrucción `switch` consiste de una serie de etiquetas `case` (pág. 174) y un caso `default` opcional (pág. 174).
- La expresión entre paréntesis después de la palabra clave `switch` se llama expresión de control (pág. 174). La instrucción `switch` compara el valor de la expresión de control con cada etiqueta `case`.
- Las etiquetas `case` consecutivas, sin instrucciones entre ellas, ejecutan el mismo conjunto de instrucciones.
- Cada etiqueta `case` sólo puede especificar una expresión integral constante.
- Cada etiqueta `case` puede tener varias instrucciones. La instrucción de selección `switch` difiere de otras instrucciones de control, en cuanto a que no requiere llaves alrededor de varias instrucciones en cada `case`.
- C++ proporciona varios tipos de datos para representar enteros: `int`, `char`, `short` y `long`. El rango de valores enteros para cada tipo depende de la plataforma.
- C++11 nos permite proporcionar un valor predeterminado para un miembro de datos al declararlo en la declaración de la clase.

Sección 5.7 Instrucciones `break` *y* `continue`

- Cuando la instrucción `break` (pág. 178) se ejecuta en una de las instrucciones de repetición (`for`, `while` y `do...while`), provoca la salida inmediata de esa instrucción.
- Cuando la instrucción `continue` (pág. 179) se ejecuta en una instrucción de repetición, omite el resto de las instrucciones en el cuerpo del ciclo y continúa con la siguiente iteración del mismo. En una instrucción `while` o `do...while`, la ejecución continúa con la siguiente evaluación de la condición. En una instrucción `for`, la ejecución continúa con la expresión de incremento en el encabezado de la instrucción `for`.

Sección 5.8 Operadores lógicos

- Los operadores lógicos (pág. 180) nos permiten formar condiciones complejas mediante la combinación de condiciones simples. Los operadores lógicos son `&&` (AND lógico), `||` (OR lógico) y `!` (negación lógica).
- El operador `&&` (AND lógico, pág. 180) asegura que dos condiciones sean *ambas* `true`.

- El operador || (OR lógico, pág. 181) asegura que de dos condiciones, una *o* ambas sean `true`.
- Una expresión que contenga los operadores `&&` o `||` se evalúa sólo hasta que se conoce si la expresión es verdadera o falsa. Esta característica de rendimiento para la evaluación de las expresiones AND lógico y OR lógico se conoce como evaluación de corto circuito (pág. 182).
- El operador ! (NOT lógico, también conocido como negación lógica; pág. 182) permite a un programador "invertir" el significado de una condición. El operador de negación lógico unario se coloca antes de una condición, para elegir una ruta de ejecución si la condición original (sin el operador de negación lógico) es `false`. En la mayoría de los casos, podemos evitar el uso de la negación lógica si expresamos la condición con un operador relacional o de igualdad apropiado.
- Cuando se utiliza en una condición, cualquier valor distinto de cero se convierte de manera implícita en `true`; 0 (cero) se convierte de manera implícita en `false`.
- De manera predeterminada, `cout` muestra a los valores `bool` `true` y `false` como `1` y `0`, respectivamente. El manipulador de flujo `boolalpha` (pág. 183) especifica que el valor de cada expresión `bool` debe mostrarse, ya sea como la palabra "true" o la palabra "false".

Sección 5.9 Confusión entre los operadores de igualdad (==) y de asignación (=)

- Cualquier expresión que produzca un valor se puede utilizar en la porción de decisión de cualquier instrucción de control. Si el valor de la expresión es cero, se trata como `false`, y si es distinto de cero, se trata como `true`.
- Una asignación produce un valor; a saber, el valor asignado a la variable del lado izquierdo del operador de asignación.

Sección 5.10 Resumen de programación estructurada

- Cualquier forma de control se puede expresar en términos de instrucciones de secuencia, selección y repetición, y éstas pueden combinarse sólo en dos formas: apilamiento y anidamiento.

Ejercicios de autoevaluación

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

a) El caso `default` es requerido en la instrucción de selección `switch`.
b) La instrucción `break` es requerida en el caso `default` de una instrucción de selección `switch`, para salir del `switch` de manera apropiada.
c) La expresión `( x > y && a < b )` es `true` si la expresión `x > y` es `true`, o si la expresión `a < b` es `true`.
d) Una expresión que contiene el operador `||` es `true` si uno o ambos de sus operandos son `true`.

5.2 Escriba una instrucción o un conjunto de instrucciones en C++, para realizar cada una de las siguientes tareas:

a) Sumar los enteros impares entre 1 y 99 utilizando una instrucción `for`. Use las variables `unsigned int` de nombre `suma` y `cuenta`.
b) Imprimir el valor `333.546372` en una anchura de campo de 15 caracteres, con precisiones de `1`, `2` y `3`. Imprimir cada número en la misma línea. Justificar a la izquierda cada número en su campo. ¿Cuáles son los tres valores que se imprimen?
c) Calcular el valor de `2.5` elevado a la potencia de `3`, utilizando la función `pow`. Imprimir el resultado con una precisión de `2` en una anchura de campo de `10` posiciones. ¿Qué se imprime?
d) Imprimir los enteros del 1 al 20, utilizando un ciclo `while` y la variable contador `unsigned int x`. Imprimir solamente cinco enteros por línea. [*Sugerencia:* cuando `x % 5` sea 0, imprima un carácter de nueva línea; de lo contrario, imprima un carácter de tabulación].
e) Repita el ejercicio 5.2 (d), usando una instrucción `for`.

5.3 Encuentre los errores en cada uno de los siguientes segmentos de código, y explique cómo corregirlos:

a)
```
unsigned int x = 1;
while ( x <= 10 );
  ++x;
}
```

b)
```
for ( double y = 0.1; y != 1.0; y += .1 )
  cout << y << endl;
```

c)
```
switch ( n )
{
  case 1:
    cout << "El numero es 1" << endl;
  case 2:
    cout << "El numero es 2" << endl;
    break;
  default:
    cout << "El número no es 1 ni 2" << endl;
    break;
}
```

d) El siguiente código debe imprimir los valores 1 a 10:
```
unsigned int n = 1;
while ( n < 10 )
  cout << n++ << endl;
```

Respuestas a los ejercicios de autoevaluación

5.1 a) Falso. El caso `default` es opcional. Sin embargo, se considera buena ingeniería de software proporcionar siempre un caso `default`.

b) Falso. La instrucción `break` se utiliza para salir de la instrucción `switch`. La instrucción `break` no se requiere cuando el caso `default` es el último. Tampoco se requerirá la instrucción `break` si tiene sentido hacer que el control se reanude en el siguiente caso.

c) Falso. Al utilizar el operador `&&`, ambas expresiones relacionales deben ser `true` para que toda la expresión sea `true`.

d) Verdadero.

5.2 a)
```
unsigned int suma = 0;
for ( unsigned int count = 1; count <= 99; count += 2 )
  suma += cuenta;
```

b)
```
cout << fixed << left
  << setprecision( 1 ) << setw( 15 ) << 333.546372
  << setprecision( 2 ) << setw( 15 ) << 333.546372
  << setprecision( 3 ) << setw( 15 ) << 333.546372
  << endl;
```

La salida es:
```
333.5          333.55         333.546
```

c)
```
cout << fixed << setprecision( 2 ) << setw( 10 ) << pow( 2.5, 3 ) << endl;
```

La salida es:
```
     15.63
```

d)
```
unsigned int x = 1;
while ( x <= 20 )
{
```

```
   if ( x % 5 == 0 )
      cout << x << endl;
   else
      cout << x << '\t';
   ++x;
}
```

e)
```
for ( unsigned int x = 1; x <= 20; ++x )
{
   if ( x % 5 == 0 )
      cout << x << endl;
   else
      cout << x << '\t';
}
```

5.3 a) *Error:* El punto y coma después del encabezado `while` provoca un ciclo infinito.
Corrección: Reemplazar el punto y coma por una llave izquierda (`{`), o eliminar tanto el punto y coma (`;`) como la llave derecha (`}`).

b) *Error:* Utilizar un número de punto flotante para controlar una instrucción de repetición `for`.
Corrección: Utilice un `unsigned int` y realice el cálculo apropiado para poder obtener los valores deseados.

```
for ( unsigned int y = 1; y != 10; ++y )
   cout << ( static_cast< double >( y ) / 10 ) << endl;
```

c) *Error:* Falta una instrucción `break` en el primer `case`.
Corrección: Agregue una instrucción `break` al final del primer `case`. Esto no es un error, si el programador desea que la instrucción del `case 2:` se ejecute siempre que lo haga la instrucción del `case 1:`.

d) *Error:* Se está utilizando un operador relacional inadecuado en la condición de continuación de ciclo.
Corrección: Use `<=` en vez de `<`, o cambie `10` a `11`.

Ejercicios

5.4 ***(Encuentre los errores en el código)*** Encuentre el (los) error(es), si los hay, en cada uno de los siguientes fragmentos de código:

a)
```
For ( unsigned int x = 100, x >= 1, ++x )
   cout << x << endl;
```

b) El siguiente código debe imprimirse sin importar que el entero `valor` sea par o impar:
```
switch ( valor % 2 )
{
   case 0:
      cout << "Entero par" << endl;
   case 1:
      cout << "Entero impar" << endl;
}
```

c) El siguiente código debe imprimir los enteros impares del 19 al 1:
```
for ( unsigned int x = 19; x >= 1; x += 2 )
   cout << x << endl;
```

d) El siguiente código debe imprimir los enteros pares del 2 al 100:
```
unsigned int counter = 2;
do
{
   cout << contador << endl;
   contador += 2;
} While ( contador < 100 );
```

5.5 ***(Suma de enteros)*** Escriba un programa que utilice una instrucción `for` para sumar una secuencia de enteros. Suponga que el primer entero leído especifica el número de valores que quedan por introducir. Su programa debe leer sólo un valor por cada instrucción de entrada. Una secuencia típica de entrada podría ser

```
5 100 200 300 400 500
```

en donde el `5` indica que se van a sumar los `5` valores subsiguientes.

5.6 ***(Promedio de enteros)*** Escriba un programa que utilice una instrucción `for` para calcular e imprimir el promedio de varios enteros. Suponga que el último valor leído es el valor centinela `9999`. Por ejemplo, la secuencia `10 8 11 7 9 9999` indica que el programa debe calcular el promedio de todos los valores antes del `9999`.

5.7 ***(¿Qué hace el siguiente programa?)*** ¿Qué hace el siguiente programa?

```
1  // Ejercicio 5.7: ex05_07.cpp
2  // ¿Qué hace este programa?
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     unsigned int x; // declara x
9     unsigned int y; // declara y
10
11    // pide al usuario los datos de entrada
12    cout << "Escriba dos enteros en el rango 1 a 20: ";
13    cin >> x >> y;  // lee valores para x y y
14
15    for ( unsigned int i = 1; i <= y; ++i ) // cuenta desde 1 hasta y
16    {
17       for ( unsigned int j = 1; j <= x; ++j ) // cuenta desde 1 hasta x
18          cout << '@'; // imprime @
19
20       cout << endl; // empieza nueva línea
21    } // fin de for exterior
22 } // fin de main
```

5.8 ***(Encontrar el entero más chico)*** Escriba un programa que utilice una instrucción `for` para encontrar el menor de varios enteros. Suponga que el primer valor leído especifica el número de valores restantes.

5.9 ***(Producto de enteros impares)*** Escriba un programa que utilice una instrucción `for` para calcular e imprimir el producto de los enteros impares del 1 al 15.

5.10 ***(Factoriales)*** La función factorial se utiliza frecuentemente en los problemas de probabilidad. Utilizando la definición de factorial del ejercicio 4.34, escriba un programa que utilice una función `for` para evaluar los factoriales de los enteros del 1 al 5. Muestre los resultados en formato tabular. ¿Qué dificultad podría impedir que usted calculara el factorial de 20?

5.11 ***(Interés compuesto)*** Modifique el programa de interés compuesto de la sección 5.4, repitiendo sus pasos para las tasas de interés del 5, 6, 7, 8, 9 y 10%. Use una instrucción `for` para variar la tasa de interés.

5.12 ***(Patrones de dibujo con ciclos `for` anidados)*** Escriba un programa que utilice ciclos `for` para imprimir los siguientes patrones por separado, uno debajo del otro. Use ciclos `for` para generar los patrones. Todos los asteriscos (`*`) deben imprimirse mediante una sola instrucción de la forma `cout << '*';` (esto hace que los asteriscos se impriman uno al lado del otro). [*Sugerencia:* los últimos dos patrones requieren que cada línea empiece con un número apropiado de espacios en blanco. *Crédito adicional:* combine su código de los cuatro problemas separados en un solo programa que imprima los cuatro patrones, uno al lado del otro, haciendo un uso inteligente de los ciclos `for` anidados].

(a)
```
*
**
***
****
*****
******
*******
********
*********
**********
```

(b)
```
**********
*********
********
*******
******
*****
****
***
**
*
```

(c)
```
**********
 *********
  ********
   *******
    ******
     *****
      ****
       ***
        **
         *
```

(d)
```
         *
        **
       ***
      ****
     *****
    ******
   *******
  ********
 *********
**********
```

5.13 ***(Gráfico de barras)*** Una aplicación interesante de las computadoras es dibujar gráficos convencionales y de barra. Escriba un programa que lea cinco números (cada uno entre 1 y 30). Suponga que el usuario sólo introduce valores válidos. Por cada número leído, su programa debe imprimir una línea que contenga ese número de asteriscos adyacentes. Por ejemplo, si su programa lee el número 7, debe mostrar *******.

5.14 ***(Cálculo de las ventas totales)*** Un almacén de pedidos por correo vende cinco productos distintos, cuyos precios de venta son los siguientes: producto 1, $2.98; producto 2, $4.50; producto 3, $9.98; producto 4, $4.49 y producto 5, $6.87. Escriba un programa que lea una serie de pares de números, como se muestra a continuación:

a) número del producto
b) cantidad vendida

Su programa debe utilizar una instrucción `switch` para determinar el precio de venta de cada producto. Debe calcular y mostrar el valor total de venta de todos los productos vendidos. Use un ciclo controlado por centinela para determinar cuándo debe el programa dejar de iterar para mostrar los resultados finales.

5.15 ***(Modificación de `LibroCalificaciones`)*** Modifique el programa `LibroCalificaciones` de las figuras 5.9 a 5.11, de manera que calcule el promedio de puntos de calificaciones. Una calificación A vale 4 puntos, B vale 3 puntos, etcétera.

5.16 ***(Cálculo del interés compuesto)*** Modifique el programa de la figura 5.6, de manera que se utilicen sólo enteros para calcular el interés compuesto. [*Sugerencia:* trate todas las cantidades monetarias como números de centavos. Luego "divida" el resultado en su porción de dólares y su porción de centavos, utilizando las operaciones de división y módulo. Inserte un punto].

5.17 ***(¿Qué se imprime?)*** Suponga que `i = 1`, `j = 2`, `k = 3` y `m = 2`. ¿Qué es lo que imprime cada una de las siguientes instrucciones?

a) `cout << ( i == 1 ) << endl;`
b) `cout << ( j == 3 ) << endl;`
c) `cout << ( i >= 1 && j < 4 ) << endl;`
d) `cout << ( m <= 99 && k < m ) << endl;`
e) `cout << ( j >= i || k == m ) << endl;`
f) `cout << ( k + m < j || 3 - j >= k ) << endl;`
g) `cout << ( !m ) << endl;`
h) `cout << ( !( j - m ) ) << endl;`
i) `cout << ( !( k > m ) ) << endl;`

5.18 ***(Tabla de sistemas numéricos)*** Escriba un programa que imprima una tabla de los equivalentes binario, octal y hexadecimal de los números decimales en el rango de 1 a 256. Si no está familiarizado con estos sistemas numéricos, lea el apéndice D. [*Sugerencia:* puede usar los manipuladores de flujo `dec`, `oct` y `hex` para mostrar los enteros en los formatos decimal, octal y hexadecimal, respectivamente].

5.19 Calcule el valor de π a partir de la serie infinita

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} + \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \cdots$$

Imprima una tabla que muestre el valor aproximado de π, después de cada uno de los primeros 1000 términos de esta serie.

5.20 ***(Triples de Pitágoras)*** Un triángulo recto puede tener lados cuyas longitudes sean valores enteros. Un conjunto de tres valores enteros para los lados de un triángulo recto se conoce como triple de Pitágoras. Estos tres lados deben satisfacer la relación que establece que la suma de los cuadrados de dos lados es igual al cuadrado de la hipotenusa. Encuentre todos los triples de Pitágoras para `lado1`, `lado2`, y la `hipotenusa`, que no sean mayores de 500. Use un ciclo `for` triplemente anidado para probar todas las posibilidades. Este método es un ejemplo de la computación de **fuerza bruta**. En cursos de ciencias computacionales más avanzados aprenderá que existen muchos problemas interesantes para los cuales no hay otra metodología algorítmica conocida, sólo el uso de la fuerza bruta.

5.21 ***(Cálculo de salarios)*** Una empresa paga a sus empleados como gerentes (quienes reciben un salario semanal fijo), trabajadores por horas (que reciben un sueldo fijo por hora para las primeras 40 horas que trabajen y "tiempo y medio": 1.5 veces su sueldo por horas, para las horas extra trabajadas), empleados por comisión (que reciben $250 más el 5.7 por ciento de sus ventas totales por semana), o trabajadores por piezas (que reciben una cantidad fija de dinero por cada artículo que producen; cada trabajador por piezas en esta empresa trabaja sólo en un tipo de artículo). Escriba un programa para calcular el sueldo semanal para cada empleado. No necesita saber cuántos empleados hay de antemano. Cada tipo de empleado tiene su propio código de pago: los gerentes tienen el código 1, los trabajadores por horas tienen el código 2, los trabajadores por comisión tienen el código 3 y los trabajadores por piezas tienen el código 4. Use una instrucción `switch` para calcular el sueldo de cada empleado, de acuerdo con el código de pago de cada uno. Dentro del `switch`, pida al usuario (es decir, el cajero de nóminas) que introduzca los hechos apropiados que su programa necesita para calcular el sueldo de cada empleado, de acuerdo con su código de pago.

5.22 (***Leyes de De Morgan***) En este capítulo hemos hablado sobre los operadores lógicos `&&`, `||` y `!`. Algunas veces, las leyes de De Morgan pueden hacer que sea más conveniente para nosotros expresar una expresión lógica. Estas leyes establecen que la expresión `!(` *condición1* `&&` *condición2* `)` es lógicamente equivalente a la expresión `(!`*condición1* `||` `!`*condición2*`)`. Además, la expresión `!(` *condición1* `||` *condición2* `)` es lógicamente equivalente a la expresión `(!`*condición1* `&&` `!`*condición2*`)`. Use las leyes de De Morgan para escribir expresiones equivalentes para cada una de las siguientes expresiones, luego escriba un programa que demuestre que, tanto la expresión original como la nueva expresión, son equivalentes en cada caso:

a) `!( x < 5 ) && !( y >= 7 )`
b) `!( a == b ) || !( g != 5 )`
c) `!( ( x <= 8 ) && ( y > 4 ) )`
d) `!( ( i > 4 ) || ( j <= 6 ) )`

5.23 ***(Rombo de asteriscos)*** Escriba un programa que imprima la siguiente figura de rombo. Puede utilizar instrucciones de salida que impriman un solo asterisco (`*`), un solo espacio en blanco o un solo carácter de nueva línea. Maximice el uso de la repetición (con instrucciones `for` anidadas) y minimice el número de instrucciones de salida.

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

5.24 ***(Modificación del rombo de asteriscos)*** Modifique el programa que escribió en el ejercicio 5.23, para que lea un número impar en el rango de 1 a 19, de manera que especifique el número de filas en el rombo y después muestre un rombo del tamaño apropiado.

5.25 ***(Eliminación de `break` y `continue`)*** Una crítica de las instrucciones `break` y `continue` es que ninguna es estructurada. En realidad, estas instrucciones pueden reemplazarse en todo momento por instrucciones estructuradas. Describa, en general, cómo eliminaría la instrucción `break` de un ciclo en un programa, para reempla-

zarlas con alguna de las instrucciones estructuradas equivalentes. [*Sugerencia:* la instrucción `break` se sale de un ciclo desde el cuerpo de éste. La otra forma de salir es que falle la prueba de continuación de ciclo. Considere utilizar en la prueba de continuación de ciclo una segunda prueba que indique una "salida anticipada debido a una condición de 'interrupción'"]. Use la técnica que desarrolló aquí para eliminar la instrucción `break` de la aplicación de la figura 5.13.

5.26 ***(¿Qué hace este código?)*** ¿Qué hace el siguiente segmento de programa?

```
for ( unsigned int i = 1; i <= 5; ++i )
{
   for ( unsigned int j = 1; j <= 3; ++j )
   {
      for ( unsigned int k = 1; k <= 4; ++k )
         cout << '*';

      cout << endl;
   } // fin del for interior

   cout << endl;
} // fin del for exterior
```

5.27 ***(Eliminación de la instrucción `continue`)*** Describa, en general, cómo eliminaría las instrucciones `continue` de un ciclo en un programa, para reemplazarlas con uno de sus equivalentes estructurados. Use la técnica que desarrolló aquí para eliminar la instrucción `continue` del programa de la figura 5.14.

5.28 ***(Canción "Los Doce Días de Navidad")*** Escriba un programa que utilice instrucciones de repetición y `switch` para imprimir la canción "Los Doce Días de Navidad". Una instrucción `switch` debe utilizarse para imprimir el día (es decir, "primer", "segundo", etcétera). Una instrucción `switch` separada debe utilizarse para imprimir el resto de cada verso. Visite el sitio Web `http://www.12days.com/library/carols/` para obtener la letra completa de la canción.

5.29 ***(Problema de Peter Minuit)*** La leyenda establece que, en 1626, Peter Minuit compró la isla de Manhattan por $24.00 en un trueque. ¿Hizo él una buena inversión? Para responder a esta pregunta, modifique el programa de interés compuesto de la figura 5.6, para empezar con un monto principal de $24.00 y calcular el monto de interés en depósito, si ese dinero se había mantenido en depósito hasta este año (por ejemplo, 387 años hasta 2013). Coloque el ciclo `for` que realiza el cálculo del interés compuesto en un ciclo `for` exterior que varíe la tasa de interés del 5 al 10%, para observar las maravillas del interés compuesto.

Hacer la diferencia

5.30 ***(Examen rápido sobre hechos del calentamiento global)*** La controversial cuestión del calentamiento global obtuvo una gran publicidad gracias a la película *An Inconvenient Truth* en la que aparece el anterior vicepresidente Al Gore. El señor Gore y una red de científicos de Naciones Unidas (N.U.), el Panel Intergubernamental sobre el Cambio Climático, compartieron el Premio Nobel a la Paz de 2007 en reconocimiento por "sus esfuerzos al generar y diseminar un mayor conocimiento sobre el cambio climatológico provocado por el hombre". Investigue *ambos* lados de la cuestión del calentamiento global en línea (tal vez quiera buscar frases como "escépticos del calentamiento global"). Cree un examen rápido de opción múltiple con cinco preguntas sobre el calentamiento global; cada pregunta debe tener cuatro posibles respuestas (enumeradas del 1 al 4). Sea objetivo y trate de representar con imparcialidad ambos lados de la cuestión. Después escriba una aplicación que administre el examen rápido, calcule el número de respuestas correctas (de cero a cinco) y devuelva un mensaje al usuario. Si éste responde de manera correcta a las cinco preguntas, imprima el mensaje «Excelente»; si responde a cuatro, imprima «Muy bien»; si responde a tres o menos, imprima «Es tiempo de aprender más sobre el calentamiento global», e incluya una lista de algunos de los sitios Web en donde encontró esos hechos.

5.31 ***(Alternativas para el plan fiscal: el "impuesto justo")*** Existen muchas propuestas para que los impuestos sean más justos. Consulte la iniciativa FairTax (impuestos justos) de Estados Unidos en el sitio:

```
www.fairtax.org/site/PageServer?pagename=calculator
```

Investigue cómo funciona la iniciativa FairTax que se propone. Nuestra sugerencia es eliminar los impuestos sobre los ingresos y otros impuestos más a favor de un 23% de impuestos sobre el consumo en todos los productos y servicios que usted compre. Algunos opositores a FairTax cuestionan la cifra del 23% y dicen que, debido a la forma en que se calculan los impuestos, sería más preciso decir que la tasa sea del 30%; revise esto con cuidado. Escriba un programa que pida al usuario que introduzca sus gastos en diversas categorías de gastos disponibles (por ejemplo, alojamiento, comida, ropa, transporte, educación, servicios médicos, vacaciones) y que después imprima el impuesto FairTax estimado que esa persona pagaría.

5.32 ***(Crecimiento de la base de usuarios de Facebook)*** Al mes de enero de 2013, hay aproximadamente 2 500 millones de personas en Internet. Facebook llegó a los mil millones de usuarios en octubre de 2012. En este ejercicio, escribirá un programa para determinar cuándo llegará Facebook a 2 500 millones de personas con tasas de crecimiento mensuales del 2, 3, 4 o 5%. Use las técnicas que aprendió en la figura 5.6.

Funciones y una introducción a la recursividad

6

La forma siempre va después de la función.
—Louis Henri Sullivan

E pluribus unum.
(Uno compuesto de varios).
—Virgilio

¡Oh! recuerdos del ayer, tratar de regresar el tiempo.
—William Shakespeare

Respóndeme en una palabra.
—William Shakespeare

Hay un punto en el cual los métodos se devoran a sí mismos.
—Frantz Fanon

Objetivos

En este capítulo aprenderá a:

- Crear programas en forma modular, a partir de funciones.
- Utilizar las funciones matemáticas comunes de la biblioteca.
- Conocer los mecanismos para pasar datos a funciones y devolver resultados.
- Comprender cómo el mecanismo de llamadas a/regreso de funciones está soportado por la pila de llamadas a funciones y los registros de activación.
- Usar la generación de números aleatorios para implementar aplicaciones de juegos.
- Comprender cómo la visibilidad de los identificadores está limitada a regiones específicas de programas.
- Escribir y usar funciones recursivas.

Plan general

6.1 Introducción

La mayoría de los programas de computadora que resuelven problemas del mundo real son mucho más grandes que los programas que se presentan en los primeros capítulos de este libro. La experiencia ha demostrado que la mejor forma de desarrollar y mantener un programa extenso es construirlo a partir de piezas (o componentes) simples y pequeñas. A esta técnica se le conoce como **divide y vencerás**.

Veremos las generalidades de una porción de las funciones matemáticas de la Biblioteca estándar de C++. Después, el lector aprenderá a declarar una función con más de un parámetro. También presentaremos información adicional acerca de los prototipos de funciones y la forma en que el compilador los utiliza para convertir el tipo de argumento en la llamada a una función, al tipo especificado en la lista de parámetros de una función, si es necesario.

Luego, daremos un pequeño giro hacia las técnicas de simulación con la generación de números aleatorios, y desarrollaremos una versión de un popular juego de dados de casino, el cual utiliza la mayoría de las técnicas de programación que el lector ha aprendido hasta este punto.

Posteriormente, presentaremos las clases de almacenamiento y reglas de alcance de C++. Éstas determinan el periodo durante el cual un objeto existe en la memoria, y en dónde se puede hacer referencia a su identificador en un programa. También aprenderá cómo C++ es capaz de llevar el registro de cuál función se ejecuta en un momento dado, cómo se mantienen los parámetros y otras variables locales de las funciones en la memoria, y cómo sabe una función a dónde regresar, una vez que termina su ejecución. Hablaremos sobre dos temas que ayudan a mejorar el rendimiento de los programas: las funciones en línea que pueden eliminar la sobrecarga de la llamada a una función, y parámetros de referencia que pueden usarse para pasar elementos extensos de datos a las funciones con eficiencia.

Muchas de las aplicaciones que desarrollará tendrán más de una función con el mismo nombre. Esta técnica, llamada sobrecarga de funciones, se utiliza para implementar funciones que realicen tareas similares para los argumentos de distintos tipos, o posiblemente para distintos números de argumentos. Consideremos las plantillas de funciones: un mecanismo para definir una familia de funciones sobre-

cargadas. Este capítulo concluye con una discusión de las funciones que se llaman a sí mismas, ya sea en forma directa o indirecta (a través de otra función): un tema conocido como recursividad.

6.2 Componentes de los programas en C++

Como hemos visto, por lo general los programas en C++ se escriben mediante la combinación de nuevas funciones y clases que escribimos con funciones y clases "preempaquetadas", disponibles en la Biblioteca estándar de C++, que proporciona una extensa colección de funciones para realizar cálculos matemáticos comunes, manipulaciones de cadenas, manipulaciones de caracteres, entrada/salida, comprobación de errores y muchas otras operaciones útiles.

Las funciones nos permiten modularizar un programa, al separar sus tareas en unidades autocontenidas. Ya ha utilizado una combinación de funciones de biblioteca y sus propias funciones en todos los programas que ha escrito. A las funciones que usted escribe se les conoce como **funciones definidas por el usuario**. Las instrucciones en los cuerpos de las funciones se escriben sólo una vez, se pueden reutilizar tal vez desde varias ubicaciones en un programa y están ocultas de las demás funciones.

Hay varias razones para modularizar un programa con funciones:

- Una de ellas es la metodología divide y vencerás.
- Otra de ellas es la reutilización de software. Por ejemplo, en los programas anteriores no tuvimos que definir cómo leer una línea de texto del teclado; C++ proporciona esta herramienta a través de la función `getline` del archivo de encabezado `<string>`.
- Una tercera motivación es la de evitar repetir código.
- Además, al dividir un programa en funciones significativas, es más fácil depurarlo y darle mantenimiento.

Observación de Ingeniería de Software 6.1

Para promover la reutilización de software, toda función debe limitarse a realizar una sola tarea bien definida, y el nombre de la función debe expresar esa tarea con eficiencia.

Como sabemos, una función se invoca mediante una llamada, y cuando la función a la que se llamó completa su tarea, devuelve un *resultado* o simplemente devuelve el *control* a la función que la llamó. Una analogía a esta estructura de un programa es la forma jerárquica de la administración (figura 6.1). Un jefe (similar a la función que hace la llamada) pide a un trabajador (similar a la función que se llamó) que realice una tarea y reporte (devuelva) los resultados, después de completarla. La función jefe *no* sabe cómo realiza la función trabajador sus tareas designadas. El trabajador también podría llamar a otras funciones trabajador, sin que el jefe supiera. Este *ocultamiento de los detalles de implementación* promueve la buena ingeniería de software. La figura 6.1 muestra cómo la función `jefe` se comunica con

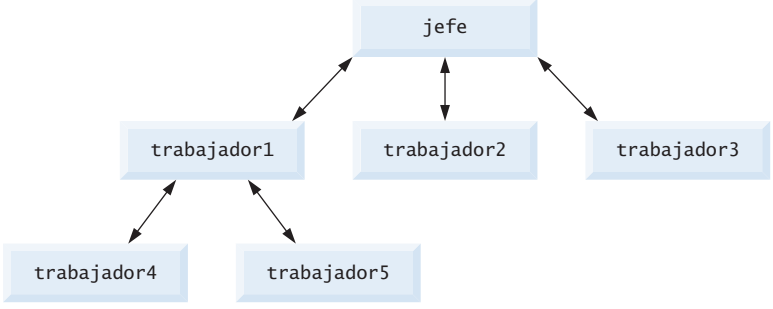


Fig. 6.1 | Relación jerárquica entre la función jefe y las funciones trabajador.

varias funciones trabajador. La función `jefe` divide las responsabilidades entre las diversas funciones `trabajador`, y `trabajador1` actúa como "función jefe" para `trabajador4` y `trabajador5`.

6.3 Funciones matemáticas de la biblioteca

Algunas veces, las funciones como `main` *no* son miembros de una clase. A éstas se les conoce como **funciones globales**. Al igual que las funciones miembro de una clase, los prototipos para las funciones globales se colocan en encabezados, de manera que las funciones globales se puedan reutilizar en cualquier programa que incluya el archivo de encabezado y pueda crear un enlace con el código objeto de la función. Por ejemplo, recuerde que utilizamos la función `pow` del archivo de encabezado `<cmath>` para elevar un valor a una potencia en la figura 5.6. Introduciremos aquí varias funciones del archivo de encabezado `<cmath>` para presentar el concepto de las funciones globales que no pertenecen a una clase específica.

El encabezado `<cmath>` proporciona una colección de funciones que nos permiten realizar cálculos matemáticos comunes. Por ejemplo, puede calcular la raíz cuadrada de `900.0` con la siguiente llamada a la función:

```
sqrt( 900.0 )
```

La expresión anterior se evalúa como `30.0`. La función `sqrt` recibe un argumento de tipo `double` y devuelve un resultado `double`. No hay necesidad de crear objetos antes de llamar a la función `sqrt`. Además, *todas* las funciones en el encabezado `<cmath>` son *globales*; por lo tanto, para llamar a cada una de ellas sólo hay que especificar el nombre de la función, seguido de paréntesis que contienen los argumentos de la misma. Si llamamos a `sqrt` con un argumento negativo, la función establece una variable global llamada `errno` con el valor constante `EDOM`. La variable `errno` y la constante `EDOM` se definen en el encabezado `<cerrno>`. En la sección 6.10 hablaremos sobre las variables globales

Tip para prevenir errores 6.1

No llame a `sqrt` con un argumento negativo. Para el código de nivel industrial, revise siempre que los argumentos que pasa a las funciones matemáticas sean válidos.

Los argumentos de una función pueden ser constantes, variables o expresiones más complejas. Si `c = 13.0, d = 3.0` y `f = 4.0`, entonces la instrucción:

```
cout << sqrt( c + d * f ) << endl;
```

muestra en pantalla la raíz cuadrada de `13.0 + 3.0 * 4.0 = 25.0`; a saber, `5.0`. Algunas funciones matemáticas de la biblioteca se sintetizan en la figura 6.2. En la figura, las variables `x` y `y` son de tipo `double`.

Función	Descripción	Ejemplo
`ceil( x )`	redondea *x* al valor más pequeño que no sea menor que *x*	`ceil( 9.2 ) es 10.0` `ceil( -9.8 ) es -9.0`
`cos( x )`	coseno trigonométrico de *x* (*x* está en radianes)	`cos( 0.0 ) es 1.0`

Fig. 6.2 | Funciones matemáticas de la biblioteca (parte 1 de 2).

Función	Descripción	Ejemplo
`exp( x )`	función exponencial e^x	`exp( 1.0 ) es 2.718282` `exp( 2.0 ) es 7.389056`

`fabs( x )`	valor absoluto de x	`fabs( 5.1 ) es 5.1`
		`fabs( 0.0 ) es 0.0`
		`fabs( -8.76 ) es 8.76`

`floor( x )`	redondea x al entero más grande, no mayor a x	`floor( 9.2 ) es 9.0` `floor( -9.8 ) es -10.0`
`fmod( x, y )`	residuo de x/y como número de punto flotante	`fmod( 2.6, 1.2 ) es 0.2`
`log( x )`	logaritmo natural de x (base e)	`log( 2.718282 ) es 1.0` `log( 7.389056 ) es 2.0`
`log10( x )`	logaritmo de x (base 10)	`log10( 10.0 ) es 1.0` `log10( 100.0 ) es 2.0`
`pow( x, y )`	x elevado a la potencia y (x^y)	`pow( 2, 7 ) es 128` `pow( 9, .5 ) es 3`
`sin( x )`	seno trigonométrico de x (x en radianes)	`sin( 0.0 ) es 0`
`sqrt( x )`	raíz cuadrada de x (en donde x es un valor no negativo)	`sqrt( 9.0 ) es 3.0`
`tan( x )`	tangente trigonométrica de x (x en radianes)	`tan( 0.0 ) es 0`

Fig. 6.2 | Funciones matemáticas de la biblioteca (parte 2 de 2).

6.4 Definiciones de funciones con varios parámetros

Consideremos las funciones con *varios* parámetros. El programa en las figuras 6.3 a 6.5 modifica la clase `LibroCalificacones`, al incluir una función definida por el usuario llamada `maximo`, la cual determina y devuelve la mayor de tres calificaciones `int`. Cuando la aplicación empieza su ejecución, la función `main` (líneas 5 a 13 de la figura 6.5) crea un objeto de la clase `LibroCalificaciones` (línea 8) y llama a la función miembro `recibirCalificaciones` del objeto (línea 11) para leer tres calificaciones enteras del usuario. En el archivo de implementación de la clase `LibroCalificaciones` (figura 6.4), en las líneas 52 y 53 de la función miembro `recibirCalificaciones` se pide al usuario que introduzca tres valores enteros y los recibe de éste. En la línea 56 se hace una llamada a la función miembro `maximo` (definida en las líneas 60 a 73). La función `maximo` determina el valor más grande, y después la instrucción `return` (línea 72) devuelve el valor al punto en el cual la función `recibirCalificaciones` invocó a `maximo` (línea 56). Entonces, la función miembro `recibirCalificaciones` almacena el valor de retorno de `maximo` en el miembro de datos `calificacionMaxima`. Después, este valor se imprime llamando a la función `mostrarReporteCalificaciones` (línea 12 de la figura 6.5). [*Nota:* a esta función la llamamos `mostrarReporteCalificaciones`, ya que versiones subsiguientes de la clase `LibroCalificaciones` utilizarán esta función para mostrar un reporte completo de calificaciones, incluyendo las calificaciones máxima y mínima]. En el capítulo 7 mejoraremos la clase `LibroCalificaciones` para procesar conjuntos de calificaciones.

```
1  // Fig. 6.3: LibroCalificaciones.h
2  // Definición de la clase LibroCalificaciones que encuentra el máximo de tres
      calificaciones.
3  // Las funciones miembro están definidas en LibroCalificaciones.cpp
4  #include <string> // el programa usa la clase string estándar de C++
5
6  // definición de la clase LibroCalificaciones
7  class LibroCalificaciones
8  {
9  public:
10   explicit LibroCalificaciones( std::string ); // inicializa el nombre del
                                                     curso
11   void establecerNombreCurso( std::string ); // establece el nombre del curso
12   std::string obtenerNombreCurso() const; //obtiene el nombre del curso
13   void mostrarMensaje() const; // muestra un mensaje de bienvenida
14   void recibirCalificaciones(); // recibe las tres calificaciones del usuario
15   void mostrarReporteCalificaciones() const; // muestra un reporte con base en
                                                   las calificaciones
16   int maximo( int, int, int ) const; // determina el máximo de 3 valores
17 private:
18   std::string nombreCurso; // nombre del curso para este LibroCalificaciones
19   int calificacionMaxima; // valor máximo de las tres calificaciones
20 }; // fin de la clase LibroCalificaciones
```

Fig. 6.3 | Definición de la clase `LibroCalificaciones` que encuentra el máximo de tres calificaciones.

```
1  // Fig. 6.4: LibroCalificaciones.cpp
2  // Definiciones de las funciones miembro para la clase LibroCalificaciones
3  // que determina el máximo de tres calificaciones.
4  #include <iostream>
5  using namespace std;
6
7  #include "LibroCalificaciones.h" // incluye la definición de la clase
                                       LibroCalificaciones
8
9  // el constructor inicializa nombreCurso con la cadena suministrada como
10 // argumento; inicializa calificacionMaxima a 0
11 LibroCalificaciones::LibroCalificaciones( string nombre )
12   : calificacionMaxima( 0 ) // este valor se reemplazará por la calificación máxima
13 {
14   establecerNombreCurso( nombre ); // valida y almacena nombreCurso
15 } // fin del constructor LibroCalificaciones
16
17 // función para establecer el nombre del curso; limita nombre a 25 o menos
     caracteres
18 void LibroCalificaciones::establecerNombreCurso( string nombre )
19 {
20   if ( nombre.size() <= 25 ) // si nombre tiene 25 o menos caracteres
21     nombreCurso = nombre; // almacena el nombre del curso en el objeto
22   else // si nombre es mayor que 25 caracteres
23   { // establece nombreCurso a los primeros 25 caracteres del parámetro nombre
24     nombreCurso = nombre.substr( 0, 25 ); // selecciona los primeros 25 caracteres
25     cerr << "El nombre \"" << name << "\" excede la longitud maxima (25).\n"
26       << "Se limito nombreCurso a los primeros 25 caracteres.\n" << endl;
27   } // fin de if...else
28 } // fin de la función establecerNombreCurso
```

Fig. 6.4 | Definiciones de las funciones miembro para la clase `LibroCalificaciones` que determina el máximo de tres calificaciones (parte 1 de 2).

```
29
30 // función para obtener el nombre del curso
31 string LibroCalificaciones::obtenerNombreCurso() const
32 {
33    return nombreCurso;
34 } // fin de la función obtenerNombreCurso
35
36 // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
37 void LibroCalificaciones::mostrarMensaje() const
38 {
39    // esta instrucción llama a obtenerNombreCurso para obtener el
40    // name of the course this GradeBook represents
41    cout << "Bienvenido al libro de calificaciones para\n"
       << obtenerNombreCurso() << "!\n"
42       << endl;
43 } // fin de la función mostrarMensaje
44
45 // recibe tres calificaciones del usuario; determina el valor máximo
46 void LibroCalificaciones::recibirCalificaciones()
47 {
48    int calificacion1; // primera calificación introducida por el usuario
49    int calificacion2; // segunda calificación introducida por el usuario
50    int calificacion3; // tercera calificación introducida por el usuario
51
52    cout << "Introduzca tres calificaciones enteras: ";
53    cin >> calificacion1 >> calificacion2 >> calificacion3;
54
55    // almacena el valor máximo en el miembro calificacionMaxima
56    calificacionMaxima = maximo( calificacion1, calificacion2, calificacion3 );
57 } // fin de la función recibirCalificaciones
58
59 // devuelve el máximo de sus tres parámetros enteros
60 int LibroCalificaciones::maximo( int x, int y, int z ) const
61 {
62    int valorMaximo = x; // supone que x es el mayor para empezar
63
64    // determina si y es mayor que valorMaximo
65    if ( y > valorMaximo )
66       valorMaximo = y; // hace a y el nuevo valorMaximo
67
68    // determina si z es mayor que valorMaximo
69    if ( z > valorMaximo )
70       valorMaximo = z; // hace a z el nuevo valorMaximo
71
72    return valorMaximo;
73 } // fin de la función maximo
74
75 // muestra un reporte con base en las calificaciones introducidas por el usuario
76 void LibroCalificaciones::mostrarReporteCalificaciones() const
77 {
78    // imprime el máximo de las calificaciones introducidas
79    cout << "Calificacion maxima introducida: " << calificacionMaxima << endl;
80 } // fin de la función mostrarReporteCalificaciones
```

Fig. 6.4 | Definiciones de las funciones miembro para la clase `LibroCalificaciones` que determina el máximo de tres calificaciones (parte 2 de 2).

```
1  // Fig. 6.5: fig06_05.cpp
2  // Crea un objeto LibroCalificaciones, introducir calificaciones y mostrar
      reporte.
3  #include "LibroCalificaciones.h" // incluye la definición de la clase
                                    LibroCalificaciones
4
5  int main()
6  {
7     // crea un objeto LibroCalificaciones
8     LibroCalificaciones miLibroCalificaciones( "CS101 Programacion en C++" );
9
10    miLibroCalificaciones.mostrarMensaje(); // muestra mensaje de bienvenida
11    miLibroCalificaciones.recibirCalificaciones(); // lee calificaciones del
                                                        usuario
12    miLibroCalificaciones.mostrarReporteCalificaciones();
      // muestra reporte con base en las calificaciones
13 } // fin de main
```

```
Bienvenido al libro de calificaciones para
CS101 Programacion en C++!

Introduzca tres calificaciones enteras: 86 67 75
Calificacion maxima introducida: 86
```

```
Bienvenido al libro de calificaciones para
CS101 Programacion en C++!

Introduzca tres calificaciones enteras: 67 86 75
Calificacion maxima introducida: 86
```

```
Bienvenido al libro de calificaciones para
CS101 Programacion en C++!

Introduzca tres calificaciones enteras: 67 75 86
Calificacion maxima introducida: 86
```

Fig. 6.5 | Crear objeto `LibroCalificaciones`, introducir calificaciones y mostrar reporte.

Observación de Ingeniería de Software 6.2

Las comas utilizadas en la línea 56 de la figura 6.4 para separar los argumentos de la función `maximo` no *son operadores coma, como vimos en la sección 5.3. El operador coma garantiza que sus operandos se evalúen de izquierda a derecha. Sin embargo, el estándar de C++ no especifica el orden de evaluación de los argumentos de una función. Por ende, distintos compiladores pueden evaluar los argumentos de una función en distintos órdenes. El estándar de C++* garantiza *que todos los argumentos en la llamada a una función se evalúen* antes *de ejecutar la función que se va a llamar.*

Tip de portabilidad 6.1

Algunas veces, cuando los argumentos de una función son expresiones, como las de las llamadas a otras funciones, el orden en el que el compilador evalúa los argumentos podría afectar a los valores de uno o más de los argumentos. Si el orden de evaluación cambia entre un compilador y otro, los valores de los argumentos que se pasan a la función podrían variar, lo cual produciría errores lógicos sutiles.

Tip para prevenir errores 6.2

Si tiene dudas acerca del orden de evaluación de los argumentos de una función, y de si el orden afectaría a los valores que se pasan a la función, evalúe los argumentos en instrucciones de asignación separadas antes de la llamada a la función, asigne el resultado de cada expresión a una variable local y después pase esas variables como argumentos para la función.

Prototipo de función para `maximo`

El prototipo de la función miembro `maximo` (figura 6.3, línea 16) indica que la función devuelve un valor entero, que el nombre de la función es `maximo` y que requiere tres parámetros enteros para realizar su tarea. La primera línea de la función `maximo` (figura 6.4, línea 60) concuerda con el prototipo de función e indica que los nombres de los parámetros son `x`, `y` y `z`. Cuando se hace una llamada a `maximo` (figura 6.4, línea 56), el parámetro `x` se inicializa con el valor del argumento `calificacion1`, el parámetro `y` se inicializa con el valor del argumento `calificacion2` y el parámetro `z` se inicializa con el valor del argumento `calificacion3`. Debe haber un argumento en la llamada a la función para cada parámetro (también conocido como **parámetro formal**) en la definición de la función.

Observe que varios parámetros se especifican en el prototipo de función y en el encabezado de la función como una lista separada por comas. El compilador hace referencia al prototipo de la función para comprobar que las llamadas a `maximo` contengan el número y tipos de argumentos correctos, y que los tipos estén en el orden correcto. Además, el compilador usa el prototipo para asegurar que el valor devuelto por la función se pueda utilizar de manera correcta en la expresión que llamó a la función (por ejemplo, la llamada a una función que devuelve `void` *no se puede* utilizar como el lado derecho de una instrucción de asignación). Cada argumento debe ser *consistente* con el tipo del parámetro correspondiente. Por ejemplo, un parámetro de tipo `double` puede recibir valores como 7.35, 22 o –0.03456, pero no una cadena como «`hola`». Si los argumentos que se pasan a una función *no* concuerdan con los tipos especificados en el prototipo de la función, el compilador trata de convertir los argumentos a esos tipos. En la sección 6.5 hablaremos sobre esta conversión.

Error común de programación 6.1

Declarar los parámetros de funciones del mismo tipo como `double x, y` *en vez de* `double x, double y` *es un error de sintaxis; se requiere un tipo para cada parámetro en la lista de parámetros.*

Error común de programación 6.2

Si el prototipo de función, el encabezado y las llamadas a la función no concuerdan en el número, tipo y orden de argumentos y parámetros, además del tipo de retorno, se producen errores de compilación. También pueden ocurrir errores en el enlazador y otros tipos de errores, como veremos más adelante en el libro.

Observación de Ingeniería de Software 6.3

Una función que tiene muchos parámetros puede estar realizando muchas tareas. Considere dividir la función en funciones más pequeñas que realicen las tareas por separado. Limite el encabezado de la función a una línea, si es posible.

Lógica de la función `máximo`

Para determinar el valor máximo (líneas 60 a 73 de la figura 6.4), empezamos con la suposición de que el parámetro `x` contiene el valor más grande, por lo que en la línea 62 de la función `maximo` se declara la variable local `valorMaximo` y se inicializa con el valor del parámetro `x`. Desde luego, es posible que el

parámetro y o z contenga el valor más grande actual, por lo que debemos comparar cada uno de estos valores con valorMaximo. La instrucción if en las líneas 65 y 66 determina si y es mayor que valorMaximo y, de ser así, asigna y a valorMaximo. La instrucción if en las líneas 69 y 70 determina si z es mayor que valorMaximo y, de ser así, asigna z a valorMaximo. En este punto, el mayor de los tres valores está en valorMaximo, por lo que la línea 72 devuelve ese valor a la llamada en la línea 56. Cuando el control del programa regresa al punto en donde se llamó a maximo, los parámetros x, y y z de maximo no están accesibles ya para el programa.

Devolver el control de una función a quien la llamó

Hay varias formas de devolver el control al punto en el que se invocó a una función. Si la función *no* devuelve un resultado (es decir, si la función tiene un tipo de valor de retorno void), el control regresa cuando el programa llega a la llave derecha de fin de la función, o mediante la ejecución de la instrucción

```
return;
```

Si la función *devuelve* un resultado, la instrucción

```
return expresion;
```

evalúa *expresion* y devuelve el valor de *expresion* a la función que hizo la llamada. Algunos compiladores generan errores y otros generan advertencias si *no* se proporciona una instrucción return apropiada en una función que se supone debe devolver un resultado.

6.5 Prototipos de funciones y coerción de argumentos

Un prototipo de función (también conocido como **declaración de función**) indica al compilador el nombre de una función, el tipo de datos devuelto por la función, el número de parámetros que espera recibir, los tipos de esos parámetros y el orden en el que éstos se esperan.

Observación de Ingeniería de Software 6.4

Los prototipos de función son obligatorios, a menos que la función se defina antes de usarla. Use directivas #include *del preprocesador para obtener prototipos de función para las funciones de la Biblioteca estándar de C++ de los encabezados para las bibliotecas apropiadas (por ejemplo, el prototipo para* sqrt *está en encabezado* <cmath>*; una lista parcial de los encabezados de la Biblioteca estándar de C++ aparece en la sección 6.6). Use también* #include *para obtener encabezados que contengan prototipos de función escritos por usted, o por otros programadores.*

Error común de programación 6.3

Si se define una función antes *de invocarla, entonces su definición también sirve como el prototipo de la misma, por lo que no es necesario un prototipo separado. Si se invoca una función* antes *de definirla, y no tiene un prototipo de función, se produce un error de compilación.*

Observación de Ingeniería de Software 6.5

Siempre debemos proporcionar prototipos de funciones, aun cuando es posible omitirlas cuando se definen las funciones antes *de utilizarlas. Al proporcionar los prototipos, evitamos fijar el código al orden en el que se definen las funciones (lo cual puede cambiar fácilmente, a medida que un programa evoluciona).*

Firmas de funciones

La porción de un prototipo de función que incluya el *nombre de la función* y los *tipos de sus argumentos* se conoce como la **firma de la función**, o simplemente **firma**. La firma de la función no especifica el tipo de valor de retorno de la función. Las *funciones en el mismo alcance deben tener firmas únicas*. El alcance de una función es la región del programa en la que la función se conoce y es accesible. En la sección 6.11 hablaremos más acerca del alcance.

En la figura 6.3, si el prototipo en la línea 16 se hubiera escrito como:

```
void maximo( int, int, int );
```

el compilador reportaría un error, ya que el tipo de valor de retorno `void` en el prototipo de la función sería distinto del tipo de valor de retorno `int` en el encabezado de la función. De manera similar, dicho prototipo haría que la instrucción

```
cout << maximo( 6, 7, 0 );
```

genere un error de compilación, ya que esa instrucción depende de `maximo` para devolver un valor que se va a mostrar en pantalla.

Coerción de argumentos

Una característica importante de los prototipos de función es la **coerción de argumentos**; es decir, obligar a que los argumentos tengan los tipos especificados por las declaraciones de los parámetros. Por ejemplo, un programa puede llamar a una función con un argumento entero, aun cuando el prototipo de función especifique un argumento `double`; la función de todas maneras trabajará correctamente.

Reglas de promoción de argumentos y conversiones implícitas[1]

Algunas veces, los valores de los argumentos que no corresponden precisamente a los tipos de los parámetros en el prototipo de función pueden ser convertidos por el compilador al tipo apropiado, antes de que se haga una llamada a la función. Estas conversiones ocurren según lo especificado por las **reglas de promoción** de C++. Las reglas de promoción indican las *conversiones implícitas* que el compilador puede realizar entre los tipos fundamentales. Un `int` se puede convertir en `double`. Un `double` también puede convertirse en un `int`, pero se *trunca* la parte fraccionaria del valor `double`. Tenga en cuenta que las variables `double` pueden contener números de una magnitud mucho mayor que las variables `int`, por lo que la pérdida de datos puede ser considerable. Los valores también pueden modificarse al convertir tipos de enteros largos en tipos de enteros pequeños (por ejemplo, de `long` a `short`), números con signo a números sin signo, o viceversa. Los enteros sin signo varían desde 0 a un valor aproximado del doble del rango positivo del tipo con signo correspondiente.

Las reglas de promoción se aplican a expresiones que contienen valores de dos o más tipos de datos; dichas expresiones se conocen también como **expresiones de tipo mixto**. El tipo de cada valor en una expresión de tipo mixto se promueve al tipo "más alto" en la expresión (en realidad, se crea y se utiliza una versión *temporal* de cada valor para la expresión; los valores originales permanecen sin cambios). La promoción también ocurre cuando el tipo de un argumento de función *no* concuerda con el tipo de parámetro especificado en la definición o prototipo de la función. La figura 6.6 lista los tipos de datos aritméticos en orden del "tipo más alto" al "tipo más bajo".

1 Las promociones y conversiones son temas complejos que se ven en la sección 4 y al principio de la sección 5 del estándar de C++. Puede comprar una copia del estándar en `bit.ly/CPlusPlus11Standard`.

Tipos de datos	
`long double`	
`double`	
`float`	
`unsigned long long int`	(sinónimo con `unsigned long long`)
`long long int`	(sinónimo con `long long`)
`unsigned long int`	(sinónimo con `unsigned long`)
`long int`	(sinónimo con `long`)
`unsigned int`	(sinónimo con `unsigned`)
`int`	
`unsigned short int`	(sinónimo con `unsigned short`)
`short int`	(sinónimo con `short`)
`unsigned char`	
`char` y `signed char`	
`bool`	

Fig. 6.6 | Jerarquía de promociones para los tipos de datos fundamentales.

Las conversiones pueden producir valores incorrectos
La conversión de valores a los tipos fundamentales *más bajos* puede producir valores incorrectos. Por lo tanto, un valor se puede convertir en un tipo fundamental menor sólo si se asigna de manera *explícita* el valor a una variable de tipo inferior (algunos compiladores generarán una advertencia en este caso), o mediante el uso de un *operador de conversión* (vea la sección 4.9). Los valores de los argumentos de una función se convierten a los tipos de los parámetros en un prototipo de función, como si se hubieran asignado de manera directa a las variables de esos tipos. Si se hace una llamada a una función `cuadrado`, que utiliza un parámetro entero, con un argumento de punto flotante, el argumento se convierte a `int` (un tipo más bajo) y `cuadrado` podría devolver un valor incorrecto. Por ejemplo, `cuadrado(4.5)` devuelve `16`, no `20.25`.

Error común de programación 6.4
Si los argumentos en la llamada a una función no concuerdan con el número y tipos de los parámetros declarados en el prototipo de función correspondiente, se produce un error de compilación. También es un error si el número de argumentos en la llamada concuerda, pero los argumentos no se pueden convertir de manera implícita a los tipos esperados.

6.6 Encabezados de la Biblioteca estándar de C++

La Biblioteca estándar de C++ está dividida en muchas porciones, cada una con su propio encabezado. Los encabezados contienen los prototipos de función para las funciones relacionadas que forman cada porción de la biblioteca. Los encabezados también contienen definiciones de varios tipos de clases y funciones, así como las constantes que necesitan esas funciones. Un encabezado "instruye" al compilador acerca de cómo interconectarse con los componentes de la biblioteca y los componentes escritos por el usuario.

En la figura 6.7 se listan algunos encabezados comunes de la Biblioteca estándar de C++, la mayoría de los cuales veremos más adelante en el libro.

Encabezado de la Biblioteca estándar de C++	Explicación
`<iostream>`	Contiene prototipos de función para las funciones de entrada y salida estándar de C++, presentadas en el capítulo 2, y que se tratan con más detalle en el capítulo 13, Entrada/salida de flujos: un análisis detallado.
`<iomanip>`	Contiene prototipos de función para los manipuladores de flujo que dan formato a flujos de datos. Este encabezado se utiliza primero en la sección 4.9 y se analiza con más detalle en el capítulo 13, Entrada/salida de flujos: un análisis detallado.
`<cmath>`	Contiene prototipos de función para las funciones de la biblioteca de matemáticas (sección 6.3).
`<cstdlib>`	Contiene prototipos de función para las conversiones de números a texto, de texto a números, asignación de memoria, números aleatorios y varias otras funciones utilitarias. En la sección 6.7 veremos partes de este encabezado; también en el capítulo 11, Sobrecarga de operadores; la clase `string`; en el capítulo 17 (en el sitio web), Manejo de excepciones: un análisis más detallado; en el capítulo 22 (en inglés, en el sitio web), Bits, caracteres, cadenas tipo C y tipos `struct`; y en el apéndice F, Temas sobre código heredado de C.
`<ctime>`	Contiene prototipos de función y tipos para manipular la hora y la fecha. Este encabezado se utiliza en la sección 6.7.
`<array>`, `<vector>`, `<list>`, `<forward_list>`, `<deque>`, `<queue>`, `<stack>`, `<map>`, `<unordered_map>`, `<unordered_set>`, `<set>`, `<bitset>`	Estos encabezados contienen clases que implementan los contenedores de la Biblioteca estándar de C++. Los contenedores almacenan datos durante la ejecución de un programa. El encabezado `<vector>` se introduce por primera vez en el capítulo 7, Plantillas de clase `array` y `vector`; cómo atrapar excepciones. En el capítulo 15 (en el sitio web), Contenedores e iteradores de la Biblioteca estándar hablaremos sobre todos estos encabezados.
`<cctype>`	Contiene prototipos de función para las funciones que evalúan caracteres en base a ciertas propiedades (por ejemplo, si el carácter es un dígito o un signo de puntuación), y prototipos de funciones que se pueden utilizar para convertir letras minúsculas a mayúsculas y viceversa. Hablaremos sobre estos temas en el capítulo 22 (en inglés, en el sitio web), Bits, caracteres, cadenas tipo C y tipos `struct`.
`<cstring>`	Contiene prototipos de funciones para las funciones de procesamiento de cadenas estilo C. Este encabezado se utiliza en el capítulo 10, Sobrecarga de operadores; la clase `string`.
`<typeinfo>`	Contiene clases para la identificación de tipos en tiempo de ejecución (determinar los tipos de datos en tiempo de ejecución). Este archivo de encabezado se describe en la sección 12.8.
`<exception>`, `<stdexcept>`	Estos encabezados contienen clases que se utilizan para manejar excepciones (se describen en el capítulo 17 (en el sitio web), Manejo de excepciones: un análisis más detallado).
`<memory>`	Contiene clases y funciones utilizadas por la Biblioteca estándar de C++ para asignar memoria a los contenedores de la Biblioteca estándar de C++. Este encabezado se utiliza en el capítulo 17 (en el sitio web), Manejo de excepciones: un análisis más detallado.
`<fstream>`	Contiene prototipos de funciones para las funciones que realizan operaciones de entrada desde archivos en disco, y operaciones de salida hacia archivos en disco (que veremos en el capítulo 14, Procesamiento de archivos).
`<string>`	Contiene la definición de la clase `string` de la Biblioteca estándar de C++ (que veremos en el capítulo 21 [en inglés, en el sitio web], La clase `String` y el procesamiento de flujos de cadena).

Fig. 6.7 | Encabezados de la Biblioteca estándar de C++ (parte 1 de 2).

Encabezado de la Biblioteca estándar de C++	Explicación
`<sstream>`	Contiene prototipos de función para las funciones que realizan operaciones de entrada a partir de cadenas en memoria, y operaciones de salida hacia cadenas en memoria (que veremos en el capítulo 21(en inglés, en el sitio web). (La clase `string` y el procesamiento de flujos de cadena).
`<functional>`	Contiene las clases y funciones utilizadas por algoritmos de la Biblioteca estándar de C++. Este encabezado se utiliza en el capítulo 15.
`<iterator>`	Contiene clases para acceder a los datos en los contenedores de la Biblioteca estándar de C++. Este encabezado se utiliza en el capítulo 15.
`<algorithm>`	Contiene las funciones para manipular los datos en los contenedores de la Biblioteca estándar de C++. Este encabezado se utiliza en el capítulo 15.
`<cassert>`	Contiene macros para agregar diagnósticos que ayuden a depurar programas (debug). Este encabezado se utiliza en el apéndice E, Preprocesador.
`<cfloat>`	Contiene los límites del sistema en cuanto al tamaño de los números de punto flotante.
`<climits>`	Contiene los límites del sistema en cuanto al tamaño de los números enteros.
`<cstdio>`	Contiene los prototipos de función para las funciones de la biblioteca de entrada/salida estándar estilo C.
`<locale>`	Contiene clases y funciones que se utilizan comúnmente en el procesamiento de flujos, para procesar datos en la forma natural para distintos lenguajes (por ejemplo, formatos monetarios, almacenamiento de cadenas, presentación de caracteres, etcétera).
`<limits>`	Contiene clases para definir los límites de los tipos de datos numéricos en cada plataforma computacional.
`<utility>`	Contiene clases y funciones utilizadas por muchos encabezados de la Biblioteca estándar de C++.

Fig. 6.7 | Encabezados de la Biblioteca estándar de C++ (parte 2 de 2).

6.7 Caso de estudio: generación de números aleatorios

[*Nota:* las técnicas de generación de números aleatorios que veremos en esta sección y en la sección 6.8 se incluyen para los lectores que todavía no utilizan compiladores de C++11. En la sección 6.9 presentaremos las herramientas mejoradas para generar números aleatorios de C++11].

Ahora analizaremos de manera breve una parte divertida de un tipo popular de aplicaciones de la programación: simulación y juegos. En ésta y en la siguiente sección desarrollaremos un programa de un juego que incluye varias funciones.

El elemento de azar puede introducirse en las aplicaciones computacionales mediante el uso de la función **rand** de la Biblioteca estándar de C++. Considere la siguiente instrucción:

```
i = rand();
```

La función `rand` genera un entero sin signo entre `0` y `RAND_MAX` (una constante simbólica definida en el encabezado `<cstdlib>`. Para determinar el valor de `RAND_MAX` para su sistema, sólo tiene que mostrar la constante. Si `rand` produce verdaderamente enteros al azar, cada número entre `0` y `RAND_MAX` tiene una oportunidad (o probabilidad) *igual* de ser elegido cada vez que se llame a `rand`.

El rango de valores producidos directamente por la función `rand` es a menudo distinto de lo que requiere una aplicación específica. Por ejemplo, un programa que simula el lanzamiento de una moneda sólo requiere 0 para "águila" y 1 para "sol". Un programa para simular el tiro de un dado de seis lados requeriría enteros aleatorios en el rango de 1 a 6. Un programa que adivine en forma aleatoria el siguiente tipo de nave espacial (de cuatro posibilidades distintas) que volará a lo largo del horizonte en un videojuego requeriría números aleatorios en el rango de 1 a 4.

Tirar un dado de seis lados

Para demostrar la función `rand`, en la figura 6.8 se simulan 20 tiros de un dado de seis lados, y se muestra el valor de cada tiro. El prototipo de la función `rand` se encuentra en `<cstdlib>`. Para producir valores enteros en el rango de 0 a 5, usamos el operador módulo (%) con `rand`, como se muestra a continuación:

```
rand() % 6
```

A esto se le conoce como **escalar**. El número 6 se conoce como el **factor de escala**. Después **desplazamos** el rango de números producidos sumando 1 a nuestro resultado anterior. En la figura 6.8 se confirma que los resultados están en el rango de 1 a 6. Si ejecuta este programa más de una vez, podrá comprobar que produce los mismos valores "aleatorios" cada vez. En la figura 6.10 le mostraremos cómo corregir esto.

```
1  // Fig. 6.8: fig06_08.cpp
2  // Enteros desplazados y escalados, producidos por 1 + rand() % 6.
3  #include <iostream>
4  #include <iomanip>
5  #include <cstdlib> // contiene el prototipo de funcion para rand
6  using namespace std;
7
8  int main()
9  {
10   // itera 20 veces
11   for ( unsigned int contador = 1; contador <= 20; ++contador )
12   {
13      // elije un número aleatorio de 1 a 6 y lo imprime
14      cout << setw( 10 ) << ( 1 + rand() % 6 );
15
16      // si contador puede dividirse entre 5, empieza una nueva línea de salida
17      if ( contador % 5 == 0 )
18         cout << endl;
19   } // fin de for
20 } // fin de main
```

```
         6         6         5         5         6
         5         1         1         5         3
         6         6         2         4         2
         6         2         3         4         1
```

Fig. 6.8 | Enteros desplazados y escalados, producidos por `1 + rand() % 6`.

Tirar un dado de seis lados 6 000 000 veces

Para mostrar que los números que produce la función `rand` ocurren con una probabilidad aproximadamente igual, la figura 6.9 simula 6 000 000 de tiros de un dado. Cada entero en el rango de 1 a 6 debe aparecer aproximadamente 1 000 000 veces. Esto se confirma mediante la salida del programa.

```
1  // Fig. 6.9: fig06_09.cpp
2  // Tiro de un dado de seis lados 6 000 000 veces.
3  #include <iostream>
4  #include <iomanip>
5  #include <cstdlib> // contiene el prototipo de la función rand
6  using namespace std;
7
8  int main()
9  {
10    unsigned int frecuencia1 = 0; // cuenta las veces que se tiró 1
11    unsigned int frecuencia2 = 0; // cuenta las veces que se tiró 2
12    unsigned int frecuencia3 = 0; // cuenta las veces que se tiró 3
13    unsigned int frecuencia4 = 0; // cuenta las veces que se tiró 4
14    unsigned int frecuencia5 = 0; // cuenta las veces que se tiró 5
15    unsigned int frecuencia6 = 0; // cuenta las veces que se tiró 6
16
17    // sintetiza los resultados de tirar un dado 6 000 000 veces
18    for ( unsigned int tiro = 1; tiro <= 6000000; ++tiro )
19    {
20       unsigned int cara = 1 + rand() % 6; // número aleatorio del 1 al 6
21
22       // determina el valor del tiro de 1 a 6 e incrementa el contador apropiado
23       switch ( cara )
24       {
25          case 1:
26             ++frecuencia1; // incrementa el contador de 1
27             break;
28          case 2:
29             ++frecuencia2; // incrementa el contador de 2
30             break;
31          case 3:
32             ++frecuencia3; // incrementa el contador de 3
33             break;
34          case 4:
35             ++frecuencia4; // incrementa el contador de 4
36             break;
37          case 5:
38             ++frecuencia5; // incrementa el contador de 5
39             break;
40          case 6:
41             ++frecuencia6; // incrementa el contador de 6
42             break;
43          default: // valor inválido
44             cout << "El programa nunca debio llegar aqui!";
45       } // fin de switch
46    } // fin de for
47
48    cout << "Cara" << setw( 13 ) << "Frecuencia" << endl; // imprime encabezados
49    cout << "   1" << setw( 13 ) << frecuencia1
50       << "\n   2" << setw( 13 ) << frecuencia2
51       << "\n   3" << setw( 13 ) << frecuencia3
52       << "\n   4" << setw( 13 ) << frecuencia4
```

Fig. 6.9 | Tiro de un dado de seis lados 6 000 000 veces (parte 1 de 2).

```
53        << "\n   5" << setw( 13 ) << frecuencia5
54        << "\n   6" << setw( 13 ) << frecuencia6 << endl;
55    } // fin de main
```

```
Cara   Frecuencia
   1       999702
   2      1000823
   3       999378
   4       998898
   5      1000777
   6      1000422
```

Fig. 6.9 | Tiro de un dado de seis lados 6 000 000 veces (parte 2 de 2).

Como se muestra en los resultados del programa, al escalar y desplazar los valores producidos por `rand`, podemos simular el tiro de un dado de seis lados. El programa *nunca* debe llegar al caso `default` (líneas 43 y 44) en la estructura `switch`, ya que la expresión de control del `switch` (`cara`) *siempre* tiene valores en el rango de 1 a 6; sin embargo, proporcionamos el caso `default` como una cuestión de buena práctica. Una vez que estudiemos los arreglos en el capítulo 7, le mostraremos cómo reemplazar toda la estructura `switch` de la figura 6.9 de una manera elegante, con una instrucción de una sola línea.

Tip para prevenir errores 6.3

Hay que proporcionar un caso `default` *en una instrucción* `switch` *para atrapar errores, ¡incluso si estamos absoluta y positivamente seguros de no tener errores!*

Randomización del generador de números aleatorios

Al ejecutar el programa de la figura 6.8 otra vez, se produce lo siguiente:

```
6       6       5       5       6
5       1       1       5       3
6       6       2       4       2
6       2       3       4       1
```

El programa imprime exactamente la *misma* secuencia de valores que se muestra en la figura 6.8. ¿Cómo pueden ser estos números aleatorios? *Al depurar un programa de simulación, esta repetitividad es esencial para demostrar que las correcciones al programa funcionan en forma apropiada.*

En realidad, la función `rand` genera **números seudoaleatorios**. Si se llama repetidas veces a `rand`, se produce una secuencia de números que parecen ser aleatorios. No obstante, la secuencia se *repite* a sí misma cada vez que se ejecuta el programa. Una vez que un programa se ha depurado extensivamente, puede condicionarse para producir una secuencia *diferente* de números aleatorios para cada ejecución. A esto se le conoce como **randomización**, y se logra mediante la función **`srand`** de la Biblioteca estándar de C++. La función `srand` recibe un argumento entero `unsigned` y **siembra** la función `rand` para que produzca una secuencia distinta de números aleatorios para cada ejecución del programa. C++11 proporciona herramientas adicionales para números aleatorios que pueden producir **números aleatorios no determinísticos**: un conjunto de números aleatorios que no pueden producirse. Dichos generadores de números aleatorios se utilizan en simulaciones y escenarios de seguridad en donde la predictibilidad es indeseable. La sección 6.9 introduce las herramientas de generación de números aleatorios de C++11.

Buena práctica de programación 6.1

Asegúrese de que su programa siembre el generador de números aleatorios de manera distinta *(y* sólo una vez*) cada vez que se ejecute el programa; de lo contrario, un atacante podría determinar con facilidad la secuencia de números seudoaleatorios que se producirían.*

Sembrar el generador de números aleatorios con `srand`

En la figura 6.10 se demuestra el uso de la función `srand`. El programa utiliza el tipo de datos `unsigned int`. Un valor `int` se representa cuando menos por dos bytes; por lo general, cuatro bytes en los sistemas de 32 bits y puede ser de hasta ocho bytes en sistemas de 64 bits. Un `int` puede tener valores positivos y negativos. Una variable de tipo `unsigned int` también se almacena en al menos dos bytes de memoria. Un valor `unsigned int` de cuatro bytes puede tener sólo valores *no negativos* en el rango de 0 a 4294967295. La función `srand` recibe un valor `unsigned int` como argumento. El prototipo para la función `srand` se encuentra en el encabezado `<cstdlib>`.

```
1  // Fig. 6.10: fig06_10.cpp
2  // Randomización del programa para tirar dados.
3  #include <iostream>
4  #include <iomanip>
5  #include <cstdlib> // contiene los prototipos para las funciones srand y rand
6  using namespace std;
7
8  int main()
9  {
10   unsigned int semilla = 0; // almacena la semilla introducida por el usuario
11
12   cout << "Introduzca la semilla: ";
13   cin >> semilla;
14   srand( semilla ); // siembra el generador de números aleatorios
15
16   // itera 10 veces
17   for ( unsigned int contador = 1; contador <= 10; ++contador )
18   {
19     // elije un número aleatorio entre 1 y 6, y lo imprime
20     cout << setw( 10 ) << ( 1 + rand() % 6 );
21
22     // si contador puede dividirse entre 5, empieza una nueva línea de salida
23     if ( contador % 5 == 0 )
24       cout << endl;
25   } // fin de for
26 } // fin de main
```

```
Introduzca la semilla: 67
         6         1         4         6         2
         1         6         1         6         4
```

```
Introduzca la semilla: 432
         4         6         3         1         6
         3         1         5         4         2
```

Fig. 6.10 | Randomización del programa para tirar dados (parte 1 de 2).

```
Introduzca la semilla: 67
      6       1       4       6       2
      1       6       1       6       4
```

Fig. 6.10 | Randomización del programa para tirar dados (parte 2 de 2).

El programa produce una secuencia *distinta* de números aleatorios cada vez que se ejecuta, siempre y cuando el usuario introduzca una semilla *distinta*. Utilizamos la *misma* semilla en la primera y la tercera ventana de resultados, por lo que se muestra la *misma* serie de 10 números en cada uno de esos resultados.

Sembrar el generador de números aleatorios con la hora actual

Para randomizar *sin* tener que introducir una semilla cada vez, podemos usar una instrucción como la siguiente:

```
srand( static_cast<unsigned int>( time( 0 ) ) );
```

Esto hace que la computadora lea su *reloj* para obtener el valor para la semilla. Por lo general, la función `time` (con el argumento `0`, como se escribe en la instrucción anterior) devuelve la hora actual como el número de segundos transcurridos desde enero 1, 1970, a media noche en Tiempo del Meridiano de Greenwich (GMT). Este valor (que es de tipo `time_t`) se convierte en un `unsigned int` y se utiliza como semilla para el generador de números aleatorios; la palabra clave `static_cast` en la instrucción anterior elimina una advertencia del compilador que se genera si pasamos un valor `time_t` a una función que espera un valor `unsigned int`. El prototipo de la función `time` está en `<ctime>`.

Escalamiento y desplazamiento de números aleatorios

Anteriormente simulamos cómo tirar un dado de seis lados con la instrucción:

```
cara = 1 + rand() % 6;
```

la cual siempre asigna un entero (al azar) a la variable `cara` en el rango $1 \leq$ `cara` ≤ 6. La amplitud de este rango (es decir, el número de enteros consecutivos en el rango) es 6, y el número inicial en el rango es 1. Si hacemos referencia a la instrucción anterior, podemos ver que la amplitud del rango se determina en base al número que se utiliza para escalar `rand` con el operador módulo (es decir, 6), y el número inicial del rango es igual al número (es decir, 1) que se agrega a la expresión `rand % 6`. Podemos generalizar este resultado de la siguiente manera:

```
numero = valorDesplazamiento + rand() % factorEscala;
```

en donde ***valorDesplazamiento*** es igual al *primer número* en el rango deseado de enteros consecutivos y ***factorEscala*** es igual a la *amplitud* del rango deseado de enteros consecutivos.

6.8 Caso de estudio: juego de probabilidad; introducción a enum

Uno de los juegos de azar más populares es el juego de dados conocido como "craps", el cual se juega en casinos y callejones por todo el mundo. Las reglas del juego son simples:

> *Un jugador tira dos dados. Cada dado tiene seis caras, las cuales contienen 1, 2, 3, 4, 5 y 6 puntos negros. Una vez que los dados dejan de moverse, se calcula la suma de los puntos negros en las dos caras superiores. Si la suma es 7 u 11 en el primer tiro, el jugador gana. Si la suma es 2, 3 o 12 en el primer tiro (llamado "craps"), el jugador pierde (es decir, la "casa" gana). Si la suma es 4, 5, 6, 8, 9 o 10 en el primer tiro, esta suma se convierte en el "punto" del jugador. Para ganar, el jugador debe seguir tirando los dados hasta que salga otra vez "su punto". El jugador pierde si tira un 7 antes de llegar a su punto.*

El programa de la figura 6.11 simula el juego de craps. En las reglas del juego, observe que el jugador debe tirar dos dados en el primer tiro y en todos los tiros subsiguientes. Definimos la función `tirarDados` (líneas 62 a 74) para tirar el dado, calcular e imprimir la suma. La función está definida sólo una vez, pero se llama desde las líneas 20 y 44. La función no recibe argumentos y devuelve la suma de los dos dados, por lo que se indican paréntesis vacíos y el tipo de valor de retorno `unsigned int` en el prototipo de la función (línea 8) y en el encabezado de la misma (línea 62).

```
1  // Fig. 6.11: fig06_11.cpp
2  // Simulación del juego de "craps".
3  #include <iostream>
4  #include <cstdlib> // contiene los prototipos para las funciones srand y rand
5  #include <ctime> // contiene el prototipo para la función time
6  using namespace std;
7
8  unsigned int rollDice(); // tira los dados, calcula y muestra la suma
9
10 int main()
11 {
12    // enumeración con constantes que representa el estado del juego
13    enum Estado { CONTINUAR, GANO, PERDIO };
      // todas las letras mayúsculas en las constantes
14
15    // randomiza el generador de números aleatorios, usando la hora actual
16    srand( static_cast<unsigned int>( time( 0 ) ) );
17
18    unsigned int miPunto = 0; // punto si no se gana o pierde en el primer tiro
19    Estado estadoJuego = CONTINUAR; // puede contener CONTINUAR, GANO o PERDIO
20    unsigned int sumaDeDados = tirarDados() ; // primer tiro del dado
21
22    // determina el estado del juego y el punto (si es necesario), con base en el
         primer tiro
23    switch ( sumaDeDados )
24    {
25       case 7: // gana con 7 en el primer tiro
26       case 11: // gana con 11 en el primer tiro
27          estadoJuego = GANO;
28          break;
29       case 2: // pierde con 2 en el primer tiro
30       case 3: // pierde con 3 en el primer tiro
31       case 12: // pierde con 12 en el primer tiro
32          estadoJuego = PERDIO;
33          break;
34       default: // no gano ni perdio, por lo que recuerda el punto
35          estadoJuego = CONTINUAR; // el juego no ha terminado
36          miPunto = sumaDeDados; // recuerda el punto
37          cout << "El punto es " << miPunto << endl;
38          break; // opcional al final del switch
39    } // fin de switch
40
41    // mientras el juego no esté completo
42    while ( CONTINUAR == estadoJuego ) // no GANO ni PERDIO
43    {
```

Fig. 6.11 | Simulación del juego de "craps" (parte 1 de 3).

```
44        sumaDeDados = tirarDados(); // tira los dados de nuevo
45
46        // determina el estado del juego
47        if ( sumaDeDados == miPunto ) // gana al hacer un punto
48           estadoJuego = GANO;
49        else
50           if ( sumaDeDados == 7 ) // pierde al tirar 7 antes del punto
51              estadoJuego = PERDIO;
52     } // fin de while
53
54     // muestra mensaje de que ganó o perdió
55     if ( GANO == estadoJuego )
56        cout << "El jugador gana" << endl;
57     else
58        cout << "El jugador pierde" << endl;
59  } // fin de main
60
61  // tira los dados, calcula la suma y muestra los resultados
62  unsigned int tirarDados()
63  {
64     // elige valores aleatorios para el dado
65     unsigned int dado1 = 1 + rand() % 6; // tiro del primer dado
66     unsigned int dado2 = 1 + rand() % 6; // tiro del segundo dado
67
68     unsigned int suma = dado1 + dado2; // calcula la suma de valores de los dados
69
70     // muestra los resultados de este tiro
71     cout << "El jugador tiro " << dado1 << " + " << dado2
72        << " = " << suma << endl;
73     return suma; // devuelve la suma de los dados
74  } // fin de la función tirarDados
```

```
El jugador tiro 2 + 5 = 7
El jugador gana
```

```
El jugador tiro 6 + 6 = 12
El jugador pierde
```

```
El jugador tiro 1 + 3 = 4
El punto es 4
El jugador tiro 4 + 6 = 10
El jugador tiro 2 + 4 = 6
El jugador tiro 6 + 4 = 10
El jugador tiro 2 + 3 = 5
El jugador tiro 2 + 4 = 6
El jugador tiro 1 + 1 = 2
El jugador tiro 4 + 4 = 8
El jugador tiro 4 + 3 = 7
El jugador pierde
```

Fig. 6.11 | Simulación del juego de "craps" (parte 2 de 3).

```
El jugador tiro 3 + 3 = 6
El punto es 6
El jugador tiro 5 + 3 = 8
El jugador tiro 4 + 5 = 9
El jugador tiro 2 + 1 = 3
El jugador tiro 1 + 5 = 6
El jugador gana
```

Fig. 6.11 | Simulación del juego de "craps" (parte 3 de 3).

El tipo de `enum Estado`

El jugador puede ganar o perder en el primer tiro, o en cualquier tiro subsiguiente. El programa utiliza la variable `estadoJuego` para llevar la cuenta de esto. La variable `estadoJuego` se declara como del nuevo tipo `Estado`. En la línea 13 se declara un tipo definido por el usuario, llamado **enumeración**, que se introduce mediante la palabra clave **`enum`** (por enumeración) y va seguida de un **nombre de tipo** (en este caso, `Estado`), además de un conjunto de constantes enteras representadas por identificadores. Los valores de estas **constantes de enumeración** empiezan en `0`, a menos que se especifique lo contrario, y se incrementan en `1`. En la enumeración anterior, la constante `CONTINUAR` tiene el valor 0, `GANO` tiene el valor 1 y `PERDIO` tiene el valor 2. Los identificadores en una enumeración deben ser únicos, pero las constantes de enumeración separadas *pueden* tener el mismo valor entero.

Buena práctica de programación 6.2

La primera letra de un identificador que se utilice como un nombre de tipo definido por el usuario debe ir en mayúscula.

Buena práctica de programación 6.3

Use sólo letras mayúsculas en los nombres de las constantes de enumeración. Esto hace que resalten y le recuerdan que las constantes de enumeración no son variables.

Las variables del tipo `Estado` definido por el usuario pueden recibir sólo uno de los tres valores declarados en la enumeración. Cuando se gana el juego, el programa establece la variable `estadoJuego` a `GANO` (líneas 27 y 48). Cuando se pierde el juego, el programa establece la variable `estadoJuego` a `PERDIO` (líneas 32 y 51). En caso contrario, el programa establece la variable `estadoJuego` a `CONTINUAR` (línea 35) para indicar que el dado se debe tirar de nuevo.

Error común de programación 6.5

Asignar el equivalente entero de una constante de enumeración (en vez de la misma constante de enumeración) a una variable del tipo de enumeración es un error de compilación.

Otra enumeración popular es

```
enum Meses { ENE = 1, FEB, MAR, ABR, MAY, JUN, JUL, AGO,
   SEP, OCT, NOV, DIC };
```

la cual crea el tipo `Meses` definido por el usuario, con constantes de enumeración que representan los meses del año. El primer valor en la enumeración anterior se establece explícitamente en `1`, por lo que el resto de los valores se incrementan a partir de `1`, lo cual produce los valores del `1` al `12`. Cualquier constante de enumeración puede recibir un valor entero en la definición de la enumeración, y cada una de las constantes de enumeración subsiguientes tiene un valor igual a 1 más que la constante anterior en la lista, hasta la siguiente configuración explícita.

Tip para prevenir errores 6.4

Use valores únicos para las constantes de una enumeración, para ayudar a evitar los errores lógicos difíciles de localizar.

Ganar o perder en el primer tiro

Después del primer tiro, si se gana o pierde el juego, el programa omite el cuerpo de la instrucción `while` (líneas 42 a 52) debido a que `estadoJuego` no es igual a `CONTINUAR`. El programa pasa a la instrucción `if...else` en las líneas 55 a 58, que imprime `"El jugador gana"` si `estadoJuego` es igual a `GANO` y `"El jugador pierde"` si `estadoJuego` es igual a `PERDIO`.

Continuar los tiros

Después del primer tiro, si el juego no ha terminado, el programa guarda la suma en `miPunto` (línea 36). La ejecución continúa con la instrucción `while`, ya que `estadoJuego` es igual a `CONTINUAR`. Durante cada iteración del `while`, el programa llama a `tirarDados` para producir una nueva `suma`. Si `suma` concuerda con `miPunto`, el programa establece `estadoJuego` en `GANO` (línea 48), la prueba del `while` falla, la instrucción `if...else` imprime `"El jugador gana"` y termina la ejecución. Si `suma` es igual a `7`, el programa establece `estadoJuego` a `PERDIO` (línea 51), falla la prueba del `while`, la instrucción `if...else` imprime `"El jugador pierde"` y termina la ejecución.

El programa de "craps" utiliza dos funciones (`main` y `tirarDados`) y las instrucciones `switch`, `while`, `if...else`, `if...else` anidadas e `if` anidadas. En los ejercicios investigamos varias características interesantes del juego de "craps".

C++11: `enum` con alcance

En la figura 6.11 presentamos las enumeraciones (`enum`). Un problema con las enumeraciones (que también se conocen como *enumeraciones sin alcance*) es que varias enumeraciones pueden contener los *mismos* identificadores. Si se utilizan dichas enumeraciones en el mismo programa, se pueden producir conflictos de nombres y errores lógicos. Para eliminar estos problemas, C++11 introduce lo que se conoce como **enumeraciones con alcance**, que se declaran con las palabras clave **`enum class`** (o con el sinónimo **`enum struct`**). Por ejemplo, podemos definir la enumeración `Estado` de la figura 6.11 como:

```
enum class Estado { CONTINUAR, GANO, PERDIO };
```

Para hacer referencia a una constante `enum` con alcance, *debemos* calificar la constante con el nombre del tipo `enum` con alcance (`Estado`) y el operador de resolución de ámbito (`::`), como en `Estado::CONTINUAR`. Esto *identifica explícitamente* a `CONTINUAR` como una constante en el *alcance* de la `enum class Estado`. De esta forma, si otra `enum` con alcance contiene el mismo identificador para una de sus constantes, siempre queda claro qué versión de la constante se está utilizando.

Tip para prevenir errores 6.5

Use `enum` con alcance para evitar conflictos de nombres y errores lógicos potenciales debido a las enumeraciones sin alcance que contienen los mismos identificadores.

C++11: especificar el tipo de las constantes de una enumeración

Las constantes en una `enum` se representan como enteros. De manera predeterminada, el tipo integral subyacente de una `enum` sin alcance depende de los valores de sus constantes; se garantiza que el tipo será lo bastante grande como para almacenar los valores constantes especificados. De manera predeterminada, el tipo integral subyacente de una `enum` con alcance es `int`. C++11 nos permite especificar el tipo integral subyacente de una `enum` siguiendo el nombre del tipo de `enum` con dos puntos (`:`) además del tipo integral. Por ejemplo, podemos especificar que las constantes en la `enum class Estado` tengan el tipo `unsigned int`, como en:

```
enum class Estado : unsigned int { CONTINUAR, GANO, PERDIO };
```

Error común de programación 6.6

Si el valor de la constante de una `enum` *está fuera del rango que puede representarse por el tipo subyacente de esa* `enum`*, se produce un error de compilación.*

6.9 Números aleatorios de C++11

De acuerdo con el CERT, la función `rand` no tiene "buenas propiedades estadísticas" y puede ser predecible, lo que hace a los programas que usan `rand` menos seguros (Lineamiento de CERT MSC30-CPP). Como mencionamos en la sección 6.7, C++11 cuenta con una nueva y *más segura* biblioteca de herramientas de generación de números aleatorios que pueden producir números aleatorios no determinísticos para simulaciones y casos de seguridad, en donde no es conveniente la predictibilidad. Estas nuevas herramientas se encuentran en el encabezado `<random>` de la Biblioteca estándar de C++.

La generación de números aleatorios es un tema matemático sofisticado, para el cual los matemáticos han desarrollado muchos algoritmos de generación de números aleatorios con diferentes propiedades estadísticas. Por cuestión de flexibilidad, con base en cómo se utilizan los números aleatorios en los programas, C++11 cuenta con muchas clases que representan diversos *motores* de generación de números aleatorios y *distribuciones*. Un motor implementa un algoritmo de generación de números aleatorios que produce números seudoaleatorios. Una distribución controla el rango de valores producidos por un motor, los tipos de esos valores (por ejemplo, `int`, `double`, etc.) y las propiedades estadísticas de los valores. En esta sección, usaremos el motor de generación de números aleatorios predeterminado (`default_random_engine`) y la distribución `uniform_int_distribution`, que distribuye *uniformemente* los enteros seudoaleatorios a través de un rango especificado de valores. El rango predeterminado es de 0 al valor máximo de un `int` en su plataforma.

Tirar un dado de seis lados

La figura 6.12 usa `default_random_engine` y `uniform_int_distribution` para tirar un dado de seis lados. La línea 14 crea un objeto `default_random_engine` llamado `motor`. El argumento de su constructor *siembra* el motor de generación de números aleatorios con la hora actual. Si no pasamos un valor al constructor, se utilizará la semilla predeterminada y el programa producirá la *misma* secuencia de números cada vez que se ejecute. La línea 15 crea `intAleatorio`: un objeto `uniform_int_distribution` que produce valores `unsigned int` (según lo especificado por `<unsigned int>`) en el rango de 1 a 6 (como lo especifican los argumentos del constructor). La expresión `intAleatorio(motor)` (línea 21) devuelve un valor `unsigned int` en el rango de 1 a 6.

```
1  // Fig. 6.12: fig06_12.cpp
2  // Uso de un motor de generación de números aleatorios y una distribución
3  // de C++11 para tirar un dado con seis lados.
4  #include <iostream>
5  #include <iomanip>
6  #include <random> // contiene herramientas de generación de números aleatorios
                     de C++11
7  #include <ctime>
8  using namespace std;
9
10 int main()
11 {
```

Fig. 6.12 | Uso de un motor de generación de números aleatorios y una distribución de C++11 para tirar un dado con seis lados (parte 1 de 2).

```
12    // usa el motor de generación de números aleatorios predeterminado para
13    // producir valores int seudoaleatorios del 1 al 6, distribuidos de manera
         uniforme
14    default_random_engine motor( static_cast<unsigned int>( time(0) ) );
15    uniform_int_distribution<unsigned int> intAleatorio( 1, 6 );
16
17    // itera 10 veces
18    for ( unsigned int contador = 1; contador <= 10; ++contador )
19    {
20       // elije un número aleatorio del 1 al 6 y lo imprime
21       cout << setw( 10 ) << intAleatorio( motor );
22
23       // si contador es divisible entre 5, comienza una nueva línea de salida
24       if ( contador % 5 == 0 )
25          cout << endl;
26    } // fin de for
27 } // fin de main
```

```
         2         1         2         3         5
         6         1         5         6         4
```

Fig. 6.12 | Uso de un motor de generación de números aleatorios y una distribución de C++11 para tirar un dado con seis lados (parte 2 de 2).

La notación <unsigned int> en la línea 15 indica que uniform_int_distribution es una *plantilla de clase*. En este caso, puede especificarse cualquier tipo entero entre los signos < y >. En el capítulo 18 (en inglés, en el sitio web) veremos cómo crear plantillas de clases y en otros capítulos le mostraremos cómo usar las plantillas de clases existentes de la Biblioteca estándar de C++. Por ahora, siéntase cómodo al usar la plantilla de clase uniform_int_distribution, imitando la sintaxis que se muestra en el ejemplo.

6.10 Clases y duración de almacenamiento

Los programas que hemos visto hasta ahora utilizan identificadores para nombres de variables y funciones. Los atributos de las variables incluyen su *nombre*, *tipo*, *tamaño* y *valor*. Cada identificador en un programa tiene también otros atributos, incluyendo la **duración del almacenamiento**, el alcance y la **vinculación**.

C++ proporciona cinco **especificadores de clase de almacenamiento** que determinan la duración del almacenamiento de una variable: **register**, **extern**, **mutable**, **static** y **thread_local**. En esta sección hablaremos sobre los especificadores de clase de almacenamiento register, extern y static. El especificador de clase de almacenamiento mutable se utiliza exclusivamente con las clases y thread_local se utiliza en las aplicaciones multihilo (hablaremos sobre ellos en los capítulos 23 y 24 [en inglés, en el sitio web], respectivamente).

Duración del almacenamiento

La *duración del almacenamiento* de un identificador determina el periodo durante el cual éste *existe en la memoria*. Algunos identificadores existen brevemente, algunos se crean y destruyen repetidas veces, y otros existen durante toda la ejecución de un programa. Primero hablaremos sobre las clases de almacenamiento **static** y **automatic**.

Alcance

El *alcance* de un identificador es la parte en la que *se puede hacer referencia a éste* en un programa. Se puede hacer referencia a algunos identificadores a lo largo de un programa; otros identificadores sólo se pueden referenciar desde ciertas partes limitadas de un programa. En la sección 6.11 hablaremos sobre el alcance de los identificadores.

Enlace

El enlace de un identificador determina si se conoce sólo en el *archivo fuente en el que se declara*, o *en varios archivos fuente que se compilen y después se enlacen*. El *especificador de clase de almacenamiento* de un identificador ayuda a determinar la duración de su almacenamiento y su enlace.

Duración del almacenamiento

Los especificadores de clases de almacenamiento se pueden dividir en cuatro duraciones de almacenamiento: *automática*, *estática*, *dinámica* y *de hilo*. Las duraciones de almacenamiento automática y estática se describen a continuación. En el capítulo 10 aprenderá que puede solicitar memoria adicional en su programa durante la ejecución del mismo; a esto se le conoce como *asignación dinámica de memoria*. Las variables que se asignan en forma dinámica tienen *duración de almacenamiento dinámica*. En el capítulo 24 (en inglés, en el sitio web) hablaremos sobre la *duración de almacenamiento de hilo*.

Variables locales y duración de almacenamiento automática

Las variables con *duración de almacenamiento automática* incluyen:

- variables locales declaradas en funciones
- parámetros de funciones
- variables locales o parámetros de funciones declarados con `register`

Dichas variables se crean cuando la ejecución del programa entra al bloque en el que se definen, existen mientras el bloque esté activo y se destruyen cuando el programa se sale del bloque. Una variable automática existe sólo en el *par circundante más cercano de llaves* dentro del cuerpo de la función en la cual aparece la definición, o durante todo el cuerpo de la función en el caso de un parámetro de función. Las variables locales son de duración de almacenamiento automática de *manera predeterminada*. En el resto del libro, nos referiremos a las variables de duración de almacenamiento automática simplemente como *variables automáticas*.

Tip de rendimiento 6.1

El almacenamiento automático es un medio de conservar la memoria, ya que las variables de duración de almacenamiento automática existen en memoria sólo cuando se ejecuta el bloque en el cual están definidas.

Observación de Ingeniería de Software 6.6

El almacenamiento automático es un ejemplo del ***principio del menor privilegio****. En el contexto de una aplicación, el principio establece que el código debe recibir* sólo *el nivel de privilegio y acceso que requiere para realizar su tarea designada,* pero no más. *¿Por qué deberíamos tener variables almacenadas en memoria y accesibles cuando no se necesitan?*

Buena práctica de programación 6.4

Declare las variables lo más cerca posible de donde se utilicen por primera vez.

Variables de registro

Los datos en la versión de lenguaje máquina de un programa se cargan generalmente en los registros, para cálculos y otros tipos de procesamiento.

El compilador podría ignorar las declaraciones `register`. Por ejemplo, tal vez no haya un número suficiente de registros disponibles. La siguiente definición *sugiere* que la variable `unsigned int contador` se coloque en uno de los registros de la computadora; sin importar que el compilador haga esto o no, `contador` se inicializa en 1:

```
register unsigned int contador = 1;
```

La palabra clave `register` se puede utilizar *sólo* con variables locales y parámetros de funciones.

Tip de rendimiento 6.2

El especificador de clase de almacenamiento `register` *se puede colocar antes de la declaración de una variable automática, para sugerir que el compilador debe mantener la variable en uno de los registros de hardware de alta velocidad de la computadora, en vez de hacerlo en memoria. Si las variables de uso intensivo, como los contadores o totales, se mantienen en los registros de hardware, se elimina la sobrecarga de cargar de manera repetitiva las variables de memoria hacia los registros, y almacenar los resultados de vuelta a la memoria.*

Tip de rendimiento 6.3

Por lo general, es innecesario el uso de `register`*. Los compiladores optimizadores de la actualidad pueden reconocer las variables de uso frecuente y colocarlas en registros, sin necesitad de una declaración* `register`*.*

Duración de almacenamiento estática

Las palabras clave `extern` y `static` declaran identificadores para variables con *duración de almacenamiento estática* y para funciones. Las variables con duración de almacenamiento estática existen en memoria a partir del punto en el que el programa empieza a ejecutarse, y dejan de existir cuando termina el programa. Dicha variable se *inicializa una vez al encontrar su declaración*. Para las funciones, el nombre de la función existe cuando el programa empieza a ejecutarse. Aun cuando los nombres de las funciones y las variables de duración estática existen desde el inicio de la ejecución del programa, esto no implica que estos identificadores se puedan utilizar a lo largo de todo el programa. La duración del almacenamiento y el alcance (en dónde se puede usar un nombre) son cuestiones separadas, como veremos en la sección 6.11.

Identificadores con duración de almacenamiento estática

Hay dos tipos de identificadores con *duración de almacenamiento estática*; los identificadores externos (como las variables globales) y las variables locales declaradas con el especificador de clase de almacenamiento `static`. Para crear **variables globales**, se colocan declaraciones de variables *fuera* de cualquier definición de clase o función. Las variables globales retienen sus valores a lo largo de la ejecución del programa. Las variables y las funciones globales se pueden referenciar mediante cualquier función que siga sus declaraciones o definiciones en el archivo fuente.

Observación de Ingeniería de Software 6.7

Al declarar una variable como global en vez de local, se permite la ocurrencia de efectos secundarios inesperados *cuando una función que no requiere acceso a la variable la modifica en forma accidental o premeditada. Esto es otro ejemplo del principio del menor privilegio. En general, con la excepción de los recursos verdaderamente globales como* `cin` *y* `cout`*, debe evitarse el uso de variables globales, excepto en ciertas situaciones con requerimientos de rendimiento únicos.*

Observación de Ingeniería de Software 6.8

Las variables que se utilizan sólo en una función específica deben declararse como locales en esa función, en vez de declararlas como variables globales.

Variables locales `static`

Las variables locales que se declaran como `static` siguen siendo conocidas sólo en la función en la que se declaran, pero a diferencia de las variables automáticas, las *variables locales* `static` *retienen sus valores cuando la función regresa a la función que la llamó*. La próxima vez que se hace una llamada a la función,

las variables locales `static` contienen los valores que tenían cuando la función se ejecutó por última vez. La siguiente instrucción declara la variable local `cuenta` como `static`, y la inicializa en 1:

```
static unsigned int cuenta = 1;
```

Todas las variables numéricas con duración de almacenamiento estática se *inicializan con cero de manera predeterminada*, pero sin duda es una buena práctica inicializar todas las variables en forma explícita.

Los especificadores de clase de almacenamiento `extern` y `static` tienen un significado especial cuando se aplican de manera explícita a los identificadores externos, como las variables globales y los nombres de funciones globales. En el apéndice F, Temas sobre código heredado de C, hablaremos sobre el uso de `extern` y `static` con identificadores externos y programas con varios archivos de código fuente.

6.11 Reglas de alcance

La porción del programa en la que se puede utilizar un identificador se conoce como su *alcance*. Por ejemplo, cuando declaramos una variable local en un bloque, sólo se puede referenciar en ese bloque y en los bloques anidados dentro de ese mismo bloque. En esta sección hablaremos sobre **alcance de bloque**, **alcance de función**, **alcance de espacio de nombres global** y **alcance de prototipo de función**. Más adelante veremos otros dos tipos de alcance: **alcance de clase** (capítulo 9) y **alcance de espacio de nombres** (capítulo 23, en inglés, en el sitio web).

Alcance de bloque

Los identificadores que se declaran *dentro* de un bloque tienen *alcance de bloque*, el cual empieza en la declaración del identificador y termina en la llave derecha de finalización (}) del bloque en el que se declara el identificador. Las variables locales tienen alcance de bloque, al igual que los parámetros de las funciones. Cualquier bloque puede contener declaraciones de variables. Cuando los bloques están anidados y un identificador en un bloque exterior tiene el mismo nombre que un identificador en un bloque interior, el identificador del bloque exterior se "oculta" hasta que termine el bloque interior. El bloque interior "ve" el valor de su propio identificador local y no el valor del identificador del bloque circundante que tiene el nombre idéntico. Las variables locales declaradas como `static` siguen teniendo alcance de bloque, aun cuando existan desde el momento en que el programa empieza su ejecución. La duración del almacenamiento *no* afecta al alcance de un identificador.

Error común de programación 6.7

El uso accidental del mismo nombre para un identificador en un bloque interior, que se utiliza para un identificador en un bloque exterior, cuando de hecho deseamos que el identificador en el bloque exterior esté activo durante la ejecución del bloque interior, es comúnmente un error lógico.

Tip para prevenir errores 6.6

Evite los nombres de variables que ocultan nombres en alcances exteriores.

Alcance de función

Las **etiquetas** (identificadores seguidos por dos puntos, como `inicio:`, o una etiqueta `case` en una instrucción `switch`) son los únicos identificadores con *alcance de función*. Las etiquetas se pueden utilizar en *cualquier parte* en la función en la que aparecen, pero no se pueden referenciar *fuera* del cuerpo de la función.

Alcance de espacio de nombres global

Un identificador que se declara *fuera* de cualquier función o clase tiene *alcance de espacio de nombres global*. Dicho identificador se "conoce" en todas las funciones a partir del punto en el que se declara, hasta llegar al final del archivo. Las variables globales, las definiciones de funciones y los prototipos de funciones que se colocan fuera de una función tienen alcance de archivo.

Alcance de prototipo de función

Los únicos identificadores con *alcance de prototipo de función* son los que se utilizan en la lista de parámetros de un prototipo de función. Como se mencionó antes, los prototipos de función *no* requieren nombres en la lista de parámetros; sólo los tipos. El compilador *ignora* los nombres que aparecen en la lista de parámetros de un prototipo de función. Los identificadores que se utilizan en un prototipo de función se pueden reutilizar en cualquier otra parte del programa sin ambigüedad.

Demostración del alcance

El programa de la figura 6.13 demuestra cuestiones sobre el alcance con las variables globales, variables locales automáticas y variables locales `static`. En la línea 10 se declara e inicializa la variable global `x` en 1. Esta variable local está oculta en cualquier bloque (o función) que declare una variable llamada `x`. En `main`, la línea 14 muestra el valor de la variable global `x`. En la línea 16 se declara una variable local `x` y se inicializa en 5. En la línea 18 se imprime esta variable para mostrar que la `x` global está oculta en `main`. A continuación, en las líneas 20 a 24 se define un nuevo bloque en `main`, en el cual otra variable local `x` se inicializa con 7 (línea 21). En la línea 23 se imprime esta variable, para mostrar que *oculta* a `x` en el bloque exterior de `main`, así como la `x` global. Cuando el bloque termina, la variable `x` que tiene el valor 7 se destruye automáticamente. A continuación, en la línea 26 se imprime la variable local `x` en el bloque exterior de `main`, para mostrar que *ya no está oculta.*

```
1  // Fig. 6.13: fig06_13.cpp
2  // Ejemplo sobre el alcance.
3  #include <iostream>
4  using namespace std;
5
6  void usarLocal(); // prototipo de función
7  void usarLocalStatic(); // prototipo de función
8  void usarGlobal(); // prototipo de función
9
10 int x = 1; // variable global
11
12 int main()
13 {
14    cout << "la x global en main es " << x << endl;
15
16    int x = 5; // variable local para main
17
18    cout << "la x local en el alcance exterior de main es " << x << endl;
19
20    { // empieza nuevo alcance
21       int x = 7; // oculta la x en el alcance exterior y la x global
22
23       cout << "la x local en el alcance interior de main es " << x << endl;
24    } // termina nuevo alcance
```

Fig. 6.13 | Ejemplo sobre el alcance (parte 1 de 3).

```
25
26    cout << "la x local en el alcance exterior de main es " << x << endl;
27
28    usarLocal(); // usarLocal tiene la x local
29    usarLocalStatic(); // usarLocalStatic tiene la x local estática
30    usarGlobal(); // usarGlobal usa la x global
31    usarLocal(); // usarLocal reinicializa su x local
32    usarLocalStatic(); // la x local estática retiene su valor anterior
33    usarGlobal(); // la x global también retiene su valor anterior
34
35    cout << "\nla x local en main es " << x << endl;
36 } // fin de main
37
38 // usarLocal reinicializa la variable x local durante cada llamada
39 void usarLocal()
40 {
41    int x = 25; // se inicializa cada vez que se llama a usarLocal
42
43    cout << "\nla x local es " << x << " al entrar a usarLocal" << endl;
44    ++x;
45    cout << "la x local es " << x << " al salir de usarLocal" << endl;
46 } // fin de la función usarLocal
47
48 // usarLocalStatic inicializa la variable x local estática sólo la
49 // primera vez que se llama a la función; el valor de x se guarda
50 // entre las llamadas a esta función
51 void usarLocalStatic()
52 {
53    static int x = 50; // se inicializa la primera vez que se llama a
                         usarLocalStatic
54
55    cout << "\nla x local estatica es " << x << " al entrar a usarLocalStatic"
56       << endl;
57    ++x;
58    cout << "la x local estatica es " << x << " al salir de usarLocalStatic"
59       << endl;
60 } // fin de la función usarLocalStatic
61
62 // usarGlobal modifica la variable global x durante cada llamada
63 void usarGlobal()
64 {
65    cout << "\nla x global es " << x << " al entrar a usarGlobal" << endl;
66    x *= 10;
67    cout << "la x global es " << x << " al salir de usarGlobal" << endl;
68 } // fin de la función usarGlobal
```

```
la x global en main es 1
la x local en el alcance exterior de main es 5
la x local en el alcance interior de main es 7
la x local en el alcance exterior de main es 5

la x local es 25 al entrar a usarLocal
la x local es 26 al salir de usarLocal
```

Fig. 6.13 | Ejemplo sobre el alcance (parte 2 de 3).

```
la x local estatica es 50 al entrar a usarLocalStatic
la x local estatica es 51 al salir de usarLocalStatic

la x global es 1 al entrar a usarGlobal
la x global es 10 al salir de usarGlobal

la x local es 25 al entrar a usarLocal
la x local es 26 al salir de usarLocal

la x local estatica es 51 al entrar a usarLocalStatic
la x local estatica es 52 al salir de usarLocalStatic

la x global es 10 al entrar a usarGlobal
la x global es 100 al salir de usarGlobal

la x local en main es 5
```

Fig. 6.13 | Ejemplo sobre el alcance (parte 3 de 3).

Para demostrar otros alcances, el programa define tres funciones, cada una de las cuales no recibe argumento y no devuelve nada. La función `usarLocal` (líneas 39 a 46) declara la variable automática `x` (línea 41) y la inicializa en 25. Cuando el programa llama a `usarLocal`, la función imprime la variable, la incrementa y la vuelve a imprimir antes de que la función devuelva el control del programa a la función que la llamó. Cada vez que el programa llama a esta función, ésta *vuelve a crear* la variable automática `x` y la *vuelve a inicializar* en 25.

La función `usarLocalStatic` (líneas 51 a 60) declara la variable `static x` y la inicializa con 50. Las variables locales que se declaran como `static` retienen sus valores aun cuando estén fuera de alcance (es decir, la función en la que se declaran no se está ejecutando). Cuando el programa llama a `usarLocalStatic`, la función imprime `x`, la incrementa y la vuelve a imprimir antes de que la función devuelva el control del programa a la función que la llamó. En la siguiente llamada a esta función, la variable local `static x` contiene el valor 51. La *inicialización* en la línea 53 *ocurre sólo una vez*: la primera vez que se llama a `usarLocalStatic`.

La función `usarGlobal` (líneas 63 a 68) no declara ninguna variable. Por lo tanto, cuando hace referencia a la variable `x`, se utiliza la `x` global (línea 10, antes de `main`). Cuando el programa llama a `usarGlobal`, la función imprime la variable global `x`, la multiplica por 10 y la imprime de nuevo, antes de que la función devuelva el control del programa a la función que la llamó. La siguiente vez que el programa llama a `usarGlobal`, se modifica el valor de la variable global, 10. Después de ejecutar las funciones `usarLocal`, `usarLocalStatic` y `usarGlobal` dos veces cada una, el programa imprime la variable local `x` en `main`, de nuevo para mostrar que ninguna de las llamadas a la función modificó el valor de `x` en `main`, debido a que todas las funciones hicieron referencia a las variables en otros alcances.

6.12 La pila de llamadas a funciones y los registros de activación

Para comprender la forma en que C++ realiza las llamadas a las funciones, primero necesitamos considerar una estructura de datos (es decir, colección de elementos de datos relacionados) conocida como **pila**. Piense en una pila como la analogía a una pila de platos. Cuando se coloca un plato en la pila, por lo general se coloca en la parte *superior*: lo que se conoce como **meter** el plato en la pila. De manera similar, cuando se extrae un plato de la pila, siempre se extrae de la parte superior: lo que se conoce como **sacar** el plato de la pila. Las pilas se denominan **estructuras de datos "último en entrar, primero en salir" (UEPS)**; el último elemento que se mete (inserta) en la pila es el primero que se saca (extrae) de ella.

La pila de llamadas a funciones

Uno de los mecanismos más importantes que los estudiantes de ciencias computacionales deben comprender es la **pila de llamadas a funciones** (conocida algunas veces como la **pila de ejecución del programa**). Esta estructura de datos (que trabaja en segundo plano) soporta el mecanismo de llamada a/regreso de las funciones. También soporta la creación, mantenimiento y destrucción de las variables automáticas de cada función a la que se llama. Como veremos en las figuras 6.15 a 6.17, este comportamiento "último en entrar, primero en salir" (UEPS) es *exactamente* lo que hace una función cuando regresa a la función que la llamó.

Marcos de pila

A medida que se hace la llamada a cada función, ésta puede a su vez, llamar a otras funciones, las cuales, a su vez, pueden llamar a otras funciones; todo ello *antes* de que regrese alguna de las funciones. En cierto momento, cada función debe regresar el control a la función que la llamó. Por ende, de alguna manera debemos llevar el registro de las *direcciones de retorno* que requiere cada función para regresar el control a la función que la llamó. La pila de llamadas a funciones es la estructura de datos perfecta para manejar esta información. Cada vez que una función llama a otra función, se *mete* una entrada en la pila. Esta entrada, conocida como **marco de pila** o **registro de activación**, contiene la *dirección de retorno* que necesita la función a la que se llamó para poder regresar a la función que hizo la llamada. También contiene cierta información adicional que veremos en breve. Si la función a la que se llamó regresa, en vez de llamar a otra función antes de regresar, se *saca* el marco de pila para la llamada a la función, y el control se transfiere a la dirección de retorno en el marco de la pila que se sacó.

La belleza de la pila de llamadas es que cada función a la que se ha llamado siempre encuentra la información que requiere para regresar a la función que la llamó en la parte *superior* de la pila de llamadas. Y, si una función hace una llamada a otra función, simplemente se *mete* a la pila de llamadas un marco de pila para esa nueva llamada a una función. Por ende, la dirección de retorno requerida por la función recién llamada para regresar a la función que la llamó se encuentra ahora en la parte *superior* de la pila.

Variables automáticas y marcos de pila

Los marcos de pila tienen otra responsabilidad importante. La mayoría de las funciones tienen variables automáticas: parámetros y cualquier variable local que declare la función. Las variables automáticas necesitan existir mientras una función se está ejecutando. Necesitan permanecer activas si la función hace llamadas a otras funciones. Pero cuando una función a la que se llamó regresa a la función que la llamó, las variables automáticas de la función a la que se llamó necesitan "desaparecer". El marco de pila de la función a la que se llamó es un lugar perfecto para reservar la memoria para las variables automáticas de la función a la que se llamó. Ese marco de pila existe mientras la función a la que se llamó esté activa. Cuando esa función regresa (y ya no necesita sus variables automáticas locales) su marco de pila se *saca* de la pila, y esas variables automáticas ya no son conocidas para el programa.

Desbordamiento de pila

Desde luego que la cantidad de memoria en una computadora es finita, por lo cual sólo se puede usar cierta cantidad de memoria para almacenar registros de activación en la pila de llamadas a funciones. Si ocurren más llamadas a funciones de las que se puedan guardar sus registros de activación en la pila de llamadas a funciones, se produce un error conocido como **desbordamiento de pila**.

La pila de llamadas a funciones en acción

Ahora vamos a considerar cómo la pila de llamadas ofrece soporte para la operación de una función `cuadrado` llamada por `main` (líneas 9 a 14 de la figura 6.14). Primero, el sistema operativo llama a `main`; esto hace que se meta un registro de activación en la pila (lo cual se muestra en la figura 6.15). El registro de activación indica a `main` cómo debe regresar al sistema operativo (es decir, transferir el control a la dirección de retorno `R1`) y contiene el espacio para la variable automática de `main` (`a`, que se inicializa en `10`).

```
1  // Fig. 6.14: fig06_14.cpp
2  // Función cuadrado utilizada para demostrar la pila
3  // de llamadas a funciones y los registros de activación.
4  #include <iostream>
5  using namespace std;
6
7  int cuadrado( int ); // prototipo para la función cuadrado
8
9  int main()
10 {
11    int a = 10; // valor para cuadrado (variable local automática en main)
12
13    cout << a << " al cuadrado: " << cuadrado( a ) << endl; // muestra a al cuadrado
14 } // fin de main
15
16 // devuelve el cuadrado de un entero
17 int cuadrado( int x ) // x es una variable local
18 {
19    return x * x; // calcula el cuadrado y devuelve el resultado
20 } // fin de la función cuadrado
```

```
10 cuadrado: 100
```

Fig. 6.14 | Función `cuadrado` utilizada para demostrar la pila de llamadas a funciones y los registros de activación.

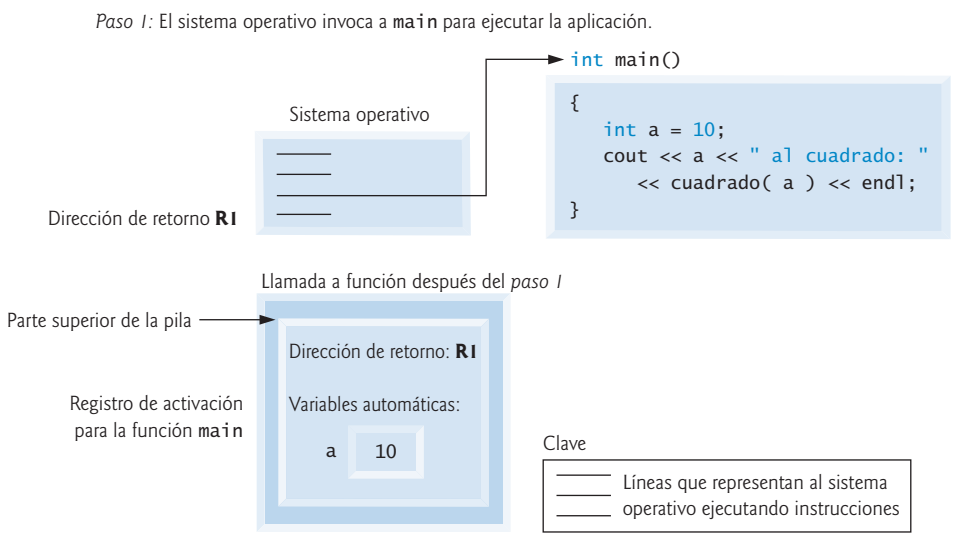


Fig. 6.15 | La pila de llamadas a funciones, después de que el sistema operativo invoca a `main` para ejecutar el programa.

La función `main` (antes de regresar al sistema operativo) llama ahora a la función `cuadrado` en la línea 13 de la figura 6.14. Esto hace que se meta un marco de pila para `cuadrado` (líneas 17 a 20) en la pila de llamadas a funciones (figura 6.16). Este marco de pila contiene la dirección de retorno que `cuadrado` necesita para regresar a `main` (es decir, `R2`) y la memoria para la variable automática de `cuadrado` (es decir, `x`).

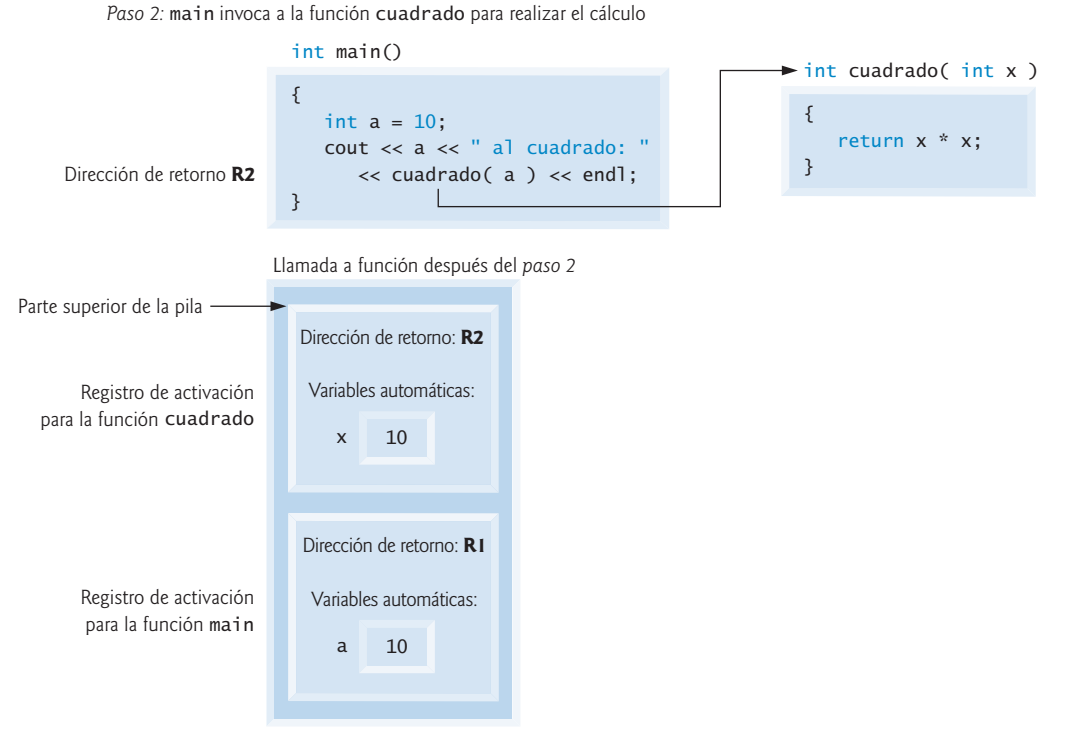


Fig. 6.16 | La pila de llamadas a funciones, después de que la función `cuadrado` regresa a `main`.

Una vez que `cuadrado` calcula el cuadrado de su argumento, necesita regresar a `main`; y ya no necesita la memoria para su variable automática, `x`. Por ende, el marco de pila de `cuadrado` se *saca* de la pila, con lo cual se proporciona a `cuadrado` la dirección de retorno en `main` (es decir, `R2`) y se pierde la variable automática de `cuadrado`. La figura 6.17 muestra la pila de llamadas a funciones *después* de sacar el registro de activación de `cuadrado`.

Ahora la función `main` muestra el resultado de llamar a `cuadrado` (figura 6.14, línea 13). Al llegar a la llave derecha de cierre de `main`, su marco de pila se *saca* de la pila, proporciona a `main` la dirección que requiere para regresar al sistema operativo (es decir, `R1` en la figura 6.15) y, en este punto, la variable automática de `main` (es decir, `a`) ya no existe.

Ahora hemos visto qué tan valiosa es la estructura de datos tipo pila para implementar un mecanismo clave que ofrezca soporte para la ejecución de un programa. Las estructuras de datos tienen muchas aplicaciones importantes en las ciencias computacionales. En el capítulo 15, Contenedores e iteradores de la Biblioteca estándar y en el capítulo 19, Custom Templatized Data Structures (en inglés, en el sitio web), hablaremos sobre las pilas, colas, listas, árboles y otras estructuras de datos.

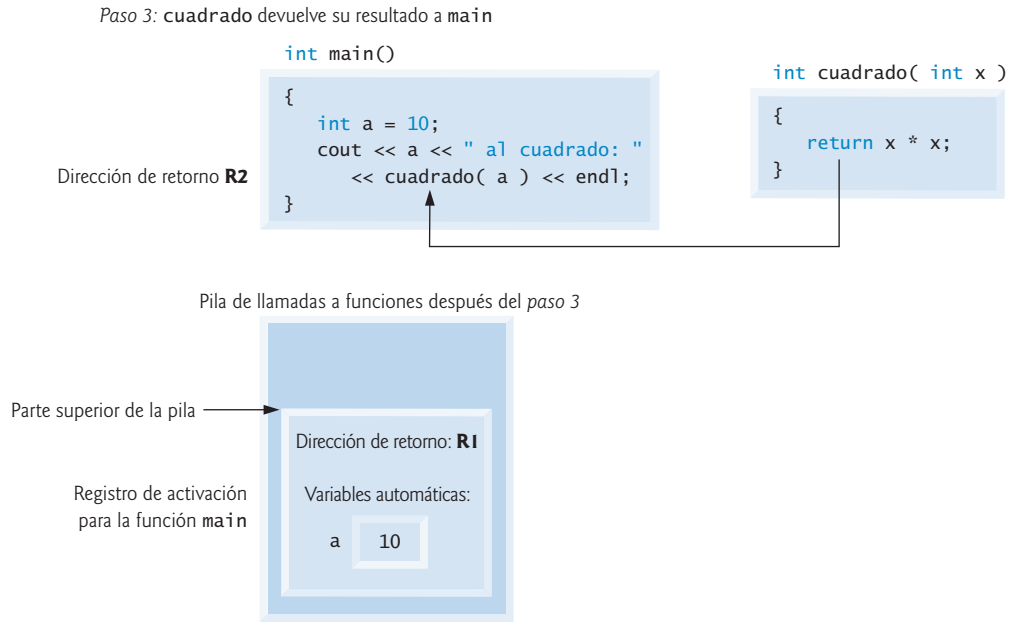


Fig. 6.17 | La pila de llamadas a funciones, después de que la función `cuadrado` regresa a `main`.

6.13 Funciones con listas de parámetros vacías

En C++, una *lista de parámetros vacía* se especifica escribiendo `void` o nada entre paréntesis. El prototipo

```
void imprimir();
```

especifica que la función `imprimir` *no* recibe argumentos y *no* devuelve un valor. La figura 6.18 demuestra ambas formas de declarar y usar funciones con listas de parámetros vacías.

```
1  // Fig. 6.18: fig06_18.cpp
2  // Funciones que no reciben argumentos.
3  #include <iostream>
4  using namespace std;
5
6  void funcion1(); // función que no recibe argumentos
7  void funcion2( void ); // función que no recibe argumentos
8
9  int main()
10 {
11    funcion1(); // llama a funcion1 sin argumentos
12    funcion2(); // llama a funcion2 sin argumentos
13 } // fin de main
14
```

Fig. 6.18 | Funciones que no reciben argumentos (parte 1 de 2).

```
15 // funcion1 usa una lista de parámetros vacía para especificar que
16 // la función no recibe argumentos
17 void funcion1()
18 {
19    cout << "funcion1 no recibe argumentos" << endl;
20 } // fin de funcion1
21
22 // funcion2 usa una lista de parámetros vacía para especificar que
23 // la función no recibe argumentos
24 void funcion2( void )
25 {
26    cout << "funcion2 tampoco recibe argumentos" << endl;
27 } // fin de funcion2
```

```
funcion1 no recibe argumentos
funcion2 tampoco recibe argumentos
```

Fig. 6.18 | Funciones que no reciben argumentos (parte 2 de 2).

6.14 Funciones en línea

Es bueno implementar un programa como un conjunto de funciones desde el punto de vista de la ingeniería de software, pero las llamadas a funciones implican una sobrecarga en tiempo de ejecución. C++ cuenta con las **funciones en línea** para ayudar a reducir la sobrecarga de las llamadas a funciones. Al colocar el calificador `inline` antes del tipo de valor de retorno de la función en su definición, se *aconseja* al compilador para que genere una copia del código del cuerpo de la función en *cada* lugar en donde se llame a la función (cuando sea apropiado), para evitar la llamada a una función. Con frecuencia, esto hace que el tamaño del programa *aumente*. El compilador puede *ignorar* el calificador `inline`, y por lo general lo hace para todas las funciones, excepto las más *pequeñas*. Por lo general las funciones `inline` reutilizables se colocan en encabezados, de modo que puedan incluirse sus definiciones en cada archivo de código fuente que las utilice.

Observación de Ingeniería de Software 6.9

Si cambia la definición de una función `inline`, deberá volver a compilar todos los clientes de esa función.

Tip de rendimiento 6.4

Los compiladores pueden poner código en línea para el que no hayamos utilizado de manera explícita la palabra clave `inline`. Los compiladores optimizadores actuales son tan sofisticados que es mejor dejarles las decisiones sobre poner o no código en línea.

La figura 6.19 utiliza la función `inline` llamada `cubo` (líneas 9 a 12) para calcular el volumen de un cubo. La palabra clave `const` en la lista de parámetros de la función `cubo` (línea 9) indica al compilador que la función *no* modifica la variable `lado`. Esto asegura que la función *no* modifique el valor de `lado` cuando se realice el cálculo (la palabra clave `const` se describe con detalle en los capítulos 7 a 9).

Observación de Ingeniería de Software 6.10

El calificador `const` se debe utilizar para hacer valer el principio del menor privilegio. El uso de este principio para diseñar software de manera apropiada puede reducir considerablemente el tiempo de depuración y los efectos secundarios inadecuados; además puede facilitar la modificación y el mantenimiento de un programa.

```
1  // Fig. 6.19: fig06_19.cpp
2  // Función inline que calcula el volumen de un cubo.
3  #include <iostream>
4  using namespace std;
5
6  // Definición de la función en línea cubo. La definición de la función aparece
7  // antes de llamar a la función, por lo que no se requiere un prototipo de función.
8  // La primera línea de la función actúa como el prototipo.
9  inline double cubo( const double lado )
10 {
11    return lado * lado * lado; // calcula el cubo
12 } // fin de la función cubo
13
14 int main()
15 {
16    double valorLado; // almacena el valor introducido por el usuario
17    cout << "Escriba la longitud del lado de su cubo: ";
18    cin >> valorLado; // lee el valor del usuario
19
20    // calcula el cubo de valorLado y muestra el resultado
21    cout << "El volumen del cubo con un lado de "
22       << valorLado << " es " << cubo( valorLado ) << endl;
23 } // end main
```

```
Escriba la longitud del lado de su cubo: 3.5
El volumen del cubo con un lado de 3.5 es 42.875
```

Fig. 6.19 | Función `inline` que calcula el volumen de un cubo.

6.15 Referencias y parámetros de referencias

Dos formas de pasar argumentos a las funciones en muchos lenguajes de programación son el **paso por valor** y el **paso por referencia**. Cuando se pasa un argumento por valor, se crea una *copia* del valor del argumento y se pasa (en la pila de llamadas a funciones) a la función que se llamó. Las modificaciones a la copia *no* afectan al valor de la variable original en la función que hizo la llamada. Esto evita los *efectos secundarios accidentales* que tanto obstaculizan el desarrollo de sistemas de software correctos y confiables. Hasta ahora, cada argumento en el libro se ha pasado por valor.

Tip de rendimiento 6.5

Una desventaja del paso por valor es que, si se va a pasar un elemento de datos extenso, el proceso de copiar esos datos puede requerir una cantidad considerable de tiempo de ejecución y espacio en memoria.

Parámetros por referencia

En esta sección presentamos los parámetros por referencia: el primero de los dos medios que proporciona C++ para realizar el paso por referencia. Mediante el *paso por referencia*, la función que hace la llamada proporciona a la función que llamó la habilidad de *acceder directamente a los datos de la primera*, y de *modificarlos*.

Tip de rendimiento 6.6

El paso por referencia es bueno por cuestiones de rendimiento, ya que puede eliminar la sobrecarga de copiar grandes cantidades de datos en el paso por valor.

Observación de Ingeniería de Software 6.11

El paso por referencia puede debilitar la seguridad, ya que la función a la que se llamó puede corromper los datos de la función que hizo la llamada.

Más adelante veremos cómo lograr la ventaja de rendimiento que ofrece el paso por referencia, *al tiempo* que se obtiene la ventaja de ingeniería de software de evitar que la corrupción de los datos de la función que hizo la llamada.

Un **parámetro por referencia** es un *alias* para su correspondiente argumento en la llamada a una función. Para indicar que un parámetro de función se pasa por referencia, simplemente hay que colocar un signo `&` después del tipo del parámetro en el prototipo de la función; use la misma convención al listar el tipo del parámetro en el encabezado de la función. Por ejemplo, la siguiente declaración en el encabezado de una función:

```
int &cuenta
```

si se lee de *izquierda* a *derecha*, significa que "`cuenta` es una referencia a un valor `int`". En la llamada a la función, simplemente hay que mencionar la variable por su nombre para pasarla por referencia. Después, al mencionar la variable por el nombre de su parámetro en el cuerpo de la función a la que se llamó, en realidad se refiere a la *variable original* en la función que hizo la llamada, y esta variable original se puede *modificar* directamente en la función a la que se llamó. Como siempre, el prototipo de función y el encabezado deben concordar.

Paso de argumentos por valor y por referencia

En la figura 6.20 se compara el paso por valor y el paso por referencia con los parámetros por referencia. Los "estilos" de los argumentos en las llamadas a la función `cuadradoPorValor` y la función `cuadradoPorReferencia` son idénticos; ambas variables sólo se mencionan por nombre en las llamadas a las funciones. *Sin comprobar los prototipos o las definiciones de las funciones, no es posible deducir sólo de las llamadas si cada función puede modificar sus argumentos*. Como los prototipos de función son obligatorios, el compilador no tiene problemas para resolver la ambigüedad.

Error común de programación 6.8

Como los parámetros por referencia se mencionan sólo por su nombre en el cuerpo de la función a la que se llama, podríamos tratar de manera inadvertida los parámetros por referencia como parámetros de paso por valor. Esto puede provocar efectos secundarios inesperados si la función modifica las copias originales de las variables.

```
1  // Fig. 6.20: fig06_20.cpp
2  // Paso de argumentos por valor y por referencia.
3  #include <iostream>
4  using namespace std;
5
6  int cuadradoPorValor( int ); // prototipo de función (paso por valor)
7  void cuadradoPorReferencia( int & ); // prototipo de función (paso por referencia)
8
9  int main()
10 {
11    int x = 2; // valor para cuadrado usando cuadradoPorValor
12    int z = 4; // valor para cuadrado usando cuadradoPorReferencia
13
```

Fig. 6.20 | Paso de argumentos por valor y por referencia (parte 1 de 2).

```
14    // demuestra cuadradoPorValor
15    cout << "x = " << x << " antes de cuadradoPorValor\n";
16    cout << "Valor devuelto por cuadradoPorValor: "
17       << cuadradoPorValor( x ) << endl;
18    cout << "x = " << x << " despues de cuadradoPorValor\n" << endl;
19
20    // demuestra cuadradoPorReferencia
21    cout << "z = " << z << " antes de cuadradoPorReferencia" << endl;
22    cuadradoPorReferencia( z );
23    cout << "z = " << z << " despues de cuadradoPorReferencia" << endl;
24 } // fin de main
25
26 // cuadradoPorValor multiplica el número por sí mismo, almacena el
27 // resultado en el número y devuelve el nuevo valor del número
28 int cuadradoPorValor( int numero )
29 {
30    return numero *= numero; // no se modificó el argumento de la función que hizo
                                  la llamada
31 } // fin de la función cuadradoPorValor
32
33 // cuadradoPorReferencia multiplica a refNumero por sí solo y almacena el
      resultado
34 // en la variable a la que refNumero hace referencia en la función main
35 void cuadradoPorReferencia( int &refNumero )
36 {
37    refNumero *= refNumero; // se modificó el argumento de la función que hizo la
                                 llamada
38 } // fin de la función cuadradoPorReferencia
```

```
x = 2 antes de cuadradoPorValor
Valor devuelto por cuadradoPorValor: 4
x = 2 despues de cuadradoPorValor

z = 4 antes de cuadradoPorReferencia
z = 16 despues de cuadradoPorReferencia
```

Fig. 6.20 | Paso de argumentos por valor y por referencia (parte 2 de 2).

El capítulo 8 habla sobre los *apuntadores*; éstos proporcionan una forma alternativa del paso por referencia, en la cual el estilo de la llamada indica claramente el paso por referencia (y el potencial de modificar los argumentos de la función que hace la llamada).

Tip de rendimiento 6.7

Para pasar objetos extensos, use un parámetro por referencia constante *para simular la apariencia y seguridad del paso por valor, evitando así la sobrecarga de pasar una copia del objeto extenso.*

Para especificar que una referencia no debe modificar el argumento, coloque el calificador `const` *antes* del especificador de tipo en la declaración del parámetro. Observe la colocación del signo `&` en la lista de parámetros de la función `cuadradoPorReferencia` (línea 35, figura 6.20). Algunos programadores de C++ prefieren escribir la forma equivalente `int& refNumero`.

Referencias como alias dentro de una función

Las referencias también se pueden usar como alias para otras variables *dentro* de una función (aunque por lo general se utilizan con las funciones, como se muestra en la figura 6.20). Por ejemplo, el código

```
int cuenta = 1; // declara la variable entera cuenta
int &cRef = cuenta; // crea cRef como alias para cuenta
++cRef; // incrementa cuenta (usando su alias cRef)
```

incrementa la variable `cuenta` mediante el uso de su alias `cRef`. Las variables de referencia *deben* inicializarse en sus declaraciones y no se pueden reasignar como *alias* para otras variables. Una vez que se *declara* una referencia como alias para otra variable, todas las operaciones que supuestamente se realizan en el *alias* (es decir, la referencia) en realidad se realizan en la variable *original*. El alias es simplemente otro nombre para la variable original. A menos que sea una referencia a una constante, un argumento por referencia debe ser un *lvalue* (por ejemplo, el nombre de una variable), no una constante o expresión que devuelva un *rvalue* (por ejemplo, el resultado de un cálculo).

Devolver una referencia de una función

Las funciones pueden devolver referencias, pero esto puede ser peligroso. Al devolver una referencia a una variable declarada en la función que se llamó, a menos que esa variable se declare como `static`, la referencia se refiere a una variable automática que se *descarta* cuando termina la función. El intento de acceder a dicha variable produce un *comportamiento indefinido*. Las referencias a variables indefinidas se llaman **referencias sueltas**.

Error común de programación 6.9

Devolver una referencia a una variable automática en una función a la que se ha llamado es un error lógico. Por lo general, los compiladores generan una advertencia cuando esto ocurre. Para el código de nivel industrial, siempre hay que *eliminar todas las advertencias de compilación para poder producir código ejecutable.*

6.16 Argumentos predeterminados

Para un programa, es algo común el invocar una función repetidas veces con el *mismo* valor de argumento para un parámetro específico. En tales casos, podemos especificar que dicho parámetro tiene un **argumento predeterminado**; es decir, que tiene un valor predeterminado que debe pasar a ese parámetro. Cuando un programa *omite* un argumento para un parámetro con un argumento predeterminado en la llamada a una función, el compilador vuelve a escribir la llamada a la función e inserta el valor predeterminado de ese argumento.

Los argumentos predeterminados *deben* ser los argumentos de más a la derecha en la lista de parámetros de una función. Al llamar a una función con dos o más argumentos predeterminados, si se omite un argumento que *no* sea el de más a la derecha en la lista de argumentos, entonces también *deben* omitirse todos los argumentos que estén a la derecha de ese argumento. Los argumentos predeterminados deben especificarse con la *primera* ocurrencia del nombre de la función; por lo general, en el prototipo de la función. Si el prototipo se omite debido a que la definición de la función también actúa como el prototipo, entonces los argumentos predeterminados se deben especificar en el encabezado de la función. Los valores predeterminados pueden ser cualquier expresión, incluyendo constantes, variables globales o llamadas a funciones. Los argumentos predeterminados también se pueden usar con funciones `inline`.

En la figura 6.21 se demuestra el uso de argumentos predeterminados para calcular el volumen de una caja. El prototipo de función para `volumenCaja` (línea 7) especifica que los tres parámetros reciben valores predeterminados de `1`. Proporcionamos nombres de variables en el prototipo de función para mejorar la legibilidad. Como siempre, los nombres de las variables no se requieren en los prototipos de función.

La primera llamada a `volumenCaja` (línea 13) especifica que no hay argumentos, con lo cual se utilizan los tres valores predeterminados de 1. En la segunda llamada (línea 17) sólo se pasa un argumento `longitud`, con lo cual se utilizan los valores predeterminados de 1 para los argumentos `anchura` y

```
1  // Fig. 6.21: fig06_21.cpp
2  // Uso de argumentos predeterminados.
3  #include <iostream>
4  using namespace std;
5
6  // prototipo de función que especifica argumentos predeterminados
7  unsigned int volumenCaja( unsigned int longitud = 1, unsigned int anchura = 1,
8     unsigned int altura = 1 );
9
10 int main()
11 {
12    // sin argumentos--usa valores predeterminados para todas las medidas
13    cout << "El volumen predeterminado de la caja es: " << volumenCaja();
14
15    // especifica la longitud; anchura y altura predeterminadas
16    cout << "\n\nEl volumen de una caja con longitud 10,\n"
17       << "anchura 1 y altura 1 es: " << volumenCaja( 10 );
18
19    // especifica longitud y anchura; altura predeterminada
20    cout << "\n\nEl volumen de una caja con longitud 10,\n"
21       << "anchura 5 y altura 1 es: " << volumenCaja( 10, 5 );
22
23    // especifica todos los argumentos
24    cout << "\n\nEl volumen de una caja con longitud 10,\n"
25       << "anchura 5 y altura 2 es: " << volumenCaja( 10, 5, 2 )
26       << endl;
27 } // fin de main
28
29 // la función volumenCaja calcula el volumen de una caja
30 unsigned int volumenCaja( unsigned int longitud, unsigned int anchura,
31    unsigned int altura )
32 {
33    return longitud * anchura * altura;
34 } // fin de la función volumenCaja
```

```
El volumen predeterminado de la caja es: 1

El volumen de una caja con longitud 10,
anchura 1 y altura 1 es: 10

El volumen de una caja con longitud 10,
anchura 5 y altura 1 es: 50

El volumen de una caja con longitud 10,
anchura 5 y altura 2 es: 100
```

Fig. 6.21 | Uso de argumentos predeterminados.

`altura`. La tercera llamada (línea 21) pasa argumentos sólo para `longitud` y `anchura`, con lo cual se utiliza un valor predeterminado de `1` para el argumento `altura`. La última llamada (línea 25) pasa argumentos para `longitud`, `anchura` y `altura`, con lo cual no utiliza valores predeterminados. Cualquier argumento que se pasa a la función de manera explícita se asigna a los parámetros de la función, de izquierda a derecha. Por lo tanto, cuando `volumenCaja` recibe un argumento, la función asigna el valor de ese argumento a su parámetro `longitud` (es decir, el parámetro de más a la izquierda en la lista de parámetros). Cuando `volumenCaja` recibe dos argumentos, la función asigna los valores de esos argumentos

a sus parámetros `longitud` y `anchura` en ese orden. Por último, cuando `volumenCaja` recibe los tres argumentos, la función asigna los valores de esos argumentos a sus parámetros `longitud`, `anchura` y `altura`, respectivamente.

Buena práctica de programación 6.5

El uso de argumentos predeterminados puede simplificar la escritura de las llamadas a funciones. Sin embargo, algunos programadores sienten que es más claro especificar de manera explícita todos los argumentos.

6.17 Operador de resolución de ámbito unario

Es posible declarar variables locales y globales con el mismo nombre. C++ proporciona el **operador de resolución de ámbito binario** (`::`) para acceder a una variable global cuando una variable local con el mismo nombre se encuentra dentro del alcance. El operador de resolución de ámbito unario no se puede utilizar para acceder a una variable *local* con el mismo nombre en un bloque exterior. Se puede acceder a una variable global directamente sin el operador de resolución de ámbito unario, si el nombre de la variable global no es el mismo que el de una variable local dentro del alcance.

En la figura 6.22 se demuestra el operador de resolución de ámbito unario con variables local y global con el mismo nombre (líneas 6 y 10). Para enfatizar que las versiones local y global de la variable `numero` son distintas, el programa declara una variable de tipo `int` y la otra de tipo `double`.

```
1  // Fig. 6.22: fig06_22.cpp
2  // Operador de resolución de ámbito unario.
3  #include <iostream>
4  using namespace std;
5
6  int numero = 7; // variable global llamada numero
7
8  int main()
9  {
10    double numero = 10.5; // variable local llamada numero
11
12    // muestra los valores de las variables local y global
13    cout << "Valor local double de numero = " << numero
14       << "\nValor global int de numero = " << ::numero << endl;
15 } // fin de main
```

```
Valor local double de numero = 10.5
Valor global int de numero = 7
```

Fig. 6.22 | Operador de resolución de ámbito unario.

Buena práctica de programación 6.6

Si utiliza siempre el operador de resolución de ámbito unario (::) para hacer referencia a las variables globales, establece claramente que está tratando de acceder a una variable global, en vez de una variable no global.

Observación de Ingeniería de Software 6.12

Al utilizar siempre el operador de resolución de ámbito unario (::) para hacer referencia a las variables globales, se facilita la modificación de los programas, al reducir el riesgo de conflictos de nombres con variables no globales.

Tip para prevenir errores 6.7

Al utilizar siempre el operador de resolución de ámbito unario (::) para hacer referencia a una variable global, se eliminan los posibles errores lógicos que podrían ocurrir si una variable no global oculta a la variable global.

Tip para prevenir errores 6.8

Evite usar variables con el mismo nombre para distintos propósitos en un programa. Aunque esto se permite en diversas circunstancias, puede producir errores.

6.18 Sobrecarga de funciones

C++ permite definir varias funciones con el mismo nombre, siempre y cuando éstas tengan distintas firmas. A esta capacidad se le conoce como **sobrecarga de funciones**. El compilador de C++ selecciona la función apropiada al examinar el número, tipos y orden de los argumentos en la llamada. La sobrecarga de funciones se utiliza para crear varias funciones con el *mismo* nombre que realicen tareas similares, pero con *distintos* tipos de datos. Por ejemplo, muchas funciones en la biblioteca de matemáticas están sobrecargadas para distintos tipos de datos numéricos; el estándar de C++ requiere versiones sobrecargadas `float`, `double` y `long double` de las funciones matemáticas de la biblioteca que se describen en la sección 6.3.

Buena práctica de programación 6.7

Al sobrecargar las funciones que realizan tareas estrechamente relacionadas, los programas pueden ser más fáciles de leer y comprender.

Funciones **cuadrado** *sobrecargadas*

La figura 6.23 utiliza funciones `cuadrado` sobrecargadas para calcular el cuadrado de un valor `int` (líneas 7a 11) y el cuadrado de un valor `double` (líneas 14 a 18). En la línea 22 se invoca a la versión `int` de la función `cuadrado`, para lo cual se le pasa el valor literal `7`. C++ trata a los valores literales numéricos enteros como de tipo `int`. De manera similar, en la línea 24 se invoca a la versión `double` de la función `cuadrado`, para lo cual se le pasa el valor literal `7.5`, que C++ trata como valor `double`. En cada caso, el compilador elije la función apropiada que va a llamar, con base en el tipo del argumento. Las últimas dos líneas en la ventana de salida confirman que se llamó a la función apropiada en cada caso.

```
1  // Fig. 6.23: fig06_23.cpp
2  // Funciones cuadrado sobrecargadas.
3  #include <iostream>
4  using namespace std;
5
6  // función cuadrado para valores int
7  int cuadrado( int x )
8  {
9     cout << "el cuadrado del entero " << x << " es "
10    return x * x;
11 } // fin de la función cuadrado con argumento int
12
```

Fig. 6.23 | Funciones `cuadrado` sobrecargadas (parte 1 de 2).

```
13  // función cuadrado para valores double
14  double cuadrado( double y )
15  {
16     cout << "el cuadrado del double " << y << " es ";
17     return y * y;
18  } // fin de la función cuadrado con argumento
19
20  int main()
21  {
22     cout << cuadrado( 7 ); // llama a la versión int
23     cout << endl;
24     cout << cuadrado( 7.5 ); // llama a la versión double
25     cout << endl;
26  } // fin de main
```

```
el cuadrado del entero 7 es 49
el cuadrado del double 7.5 es 56.25
```

Fig. 6.23 | Funciones `cuadrado` sobrecargadas (parte 2 de 2).

Cómo diferencia el compilador las funciones sobrecargadas

Las funciones sobrecargadas se diferencian mediante sus *firmas*. Una firma es una combinación del nombre de una función y los tipos de sus parámetros (en orden). El compilador codifica cada identificador de función con los tipos de sus parámetros (lo que algunas veces se conoce como **manipulación de nombres** o **decoración de nombres**) para permitir el **enlace seguro de tipos**. Este tipo de enlace asegura que se llame a la función sobrecargada apropiada, y que los tipos de los argumentos se conformen a los tipos de los parámetros.

La figura 6.24 se compiló con GNU C++. En vez de mostrar los resultados de la ejecución del programa (como se haría normalmente), mostramos los nombres manipulados de la función que GNU C++ produce en lenguaje ensamblador. Cada nombre manipulado (distinto de `main`) empieza con dos guiones bajos (__) seguidos de la letra `Z`, un número y el nombre de la función. El número después de `Z` especifica cuántos caracteres hay en el nombre de la función. Por ejemplo, la función `cuadrado` tiene 8 caracteres en su nombre, por lo que su nombre manipulado recibe el prefijo `__Z8`. Después, el nombre de la función va seguido de una codificación de su lista de parámetros. En la lista de parámetros para la función `nada2` (línea 25; vea la cuarta línea de salida), `c` representa a un valor `char`, `i` representa a un valor `int`, `Rf` representa a un valor `float &` (es decir, una referencia a un valor `float`) y `Rd` representa a un valor `double &` (es decir, una referencia a un valor `double`). En la lista de parámetros para la función `nada1`, `i` representa a un valor `int`, `f` representa a un valor `float`, `c` representa a un valor `char` y `Ri` representa a un valor `int &`. Las dos funciones `cuadrado` se diferencian en base a sus listas de parámetros; una especifica `d` para `double` y la otra especifica `i` para `int`. Los tipos de valores de retorno de las funciones *no* se especifican en los nombres manipulados. Las *funciones sobrecargadas pueden tener distintos tipos de valores de retorno, pero si es así, también deben tener distintas listas de parámetros*. De nuevo, *no se pueden* tener dos funciones con la *misma* firma y *distintos* tipos de valores de retorno. La manipulación de nombres de funciones es específica para cada compilador. Además, la función `main` no está *manipulada*, ya que *no puede* sobrecargarse.

Error común de programación 6.10

Crear funciones sobrecargadas con las listas de parámetros idénticos y distintos tipos de valores de retorno es un error de compilación.

```
1  // Fig. 6.24: fig06_24.cpp
2  // Manipulación de nombres para permitir la vinculación segura de tipos.
3
4  // función cuadrado para valores int
5  int cuadrado( int x )
6  {
7     return x * x;
8  } // fin de la función cuadrado
9
10 // función cuadrado para valores double
11 double cuadrado( double y )
12 {
13    return y * y;
14 } // fin de la función cuadrado
15
16 // función que obtiene argumentos de los tipos
17 // int, float, char e int &
18 void nada1( int a, float b, char c, int &d )
19 {
20    // cuerpo vacío de la función
21 } // fin de la función nada1
22
23 // función que recibe argumentos de los tipos
24 // char, int, float & y double &
25 int nada2( char a, int b, float &c, double &d )
26 {
27    return 0;
28 } // fin de la función nada2
29
30 int main()
31 {
32 } // fin de main
```

```
__Z8cuadradoi
__Z8cuadradod
__Z5nada1ifcRi
__Z5nada2ciRfRd
main
```

Fig. 6.24 | Manipulación de nombres para permitir la vinculación segura de tipos.

El compilador utiliza sólo las listas de parámetros para diferenciar entre las funciones sobrecargadas. Dichas funciones *no* necesitan tener el mismo número de parámetros. Hay que tener cuidado al sobrecargar funciones con parámetros predeterminados, ya que esto puede provocar ambigüedades.

Error común de programación 6.11

Una función en la que se omiten sus argumentos predeterminados se podría invocar de una manera idéntica a otra función sobrecargada; esto es un error de compilación. Por ejemplo, al tener en un programa tanto una función que no reciba argumentos de manera explícita, como una función con el mismo nombre que contenga todos los argumentos predeterminados, se produce un error de compilación cuando tratamos de usar el nombre de esa función en una llamada en la que no se pasen argumentos. El compilador no sabe cuál versión de la función elegir.

Operadores sobrecargados
En el capítulo 10 hablaremos acerca de cómo sobrecargar *operadores*, para definir la forma en que deben operar con objetos de tipos de datos definidos por el usuario (de hecho, hemos estado utilizando muchos operadores sobrecargados hasta este punto, incluyendo el operador de inserción de flujo << y el operador de extracción de flujo >>, que se sobrecargan para *todos* los tipos fundamentales. En el capítulo 10 hablaremos más acerca de cómo sobrecargar los operadores << y >> para poder manejar objetos de tipos definidos por el usuario).

6.19 Plantillas de funciones

Por lo general, las funciones sobrecargadas se utilizan para realizar operaciones similares que involucren distintos tipos de lógica de programa en distintos tipos de datos. Si la lógica del programa y las operaciones son *idénticas* para cada tipo de datos, la sobrecarga se puede llevar a cabo de una forma más compacta y conveniente, mediante el uso de **plantillas de funciones**. El programador escribe una sola definición de plantilla de función. Dados los tipos de los argumentos que se proporcionan en las llamadas a esta función, C++ genera de manera automática **especializaciones de plantilla de función** separadas para manejar cada tipo de llamada de manera apropiada. Por ende, al definir una sola plantilla de función, en esencia se define toda una familia de funciones sobrecargadas.

La figura 6.25 define una plantilla de función `maximo` (líneas 3 a 17) que determina el mayor de tres valores. Todas las definiciones de plantillas de función empiezan con la **palabra clave `template`** (línea 3), seguidas de una **lista de parámetros de plantilla** para la plantilla de función encerrada entre los paréntesis angulares (< y >). Antes de cada parámetro en la lista de parámetros (que a menudo se conoce como un **parámetro de tipo formal**) se coloca la palabra clave `typename` o la palabra clave `class` (que son sinónimos en este contexto). Los parámetros de tipo formal son receptáculos para los tipos fundamentales, o los tipos definidos por el usuario. Estos receptáculos, en este caso `T`, se utilizan para especificar los tipos de los parámetros de la función (línea 4), para especificar el tipo de valor de retorno de la función (línea 4) y para declarar variables dentro del cuerpo de la definición de la función (línea 6). Una plantilla de función se define de igual forma que cualquier otra función, pero utiliza los parámetros de tipo formal como receptáculos para los tipos de datos actuales.

```
1  // Fig. 6.25: maximo.h
2  // Archivo de encabezado de la plantilla de la función maximo.
3  template < typename T >  // o template< class T >
4  T maximo( T valor1, T valor2, T valor3 )
5  {
6     T valorMaximo = valor1; // asume que valor1 es maximo
7
8     // determina si valor2 es mayor que valorMaximo
9     if ( valor2 > valorMaximo )
10       valorMaximo = valor2;
11
12    // determina si valor3 es mayor que valorMaximo
13    if ( valor3 > valorMaximo )
14       valorMaximo = valor3;
15
16    return valorMaximo;
17 } // fin de la plantilla de función maximo
```

Fig. 6.25 | Archivo de encabezado de la plantilla de la función `maximo`.

La plantilla de función declara un solo parámetro de tipo formal `T` (línea 3) como receptáculo para el tipo de datos a evaluar por la función `maximo`. El nombre de un parámetro de tipo debe ser único en la lista de parámetros de la plantilla para una definición específica de ésta. Cuando el compilador detecta una invocación a `maximo` en el código fuente del programa, el *tipo de* los datos que se pasan a `maximo` se sustituye por `T` en toda la definición de la plantilla, y C++ crea una función completa para determinar el máximo de tres valores del tipo de datos especificado; los tres deben tener el mismo tipo, ya que sólo usamos un parámetro de tipo en este ejemplo. Después se compila la función recién creada; las plantillas son un medio para *generar código*.

La figura 6.26 utiliza la plantilla de función `maximo` para determinar el mayor de tres valores `int`, tres valores `double` y tres valores `char`, respectivamente (líneas 17, 27 y 37). Se crean funciones separadas como resultado de las llamadas en las líneas 17, 27 y 37: esperar tres valores `int`, tres valores `double` y tres valores `char`, respectivamente.

```
1  // Fig. 6.26: fig06_26.cpp
2  // Programa de prueba de la plantilla de función maximo.
3  #include <iostream>
4  #include "maximo.h" // incluye la definición de la plantilla de función maximo
5  using namespace std;
6
7  int main()
8  {
9     // demuestra la función maximo con valores int
10    int int1, int2, int3;
11
12    cout << "Introduzca tres valores enteros: ";
13    cin >> int1 >> int2 >> int3;
14
15    // invoca a la versión int de maximo
16    cout << "El valor int de maximo es: "
17       << maximo( int1, int2, int3 );
18
19    // demuestra la función maximo con valores double
20    double double1, double2, double3;
21
22    cout << "\n\nIntroduzca tres valores double: ";
23    cin >> double1 >> double2 >> double3;
24
25    // invoca a la versión double de maximo
26    cout << "El valor double de maximo es: "
27       << maximo( double1, double2, double3 );
28
29    // demuestra la función maximo con valores char
30    char char1, char2, char3;
31
32    cout << "\n\nIntroduzca tres caracteres: ";
33    cin >> char1 >> char2 >> char3;
34
35    // invoca a la versión char de maximo
36    cout << "El valor char de maximo es: "
37       << maximo( char1, char2, char3 ) << endl;
38 } // fin de main
```

Fig. 6.26 | Programa de prueba de la plantilla de función `maximo` (parte 1 de 2).

```
Introduzca tres valores enteros: 1 2 3
El valor int de maximo es: 3

Introduzca tres valores double: 3.3 2.2 1.1
El valor double de maximo es: 3.3

Introduzca tres caracteres: A C B
El valor char de maximo es: C
```

Fig. 6.26 | Programa de prueba de la plantilla de función `maximo` (parte 2 de 2).

La especialización de plantilla de función que se crea para el tipo `int` reemplaza cada ocurrencia de `T` con `int`, como se muestra a continuación:

```
int maximo( int valor1, int valor2, int valor3 )
{
   int valorMaximo = valor1; // asume que valor1 es el máximo

   // determina si valor2 es mayor que valorMaximo
   if ( valor2 > valorMaximo )
      valorMaximo = valor2;

   // determina si valor3 es mayor que valorMaximo
   if ( valor3 > valorMaximo )
      valorMaximo = valor3;

   return valorMaximo;
} // fin de la plantilla de función maximo
```

C++11: tipos de valores de retorno al final para las funciones

C++ introduce los **tipos de valores de retorno al final** para las funciones. Para especificar un tipo de valor de retorno al final, se coloca la palabra clave `auto` antes del nombre de la función, luego se coloca después de la lista de parámetros de la función el símbolo -> y el tipo de valor de retorno. Por ejemplo, para especificar un tipo de valor de retorno al final para la plantilla de función `maximo` (figura 6.25), escribiríamos:

```
template < typename T >
auto maximo( T x, T y, T z ) -> T
```

A medida que construya plantillas de función más complejas, habrá casos en los que sólo se permitan tipos de valores de retorno al final. Dichas plantillas de función complejas están más allá del alcance de este libro.

6.20 Recursividad

Para algunos problemas, es conveniente hacer que las funciones se *llamen a sí mismas*. Una **función recursiva** es una función que se llama a sí misma, ya sea en forma directa o indirecta (a través de otra función). [*Nota:* el estándar de C++ indica que `main` no debe llamarse dentro de un programa ni en forma recursiva. Su único propósito es servir de punto inicial para la ejecución del programa]. En esta sección y en la siguiente presentaremos ejemplos simples de recursividad. La recursividad se discute extensamente en los cursos de ciencias computacionales de nivel superior. En la figura 6.32 (al final de la sección 6.22) se sintetizan los ejemplos y ejercicios de recursividad que se incluyen en el libro.

Conceptos de recursividad

Primero hablaremos sobre la recursividad en forma conceptual, y después analizaremos programas que contienen funciones recursivas. Las metodologías recursivas de solución de problemas tienen varios elementos en común. Se hace una llamada a una función recursiva para resolver un problema. La función

en realidad sabe cómo resolver sólo el (los) *caso(s) más simple(s)*, o **caso(s) base**. Si se hace la llamada a la función con un caso base, ésta simplemente devuelve un resultado. Si se hace la llamada a la función con un problema más complejo, la función comúnmente divide el problema en dos piezas conceptuales: una pieza que sabe cómo resolver y otra pieza que no sabe cómo resolver. Para que la recursividad sea factible, esta última pieza *debe* ser similar al problema original, pero una versión ligeramente más sencilla o simple del mismo. Debido a que este nuevo problema se parece al problema original, la función llama a una copia de sí misma para trabajar en el problema más pequeño; a esto se le conoce como **llamada recursiva**, y también como **paso recursivo**. Por lo general, el paso recursivo incluye la palabra clave `return`, ya que su resultado se combina con la parte del problema que la función supo cómo resolver, para formar un resultado que se pasará de vuelta a la función original que hizo la llamada, que posiblemente sea `main`.

Error común de programación 6.12

Omitir el caso base o escribir el paso de recursividad en forma incorrecta, de modo que no converja en el paso base, producirá un error de recursividad infinito y tal vez un desbordamiento de pila. Esto es análogo al problema de un ciclo infinito en una solución iterativa (no recursiva).

El paso recursivo se ejecuta mientras siga "abierta" la llamada original a la función (es decir, que no haya terminado su ejecución). Este paso puede producir muchas llamadas recursivas más, a medida que la función divide cada nuevo subproblema, con el que se llama a la función, en dos piezas conceptuales. Para que la recursividad termine en un momento dado, cada vez que la función se llama a sí misma con una versión más simple del problema original, esta secuencia de problemas cada vez más pequeños debe *converger* en un caso base. En ese punto, la función reconoce el caso base y devuelve un resultado a la copia anterior de la función; después se origina una secuencia de retornos, hasta que la llamada a la función original devuelve el resultado final al método `main`. Todo esto suena bastante extravagante, en comparación con el tipo de solución de problemas que hemos usado hasta este punto. Como ejemplo de estos conceptos en la práctica, vamos a escribir un programa recursivo para realizar un popular cálculo matemático.

Factorial

El factorial de un entero positivo n, que se escribe como $n!$ (y se pronuncia como "factorial de n"), viene siendo el producto

$$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

en donde 1! es igual a 1 y 0! se define como 1. Por ejemplo, 5! es el producto $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, que es igual a 120.

Factorial iterativo

El factorial de un `numero` entero más grande o igual a 0, puede calcularse de manera **iterativa** (sin recursividad), usando una instrucción `for` de la siguiente manera:

```
factorial = 1;
for ( unsigned int contador = numero; contador >= 1; --contador )
   factorial *= contador;
```

Factorial recursivo

Podemos llegar a una definición *recursiva* de la función factorial, si observamos la siguiente relación algebraica:

$$n! = n \cdot (n-1)!$$

Por ejemplo, 5! es sin duda igual a 5 * 4!, como se muestra en las siguientes ecuaciones:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$
$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$
$$5! = 5 \cdot (4!)$$

Evaluación de 5!

La evaluación de 5! procedería como se muestra en la figura 6.27, que ilustra cómo procede la sucesión de llamadas recursivas hasta que 1! se evalúa como 1, lo cual termina la recursividad. La figura 6.27(b) muestra los valores devueltos de cada llamada recursiva a la función que hizo la llamada, hasta que se calcula y devuelve el valor final.

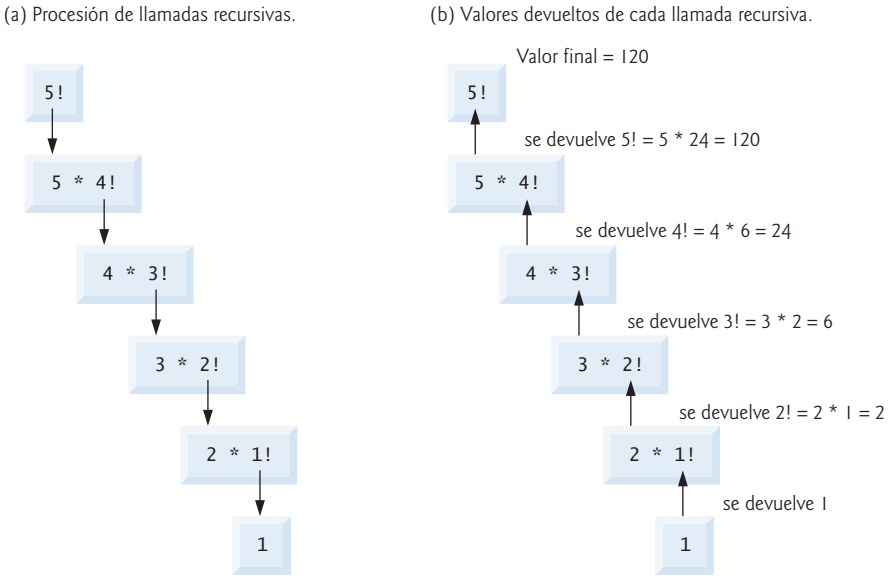


Fig. 6.27 | Evaluación recursiva de 5!.

Uso de una función `factorial` *recursiva para calcular los factoriales*

La figura 6.28 utiliza la recursividad para calcular e imprimir los factoriales de los enteros del 0 al 10. (En unos momentos explicaremos la elección del tipo de datos `unsigned long`). La función recursiva `factorial` (líneas 18 a 24) determina primero si la condición de terminación `numero <= 1` (línea 20) es verdadera. Si `numero` es menor o igual que 1, la función `factorial` devuelve 1 (línea 21), ya no es necesaria más recursividad y la función termina. Si `numero` es mayor que `1`, en la línea 23 se expresa el problema como el producto de `numero` y una llamada recursiva a `factorial` en la que se evalúa el factorial de `numero - 1`, que es un problema un poco más simple que el cálculo original, `factorial (numero)`.

Por qué elegimos el tipo `unsigned long` *en este ejemplo*

La función `factorial` se ha declarado para recibir un parámetro de tipo `unsigned long` y devolver un resultado de tipo `unsigned long`. Ésta es una notación abreviada para `unsigned long int`. El estándar de C++ requiere que una variable de tipo `unsigned long int` sea *por lo menos tan grande como* un valor

```
1  // Fig. 6.28: fig06_28.cpp
2  // Función recursiva factorial.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  unsigned long factorial( unsigned long ); // prototipo de función
8
9  int main()
10 {
11    // calcula los factoriales del 0 al 10
12    for ( unsigned int contador = 0; contador <= 10; ++contador  )
13       cout << setw( 2 ) << contador << "! = " << factorial( contador )
14          << endl;
15 } // fin de main
16
17 // definición recursiva de la función factorial
18 unsigned long factorial( unsigned long numero )
19 {
20    if ( numero <= 1 ) // evalúa el caso base
21       return 1; // casos base: 0! = 1 y 1! = 1
22    else // paso recursivo
23       return numero * factorial( numero - 1 );
24 } // fin de la función factorial
```

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

Fig. 6.28 | Función recursiva `factorial`.

`int`. Por lo general, un valor `unsigned long int` se almacena en por lo menos cuatro bytes (32 bits); dicha variable puede contener un valor en el rango desde 0 hasta (por lo menos) 4 294 967 295. (El tipo de datos `long int` también se almacena por lo menos en cuatro bytes, y puede contener un valor en por lo menos el rango de –2 147 483 647 a 2 147 483 647). Como se puede ver en la figura 6.28, los valores de los factoriales aumentan su tamaño rápidamente. Elegimos el tipo de datos `unsigned long` de manera que el programa pueda calcular factoriales mayores que 7! en las computadoras con enteros pequeños (como los de dos bytes). Por desgracia, la función `factorial` produce valores extensos con tanta rapidez que, incluso el tipo `unsigned long` no nos ayuda a calcular muchos valores de factorial antes de que se exceda el valor de una variable `unsigned long`.

El tipo `unsigned long long int` *de C++11*

El nuevo tipo `unsigned long long int` de C++ (que puede abreviarse como `unsigned long long`) en algunos sistemas nos permite almacenar valores en 8 bytes (64 bits), para guardar números tan grandes como 18 446 744 073 709 551 615.

Representar números aún más grandes
Podrían usarse variables de tipo de datos `double` para calcular los factoriales de números más grandes. Esto apunta a una debilidad en la mayoría de los lenguajes de programación, a saber, que los lenguajes no se extienden fácilmente para manejar los requerimientos únicos de varias aplicaciones. Como veremos cuando hablemos sobre la programación orientada a objetos con más detalle, C++ es un lenguaje *extensible* que nos permite crear clases que puedan representar enteros arbitrariamente grandes, si lo deseamos. Dichas clases ya están disponibles en bibliotecas de clases populares, y trabajaremos en nuestras propias clases similares en los ejercicios 9.14 y 10.9.

6.21 Ejemplo sobre el uso de la recursividad: serie de Fibonacci

La serie de Fibonacci,

```
0, 1, 1, 2, 3, 5, 8, 13, 21, …
```

empieza con 0 y 1, y tiene la propiedad de que cada número subsiguiente de Fibonacci es la suma de los dos números Fibonacci anteriores.

Esta serie ocurre en la naturaleza y, en específico, describe una forma de espiral. La proporción de números de Fibonacci sucesivos converge en un valor constante de 1.618…. Este número también ocurre con frecuencia en la naturaleza, y se le ha denominado **proporción dorada**, o **media dorada**. Los humanos tienden a descubrir que la media dorada es estéticamente placentera. A menudo, los arquitectos diseñan ventanas, cuartos y edificios cuya longitud y anchura se encuentran en la proporción de la media dorada. A menudo se diseñan tarjetas postales con una proporción de anchura/altura de media dorada.

Definición recursiva de Fibonacci
La serie de Fibonacci se puede definir de manera recursiva como:

$$\text{fibonacci}(0) = 0$$
$$\text{fibonacci}(1) = 1$$
$$\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$$

El programa de la figura 6.29 calcula el *n*-ésimo número de Fibonacci en forma recursiva, usando la función `fibonacci`. Los números de Fibonacci tienden a aumentar con bastante rapidez, aunque son más lentos que los factoriales. Por lo tanto, elegimos el tipo de datos `unsigned long` para el tipo del parámetro y el tipo de valor de retorno en la función `fibonacci`. En la figura 6.29 se muestra la ejecución del programa, que muestra los valores de Fibonacci para varios números.

La aplicación empieza con una instrucción `for` que calcula y muestra los valores de `Fibonacci` para los enteros 0 a 10, y va seguida de tres llamadas para calcular los valores Fibonacci de los enteros 20, 30 y 35 (líneas 16 a 18). Las llamadas a la función `fibonacci` (líneas 13 y 16 a 18) de `main` *no* son llamadas recursivas, pero las llamadas de la línea 27 de `fibonacci` son recursivas. Cada vez que el programa invoca a `fibonacci` (líneas 22 a 28), la función evalúa de inmediato el caso base para determinar si `numero` es igual a `0` o `1` (línea 24). Si esto es verdadero, en la línea 25 se devuelve `numero`. Lo interesante es que, si `numero` es mayor que 1, el paso recursivo (línea 27) genera *dos* llamadas recursivas, cada una para un problema ligeramente más pequeño que la llamada original a `fibonacci`.

```
1  // Fig. 6.29: fig06_29.cpp
2  // Función recursiva fibonacci.
3  #include <iostream>
```

Fig. 6.29 | Función recursiva `fibonacci` (parte 1 de 2).

```
4  using namespace std;
5
6  unsigned long fibonacci( unsigned long ); // prototipo de función
7
8  int main()
9  {
10    // calcula los valores de fibonacci del 0 al 10
11    for ( unsigned int contador = 0; contador <= 10; ++contador )
12       cout << "fibonacci( " << contador << " ) = "
13          << fibonacci( contador ) << endl;
14
15    // muestra valores de fibonacci mayores
16    cout << "\nfibonacci( 20 ) = " << fibonacci( 20 ) << endl;
17    cout << "fibonacci( 30 ) = " << fibonacci( 30 ) << endl;
18    cout << "fibonacci( 35 ) = " << fibonacci( 35 ) << endl;
19 } // fin de main
20
21 // método fibonacci recursivo
22 unsigned long fibonacci( unsigned long numero )
23 {
24    if ( ( 0 == numero ) || ( 1 == numero ) ) // casos base
25       return numero;
26    else // paso recursivo
27       return fibonacci( numero - 1 ) + fibonacci( numero - 2 );
28 } // fin de la función fibonacci
```

```
fibonacci( 0 ) = 0
fibonacci( 1 ) = 1
fibonacci( 2 ) = 1
fibonacci( 3 ) = 2
fibonacci( 4 ) = 3
fibonacci( 5 ) = 5
fibonacci( 6 ) = 8
fibonacci( 7 ) = 13
fibonacci( 8 ) = 21
fibonacci( 9 ) = 34
fibonacci( 10 ) = 55

fibonacci( 20 ) = 6765
fibonacci( 30 ) = 832040
fibonacci( 35 ) = 9227465
```

Fig. 6.29 | Función recursiva `fibonacci` (parte 2 de 2).

Evaluación de `fibonacci(3)`

La figura 6.30 muestra cómo la función `fibonacci` evaluaría `fibonacci(3)`. Esta figura genera ciertas preguntas interesantes, en cuanto al *orden* en el que los compiladores de C++ evalúan los operandos de los operadores. Este orden es *distinto* del orden en el que se aplican los operadores a sus operandos; a saber, el orden que dictan las reglas de la precedencia y asociatividad de los operadores. La figura 6.30 muestra que al evaluar `fibonacci(3)`, se producen dos llamadas recursivas: `fibonacci(2)` y `fibonacci(1)`. ¿En qué orden se hacen estas llamadas?

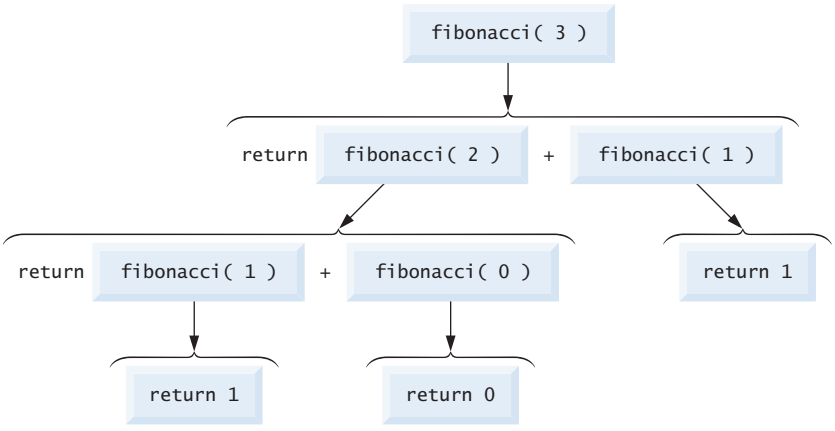


Fig. 6.30 | Conjunto de llamadas recursivas a la función `fibonacci`.

Orden de evaluación de los operandos

La mayoría de los programadores simplemente suponen que los operandos se evalúan de izquierda a derecha. C++ *no* especifica el orden en el que se van a evaluar los operandos de la mayoría de los operadores (incluyendo +). Por lo tanto, no debemos hacer suposiciones en cuanto al orden en el que se evaluarán estas llamadas. De hecho, se podría ejecutar `fibonacci(2)` primero y después `fibonacci(1)`, o se podrían ejecutar en el orden inverso: `fibonacci(1)` y luego `fibonacci(2)`. En este programa y en la mayoría de los otros programas, el resultado sería el mismo. Sin embargo, en algunos programas la evaluación de un operando puede tener **efectos secundarios** (cambios en los valores de los datos) que podrían afectar al resultado final de la expresión.

C++ especifica el orden de evaluación de los operandos sólo para *cuatro* operadores: `&&`, `||`, el operador coma (`,`) y `?:`. Los primeros tres son operadores binarios, y se garantiza que sus dos operandos se evaluarán de izquierda a derecha. El último operador es el único operador *ternario* de C++. Su operando de más a la izquierda siempre se evalúa primero; si se evalúa como *verdadero*, el operando intermedio se evalúa a continuación y se ignora el último operando; si el operando de más a la izquierda se evalúa como *falso*, el tercer operando se evalúa a continuación y se ignora el operando intermedio.

Tip de portabilidad 6.2

Los programas que dependen del orden de evaluación de los operandos de operadores distintos de `&&`, `||`, `?:` y el operador coma (`,`) pueden funcionar de manera distinta en sistemas con distintos compiladores y producir errores lógicos.

Error común de programación 6.13

La escritura de programas que dependan del orden de evaluación de los operandos de operadores distintos de `&&`, `||`, `?:` y el operador coma (`,`) pueden producir errores lógicos.

Tip para prevenir errores 6.9

No dependa del orden en el que se evalúan los operandos. Para asegurar que los efectos secundarios se apliquen en el orden correcto, divida las expresiones complejas en instrucciones separadas.

Error común de programación 6.14

Recuerde que los operadores && y || usan la evaluación de corto circuito. Colocar una expresión con un efecto secundario del lado derecho de un operador && o || es un error lógico si esa expresión siempre debe evaluarse.

Complejidad exponencial

Hay que tener cuidado con los programas recursivos, como el que usamos aquí para generar números de Fibonacci. Cada nivel de recursividad en la función `fibonacci` tiene un efecto de *duplicación* sobre el número de llamadas a funciones; es decir, el número de llamadas recursivas que se requieren para calcular el *n*-ésimo número de Fibonacci se encuentra en el orden de 2^n. Esto se sale rápidamente de control. Para calcular sólo el 20vo. número de Fibonacci se requeriría un orden de 2^{20}, o cerca de un millón de llamadas, para calcular el 30vo. número de Fibonacci se requeriría un orden de 2^{30}, o aproximadamente mil millones de llamadas, y así en lo sucesivo. Los científicos computacionales se refieren a esto como **complejidad exponencial**. Los problemas de esta naturaleza pueden humillar incluso hasta a las computadoras más poderosas del mundo. Las cuestiones relacionadas con la complejidad, tanto en general como en particular, se discuten con detalle en un curso del plan de estudios de ciencias computacionales de nivel superior, al que generalmente se le llama "Algoritmos".

Tip de rendimiento 6.8

Evite los programas recursivos al estilo Fibonacci que produzcan una "explosión" exponencial de llamadas.

6.22 Comparación entre recursividad e iteración

En las dos secciones anteriores, estudiamos dos funciones recursivas que también pueden implementarse mediante programas iterativos simples. En esta sección comparamos las dos metodologías, y hablamos acerca del por qué podríamos elegir una metodología sobre la otra en una situación específica.

- Tanto la iteración como la recursividad se *basan en una instrucción de control*: la iteración utiliza una *estructura de repetición*; la recursividad utiliza una *estructura de selección*.
- Tanto la iteración como la recursividad implican la *repetición*: la iteración utiliza en forma explícita una *estructura de repetición*; la recursividad logra la repetición a través de *llamadas repetidas a una función*.
- La iteración y la recursividad implican una *prueba de terminación*: la iteración termina cuando *falla la condición de continuación de ciclo*; la recursividad termina cuando *se reconoce un caso base*.
- La iteración mediante la repetición controlada por un contador y la recursividad *se acercan gradualmente a la terminación*: la iteración *modifica un contador* hasta que éste asuma un valor que haga que falle la condición de continuación de ciclo; la recursividad produce versiones *más simples del problema original* hasta que se llega al caso base.
- Tanto la iteración como la recursividad *pueden ocurrir infinitamente*: un *ciclo infinito* ocurre con la iteración si la prueba de continuación de ciclo nunca se vuelve falsa; la *recursividad infinita* ocurre si el paso recursivo no reduce el problema durante cada llamada recursiva, de forma tal que llegue a converger en el caso base.

Implementación del factorial iterativo

Para ilustrar las diferencias entre la iteración y la recursividad, vamos a examinar una solución iterativa para el problema del factorial (figura 6.31). Se utiliza una instrucción de repetición (líneas 23 y 24 de la figura 6.31) en vez de la instrucción de selección de la solución recursiva (líneas 20 a 23 de la

figura 6.28). Ambas soluciones usan una prueba de terminación. En la solución recursiva, en la línea 20 (figura 6.28) se evalúa el caso base. En la solución iterativa, en la línea 23 (figura 6.31) se evalúa la condición de continuación de ciclo; si la prueba falla, el ciclo termina. Por último, en vez de producir versiones cada vez más simples del problema original, la solución iterativa utiliza un contador que se modifica hasta que la condición de continuación de ciclo se vuelve falsa.

```
1  // Fig. 6.31: fig06_31.cpp
2  // Función iterativa factorial.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  unsigned long factorial( unsigned int ); // prototipo de función
8
9  int main()
10 {
11    // calcula los factoriales del 0 al 10
12    for ( unsigned int contador = 0; contador <= 10; ++contador )
13       cout << setw( 2 ) << contador << "! = " << factorial( contador )
14          << endl;
15 } // fin de main
16
17 // función factorial iterativa
18 unsigned long factorial( unsigned int numero )
19 {
20    unsigned long resultado = 1;
21
22    // cálculo iterativo del factorial
23    for ( unsigned int i = numero; i >= 1; --i )
24       resultado *= i;
25
26    return resultado;
27 } // fin de la función factorial
```

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

Fig. 6.31 | Función iterativa `factorial`.

Desventajas de la recursividad

La recursividad tiene muchas desventajas. Invoca al mecanismo en forma repetida, y en consecuencia se produce una *sobrecarga de las llamadas a la función*. Esta repetición puede ser perjudicial, en términos de tiempo del procesador y espacio de la memoria. Cada llamada recursiva crea *otra copia de las variables de la función*; esto puede consumir una cantidad considerable de memoria. Como la iteración ocurre

comúnmente dentro de una función, se evita la sobrecarga de las llamadas repetidas a la función y la asignación adicional de memoria. Entonces, ¿por qué elegir la recursividad?

Observación de Ingeniería de Software 6.13

Cualquier problema que se pueda resolver mediante la recursividad, se puede resolver también mediante la iteración (sin recursividad). Por lo general se prefiere un método recursivo a uno iterativo cuando el primero refleja con más naturalidad el problema, y se produce un programa más fácil de entender y de depurar. Otra razón por la que es preferible elegir una solución recursiva es que una iterativa podría no ser aparente.

Tip de rendimiento 6.9

Evite usar la recursividad en situaciones en las que se requiera un alto rendimiento. Las llamadas recursivas requieren tiempo y consumen memoria adicional.

Error común de programación 6.15

Hacer que una función no recursiva se llame a sí misma por accidente, ya sea en forma directa o indirecta (a través de otra función), es un error lógico.

Resumen de los ejemplos y ejercicios de recursividad en este libro

En la figura 6.32 se sintetizan los ejemplos de recursividad y los ejercicios que se incluyen en el libro.

Ubicación en el libro	Ejemplos y ejercicios de recursividad
Capítulo 6	
Sección 6.20, fig. 6.28	Función factorial
Sección 6.21, fig. 6.29	Función Fibonacci
Ejercicio 6.36	Exponenciación recursiva
Ejercicio 6.38	Torres de Hanoi
Ejercicio 6.40	Visualización de la recursividad
Ejercicio 6.41	Máximo común divisor
Ejercicios 6.44 y 6.45	Ejercicio “¿Qué hace este programa?”
Capítulo 7	
Ejercicio 7.17	Ejercicio “¿Qué hace este programa?”
Ejercicio 7.20	Ejercicio “¿Qué hace este programa?”
Ejercicio 7.28	Determinar si una cadena es un palíndromo
Ejercicio 7.29	Ocho reinas
Ejercicio 7.30	Imprimir un arreglo
Ejercicio 7.31	Imprimir una cadena en forma inversa
Ejercicio 7.32	Valor mínimo en un arreglo
Ejercicio 7.33	Recorrido de laberinto
Ejercicio 7.34	Generación de laberintos al azar

Fig. 6.32 | Resumen de los ejemplos y ejercicios de recursividad en el libro (parte 1 de 2).

Ubicación en el libro	Ejemplos y ejercicios de recursividad
Capítulo 19 (en el sitio web)	
Sección 19.6, figs. 19.20 y 19.22	Inserción de árboles binarios
Sección 19.6, figs. 19.20 y 19.22	Recorrido preorden de un árbol binario
Sección 19.6, figs. 19.20 y 19.22	Recorrido inorden de un árbol binario
Sección 19.6, figs. 19.20 y 19.22	Recorrido postorden de un árbol binario
Ejercicio 19.20	Imprimir una lista enlazada en forma inversa
Ejercicio 19.21	Buscar en una lista enlazada
Ejercicio 19.22	Eliminación de árboles binarios
Ejercicio 19.23	Búsqueda de árboles binarios
Ejercicio 19.24	Recorrido por orden de nivel de un árbol binario
Ejercicio 19.25	Árbol de impresión
Capítulo 20 (en el sitio web)	
Sección 20.3.3, fig. 20.6	Ordenamiento por combinación
Ejercicio 20.8	Búsqueda lineal
Ejercicio 20.9	Búsqueda binaria
Ejercicio 20.10	Quicksort

Fig. 6.32 | Resumen de los ejemplos y ejercicios de recursividad en el libro (parte 2 de 2).

6.23 Conclusión

En este capítulo aprendió más acerca de las declaraciones de funciones, incluyendo los prototipos, las firmas, los encabezados y los cuerpos de las funciones. Vimos las generalidades sobre las funciones matemáticas de la biblioteca. Aprendió acerca de la coerción de argumentos, o la acción de forzar a que los argumentos sean de los tipos apropiados que se especifiquen mediante las declaraciones de los parámetros de una función. Demostramos cómo usar las funciones `rand` y `srand` para generar conjuntos de números aleatorios que se puedan utilizar para las simulaciones. Le mostramos cómo definir conjuntos de constantes mediante `enum`. Aprendió también acerca del alcance de las variables, los especificadores de clase de almacenamiento y la duración del almacenamiento. Vimos dos formas distintas de pasar argumentos a las funciones: paso por valor y paso por referencia. Para el paso por referencia, las referencias se utilizan como un alias para una variable. Le mostramos cómo implementar funciones en línea y funciones que reciban argumentos predeterminados. Aprendió que varias funciones en una clase se pueden sobrecargar, usando el mismo nombre y distintas firmas. Dichas funciones se pueden utilizar para realizar las mismas tareas (o similares), usando distintos tipos o números de parámetros. Después, demostramos una manera más simple de sobrecargar funciones mediante las plantillas de función, en donde una función se define una sola vez, pero se puede usar para varios tipos distintos. Posteriormente le presentamos el concepto de la recursividad, en donde una función se llama a sí misma para resolver un problema.

En el capítulo 7, aprenderá a mantener listas y tablas de datos en arreglos y vectores (`vector`) orientados a objetos. Veremos una implementación basada en arreglo más elegante de la aplicación para tirar dados, y dos versiones mejoradas del caso de estudio `LibroCalificaciones` que estudiamos en los capítulos 3 a 6, en donde utilizaremos arreglos para almacenar las calificaciones introducidas.

Resumen

Sección 6.1 Introducción

- La experiencia ha demostrado que la mejor forma de desarrollar y mantener un programa extenso es construirlo a partir de piezas simples y pequeñas. A esta técnica se le conoce como divide y vencerás (pág. 202).

Sección 6.2 Componentes de los programas en C++

- Por lo general, los programas en C++ se escriben mediante la combinación de nuevas funciones y clases que escribimos con funciones "pre-empaquetadas", y clases disponibles en la Biblioteca estándar de C++.
- Las funciones permiten al programador modularizar un programa, al separar sus tareas en unidades autocontenidas.
- Las instrucciones en los cuerpos de las funciones se escriben sólo una vez, se pueden reutilizar tal vez desde varias ubicaciones en un programa y además están ocultas de las demás funciones.

Sección 6.3 Funciones matemáticas de la biblioteca

- Algunas veces las funciones no son miembros de una clase. A dichas funciones se les conoce como funciones globales (pág. 204).
- A menudo, los prototipos para las funciones globales se colocan en encabezados, de manera que las funciones globales se puedan reutilizar en cualquier programa que incluya el encabezado y se pueda crear un enlace con el código objeto de la función.

Sección 6.4 Definiciones de funciones con varios parámetros

- El compilador hace referencia al prototipo de función, para comprobar que las llamadas a una función tengan el número y tipos de argumentos correctos, que los tipos de los argumentos estén en el orden correcto y que el valor devuelto por la función se pueda utilizar de manera correcta en la expresión que llamó a la función.
- Si una función no devuelve un resultado, el control regresa cuando el programa llega a la llave derecha de fin de la función, o mediante la ejecución de la instrucción

  ```
  return;
  ```

  Si una función devuelve un resultado, la instrucción

  ```
  return expresión;
  ```

  evalúa *expresión* y devuelve el valor de *expresión* a la función que hizo la llamada.

Sección 6.5 Prototipos de funciones y coerción de argumentos

- La porción de un prototipo de función que incluye el nombre de la función y los tipos de sus argumentos se conoce como la firma de la función (pág. 211), o simplemente firma.
- Una característica importante de los prototipos de función es la coerción de argumentos (pág. 211); es decir, obligar a que los argumentos tengan los tipos especificados por las declaraciones de los parámetros.
- El compilador puede convertir los argumentos a los tipos de parámetros según lo especificado por las reglas de promoción de C++ (pág. 211). Las reglas de promoción indican las conversiones implícitas que el compilador puede realizar entre tipos fundamentales.

Sección 6.6 Encabezados de la Biblioteca estándar de C++

- La Biblioteca estándar de C++ está dividida en muchas porciones, cada una con su propio encabezado. Los encabezados también contienen definiciones de varios tipos de clases, funciones y constantes.
- Un encabezado "instruye" al compilador acerca de cómo interconectarse con los componentes de la biblioteca.

Sección 6.7 Caso de estudio: generación de números aleatorios

- Llamar a `rand` (pág. 214) en forma repetida produce una secuencia de números seudoaleatorios (pág. 217). La secuencia se repite cada vez que se ejecute el programa.

- Para randomizar los números producidos por `rand`, se pasa un argumento `unsigned integer` (por lo general de la función `time`; pág. 219) a la función `srand` (pág. 217), la cual siembra la función `rand`.
- Los números aleatorios en un rango se pueden generar de la siguiente manera:

  numero = *valorDesplazamiento* + `rand() %` *factorEscala*;

  en donde *valorDesplazamiento* (pág. 219) especifica el primer número en el rango deseado de enteros consecutivos y *factorEscala* (pág. 219) es igual a la anchura del rango deseado de enteros consecutivos.

Sección 6.8 Caso de estudio: juego de probabilidad: introducción a las `enum`

- Una enumeración, que se introduce mediante la palabra clave `enum` y va seguida de un nombre de tipo (pág. 222), es un conjunto de constantes enteras con nombres (pág. 222) que empiezan en `0`, a menos que se especifique lo contrario, y se incrementan en `1`.
- `enum` sin alcance pueden producir conflictos de nombres y errores lógicos. Para eliminar estos problemas, C++11 introduce `enum` con alcance (pág. 223), que se declaran con las palabras clave `enum class` (o con el sinónimo `enum struct`).
- Para hacer referencia a una constante `enum` con alcance, debemos calificar la constante con el nombre del tipo de la `enum` con alcance y con el operador de resolución de ámbito (`::`). Si otra `enum` con alcance contiene el mismo identificador que para una de sus constantes, siempre queda claro cuál versión de la constante se está utilizando.
- Las constantes en una `enum` se representan como enteros.
- El tipo integral subyacente de una `enum` sin alcance depende de los valores de sus constantes; se garantiza que el tipo será lo bastante grande como para poder almacenar los valores constantes especificados.
- El tipo integral subyacente de la `enum` con alcance es `int`, de manera predeterminada.
- C++11 nos permite especificar el tipo integral subyacente de una `enum`, colocando después del nombre del tipo de `enum` dos puntos (`:`) y el tipo integral.
- Se produce un error de compilación si el valor de la constante de una `enum` está fuera del rango que puede representarse por el tipo subyacente de la `enum`.

Sección 6.9 Números aleatorios de C++11

- De acuerdo con CERT, la función `rand` no tiene "buenas propiedades estadísticas" y puede ser predecible, lo que hace a los programas que usan `rand` menos seguros.
- C++11 cuenta con una nueva biblioteca más segura de herramientas de números aleatorios, que puede producir números aleatorios no determinísticos para simulaciones y casos de seguridad, en donde la predictibilidad es indeseable. Estas nuevas herramientas se encuentran en el encabezado `<random>` de la Biblioteca estándar de C++.
- Para tener flexibilidad con base en la forma en que se utilizan los números aleatorios en los programas, C++11 provee muchas clases que representan varios motores de generación de números aleatorios y distribuciones. Un motor implementa un algoritmo de generación de números aleatorios que produce números seudoaleatorios. Una distribución controla el rango de valores producidos por un motor, los tipos de esos valores y las propiedades estadísticas de los valores.
- El tipo `default_random_engine` (pág. 224) representa el motor de generación de números aleatorios predeterminado.
- La distribución `uniform_int_distribution` (pág. 224) distribuye de manera uniforme los enteros seudoaleatorios a través de un rango especificado de valores. El rango predeterminado es desde 0 hasta el valor máximo de un `int` en su plataforma.

Sección 6.10 Clases y duración de almacenamiento

- La duración de almacenamiento de un identificador (pág. 225) determina el periodo durante el cual éste existe en la memoria.
- El alcance de un identificador es la parte en la que se puede hacer referencia a éste en un programa.
- El enlace de un identificador (pág. 225) determina si se conoce sólo en el archivo fuente en el que se declara, o en varios archivos fuente que se compilan y después se enlazan juntos.

- Las variables con duración de almacenamiento automática incluyen las variables locales declaradas en las funciones, los parámetros de función y las variables locales o los parámetros de función declarados con `register` (pág. 225). Dichas variables se crean cuando la ejecución del programa entra en el bloque en el que están definidas, existen mientras el bloque está activo y se destruyen cuando el programa sale del bloque.
- Las palabras clave `extern` (pág. 225) y `static` declaran identificadores para variables de la duración de almacenamiento estática (pág. 225) y para funciones. Las variables de duración de almacenamiento estática existen a partir del punto en el que el programa empieza a ejecutarse, y dejan de existir cuando termina el programa.
- El almacenamiento de una variable con duración de almacenamiento estático se asigna cuando el programa empieza su ejecución. Dicha variable se inicializa una vez al encontrar su declaración. Para las funciones, el nombre de la función existe cuando el programa empieza a ejecutarse, de igual forma que para las otras funciones.
- Los identificadores externos (como las variables globales) y las variables locales declaradas con el especificador de clase de almacenamiento `static` tienen duración de almacenamiento estática (pág. 225).
- Las declaraciones de variables globales (pág. 227) se colocan fuera de cualquier definición de clase o función. Las variables globales retienen sus valores a lo largo de la ejecución del programa. Las variables y las funciones globales se pueden referenciar mediante cualquier función que siga sus declaraciones o definiciones.
- A diferencia de las variables automáticas, las variables locales `static` retienen sus valores cuando la función en la que se declaran regresa a la que hizo la llamada.

Sección 6.11 Reglas de alcance

- Un identificador que se declara fuera de cualquier función o clase tiene alcance de espacio de nombres global (pág. 228).
- Los identificadores que se declaran dentro de un bloque tienen alcance de bloque (pág. 228), el cual empieza en la declaración del identificador y termina en la llave derecha de finalización (}) del bloque en el que se declara el identificador.
- Las etiquetas son los únicos identificadores con alcance de función (pág. 228). Las etiquetas pueden usarse en cualquier parte de la función en la que aparezcan, pero no pueden referenciarse fuera del cuerpo de la función.
- Un identificador que se declara fuera de una función o clase tiene alcance de espacio de nombres global. Dicho identificador es "conocido" en todas las funciones, desde el punto en el que se declara hasta el final del archivo.
- Los identificadores en la lista de parámetros de un prototipo de función tienen alcance de prototipo de función (pág. 228).

Sección 6.12 La pila de llamadas a funciones y los registros de activación

- Las pilas (pág. 231) se denominan estructuras de datos "último en entrar, primero en salir" (UEPS); el último elemento que se mete (inserta; pág. 231) en la pila es el primero que se saca (extrae; pág. 231) de ella.
- La pila de llamadas a funciones (pág. 232) soporta el mecanismo de llamadas/regresos a funciones y la creación, mantenimiento y destrucción de las variables automáticas de cada función a la que se llama.
- Cada vez que una función llama a otra, se mete un marco de pila o registro de activación (pág. 232) a la pila, el cual contiene la dirección de retorno que necesita la función a la que se llamó para poder regresar a la función que hizo la llamada, junto con las variables automáticas y parámetros de la llamada a la función.
- El marco de pila (pág. 232) existe mientras la función a la que se llamó esté activa. Cuando esa función regresa, su marco de pila se saca de la pila y sus variables automáticas locales dejan de existir.

Sección 6.13 Funciones con listas de parámetros vacías

- En C++, una lista de parámetros vacía se especifica mediante `void` o nada entre paréntesis.

Sección 6.14 Funciones en línea

- C++ cuenta con las funciones en línea (pág. 236) para ayudar a reducir la sobrecarga de las llamadas a funciones; en especial para las funciones pequeñas. Al colocar el calificador `inline` (pág. 236) antes del tipo de valor de retorno de la función en su definición, se aconseja al compilador para que genere una copia del código de la función en cada lugar en donde se llame a la función, para evitar la llamada a una función.

- Los compiladores pueden poner código en línea para el que no hayamos utilizado de manera explícita la palabra clave `inline`. Los compiladores optimizadores modernos son tan sofisticados, que es mejor dejar a ellos las decisiones sobre poner código en línea.

Sección 6.15 Referencias y parámetros por referencia

- Cuando se pasa un argumento por valor (pág. 237), se crea una *copia* del valor del argumento y se pasa a la función que se llamó. Las modificaciones a la copia no afectan al valor de la variable original en la función que hizo la llamada.
- Mediante el paso por referencia (pág. 237), la función que hace la llamada proporciona a la función que llamó la habilidad de acceder directamente a los datos de la primera, y de modificar esos datos en caso de que la función que se llamó así lo decida.
- Un parámetro por referencia (pág. 238) es un alias para su correspondiente argumento en la llamada a una función.
- Para indicar que un parámetro de función se pasa por referencia, simplemente hay que colocar un signo "`&`" después del tipo del parámetro en el prototipo de la función y en su encabezado.
- Todas las operaciones que se realizan en una referencia en realidad se realizan en la variable original.

Sección 6.16 Argumentos predeterminados

- Cuando una función se invoca repetidas veces con el mismo argumento para un parámetro específico, podemos especificar que dicho parámetro tiene un argumento predeterminado (pág. 240).
- Cuando un programa omite un argumento para un parámetro con un argumento predeterminado, el compilador inserta el valor predeterminado del argumento para pasarlo a la llamada a la función.
- Los argumentos predeterminados deben ser los argumentos de más a la derecha en la lista de parámetros de una función.
- Los argumentos predeterminados deben especificarse en el prototipo de la función.

Sección 6.17 Operador de resolución de ámbito unario

- C++ proporciona el operador de resolución de ámbito unario (`::`) (pág. 242) para acceder a una variable global cuando una variable local con el mismo nombre se encuentra dentro del alcance.

Sección 6.18 Sobrecarga de funciones

- C++ permite definir varias funciones con el mismo nombre, siempre y cuando éstas tengan diferentes conjuntos de parámetros. A esta capacidad se le conoce como sobrecarga de funciones (pág. 243).
- Cuando se hace una llamada a una función sobrecargada, el compilador de C++ selecciona la función apropiada al examinar el número, tipos y orden de los argumentos en la llamada.
- Las funciones sobrecargadas se diferencian mediante sus firmas.
- El compilador codifica cada identificador de función con los tipos de sus parámetros para permitir un enlace seguro de tipos (pág. 244). Este tipo de enlace asegura que se llame a la función sobrecargada apropiada, y que los tipos de los argumentos se conformen a los tipos de los parámetros.

Sección 6.19 Plantillas de funciones

- Por lo general, las funciones sobrecargadas realizan operaciones similares que involucren distintos tipos de lógica de programa en distintos tipos de datos. Si la lógica del programa y las operaciones son idénticas para cada tipo de datos, la sobrecarga se puede llevar a cabo de una forma más compacta y conveniente, mediante el uso de plantillas de funciones (pág. 246).
- Dados los tipos de los argumentos que se proporcionan en las llamadas a una plantilla de función, C++ genera de manera automática especializaciones de plantilla de función separadas (pág. 246) para manejar cada tipo de llamada de manera apropiada.
- Todas las definiciones de plantillas de función empiezan con la palabra clave `template` (pág. 246) seguida de una lista de parámetros de plantilla (pág. 246) para la plantilla de función encerrada entre los paréntesis angulares (< y >).

- Los parámetros de tipo formal (pág. 246) son precedidos por la palabra clave `typename` (o `class`) y son receptáculos para los tipos fundamentales, o los tipos definidos por el usuario. Estos receptáculos se utilizan para especificar los tipos de los parámetros de la función, para especificar el tipo de valor de retorno de la función y para declarar variables dentro del cuerpo de la definición de la función.
- C++11 introduce los tipos de valores de retorno al final para las funciones. Para especificar este tipo de valor de retorno al final, coloque la palabra clave `auto` antes del nombre de la función y coloque después de la lista de parámetros de la función el signo `->` y el tipo de valor de retorno.

Sección 6.20 Recursividad

- Una función recursiva (pág. 248) es una función que se llama a sí misma, ya sea en forma directa o indirecta.
- Una función recursiva sabe cómo resolver sólo el (los) caso(s) más simple(s), o caso(s) base. Si se hace la llamada a la función con un caso base (pág. 249), ésta simplemente devuelve un resultado.
- Si se hace la llamada a la función con un problema más complejo, la función comúnmente divide el problema en dos piezas conceptuales: una que sabe cómo resolver y otra que no sabe cómo. Para que la recursividad sea factible, esta última pieza debe ser similar al problema original, pero una versión ligeramente más sencilla o simple del mismo.
- Para que la recursividad termine, la secuencia llamadas recursivas (pág. 249) debe converger en el caso base.
- El nuevo tipo `unsigned long long int` de C++11 (que puede abreviarse como `unsigned long long`) en algunos sistemas nos permite almacenar valores en 8 bytes (64 bits), que pueden contener números tan grandes como 18 446 744 073 709 551 615.

Sección 6.21 Ejemplo sobre el uso de la recursividad: serie de Fibonacci

- La proporción de números de Fibonacci sucesivos converge en un valor constante de 1.618… Este número también ocurre con frecuencia en la naturaleza, y se le ha denominado proporción dorada, o media dorada (pág. 252).

Sección 6.22 Comparación entre recursividad e iteración

- La iteración (pág. 249) y la recursividad tienen muchas similitudes: ambas se basan en una instrucción de control, implican la repetición, implican una prueba de terminación, se acercan gradualmente a la terminación y pueden ocurrir infinitamente.
- La recursividad invoca al mecanismo en forma repetida, y en consecuencia se produce una sobrecarga de las llamadas a la función. Cada llamada recursiva (pág. 249) crea otra copia de las variables de la función; este conjunto de copias puede consumir una cantidad considerable de espacio en memoria.

Ejercicios de autoevaluación

6.1 Complete las siguientes oraciones:

a) En C++, los componentes de un programa se llaman _____ y _____.
b) Una función se invoca con un(a) _____.
c) A una variable que se conoce sólo dentro de la función en la que está declarada, se le llama _____.
d) La instrucción _____ en una función a la que se llamó pasa el valor de una expresión, de vuelta a la función que hizo la llamada.
e) La palabra clave _____ se utiliza en un encabezado de función para indicar que una función no devuelve ningún valor, o para indicar que esa función no contiene parámetros.
f) El _____ de un identificador es la porción del programa en la que puede usarse.
g) Las tres formas de regresar el control de una llamada a una función a la función que la llamó son _____, _____ y _____.
h) Un _____ permite al compilador comprobar el número, tipos y orden de los argumentos que se pasan a una función.
i) La función _____ se utiliza para producir números aleatorios.
j) La función _____ se utiliza para establecer la semilla de números aleatorios, para randomizar la secuencia de números generada por la función `rand`.

k) El especificador de clase de almacenamiento _____ es una recomendación que se hace al compilador para que almacene una variable en uno de los registros de la computadora.
l) Una variable que se declara fuera de cualquier bloque o función es una variable _____.
m) Para que una variable local en una función retenga su valor entre las llamadas a la función, debe declararse con el especificador de clase de almacenamiento _____.
n) Una función que se llama a sí misma, ya sea en forma directa o indirecta (a través de otra función), es una función _____.
o) Por lo general, una función recursiva tiene dos componentes: uno que proporciona el medio para que termine la recursividad, al evaluar un caso _____, y uno que expresa el problema como una llamada recursiva para un problema más simple que el de la llamada original.
p) Es posible tener varias funciones con el mismo nombre, que operen con distintos tipos o números de argumentos. A esto se le conoce como _____ de funciones.
q) El _____ permite acceder a una variable global con el mismo nombre que una variable en el alcance actual.
r) El calificador _____ se usa para declarar variables de sólo lectura.
s) Una _____ de función permite definir una sola función para realizar una tarea en muchos tipos de datos distintos.

6.2 Para el programa de la figura 6.33, indique el alcance (ya sea de función, de espacio de nombres global, de bloque o de prototipo de función) de cada uno de los siguientes elementos:

a) La variable `x` en `main`.
b) La variable `y` en `cubo`.
c) La función `cubo`.
d) La función `main`.
e) El prototipo de función para `cubo`.
f) El identificador `y` en el prototipo de función para `cubo`.

```
1  // Ejercicio 6.2: ej06_02.cpp
2  #include <iostream>
3  using namespace std;
4
5  int cubo( int y ); // prototipo de función
6
7  int main()
8  {
9     int x = 0;
10
11    for ( x = 1; x <= 10; x++ ) // itera 10 veces
12       cout << cubo( x ) << endl; // calcula el cubo de x e imprime los resultados
13 } // fin de main
14
15 // definición de la función cubo
16 int cubo( int y )
17 {
18    return y * y * y;
19 } // fin de la función cubo
```

Fig. 6.33 | Programa para el ejercicio 6.2.

6.3 Escriba un programa que pruebe si los ejemplos de las llamadas a las funciones matemáticas de la biblioteca que se muestran en la figura 6.2 realmente producen los resultados indicados.

6.4 Proporcione el encabezado para cada una de las siguientes funciones:

a) La función `hipotenusa`, que toma dos argumentos de punto flotante con doble precisión, llamados `lado1` y `lado2`, y que devuelve un resultado de punto flotante, con doble precisión.
b) La función `menor`, que toma tres enteros `x`, `y` y `z`, y devuelve un entero.
c) La función `instrucciones`, que no recibe argumentos y no devuelve ningún valor. [*Nota:* dichas funciones se utilizan comúnmente para mostrar instrucciones a un usuario].

d) La función `intADouble`, que recibe un argumento entero llamado `numero` y devuelve un resultado de punto flotante, con doble precisión.

6.5 Proporcione el prototipo de función (sin nombres de parámetros) para cada una de las siguientes situaciones:

a) La función descrita en el ejercicio 6.4(a).
b) La función descrita en el ejercicio 6.4(b).
c) La función descrita en el ejercicio 6.4(c).
d) La función descrita en el ejercicio 6.4(d).

6.6 Escriba una declaración para cada uno de los siguientes casos:

a) Una variable `int` llamada `cuenta`, que deba mantenerse en un registro. Inicialice `cuenta` en `0`.
b) La variable de punto flotante con doble precisión llamada `ultimoVal`, que debe retener su valor entre las llamadas a la función en la que está definida.

6.7 Encuentre el o los errores en cada uno de los siguientes segmentos de programas, y explique cómo se pueden corregir (vea también el ejercicio 6.47):

a)
```
int g()
{
   cout << "Dentro de la funcion g" << endl;
   int h()
   {
      cout << "Dentro de la función h" << endl;
   }
}
```

b)
```
int suma( int x, int y )
{
   int resultado = 0;

   resultado = x + y;
}
```

c)
```
int suma( int n )
{
   if ( 0 == n )
      return 0;
   else
      n + suma( n - 1 );
}
```

d)
```
void f( double a );
{
   float a;
   cout << a << endl;
}
```

e)
```
void producto()
{
   int a = 0;
   int b = 0;
   int c = 0;
   cout << "Escribe tres enteros: ";
   cin >> a >> b >> c;
   int resultado = a * b * c;
   cout << "El resultado es " << resultado;
   return resultado;
}
```

6.8 ¿Por qué un prototipo de función podría contener la declaración del tipo de un parámetro, como `double &`?

6.9 *(Verdadero/Falso)* Todos los argumentos a las llamadas a funciones en C++ se pasan por valor.

6.10 Escriba un programa completo que pida al usuario el radio de una esfera, calcule e imprima el volumen de esa esfera. Use una función `inline` llamada `volumenEsfera` que devuelva el resultado de la siguiente expresión: `(4.0 / 3.0 * 3.14159 * pow(radio, 3))`.

Respuestas a los ejercicios de autoevaluación

6.1 a) funciones, clases. b) llamada a una función. c) variable local. d) `return`. e) `void`. f) alcance. g) `return;` `return` *expresión*; o encontrar la llave derecha de cierre de una función. h) prototipo de función. i) `rand`. j) `srand`. k) `register`. l) global. m) `static`. n) recursiva. o) base. p) sobrecarga q) operador de resolución de ámbito unario (`::`). r) `const`. s) plantilla.

6.2 a) alcance de bloque. b) alcance de bloque. c) alcance de espacio de nombres global. d) alcance de espacio de nombres global. e) alcance de espacio de nombres global. f) alcance de prototipo de función.

6.3 Vea el siguiente programa:

```
1  // Ejercicio 6.3: ej06_03.cpp
2  // Prueba de las funciones matemáticas de la biblioteca.
3  #include <iostream>
4  #include <iomanip>
5  #include <cmath>
6  using namespace std;
7
8  int main()
9  {
10   cout << fixed << setprecision( 1 );
11
12   cout << "sqrt(" << 9.0 << ") = " << sqrt( 9.0 );
13   cout << "\nexp(" << 1.0 << ") = " << setprecision( 6 )
14     << exp( 1.0 ) << "\nexp(" << setprecision( 1 ) << 2.0
15     << ") = " << setprecision( 6 ) << exp( 2.0 );
16   cout << "\nlog(" << 2.718282 << ") = " << setprecision( 1 )
17     << log( 2.718282 )
18     << "\nlog(" << setprecision( 6 ) << 7.389056 << ") = "
19     << setprecision( 1 ) << log( 7.389056 );
20   cout << "\nlog10(" << 10.0 << ") = " << log10( 10.0 )
21     << "\nlog10(" << 100.0 << ") = " << log10( 100.0 ) ;
22   cout << "\nfabs(" << 5.1 << ") = " << fabs( 5.1 )
23     << "\nfabs(" << 0.0 << ") = " << fabs( 0.0 )
24     << "\nfabs(" << -8.76 << ") = " << fabs( -8.76 );
25   cout << "\nceil(" << 9.2 << ") = " << ceil( 9.2 )
26     << "\nceil(" << -9.8 << ") = " << ceil( -9.8 );
27   cout << "\nfloor(" << 9.2 << ") = " << floor( 9.2 )
28     << "\nfloor(" << -9.8 << ") = " << floor( -9.8 );
29   cout << "\npow(" << 2.0 << ", " << 7.0 << ") = "
30     << pow( 2.0, 7.0 ) << "\npow(" << 9.0 << ", "
31     << 0.5 << ") = " << pow( 9.0, 0.5 );
32   cout << setprecision( 3 ) << "\nfmod("
33     << 2.6 << ", " << 1.2 << ") = "
34     << fmod( 2.6, 1.2 ) << setprecision( 1 );
35   cout << "\nsin(" << 0.0 << ") = " << sin( 0.0 );
36   cout << "\ncos(" << 0.0 << ") = " << cos( 0.0 );
37   cout << "\ntan(" << 0.0 << ") = " << tan( 0.0 ) << endl;
38 } // fin de main
```

```
sqrt(9.0) = 3.0
exp(1.0) = 2.718282
exp(2.0) = 7.389056
log(2.718282) = 1.0
log(7.389056) = 2.0
log10(10.0) = 1.0
log10(100.0) = 2.0
fabs(5.1) = 5.1
fabs(0.0) = 0.0
fabs(-8.8) = 8.8
ceil(9.2) = 10.0
ceil(-9.8) = -9.0
floor(9.2) = 9.0
floor(-9.8) = -10.0
pow(2.0, 7.0) = 128.0
pow(9.0, 0.5) = 3.0
fmod(2.600, 1.200) = 0.200
sin(0.0) = 0.0
cos(0.0) = 1.0
tan(0.0) = 0.0
```

6.4 a) `double hipotenusa( double lado1, double lado2 )`
b) `int menor( int x, int y, int z )`
c) `void instrucciones()`
d) `double intADouble( int numero )`

6.5 a) `double hipotenusa( double, double );`
b) `int menor( int, int, int );`
c) `void instrucciones();`
d) `double intADouble( int );`

6.6 a) `register int cuenta = 0;`
b) `static double ultimoVal;`

6.7 a) *Error:* la función `h` está definida en la función `g`.
Corrección: mueva la definición de `h` fuera de la definición de `g`.
b) *Error:* se supone que la función debe devolver un entero, pero no es así.
Corrección: coloque una instrucción `return resultado;` al final del cuerpo de la función, o elimine la variable `resultado` y coloque la siguiente instrucción en la función:

```
return x + y;
```

c) *Error:* no se devuelve el resultado de `n + suma( n - 1 );` `suma` devuelve un resultado incorrecto.
Corrección: vuelva a escribir la instrucción en la cláusula `else` de la siguiente manera:

```
return n + sum( n - 1 );
```

d) *Errores:* el punto y coma que va después del paréntesis derecho de la lista de parámetros y la redefinición del parámetro a en la definición de la función.
Correcciones: elimine el punto y coma que va después del paréntesis derecho de la lista de parámetros, y elimine la declaración `float a;`.
e) *Error:* la función devuelve un valor cuando no debe hacerlo.
Corrección: elimine la instrucción `return` o cambie el tipo de valor de retorno.

6.8 Esto crea un parámetro de referencia de tipo "referencia a `double`", el cual permite que la función modifique la variable original en la función que hace la llamada.

6.9 Falso. C++ permite el paso por referencia mediante el uso de parámetros por referencia (y apuntadores, como veremos en el capítulo 8).

6.10 Vea el siguiente programa:

```
// Solución al ejercicio 6.10: Ej06_10.cpp
// Función en línea que calcula el volumen de una esfera.
#include <iostream>
#include <cmath>
using namespace std;

const double PI = 3.14159; // define la constante global PI

// calcula el volumen de una esfera
inline double volumenEsfera( const double radio )
{
   return 4.0 / 3.0 * PI * pow( radio, 3 );
} // fin de la función en línea volumenEsfera

int main()
{
   double valorRadio = 0;

   // pide el radio al usuario
   cout << "Escriba la longitud del radio de su esfera: ";
   cin >> valorRadio; // recibe el radio

   // usa valorRadio para calcular el volumen de la esfera y mostrar el resultado
   cout << "El volumen de la esfera con radio "  << valorRadio
      << " es " << volumenEsfera( valorRadio ) << endl;
} // fin de main
```

Ejercicios

6.11 Muestre el valor de x después de ejecutar cada una de las siguientes instrucciones:

a) `x = fabs( 7.5 )`
b) `x = floor( 7.5 )`
c) `x = fabs( 0.0 )`
d) `x = ceil( 0.0 )`
e) `x = fabs( -6.4 )`
f) `x = ceil( -6.4 )`
g) `x = ceil( -fabs( -8 + floor( -5.5 ) ) )`

6.12 ***(Cargos de estacionamiento)*** Un estacionamiento cobra una cuota mínima de $2.00 por estacionarse hasta tres horas. El estacionamiento cobra $0.50 adicionales por cada hora o *fracción* que se pase de tres horas. El cargo máximo para cualquier periodo dado de 24 horas es de $10.00. Suponga que ningún automóvil se estaciona durante más de 24 horas a la vez. Escriba un programa que calcule e imprima los cargos por estacionamiento para cada uno de tres clientes que estacionaron su automóvil ayer en este estacionamiento. Debe introducir las horas de estacionamiento para cada cliente. El programa debe imprimir los resultados en un formato tabular ordenado, debe calcular e imprimir el total de los recibos de ayer. El programa debe utilizar la función `calcularCargos` para determinar el cargo para cada cliente. Sus resultados deben aparecer en el siguiente formato:

```
Automóvil      Horas        Cargo
1                1.5         2.00
2                4.0         2.50
3               24.0        10.00
TOTAL           29.5        14.50
```

6.13 ***(Redondeo de números)*** Una aplicación de la función `floor` es redondear un valor al siguiente entero. La instrucción

```
y = floor( x + 0.5 );
```

redondea el número `x` al entero más cercano y asigna el resultado a `y`. Escriba un programa que lea varios números y que utilice la instrucción anterior para redondear cada uno de los números a su entero más cercano. Para cada número procesado, muestre tanto el número original como el redondeado.

6.14 ***(Redondeo de números)*** La función `floor` puede utilizarse para redondear un número hasta un lugar decimal específico. La instrucción

```
y = floor( x * 10 + 0.5 ) / 10;
```

redondea `x` en la posición de las décimas (la primera posición a la derecha del punto decimal). La instrucción

```
y = floor( x * 100 + 0.5 ) / 100;
```

redondea `x` en la posición de las centésimas (la segunda posición a la derecha del punto decimal). Escriba un programa que defina cuatro funciones para redondear un número `x` en varias formas:

a) `redondearAEntero( numero )`
b) `redondearADecimas( numero )`
c) `redondearACentesimas( numero )`
d) `redondearAMilesimas( numero )`

Para cada valor leído, su programa debe imprimir el valor original, el número redondeado al entero más cercano, el número redondeado a la décima más cercana, el número redondeado a la centésima más cercana y el número redondeado a la milésima más cercana.

6.15 ***(Preguntas con respuestas cortas)*** Responda a cada una de las siguientes preguntas:

a) ¿Qué significa elegir números "al azar"?
b) ¿Por qué es la función `rand` útil para simular juegos al azar?
c) ¿Por qué se debe randomizar un programa mediante `srand`? ¿Bajo qué circunstancias es aconsejable no randomizar?
d) ¿Por qué a menudo es necesario escalar o desplazar los valores producidos por `rand`?
e) ¿Por qué es la simulación computarizada de las situaciones reales una técnica útil?

6.16 ***(Números aleatorios)*** Escriba instrucciones que asignen enteros aleatorios a la variable n en los siguientes rangos:

a) $1 \leq n \leq 2$
b) $1 \leq n \leq 100$
c) $0 \leq n \leq 9$
d) $1000 \leq n \leq 1112$
e) $-1 \leq n \leq 1$
f) $-3 \leq n \leq 11$

6.17 ***(Números aleatorios)*** Escriba una sola instrucción que imprima un número al azar de cada uno de los siguientes conjuntos:

a) 2, 4, 6, 8, 10.
b) 3, 5, 7, 9, 11.
c) 6, 10, 14, 18, 22.

6.18 ***(Exponenciación)*** Escriba una función llamada `enteroPotencia(` *base, exponente* `)` que devuelva el valor de

$base^{exponente}$

Por ejemplo, `enteroPotencia(3, 4) = 3 * 3 * 3 * 3`. Suponga que *exponente* es un entero positivo distinto de cero y que *base* es un entero. No utilice ninguna función matemática de la biblioteca.

6.19 *(Cálculos de hipotenusa)* Defina una función llamada `hipotenusa` que calcule la hipotenusa de un triángulo recto, cuando se proporcionen las longitudes de los otros dos lados. La función debe recibir dos argumentos `double` y devolver la hipotenusa como `double`. Use esta función en un programa para determinar la hipotenusa para cada uno de los triángulos que se muestran a continuación.

Triángulo	Lado 1	Lado 2
1	3.0	4.0
2	5.0	12.0
3	8.0	15.0

6.20 *(Múltiples)* Escriba una función llamada `multiple` que determine, para un par de enteros, si el segundo entero es múltiplo del primero. La función debe tomar dos argumentos enteros y devolver `true` si el segundo es múltiplo del primero, y `false` en caso contrario. Use esta función en un programa que reciba como entrada una serie de pares de enteros.

6.21 *(Números pares)* Escriba un programa que reciba una serie de enteros y los pase, uno a la vez, a una función llamada `esPar`, que utilice el operador módulo para determinar si un entero dado es par. La función debe tomar un argumento entero y devolver `true` si el entero es par, y `false` en caso contrario.

6.22 *(Cuadrado de asteriscos)* Escriba una función que muestre en el margen izquierdo de la pantalla un cuadrado relleno de asteriscos, cuyo lado se especifique en el parámetro entero `lado`. Por ejemplo, si `lado` es `4`, la función debe mostrar lo siguiente:

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

6.23 *(Cuadrado de cualquier carácter)* Modifique la función creada en el ejercicio 6.22 para formar el cuadrado de cualquier carácter que esté contenido en el parámetro tipo carácter `caracterRelleno`. Por ejemplo, si `lado` es `5` y `caracterRelleno` es "`#`", esta función debe imprimir lo siguiente:

```
#####
#####
#####
#####
#####
```

6.24 *(Separar dígitos)* Escriba segmentos de programas que realicen cada una de las siguientes tareas:

a) Calcular la parte entera del cociente, cuando el entero `a` se divide entre el entero `b`.
b) Calcular el residuo entero cuando el entero `a` se divide entre el entero `b`.
c) Utilizar las piezas de los programas desarrollados en las partes (a) y (b) para escribir una función que reciba un entero entre `1` y `32767`, y que lo imprima como una serie de dígitos, separando cada par de dígitos por dos espacios. Por ejemplo, el entero `4562` debe imprimirse de la siguiente manera:

```
4  5  6  2
```

6.25 *(Calcular el número de segundos)* Escriba una función que reciba la hora en forma de tres argumentos enteros (horas, minutos y segundos) y devuelva el número de segundos transcurridos desde la última vez que el reloj "marcó las 12". Use esta función para calcular el monto de tiempo en segundos entre dos horas, las cuales deben estar dentro de un ciclo de 12 horas del reloj.

6.26 ***(Temperaturas en Centígrados y Fahrenheit)*** Implemente las siguientes funciones enteras:

a) La función `centigrados` que devuelve la equivalencia en grados Centígrados de una temperatura en grados Fahrenheit.

b) La función `fahrenheit` que devuelve la equivalencia en grados Fahrenheit de una temperatura en grados Centígrados.

c) Utilice estas funciones para escribir un programa que imprima gráficos que muestren los equivalentes en grados Fahrenheit de todas las temperaturas en grados Centígrados, desde 0 hasta 100, y los equivalentes en grados Centígrados de todas las temperaturas en grados Fahrenheit, desde 32 hasta 212. Imprima los resultados en un formato tabular ordenado que minimice el número de líneas de salida, al tiempo que permanezca legible.

6.27 ***(Encontrar el mínimo)*** Escriba un programa que reciba tres números de punto flotante de doble precisión, y que los pase a una función que devuelva el número más pequeño.

6.28 ***(Números perfectos)*** Se dice que un número entero es un *número perfecto* si la suma de sus divisores, incluyendo 1 (pero no el número en sí), es igual al número. Por ejemplo, 6 es un número perfecto ya que 6 = 1 + 2 + 3. Escriba una función llamada `esPerfecto` que determine si el parámetro `numero` es un número perfecto. Use esta función en un programa que determine e imprima todos los números perfectos entre 1 y 1000. Imprima los divisores de cada número perfecto para confirmar que el número sea realmente perfecto. Ponga a prueba el poder de su computadora, evaluando números mucho más grandes que 1000.

6.29 ***(Números primos)*** Se dice que un entero es *primo* si puede dividirse solamente por 1 y por sí mismo. Por ejemplo, 2, 3, 5 y 7 son primos, pero 4, 6, 8 y 9 no.

a) Escriba una función que determine si un número es primo.

b) Use esta función en un programa que determine e imprima todos los números primos entre 2 y 10,000. ¿Cuántos de estos números hay que probar realmente para asegurarse de encontrar todos los números primos?

c) Al principio podría pensarse que $n/2$ es el límite superior para evaluar si un número es primo, pero lo máximo que se necesita es ir hasta la raíz cuadrada de n. ¿Por qué? Vuelva a escribir el programa y ejecútelo de ambas formas. Estime la mejora en el rendimiento.

6.30 ***(Dígitos inversos)*** Escriba una función que reciba un valor entero y devuelva el número con sus dígitos invertidos. Por ejemplo, para el número 7631, la función debe regresar 1367.

6.31 ***(Máximo común divisor)*** El *máximo común divisor (MCD)* de dos enteros es el entero más grande que puede dividir uniformemente a cada uno de los dos números. Escriba una función llamada `mcd` que devuelva el máximo común divisor de dos enteros.

6.32 ***(Puntos de calidad para calificaciones numéricas)*** Escriba una función llamada `puntosCalidad` que reciba como entrada el promedio de un estudiante y devuelva `4` si el promedio se encuentra entre `90` y `100`, `3` si el promedio se encuentra entre `80` y `89`, `2` si el promedio se encuentra entre `70` y `79`, `1` si el promedio se encuentra entre `60` y `69`, y `0` si el promedio es menor de `60`.

6.33 ***(Lanzar monedas)*** Escriba un programa que simule el lanzamiento de monedas. Cada vez que se lance la moneda, el programa debe imprimir `Cara` o `Cruz`. Deje que el programa lance la moneda 100 veces y cuente el número de veces que aparezca cada uno de los lados de la moneda. Imprima los resultados. El programa debe llamar a una función separada llamada `tirar`, que no reciba argumentos y devuelva `0` en caso de cara y `1` en caso de cruz. [*Nota*: si el programa simula en forma realista el lanzamiento de monedas, cada lado de la moneda debe aparecer aproximadamente la mitad del tiempo`].

6.34 ***(Juego "Adivina el número")*** Escriba un programa que juegue a "adivina el número" de la siguiente manera: su programa elige el número a adivinar, seleccionando un entero aleatorio en el rango de 1 a 1000. Después, el programa muestra lo siguiente:

```
Tengo un numero entre 1 y 1000.
Puedes adivinar mi numero?
Por favor escribe tu primera respuesta.
```

El jugador escribe su primer intento. El programa responde con uno de los siguientes mensajes:

```
1. Excelente!  Adivinaste el numero!
   Te gustaria jugar de nuevo (s/n)?
2. Demasiado bajo. Intenta de nuevo.
3. Demasiado alto. Intenta de nuevo.
```

Si la respuesta del jugador es incorrecta, su programa deberá iterar hasta que el jugador adivine correctamente. Su programa deberá seguir indicando al jugador los mensajes `Demasiado alto` o `Demasiado bajo`, para ayudar a que el jugador "se acerque" a la respuesta correcta.

6.35 ***(Modificación del juego "Adivina el número")*** Modifique el programa del ejercicio 6.34 para contar el número de intentos que haga el jugador. Si el número es 10 o menos, imprima el mensaje "`O ya sabia usted el secreto, o tuvo suerte!`". Si el jugador adivina el número en 10 intentos, imprima el mensaje "`Aja! Sabia usted el secreto!`". Si el jugador hace más de 10 intentos, imprima el mensaje "`Deberia haberlo hecho mejor!`". ¿Por qué no se deben requerir más de 10 intentos? Bueno, en cada "buen intento", el jugador debe poder eliminar la mitad de los números. Ahora muestre por qué cualquier número de 1 a 1000 puede adivinarse en 10 intentos o menos.

6.36 ***(Exponenciación recursiva)*** Escriba una función recursiva llamada `potencia( base, exponente )` que, cuando sea llamada, devuelva

$$base^{exponente}$$

Por ejemplo, `potencia( 3, 4 ) = 3 * 3 * 3 * 3`. Suponga que `exponente` es un entero mayor o igual que `1`. *Sugerencia:* el paso recursivo debe utilizar la relación

$$base^{exponente} = base \cdot base^{exponente - 1}$$

y la condición de terminación ocurre cuando `exponente` es igual a `1`, ya que

$$base^{1} = base$$

6.37 ***(Serie de Fibonacci: solución iterativa)*** Escriba una versión *no recursiva* de la función `fibonacci` de la figura 6.29.

6.38 ***(Torres de Hanoi)*** En este capítulo estudiamos funciones que pueden implementarse con facilidad, tanto en forma recursiva como iterativa. En este ejercicio presentamos un problema cuya solución recursiva demuestra la elegancia de la recursividad, y cuya solución iterativa tal vez no sea tan aparente.

Las **Torres de Hanoi** son uno de los problemas clásicos más famosos con los que todo científico computacional en ciernes tiene que lidiar. Cuenta la leyenda que en un templo del Lejano Oriente, los sacerdotes intentan mover una pila de discos dorados, de una aguja de diamante a otra (figura 6.34). La pila inicial tiene 64 discos insertados en una aguja y se ordenan de abajo hacia arriba, de mayor a menor tamaño. Los sacerdotes intentan mover la pila de una aguja a otra, con las restricciones de que sólo se puede mover un disco a la vez, y en ningún momento se puede colocar un disco más grande encima de uno más pequeño. Se cuenta con tres agujas, una de las cuales se utiliza para almacenar discos temporalmente. Se supone que el mundo acabará cuando los sacerdotes completen su tarea, por lo que hay pocos incentivos para que nosotros podamos facilitar sus esfuerzos.

Vamos a suponer que los sacerdotes intentan mover los discos de la aguja 1 a la aguja 3. Deseamos desarrollar un algoritmo que imprima la secuencia precisa de transferencias de los discos de una aguja a otra.

Si atacamos este problema con los métodos convencionales, rápidamente terminaríamos "atados" manejando los discos sin esperanza. En vez de ello, si atacamos este problema teniendo en mente la recursividad, los pasos serán más simples. La acción de mover n discos puede verse en términos de mover sólo $n - 1$ discos (de ahí la recursividad) de la siguiente forma:

a) Mover $n - 1$ discos de la aguja 1 a la aguja 2, usando la aguja 3 como un área de almacenamiento temporal.
b) Mover el último disco (el más grande) de la aguja 1 a la aguja 3.
c) Mover $n - 1$ discos de la aguja 2 a la aguja 3, usando la aguja 1 como área de almacenamiento temporal.

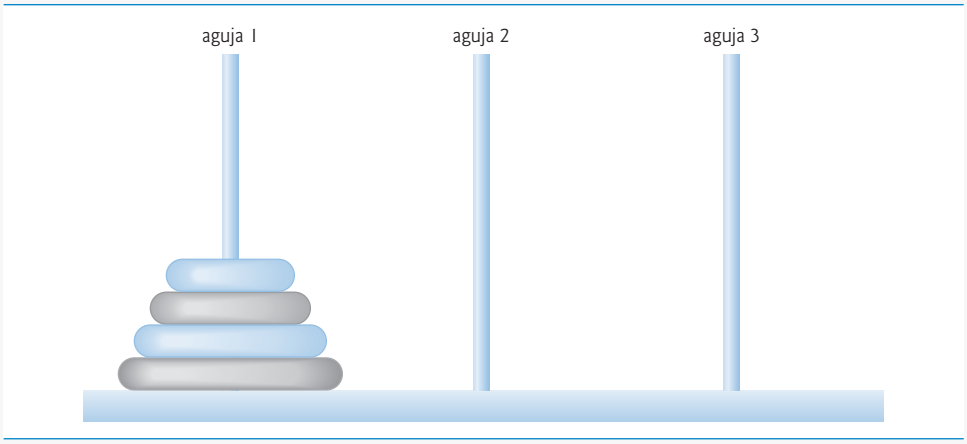


Fig. 6.34 | Las Torres de Hanoi para el caso con cuatro discos.

El proceso termina cuando la última tarea implica mover $n = 1$ disco (es decir, el caso base). Esta tarea se logra con sólo mover el disco, sin necesidad de un área de almacenamiento temporal. Escriba un programa para resolver el problema de las Torres de Hanoi. Use una función recursiva con cuatro parámetros:

a) El número de discos a mover.
b) La aguja en la que están insertados estos discos en un principio.
c) La aguja a la que se va a mover esta pila de discos.
d) La aguja que se va a utilizar como área de almacenamiento temporal.

Imprima las instrucciones precisas para mover los discos de la aguja inicial a la aguja de destino. Para mover una pila de tres discos de la aguja 1 a la aguja 3, el programa muestra la siguiente serie de movimientos:

1 → 3 (Esto significa mover un disco de la aguja 1 a la aguja 3).
1 → 2
3 → 2
1 → 3
2 → 1
2 → 3
1 → 3

6.39 ***(Torres de Hanoi: versión iterativa)*** Cualquier programa que se pueda implementar en forma recursiva se puede implementar en forma iterativa, aunque algunas veces con mayor dificultad y menor claridad. Pruebe escribir una versión iterativa de las Torres de Hanoi. Si tiene éxito, compare su versión iterativa con la versión recursiva desarrollada en el ejercicio 6.38. Investigue las cuestiones relacionadas con el rendimiento, la claridad y su habilidad de demostrar que los programas estén correctos.

6.40 ***(Visualización de la recursividad)*** Es interesante observar la recursividad "en acción". Modifique la función factorial de la figura 6.28 para imprimir su variable local y su parámetro de llamada recursiva. Para cada llamada recursiva, muestre los resultados en una línea separada y agregue un nivel de sangría. Haga su máximo esfuerzo por hacer que los resultados sean claros, interesantes y significativos. Su meta aquí es diseñar e implementar un formato de salida que ayude a una persona a comprender mejor la recursividad. Tal vez desee agregar dichas capacidades de visualización a otros ejemplos y ejercicios recursivos a lo largo de este libro.

6.41 ***(Máximo común divisor recursivo)*** El máximo común divisor de los enteros `x` y `y` es el entero más grande que se puede dividir entre `x` y `y` de manera uniforme. Escriba una función recursiva llamada `mcd`, que devuelva el máximo común divisor de `x` y `y`, definida mediante la recursividad, de la siguiente manera: si `y` es igual a `0`, entonces `mcd(x,y)` es `x`; en caso contrario, `mcd(x,y)` es `mcd(y, x % y)`, en donde `%` es el operador módulo. [*Nota:* para este algoritmo, `x` debe ser mayor que `y`].

6.42 ***(Distancia entre puntos)*** Escriba una función llamada `distancia` que calcule la distancia entre dos puntos *(x1, y1)* y *(x2, y2)*. Todos los números y valores de retorno deben ser de tipo `double`.

6.43 ¿Qué error tiene el siguiente programa?

```
1  // Ejercicio 6.43: ej06_43.cpp
2  // ¿Qué error tiene este programa?
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int c = 0;
9
10    if ( ( c = cin.get() ) != EOF )
11    {
12       main();
13       cout << c;
14    } // fin de if
15 } // fin de main
```

6.44 ¿Qué hace el siguiente programa?

```
1  // Ejercicio 6.44: ej06_44.cpp
2  // ¿Qué hace este programa?
3  #include <iostream>
4  using namespace std;
5
6  int misterio( int, int ); // prototipo de función
7
8  int main()
9  {
10    int x = 0;
11    int y = 0;
12
13    cout << "Escriba dos enteros: ";
14    cin >> x >> y;
15    cout << "El resultado es " << misterio( x, y ) << endl;
16 } // fin de main
17
18 // el parámetro b debe ser un entero positivo para prevenir la recursividad infinita
19 int misterio( int a, int b )
20 {
21    if ( 1 == b ) // caso base
22       return a;
23    else // paso recursivo
24       return a + misterio( a, b - 1 );
25 } // fin de la función misterio
```

6.45 Una vez que determine qué es lo que hace el programa del ejercicio 6.44, modifíquelo para que funcione de manera apropiada, después de eliminar la restricción de que el segundo argumento no debe ser negativo.

6.46 ***(Funciones matemáticas de la biblioteca)*** Escriba un programa que evalúe todas, si es posible, las funciones matemáticas de la biblioteca en la figura 6.2. Ejercite cada una de estas funciones, haciendo que su programa imprima tablas de valores de retornos para una diversidad de valores de los argumentos.

6.47 ***(Encuentre el error)*** Encuentre el error en cada uno de los siguientes segmentos de programa, y explique cómo corregirlo:

a)
```
float cubo( float ); // prototipo de función

cubo( float numero ) // definición de función
{
   return numero * numero * numero;
}
```
b)
```
int numeroAleatorio = srand();
```
c)
```
float y = 123.45678;
int x;

x = y;
cout << static_cast< float >( x ) << endl;
```
d)
```
double cuadrado( double number )
{
   double numero = 0;
   return numero * numero;
}
```
e)
```
int suma( int n )
{
   if ( 0 == n )
      return 0;
   else
      return n + suma( n );
}
```

(Modificación del juego de Craps) Modifique el programa Craps de la figura 6.11 para permitir apuestas. Empaquete como función la parte del programa que ejecuta un juego de craps. Inicialice la variable `saldoBanco` con 1000 dólares. Pida al jugador que introduzca una `apuesta`. Use un ciclo `while` para comprobar que esa `apuesta` sea menor o igual al `saldoBanco` y, si no lo es, haga que el usuario vuelva a introducir la `apuesta` hasta que se introduzca un valor válido. Después de esto, comience un juego de craps. Si el jugador gana, agregue la `apuesta` al `saldoBanco` e imprima el nuevo `saldoBanco`. Si el jugador pierde, reste la `apuesta` al `saldoBanco`, imprima el nuevo `saldoBanco`, compruebe si `saldoBanco` se ha vuelto cero y, de ser así, imprima el mensaje `"Lo siento. Se quedo sin fondos!"`. A medida que el juego progrese, imprima varios mensajes para crear algo de "charla", como `"Oh, se esta yendo a la quiebra, verdad?"`, o `"Oh, vamos, arriesguese!"`, o `"La hizo en grande. Ahora es tiempo de cambiar sus fichas por efectivo!"`.

6.48 *(Área de círculo)* Escriba un programa en C++ que pida al usuario el radio de un círculo y después llame a la función `inline areaCirculo` para calcular el área de ese círculo.

6.49 *(Paso por valor y paso por referencia)* Escriba un programa completo en C++ con las dos funciones alternativas que se especifican a continuación, de las cuales cada una simplemente triplica la variable `cuenta` definida en `main`. Después compare y contraste ambos métodos. Estas dos funciones son:

a) la función `triplicarPorValor`, que pasa una copia de `cuenta` por valor, triplica la copia y devuelve el nuevo valor, y
b) la función `triplicarPorReferencia`, que pasa `cuenta` por referencia a través de un parámetro por referencia y triplica el valor original de `cuenta` a través de su alias (es decir, el parámetro por referencia).

6.50 ¿Cuál es el propósito del operador de resolución de ámbito unario?

6.51 *(Plantilla de función `minimo`)* Escriba un programa que use una plantilla de función llamada `minimo` para determinar el menor de dos argumentos. Pruebe el programa usando argumentos tipo entero, carácter y número de punto flotante.

6.52 ***(Plantilla de función `maximo`)*** Escriba un programa que utilice una plantilla de función llamada `maximo` para determinar el mayor de dos argumentos. Pruebe el programa usando argumentos tipo entero, carácter y número de punto flotante.

6.53 ***(Encuentre el error)*** Determine si los siguientes segmentos de programa contienen errores. Para cada error, explique cómo puede corregirse. [*Nota:* para un segmento de programa específico, es posible que no haya errores presentes en el segmento].

a)
```
template < class A >
int suma( int num1, int num2, int num3 )
{
   return num1 + num2 + num3;
}
```

b)
```
void imprimirResultados( int x, int y )
{
   cout << "La suma es " << x + y << '\n';
   return x + y;
}
```

c)
```
template < A >
A producto( A num1, A num2, A num3 )
{
   return num1 * num2 * num3;
}
```

d)
```
double cubo( int );
int cubo( int );
```

6.54 ***(Números aleatorios de C++11: juego de craps modificado)*** Modifique el programa de la figura 6.11 para usar las nuevas herramientas de generación de números aleatorios que se muestran en la sección 6.9.

6.55 ***(Enumeración con alcance de C++11)*** Cree una `enum` con alcance llamada `TipoCuenta` que contenga las constantes llamadas `AHORROS`, `CHEQUES` e `INVERSION`.

Hacer la diferencia

A medida que disminuyen los costos de las computadoras, aumenta la posibilidad de que cada estudiante, sin importar su economía, tenga una y la utilice en la escuela. Esto crea excitantes posibilidades para mejorar la experiencia educativa de todos los estudiantes a nivel mundial, según lo sugieren los siguientes cinco ejercicios. [*Nota:* vea nuestras iniciativas, como el proyecto One Laptop Per Child (`www.laptop.org`). Investigue también acerca de las laptops "verdes" o ecológicas y observe las características "ecológicas" clave de estos dispositivos. Investigue también la Herramienta de evaluación ambiental de productos electrónicos (`www.epeat.net`), que le puede ayudar a evaluar las características "ecológicas" de las computadoras de escritorio, notebooks y monitores para poder decidir qué productos comprar].

6.56 ***(Instrucción asistida por computadora)*** El uso de las computadoras en la educación se conoce como *instrucción asistida por computadora (CAI)*. Escriba un programa que ayude a un estudiante de escuela primaria, para que aprenda a multiplicar. Use la función `rand` para producir dos enteros positivos de un dígito. El programa debe entonces mostrar una pregunta al usuario, como:

```
¿Cuánto es 6 por 7?
```

El estudiante entonces debe escribir la respuesta. Luego, el programa debe verificar la respuesta del estudiante. Si es correcta, muestre el mensaje `"Muy bien!"` y haga otra pregunta de multiplicación. Si la respuesta es incorrecta, muestre el mensaje `"No. Por favor intenta de nuevo."` y deje que el estudiante intente la misma pregunta varias veces, hasta que esté correcta. Debe utilizarse una función separada para generar cada pregunta nueva. Esta función debe llamarse una vez cuando la aplicación empiece a ejecutarse, y cada vez que el usuario responda correctamente a la pregunta.

6.57 ***(Instrucción asistida por computadora: reducción de la fatiga de los estudiantes)*** Un problema que se desarrolla en los entornos CAI es la fatiga de los estudiantes. Este problema puede eliminarse si se varían las contestaciones de la computadora para mantener la atención del estudiante. Modifique el programa del ejercicio 6.57 de manera que se muestren diversos comentarios para cada respuesta, de la siguiente manera:

Posibles contestaciones a una respuesta correcta:

```
Muy bien!
Excelente!
Buen trabajo!
Sigue asi!
```

Posibles contestaciones a una respuesta incorrecta:

```
No. Por favor intenta de nuevo.
Incorrecto. Intenta una vez mas.
No te rindas!
No. Sigue intentando.
```

Use la generación de números aleatorios para elegir un número entre 1 y 4 que se utilice para seleccionar una de las cuatro contestaciones apropiadas a cada respuesta correcta o incorrecta. Use una instrucción `switch` para emitir las contestaciones.

6.58 ***(Instrucción asistida por computadora: supervisión del rendimiento de los estudiantes)*** Los sistemas de instrucción asistida por computadora más sofisticados supervisan el rendimiento del estudiante durante cierto tiempo. La decisión de empezar un nuevo tema se basa a menudo en el éxito del estudiante con los temas anteriores. Modifique el programa del ejercicio 6.58 para contar el número de respuestas correctas e incorrectas introducidas por el estudiante. Una vez que el estudiante escriba 10 respuestas, su programa debe calcular el porcentaje de respuestas correctas. Si éste es menor del 75%, imprima `"Por favor pida ayuda adicional a su profesor"` y reinicie el programa, para que otro estudiante pueda probarlo. Si el porcentaje es del 75% o mayor, muestre el mensaje `"Felicidades, esta listo para pasar al siguiente nivel!"` y luego reinicie el programa, para que otro estudiante pueda probarlo.

6.59 ***(Instrucción asistida por computadora: niveles de dificultad)*** En los ejercicios 6.57 al 6.59 se desarrolló un programa de instrucción asistida por computadora para enseñar a un estudiante de escuela primara cómo multiplicar. Modifique el programa para que permita al usuario introducir un nivel de dificultad. Un nivel de 1 significa que el programa debe usar sólo números de un dígito en los problemas; un nivel 2 significa que el programa debe utilizar números de dos dígitos máximo, y así en lo sucesivo.

6.60 ***(Instrucción asistida por computadora: variación de los tipos de problemas)*** Modifique el programa del ejercicio 6.60 para permitir al usuario que elija el tipo de problemas aritméticos que desea estudiar. Una opción de `1` significa problemas de suma solamente, `2` significa problemas de resta, `3` significa problemas de multiplicación, `4` significa problemas de división y `5` significa una mezcla aleatoria de problemas de todos estos tipos.

7

Plantillas de clase `array` y `vector`; cómo atrapar excepciones

Ahora ve, escríbelo
ante ellos en una tabla,
y anótalo en un libro.
—Isaías 30:8

Comienza en el principio…
y continúa hasta que llegues
al final; después detente.
—Lewis Carroll

Ir más allá es tan malo
como no llegar.
—Confucio

Objetivos

En este capítulo aprenderá a:

- Utilizar la plantilla de clase `array` de la Biblioteca estándar de C++: una colección de tamaño fijo de elementos de datos relacionados.
- Utilizar arreglos para almacenar, ordenar y buscar datos en listas y tablas de valores.
- Declarar arreglos, inicializarlos y hacer referencia a elementos individuales de los arreglos.
- Usar la instrucción `for` basada en rango.
- Pasar arreglos a las funciones.
- Declarar y manipular arreglos multidimensionales.
- Utilizar la plantilla de clase `vector` de la Biblioteca estándar de C++: una colección de tamaño variable de elementos de datos relacionados.

Plan general

7.1 Introducción

En este capítulo presentamos el tema de las **estructuras de datos**: *colecciones* de elementos de datos relacionados. Hablaremos sobre los **`arreglos`**, que son colecciones *de tamaño fijo* que consisten de elementos de datos del *mismo* tipo, y de los **`vectores`**, que son colecciones (también de elementos de datos del *mismo* tipo) que pueden aumentar y reducir su tamaño en forma *dinámica* en tiempo de ejecución. Tanto `array` como `vector` son plantillas de clase de la biblioteca estándar de C++. Para usarlas, hay que incluir los encabezados `<array>` y `<vector>`, respectivamente.

Después de hablar acerca de cómo se declaran, se crean y se inicializan los arreglos, presentaremos ejemplos que demuestran varias manipulaciones comunes de ellos. Le mostraremos cómo realizar *búsquedas* en los arreglos para encontrar elementos específicos, y cómo *ordenar* los arreglos para poner sus datos en *orden*.

Mejoraremos la clase `LibroCalificaciones` mediante el uso de arreglos unidimensionales y bidimensionales para mantener un conjunto de calificaciones en memoria y analizar las calificaciones de distintos exámenes. Introduciremos el mecanismo de *manejo de excepciones* y lo usaremos para que un programa pueda seguir ejecutándose cuando intente acceder al elemento de un `vector` o `array` que no exista.

7.2 Arreglos

Un arreglo es un grupo *contiguo* de localidades de memoria, todas ellas del *mismo* tipo. Para hacer referencia a una localidad o elemento específico en el arreglo, especificamos su nombre y el **número de posición** del elemento en el arreglo.

La figura 7.1 muestra un arreglo de enteros llamado `c`, el cual contiene 12 **elementos**. Para hacer referencia a cualquiera de estos elementos en un programa, se proporciona el nombre del arreglo seguido del número de posición del elemento específico entre corchetes (`[]`). Al número de posición se le conoce más formalmente como el **índice** o **subíndice** (este número especifica el número de elementos a partir del inicio del arreglo). El primer elemento tiene el **subíndice 0** (**cero**) y se conoce algunas veces como el **elemento cero**. Por ende, los elementos del `array c` son `c[0]` (se pronuncia como "`c` sub cero"),

c[1], c[2] y así en lo sucesivo. El subíndice más alto en el arreglo c es 11, el cual es 1 menos que el número de elementos en el arreglo (12). Los nombres de los arreglos siguen las mismas convenciones que los demás nombres de variables.

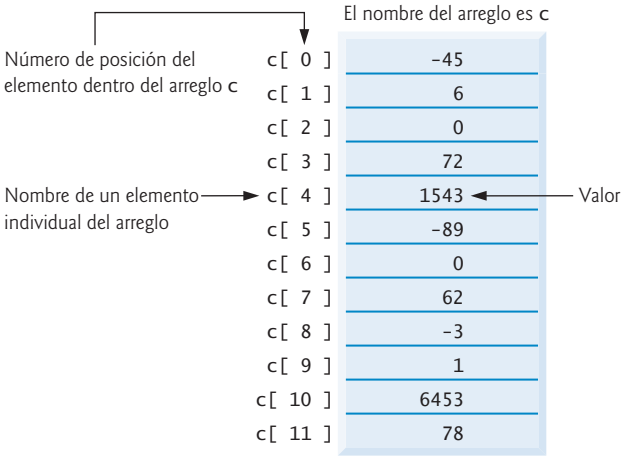


Fig. 7.1 | Un arreglo con 12 elementos.

Un subíndice debe ser un entero o una expresión entera (usando cualquier tipo integral). Si un programa utiliza una expresión como un subíndice, entonces el programa evalúa la expresión para determinar el subíndice. Por ejemplo, si suponemos que la variable a es igual a 5 y que la variable b es igual a 6, entonces la instrucción

```
c[ a + b ] += 2;
```

suma 2 al elemento c[11] del arreglo. El nombre del arreglo con subíndice es un *lvalue*: se puede utilizar en el lado izquierdo de una asignación, de igual forma que los nombres de las variables que no son arreglos.

Examinaremos el arreglo c de la figura 7.1 con más detalle. El **nombre** del arreglo completo es c. Cada arreglo *conoce su propio tamaño*, que puede determinarse mediante una llamada a su función miembro **size**, como en c.size(). La manera en que se hace referencia a los 12 elementos de este arreglo es de c[0] a c[11]. El **valor** de c[0] es -45, el valor de c[7] es 62 y el valor de c[11] es 78. Para imprimir la suma de los valores contenidos en los primeros tres elementos del arreglo c, escribiríamos lo siguiente:

```
cout << c[ 0 ] + c[ 1 ] + c[ 2 ] << endl;
```

Para dividir el valor de c[6] entre 2 y asignar el resultado a la variable x, escribiríamos lo siguiente:

```
x = c[ 6 ] / 2;
```

Error común de programación 7.1

Observe la diferencia entre el "séptimo elemento del arreglo" y el "elemento 7 del arreglo". Los subíndices de los arreglos empiezan en 0, por lo que el "séptimo elemento del arreglo" tiene un subíndice de 6, mientras que el "elemento 7 del arreglo" tiene un subíndice de 7 y es en realidad el octavo *elemento del arreglo. Por desgracia, esta distinción genera con frecuencia* ***errores de desplazamiento en 1****. Para evitar dichos errores, nos referimos explícitamente a los elementos específicos de un arreglo por medio del nombre del arreglo y el número de subíndice (por ejemplo, c[6] o c[7]).*

Los corchetes que se utilizan para encerrar el subíndice de un arreglo son en realidad un *operador* que tiene la misma precedencia que los paréntesis. En la figura 7.2 se muestran la precedencia y la aso-

ciatividad de los operadores introducidos hasta ahora. Los operadores se muestran de arriba hacia abajo, en orden descendente de precedencia, con su asociatividad y su tipo.

Operadores	Asociatividad	Tipo
`::  ()`	izquierda a derecha *[Vea la precaución en la figura 2.10 con respecto al agrupamiento de paréntesis].*	primario
`() [] ++ -- static_cast<`*tipo*`>(`*operando*`)`	izquierda a derecha	postfijo
`++ -- + - !`	derecha a izquierda	unario (prefijo)
`* / %`	izquierda a derecha	multiplicativo
`+ -`	izquierda a derecha	aditivo
`<< >>`	izquierda a derecha	inserción/extracción
`< <= > >=`	izquierda a derecha	relacional
`== !=`	izquierda a derecha	igualdad
`&&`	izquierda a derecha	AND lógico
`||`	izquierda a derecha	OR lógico
`?:`	derecha a izquierda	condicional
`= += -= *= /= %=`	derecha a izquierda	asignación
`,`	izquierda a derecha	coma

Fig. 7.2 | Precedencia y asociatividad de los operadores introducidos hasta ahora.

7.3 Declaración de arreglos

Los arreglos (objetos `array`) ocupan espacio en memoria. Para especificar el tipo de los elementos y el número de elementos requerido por un arreglo, use una declaración de la forma:

```
array< tipo, tamañoArreglo > nombreArreglo;
```

La notación <*tipo*, *tamañoArreglo*> indica que el `array` es una plantilla de clase. El compilador reserva la cantidad apropiada de memoria con base en el *tipo* de los elementos y el *tamañoArreglo*. (Recuerde que una declaración que reserva memoria se conoce en forma más apropiada como *definición*). El *tamañoArreglo* debe ser un entero sin signo. Para indicar al compilador que debe reservar 12 elementos para el arreglo `c` de enteros, use la siguiente declaración:

```
array< int, 12 > c; // c es un arreglo de 12 valores enteros
```

Se pueden declarar arreglos para contener valores de la mayoría de los tipos de datos. Por ejemplo, es posible usar un arreglo de tipo `string` para almacenar cadenas de caracteres.

7.4 Ejemplos acerca del uso de los arreglos

Los siguientes ejemplos demuestran cómo declarar, inicializar y manipular arreglos.

7.4.1 Declaración de un arreglo y uso de un ciclo para inicializar los elementos del arreglo

El programa de la figura 7.3 declara el arreglo `n` con cinco elementos enteros (línea 10). La línea 5 incluye el encabezado <array>, que contiene la definición de la plantilla de clase `array`. En las líneas 13 y 14

se utiliza una instrucción `for` para inicializar los elementos del arreglo con cero. Al igual que otras variables automáticas, los arreglos automáticos *no* se inicializan de manera implícita con cero, aunque los arreglos `static` sí. La primera instrucción de salida (línea 16) muestra los encabezados de columna para las columnas impresas en la instrucción `for` subsiguiente (líneas 19 y 20), la cual imprime el arreglo en formato tabular. Recuerde que `setw` especifica la anchura de campo en la que sólo se va a imprimir el *siguiente* valor.

```
1  // Fig. 7.3: fig07_03.cpp
2  // Inicialización de los elementos de un arreglo con ceros, e impresión del arreglo.
3  #include <iostream>
4  #include <iomanip>
5  #include <array>
6  using namespace std;
7
8  int main()
9  {
10   array< int, 5 > n; // n es un arreglo de 5 valores int
11
12   // inicializa los elementos del arreglo n con 0
13   for ( size_t i = 0; i < n.size(); ++i )
14     n[ i ] = 0; // establece el elemento en la ubicación i a 0
15
16   cout << "Elemento" << setw( 13 ) << "Valor" << endl;
17
18   // imprime el valor de cada elemento del arreglo
19   for ( size_t j = 0; j < n.size(); ++j )
20     cout << setw( 7 ) << j << setw( 13 ) << n[ j ] << endl;
21 } // fin de main
```

```
Elemento      Valor
       0          0
       1          0
       2          0
       3          0
       4          0
```

Fig. 7.3 | Inicialización de los elementos de un arreglo con ceros, e impresión del arreglo.

En este programa, las variables de control `i` (línea 13) y `j` (línea 19) que especifican los subíndices del arreglo se declaran como de tipo **`size_t`**. De acuerdo con el C++ estándar, `size_t` representa un tipo integral sin signo. Este tipo se recomienda para cualquier variable que representa el tamaño de un arreglo o los subíndices de éste. El tipo `size_t` está definido en el espacio nombre `std` y se encuentran en el encabezado `<cstddef>`, que se incluye a través de muchos otros encabezados. Si intenta compilar un programa que utilice el tipo `size_t` y recibe errores que indiquen que no está definido, sólo tiene que incluir `<cstddef>` en su programa.

7.4.2 Inicialización de un arreglo en una declaración mediante una lista inicializadora

Los elementos de un arreglo también se pueden inicializar en la declaración del arreglo, para lo cual colocamos después del nombre del arreglo un signo igual y entre llaves, una lista de **inicializadores** separados por comas. El programa de la figura 7.4 utiliza una **lista inicializadora** para inicializar un arreglo de enteros con cinco valores (línea 11) y lo imprime en formato tabular (líneas 13 a 17).

```
1  // Fig. 7.4: fig07_04.cpp
2  // Inicialización de un arreglo en una declaración.
3  #include <iostream>
4  #include <iomanip>
5  #include <array>
6  using namespace std;
7
8  int main()
9  {
10    // usa la lista inicializadora para inicializar el arreglo n
11    array< int, 5 > n = { 32, 27, 64, 18, 95 };
12
13    cout << "Elemento" << setw( 13 ) << "Valor" << endl;
14
15    // imprime el valor de cada elemento del arreglo
16    for ( size_t i = 0; i < n.size(); ++i )
17       cout << setw( 7 ) << i << setw( 13 ) << n[ i ] << endl;
18 } // fin de main
```

```
Elemento      Valor
       0         32
       1         27
       2         64
       3         18
       4         95
```

Fig. 7.4 | Inicialización de un arreglo en una declaración.

Si hay *menos* inicializadores que elementos en el arreglo, el resto de los elementos del arreglo se inicializan con cero. Por ejemplo, los elementos del arreglo `n` en la figura 7.3 podrían haberse inicializado en ceros con la declaración

```
array< int, 5 > n = {}; // inicializa los elementos del arreglo n con 0
```

la declaración inicializa los elementos con cero, ya que hay menos inicializadores (ninguno en este caso) que elementos en el arreglo. Esta técnica sólo se puede utilizar en la declaración del arreglo, mientras que la técnica de inicialización que se muestra en la figura 7.3 se puede utilizar de manera repetida durante la ejecución del programa, para "reinicializar" los elementos de un arreglo.

Si se especifican el tamaño del arreglo y una lista inicializadora en la declaración de un arreglo, el número de inicializadores debe ser menor o igual que el tamaño del arreglo. La declaración del arreglo

```
array< int, 5 > n = { 32, 27, 64, 18, 95, 14 };
```

produce un error de compilación, ya que hay seis inicializadores y sólo cinco elementos en el arreglo.

7.4.3 Especificación del tamaño de un arreglo con una variable constante y establecimiento de los elementos de un arreglo con cálculos

En la figura 7.5 se establecen los cinco elementos de un arreglo `s` con los enteros pares `2`, `4`, `6`, `8` y `10` (líneas 15 y 16), y se imprime el arreglo en formato tabular (líneas 18 a 22). Para generar estos números (línea 16), se multiplica cada valor sucesivo del contador de ciclo por `2`, y se le suma `2`.

En la línea 11 se utiliza el **calificador `const`** para declarar lo que se conoce como una **variable constante** llamada `tamanioArreglo` con el valor `5`. Una variable constante que se utiliza para especificar el tamaño de un arreglo *debe* inicializarse con una expresión constante cuando se declara y *no puede*

```
1  // Fig. 7.5: fig07_05.cpp
2  // Establece el arreglo s con los enteros pares del 2 al 10.
3  #include <iostream>
4  #include <iomanip>
5  #include <array>
6  using namespace std;
7
8  int main()
9  {
10   // la variable constante se puede usar para especificar el tamaño de los arreglos
11   const size_t tamanioArreglo = 5; // debe inicializarse en la declaración
12
13   array< int, tamanioArreglo > s; // el arreglo s tiene 5 elementos
14
15   for ( size_t i = 0; i < s.size(); ++i ) // establece los valores
16     s[ i ] = 2 + 2 * i;
17
18   cout << "Elemento" << setw( 13 ) << "Valor" << endl;
19
20   // imprime el contenido del arreglo s en formato tabular
21   for ( size_t j = 0; j < s.size(); ++j )
22     cout << setw( 7 ) << j << setw( 13 ) << s[ j ] << endl;
23 } // fin de main
```

```
Elemento      Valor
       0          2
       1          4
       2          6
       3          8
       4         10
```

Fig. 7.5 | Establece el arreglo s con los enteros pares del 2 al 10.

modificarse de ahí en adelante (como se muestra en las figuras 7.6 y 7.7). Estas variables también se conocen como **constantes con nombre** o **variables de sólo lectura**.

Error común de programación 7.2

Si no se inicializa una variable constante al momento de declararla, se produce un error de compilación.

Error común de programación 7.3

Asignar un valor a una variable constante en una instrucción ejecutable es un error de compilación.

```
1  // Fig. 7.6: fig07_06.cpp
2  // Uso de una variable constante inicializada en forma apropiada.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
```

Fig. 7.6 | Uso de una variable constante inicializada en forma apropiada (parte 1 de 2).

```
8  const int x = 7; // variable constante inicializada
9
10    cout << "El valor de la variable constante x es: " << x << endl;
11 } // fin de main
```

```
El valor de la variable constante x es: 7
```

Fig. 7.6 | Uso de una variable constante inicializada en forma apropiada (parte 2 de 2).

```
1  // Fig. 7.7: fig07_07.cpp
2  // Una variable const debe inicializarse.
3
4  int main()
5  {
6     const int x; // Error: x debe inicializarse
7
8     x = 7; // Error: no se puede modificar una variable const
9  } // fin de main
```

Mensaje de error del compilador de Microsoft Visual C++:

```
error C2734: 'x' : const object must be initialized if not extern
error C3892: 'x' : you cannot assign to a variable that is const
```

Mensaje de error del compilador GNU C++:

```
fig07_07.cpp:6:14: error: uninitialized const 'x' [-fpermissive]
fig07_07.cpp:8:8: error: assignment of read-only variable 'x'
```

Mensaje de error del compilador LLVM:

```
Default initialization of an object of const type 'const int'
```

Fig. 7.7 | Una variable `const` debe inicializarse.

En la figura 7.7, el error de compilación producido por Microsoft Visual C++ se refiere a la variable `int x` como un "objeto `const`". El estándar de C++ define a un "objeto" como una "región de almacenamiento". Al igual que los objetos de las clases, las variables de tipo fundamental también ocupan espacio en memoria, por lo que se conocen comúnmente como "objetos".

Las variables constantes se pueden colocar en cualquier parte en la que se espera una expresión constante. En la figura 7.5, la variable constante `tamanioArreglo` especifica el tamaño del arreglo `s` en la línea 13.

Buena práctica de programación 7.1

Definir el tamaño de un arreglo como una variable constante, en vez de una constante literal, hace a los programas más claros. Esta técnica elimina lo que se conoce como ***números mágicos****: valores numéricos que no se explican. Al usar una variable constante podemos proporcionar un nombre para una constante literal; esto ayuda a explicar el propósito del valor en el programa.*

7.4.4 Suma de los elementos de un arreglo

A menudo, los elementos de un arreglo representan una serie de valores para ser utilizados en un cálculo. Por ejemplo, si los elementos de un arreglo representan calificaciones de un examen, tal vez un profesor desea obtener el total de los elementos del arreglo y usar esa suma para calcular el promedio de la clase para el examen.

El programa en la figura 7.8 suma los valores contenidos en el arreglo a de cuatro elementos enteros. El programa declara, crea e inicializa el arreglo en la línea 10. La instrucción `for` (líneas 14 y 15) realiza los cálculos. Los valores que se suministran como inicializadores para el arreglo a también se podrían haber pedido, en el programa, al usuario mediante el teclado, o mediante un archivo en el disco (vea el capítulo 14, Procesamiento de archivos). Por ejemplo, la instrucción `for`

```
for ( size_t j = 0; j < a.size(); ++j )
  cin >> a[ j ];
```

lee un valor a la vez del teclado y lo almacena en el elemento `a[j]`.

```
1  // Fig. 7.8: fig07_08.cpp
2  // Cálculo de la suma de los elementos de un arreglo.
3  #include <iostream>
4  #include <array>
5  using namespace std;
6
7  int main()
8  {
9     const size_t tamanioArreglo = 4; // especifica el tamaño del arreglo
10    array< int, tamanioArreglo > a = { 10, 20, 30, 40 };
11    int total = 0;
12
13    // suma el contenido del arreglo a
14    for ( size_t i = 0; i < a.size(); ++i )
15       total += a[ i ];
16
17    cout << "Total de elementos del arreglo: " << total << endl;
18 } // fin de main
```

```
Total de elementos del arreglo: 100
```

Fig. 7.8 | Cálculo de la suma de los elementos de un arreglo.

7.4.5 Uso de gráficos de barra para mostrar los datos de un arreglo en forma gráfica

Muchos programas presentan datos a los usuarios en forma gráfica. Por ejemplo, los valores numéricos se muestran comúnmente como barras en un gráfico de barras. En dicho gráfico, las barras más extensas representan valores numéricos proporcionalmente más grandes. Una manera simple de mostrar datos numéricos en forma gráfica es mediante un gráfico de barras que muestra cada valor numérico como una barra de asteriscos (*).

A menudo, a los profesores les gusta examinar la distribución de calificaciones en un examen. Un profesor podría graficar el número de calificaciones en cada una de varias categorías, para visualizar la distribución de calificaciones. Suponga que las calificaciones fueron 87, 68, 94, 100, 83, 78, 85, 91, 76 y 87. Hubo una calificación de 100, dos calificaciones entre 90 y 99, cuatro entre 80 y 89, dos entre 70 y 79, una calificación entre 60 y 69, y ninguna menor a 60. Nuestro siguiente programa (figura 7.9) almacena estos datos en un arreglo de 11 elementos, cada uno de los cuales corresponde a una categoría de calificaciones. Por ejemplo, `n[0]` indica el número de calificaciones en el rango de 0 a 9, `n[7]` indica

```
1  // Fig. 7.9: fig07_09.cpp
2  // Programa para imprimir gráficos de barra.
3  #include <iostream>
4  #include <iomanip>
5  #include <array>
6  using namespace std;
7
8  int main()
9  {
10    const size_t tamanioArreglo = 11;
11    array< unsigned int, tamanioArreglo > n =
12       { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };
13
14    cout << "Distribucion de calificaciones:" << endl;
15
16    // para cada elemento del arreglo n, imprime una barra del gráfico
17    for ( size_t i = 0; i < n.size(); ++i )
18    {
19       // imprime etiquetas de las barras ("0-9:", ..., "90-99:", "100:" )
20       if ( 0 == i )
21          cout << "  0-9: ";
22       else if ( 10 == i )
23          cout << "  100: ";
24       else
25          cout << i * 10 << "-" << ( i * 10 ) + 9 << ": ";
26
27       // imprime barra de asteriscos
28       for ( unsigned int estrellas = 0; estrellas < n[ i ]; ++estrellas )
29          cout << '*';
30
31       cout << endl; // inicia una nueva línea de salida
32    } // fin de for externo
33 } // fin de main
```

```
Distribución de calificaciones:
  0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Fig. 7.9 | Programa para imprimir gráficos de barra.

el número de calificaciones en el rango de 70 a 79 y `n[10]` indica el número de calificaciones de 100. Las versiones de la clase `LibroCalificaciones` de las figuras 7.15 y 7.16, y de las figuras 7.22 y 7.23, contienen código que calcula estas frecuencias de calificaciones, con base en un conjunto de calificaciones. Por ahora crearemos el arreglo en forma manual, analizando el conjunto de calificaciones.

El programa lee los números del arreglo y grafica la información como un gráfico de barras, mostrando cada rango de calificaciones seguido de una barra de asteriscos, los cuales indican el número de calificaciones en ese rango. Para etiquetar cada barra, en las líneas 20 a 25 se imprime un rango de calificaciones (por ejemplo, `"70-79: "`) con base en el valor actual de la variable contador `i`. La instrucción `for`

anidada (líneas 28 y 29) imprime las barras. Observe la condición de continuación de ciclo en la línea 28 (`estrellas < n[i]`). Cada vez que el programa llega al `for` *interior*, el ciclo cuenta desde `0` hasta `n[i]`, con lo cual usa un valor en el arreglo `n` para determinar el número de asteriscos a mostrar. En este ejemplo, `n[0] – n[5]` contiene ceros, ya que ningún estudiante recibió una calificación menor a 60. Por ende, el programa no muestra asteriscos enseguida de los primeros seis rangos de calificaciones.

7.4.6 Uso de los elementos de un arreglo como contadores

Algunas veces, los programas usan variables contadores para sintetizar datos, como los resultados de una encuesta. En la figura 6.9, utilizamos contadores separados en nuestro programa para tirar dados, para rastrear el número de ocurrencias de cada lado de un dado, a medida que el programa tiraba el dado 6 000 000 veces. En la figura 7.10 se muestra una versión de este programa, en la que se utiliza un arreglo. Esta versión también utiliza las nuevas herramientas de generación de números aleatorios de C++11 que se introdujeron en la sección 6.9.

La figura 7.10 utiliza el arreglo `frecuencia` (línea 18) para contar las ocurrencias de cada lado del dado. *La instrucción individual en la línea 22 de este programa reemplaza a la instrucción* `switch` *en las líneas 23 a 45 de la figura 6.9.* En la línea 22 se utiliza un valor aleatorio para determinar cuál elemento de `frecuencia` incrementar durante cada iteración del ciclo. El cálculo en la línea 22 produce un subíndice aleatorio de 1 a 6, por lo que el arreglo `frecuencia` debe ser lo suficientemente grande como para almacenar seis contadores. Sin embargo, usamos un arreglo de siete elementos en el que ignoramos

```
1  // Fig. 7.10: fig07_10.cpp
2  // Programa para tirar dados que utiliza un arreglo en vez de una instrucción switch.
3  #include <iostream>
4  #include <iomanip>
5  #include <array>
6  #include <random>
7  #include <ctime>
8  using namespace std;
9
10 int main()
11 {
12    // usa el motor predeterminado de generación de números aleatorios para
13    // producir valores int seudoaleatorios distribuidos de manera uniforme de 1 a 6
14    default_random_engine motor( static_cast< unsigned int >( time(0) ) );
15    uniform_int_distribution< unsigned int > intAleatorio( 1, 6 );
16
17    const size_t tamanioArreglo = 7; // ignora el elemento cero
18    array< unsigned int, tamanioArreglo > frecuencia = {}; // inicializa con ceros
19
20    // tira el dado 6000000 de veces; usa el valor del dado como índice de
         frecuencia
21    for ( unsigned int tiro = 1; tiro <= 6000000; ++tiro )
22       ++frecuencia[ intAleatorio( motor ) ];
23
24    cout << "Cara" << setw( 13 ) << "Frecuencia" << endl;
25
26    // imprime el valor de cada elemento del arreglo
27    for ( size_t cara = 1; cara < frecuencia.size(); ++cara )
28       cout << setw( 4 ) << cara << setw( 13 ) << frecuencia[ cara ]
29          << endl;
30 } // fin de main
```

Fig. 7.10 | Programa para tirar dados que utiliza un arreglo en vez de una instrucción `switch` (parte 1 de 2).

```
Cara   Frecuencia
   1      1000167
   2      1000149
   3      1000152
   4       998748
   5       999626
   6      1001158
```

Fig. 7.10 | Programa para tirar dados que utiliza un arreglo en vez de una instrucción `switch` (parte 2 de 2).

`frecuencia[0]`; es más lógico hacer que la cara del dado con el valor 1 incremente a `frecuencia[1]` que a `frecuencia[0]`. Por ende, el valor de cada cara se utiliza como subíndice para el arreglo `frecuencia`. También reemplazamos las líneas 49 a 54 de la figura 6.9, iterando a través del arreglo `frecuencia` para imprimir los resultados (figura 7.10, líneas 27 a 29).

7.4.7 Uso de arreglos para sintetizar los resultados de una encuesta

Nuestro siguiente ejemplo utiliza arreglos para sintetizar los resultados de los datos recolectados en una encuesta. Considere el siguiente enunciado del problema:

> *Se pidió a veinte estudiantes que calificaran la calidad de la comida en la cafetería estudiantil, en una escala del 1 al 5, en donde 1 significa "pésimo" y 5 significa "excelente". Coloque las 20 respuestas en un arreglo entero y determine la frecuencia de cada calificación.*

Ésta es un tipo popular de aplicación de procesamiento de arreglos (figura 7.11). Deseamos resumir el número de respuestas de cada tipo (es decir, del 1 al 5). El arreglo `respuestas` (líneas 15 a 16) es un arreglo entero de 20 elementos, y contiene las respuestas de los estudiantes a la encuesta. El arreglo `respuestas` se declara como `const`, ya que sus valores no cambian (y no deben hacerlo). Utilizamos un arreglo de seis elementos llamado `frecuencia` (línea 19) para contar el número de ocurrencias de cada respuesta. Cada elemento del arreglo se utiliza como un contador para una de las respuestas de la encuesta, y se inicializa con cero. Al igual que en la figura 7.10, ignoramos `frecuencia[0]`.

```
1  // Fig. 7.11: fig07_11.cpp
2  // Programa para analizar encuestas.
3  #include <iostream>
4  #include <iomanip>
5  #include <array>
6  using namespace std;
7
8  int main()
9  {
10    // define los tamaños de los arreglos
11    const size_t tamanioRespuesta = 20; // tamaño del arreglo respuestas
12    const size_t tamanioFrecuencia = 6; // tamaño del arreglo frecuencia
13
14    // coloca las respuestas de la encuesta en el arreglo respuestas
15    const array< unsigned int, tamanioRespuesta > respuestas =
16       { 1, 2, 5, 4, 3, 5, 2, 1, 3, 1, 4, 3, 3, 3, 2, 3, 3, 2, 2, 5 };
```

Fig. 7.11 | Programa para analizar encuestas (parte 1 de 2).

```
17
18    // inicializa los contadores de frecuencia con 0
19    array< unsigned int, tamanioFrecuencia > frecuencia = {};
20
21    // para cada respuesta, selecciona el elemento de respuestas y usa ese valor
22    // acomo subíndice de frecuencia para determinar el elemento a incrementar
23    for ( size_t respuesta = 0; respuesta < respuestas.size(); ++respuesta )
24       ++frecuencia[ respuestas[ respuesta ] ];
25
26    cout << "Calificacion" << setw( 17 ) << "Frecuencia" << endl;
27
28    // imprime el valor de cada elemento del arreglo
29    for ( size_t calificacion = 1; calificacion < frecuencia.size(); ++calificacion )
30       cout << setw( 6 ) << calificacion << setw( 17 ) << frecuencia[ calificacion ]
31          << endl;
32 } // fin de main
```

```
Calificacion      Frecuencia
           1                3
           2                5
           3                7
           4                2
           5                3
```

Fig. 7.11 | Programa para analizar encuestas (parte 2 de 2).

La primera instrucción `for` (líneas 23 y 24) recibe las respuestas, una a la vez, del arreglo `respuestas` e incrementa uno de los cinco contadores en el arreglo `frecuencia` (de `frecuencia[1]` a `frecuencia[5]`). La instrucción clave en el ciclo es la línea 24, la cual incrementa el contador de `frecuencia` apropiado, dependiendo del valor de `respuestas[respuesta]`.

Consideraremos varias iteraciones del ciclo `for`. Cuando la variable de control `respuesta` es `0`, el valor de `respuestas[respuesta]` es el valor de `respuestas[0]` (es decir, `1` en la línea 16), por lo que el programa interpreta a `++frecuencia[respuestas[respuesta]]` como

```
++frecuencia[ 1 ]
```

con lo cual se incrementa el valor en el elemento 1 del arreglo. Para evaluar la expresión, empiece con el valor en el conjunto *más interno* de corchetes (`respuesta`). Una vez que conozca el valor de `respuesta` (que viene siendo el valor de la variable de control de ciclo en la línea 23), insértelo en la expresión y evalúe el siguiente conjunto más externo de corchetes (`respuestas[respuesta]`, que es un valor seleccionado del arreglo `respuestas` en las líneas 15 a 16). Después utilice el valor resultante como subíndice del arreglo `frecuencia`, para especificar cuál contador se va a incrementar.

Cuando `respuesta` es `1`, `respuestas[respuesta]` es el valor de `respuestas[1]`, que es `2`, por lo que el programa interpreta a `++frecuencia[respuestas[respuesta]]` como

```
++frecuencia[ 2 ]
```

con lo cual se incrementa el elemento 2 del arreglo.

Cuando `respuesta` es `2`, `respuestas[respuesta]` es el valor de `respuestas[2]`, que es `5`, por lo que el programa interpreta a `++frecuencia[respuestas[respuesta]]` como

```
++frecuencia[ 5 ]
```

con lo cual se incrementa el elemento 5 del arreglo, y así en lo sucesivo. Sin importar el número de respuestas procesadas en la encuesta, el programa *sólo* requiere un arreglo de seis elementos (en el cual se ignora el elemento cero) para resumir los resultados, ya que todos los valores de las respuestas se encuentran entre 1 y 5, y los valores de subíndice para un arreglo de seis elementos son del 0 al 5.

Comprobación de límites para los subíndices de un arreglo

Si los datos en el arreglo `respuestas` tuvieran valores inválidos, como 13, el programa trataría de sumar `1` a `frecuencia[13]`, lo cual se encuentra *fuera* de los límites del arreglo. *Cuando usamos el operador* `[]` *para acceder al elemento de un arreglo, C++ no cuenta con* ***comprobación de límites*** *automática de arreglos para evitar que hagamos referencia a un elemento que no existe*. Por lo tanto, un programa en ejecución puede "salirse" de cualquier extremo de un arreglo sin advertencia. En la sección 7.10 demostraremos la función `at` de la plantilla de clase `vector`, que realiza la comprobación de límites por el usuario. La plantilla de clase `array` también tiene una función `at`.

Es importante asegurar que cada subíndice que utilicemos para acceder al elemento de un arreglo se encuentre dentro de los límites de ese arreglo; es decir, mayor o igual a 0 y menor que el número de elementos del arreglo.

A la acción de permitir que los programas lean de, o escriban en, los elementos de un arreglo fuera de los límites, se le conoce como *falla de seguridad*. Si se intentan leer elementos fuera de los límites de un arreglo, el programa puede fallar o incluso tal vez parezca ejecutarse correctamente al utilizar datos incorrectos. Si escribimos en un elemento fuera de los límites (lo que se conoce como *desbordamiento de búfer*), los datos del programa en memoria podrían corromperse, el programa podría fallar y dejar que algún atacante explote el sistema y ejecute su propio código. Para obtener más información sobre desbordamientos de búfer, consulte `en.wikipedia.org/wiki/Buffer_overflow` (`http://es.wikipedia.org/wiki/Desbordamiento_de_b%C3%BAfer` en español).

Error común de programación 7.4

Hacer referencia a un elemento fuera de los límites del arreglo es un error lógico en tiempo de ejecución. *No es un error de sintaxis.*

Tip para prevenir errores 7.1

Al iterar a través de un arreglo, el subíndice del arreglo debe ser mayor o igual a 0 y siempre debe ser menor que el número total de elementos en el arreglo (uno menos que el tamaño del arreglo). Asegúrese que la condición de terminación de ciclo evite acceder a los elementos fuera de este rango. En los capítulos 15 y 16 aprenderá sobre los iteradores, que pueden ayudar a evitar el acceso a los elementos fuera de los límites de un arreglo (o de otro contenedor).

7.4.8 Arreglos locales estáticos y arreglos locales automáticos

En el capítulo 6 hablamos sobre el especificador de clase de almacenamiento `static`. Una variable local `static` en la definición de una función existe durante todo el programa, pero *sólo* puede verse en el cuerpo de la función.

Tip de rendimiento 7.1

Podemos aplicar `static` *a la declaración de un arreglo local, de manera que el arreglo no se cree e inicialice cada vez que el programa llame a la función, y no se destruya cada vez que termine la función en el programa. Esto puede mejorar el rendimiento, en especial cuando se utilizan arreglos extensos.*

Un programa inicializa los arreglos locales `static` la primera vez que encuentra sus declaraciones. Si el programador no inicializa un arreglo `static` de manera explícita, el compilador inicializa con *cero* cada elemento de ese arreglo al momento de su creación. Recuerde que C++ *no* realiza dicha inicialización predeterminada para las variables automáticas.

La figura 7.12 demuestra la función `inicArregloStatic` (líneas 24 a 40) con un arreglo local `static` (línea 27) y la función `inicArregloAutomatico` (líneas 43 a 59) con un arreglo local automático (línea 46).

```
1  // Fig. 7.12: fig07_12.cpp
2  // Inicialización de un arreglo static e inicialización de un arreglo automático.
3  #include <iostream>
4  #include <array>
5  using namespace std;
6
7  void inicArregloStatic(); // prototipo de función
8  void inicArregloAutomatico(); // prototipo de función
9  const size_t tamanioArreglo = 3;
10
11 int main()
12 {
13    cout << "Primera llamada a cada funcion:\n";
14    inicArregloStatic();
15    inicArregloAutomatico();
16
17    cout << "\n\nSegunda llamada a cada funcion:\n";
18    inicArregloStatic();
19    inicArregloAutomatico();
20    cout << endl;
21 } // fin de main
22
23 // función para demostrar un arreglo local static
24 void inicArregloStatic( void )
25 {
26    // inicializa los elementos con 0 la primera vez que se llama a la función
27    static array< int, tamanioArreglo > arreglo1; // arreglo local static
28
29    cout << "\nValores al entrar en inicArregloStatic:\n";
30
31    // imprime el contenido de arreglo1
32    for ( size_t i = 0; i < arreglo1.size(); ++i )
33       cout << "arreglo1[" << i << "] = " << arreglo1[ i ] << "  ";
34
35    cout << "\nValores al salir de inicArregloStatic:\n";
36
37    // modifica e imprime el contenido de arreglo1
38    for ( size_t j = 0; j < arreglo1.size(); ++j )
39       cout << "arreglo1[" << j << "] = " << ( arreglo1[ j ] += 5 ) << "  ";
40 } // fin de la función inicArregloStatic
41
42 // función para demostrar un arreglo local automático
43 void inicArregloAutomatico( void )
44 {
45    // inicializa los elementos cada vez que se llama a la función
46    array< int, tamanioArreglo > arreglo2 = { 1, 2, 3 }; // arreglo local automático
47
48    cout << "\n\nValores al entrar a inicArregloAutomatico:\n";
```

Fig. 7.12 | Inicialización de un arreglo `static` e inicialización de un arreglo automático (parte 1 de 2).

```
49
50    // imprime el contenido de arreglo2
51    for ( size_t i = 0; i < arreglo2.size(); ++i )
52       cout << "arreglo2[" << i << "] = " << arreglo2[ i ] << "  ";
53
54    cout << "\nValores al salir de inicArregloAutomatico:\n";
55
56    // modifica e imprime el contenido de arreglo2
57    for ( size_t j = 0; j < arreglo2.size(); ++j )
58       cout << "arreglo2[" << j << "] = " << ( arreglo2[ j ] += 5 ) << "  ";
59 } // fin de la función inicArregloAutomatico
```

```
Primera llamada a cada funcion:

Valores al entrar en inicArregloStatic:
arreglo1[0] = 0  arreglo1[1] = 0  arreglo1[2] = 0
Valores al salir de inicArregloStatic:
arreglo1[0] = 5  arreglo1[1] = 5  arreglo1[2] = 5

Valores al entrar a inicArregloStatic:
arreglo2[0] = 1  arreglo2[1] = 2  arreglo2[2] = 3
Valores al salir de inicArregloStatic:
arreglo2[0] = 6  arreglo2[1] = 7  arreglo2[2] = 8

Segunda llamada a cada funcion:

Valores al entrar en inicArregloStatic:
arreglo1[0] = 5  arreglo1[1] = 5  arreglo1[2] = 5
Valores al salir de inicArregloStatic:
arreglo1[0] = 10  arreglo1[1] = 10  arreglo1[2] = 10

Valores al entrar a inicArregloAutomatico:
arreglo2[0] = 1  arreglo2[1] = 2  arreglo2[2] = 3
Valores al salir de inicArregloAutomatico:
arreglo2[0] = 6  arreglo2[1] = 7  arreglo2[2] = 8
```

Fig. 7.12 | Inicialización de un arreglo `static` e inicialización de un arreglo automático (parte 2 de 2).

La función `inicArregloStatic` se llama dos veces (líneas 14 y 18). El compilador *inicializa* el arreglo local `static` `arreglo1` con *cero* la primera vez que se hace una llamada a la función. Ésta imprime el arreglo, suma 5 a cada elemento e imprime el arreglo de nuevo. La segunda vez que se llama a la función, el arreglo `static` contiene los valores *modificados* que se almacenan durante la primera llamada a la función.

La función `inicArregloAutomatico` también se llama dos veces (líneas 15 y 19). Los elementos del arreglo local automático `arreglo2` se inicializan (línea 46) con los valores 1, 2 y 3. La función imprime el arreglo, suma 5 a cada elemento e imprime el arreglo de nuevo. La segunda vez que se llama a la función, los elementos del arreglo se *reinicializan* con 1, 2 y 3. El arreglo tiene una *duración de almacenamiento automática*, por lo que se vuelve a crear y se reinicializa durante cada llamada a `inicArregloAutomatico`.

7.5 Instrucción for basada en rango

Como hemos visto, es común procesar *todos* los elementos de un arreglo. La nueva **instrucción for basada en rango** de C++11 nos permite hacerlo *sin usar un contador*, con lo que evitamos la posibilidad de "salirnos" del arreglo y eliminamos la necesidad de implementar nuestra propia comprobación de límites.

Tip para prevenir errores 7.2

Al procesar todos los elementos de un arreglo, si no necesita acceso al subíndice de los elementos del mismo, use la instrucción `for` *basada en rango.*

La sintaxis de la instrucción `for` basada en rango es:

```
for ( declaracionVariableRango : expresión )
   instrucción
```

en donde *declaracionVariableRango* tiene un tipo y un identificador (como `int elemento`), y *expresión* es el arreglo a través del cual se va a iterar. El tipo en la *declaracionVariableRango* debe ser consistente con el tipo de los elementos del arreglo. El identificador representa los valores sucesivos de los elementos del arreglo en iteraciones sucesivas del ciclo. Podemos usar la instrucción `for` basada en rango con la mayoría de las estructuras de datos preconstruidas de la Biblioteca estándar de C++ (que se conocen comúnmente como *contenedores*), incluyendo las clases `array` y `vector`.

La figura 7.13 usa la instrucción `for` basada en rango para mostrar el contenido de un arreglo (líneas 13 y 14; líneas 22 y 23) y para multiplicar cada uno de los valores de los elementos del arreglo por 2 (líneas 17 y 18).

```
1  // Fig. 7.13: fig07_13.cpp
2  // Uso de la instrucción for basada en rango para multiplicar los elementos de un
   //    arreglo por 2.
3  #include <iostream>
4  #include <array>
5  using namespace std;
6
7  int main()
8  {
9     array< int, 5 > items = { 1, 2, 3, 4, 5 };
10
11    // muestra los items antes de modificarlos
12    cout << "items antes de modificarlos: ";
13    for ( int item : items )
14       cout << item << " ";
15
16    // multiplica los elementos de los ítems por 2
17    for ( int &refItem : items )
18       refItem *= 2;
19
20    // muestra los ítems después de modificarlos
21    cout << "\nitems despues de modificarlos: ";
22    for ( int item : items )
23       cout << item << " ";
24
25    cout << endl;
26 } // fin de main
```

```
items antes de modificarlos: 1 2 3 4 5
items después de modificarlos: 2 4 6 8 10
```

Fig. 7.13 | Uso de la instrucción `for` basada en rango para multiplicar los elementos de un arreglo por 2.

Uso del `for` *basado en rango para mostrar el contenido de un arreglo*

La instrucción `for` basada en rango simplifica el código para iterar por un arreglo. La línea 13 puede leerse como "para cada iteración, asignar el siguiente elemento de `items` a la variable `item`, después

ejecutar la siguiente instrucción". Así, para cada iteración, el identificador `item` representa un elemento en `items`. Las líneas 13 y 14 son equivalentes a la siguiente repetición controlada por contador:

```
for ( int contador = 0; contador < items.size(); ++contador )
   cout << items[ contador ] << " ";
```

Uso de la instrucción `for` basada en rango para modificar el contenido de un arreglo

Las líneas 17 y 18 usan una instrucción `for` basada en rango para multiplicar cada elemento de `items` por 2. En la línea 17, la *declaracionVariableRango* indica que `refItem` es una *referencia* `int` (`&`). Recuerde que una referencia es un alias para otra variable en memoria; en este caso, uno de los elementos del arreglo. Usamos una referencia `int` debido a que `items` contiene valores `int` y queremos *modificar* el valor de cada elemento; ya que `refItem` se declara como *referencia*, cualquier modificación realizada en `refItem` cambia el valor del elemento correspondiente en el arreglo.

Uso del subíndice de un elemento

La instrucción `for` basada en rango puede usarse en vez de la instrucción `for` controlada por contador, cada vez que el código que itera a través de un arreglo *no* requiera acceso al subíndice del elemento. Por ejemplo, para obtener el total de los enteros en un arreglo (como en la figura 7.8) se requiere acceso sólo a los valores de los elementos; sus subíndices son irrelevantes. No obstante, si un programa debe usar subíndices por alguna razón que no sea iterar a través de un arreglo (por ejemplo, para imprimir un número de subíndice enseguida del valor de cada elemento del arreglo, como en los primeros ejemplos de este capítulo), debe usar la instrucción `for` controlada por contador.

7.6 Caso de estudio: la clase `LibroCalificaciones` que usa un arreglo para almacenar las calificaciones

En esta sección desarrollaremos aún más la clase `LibroCalificaciones`, que presentamos en el capítulo 3 y ampliamos en los capítulos 4 a 6. Recuerde que esta clase representa un libro de calificaciones utilizado por un instructor para almacenar y analizar un conjunto de calificaciones de estudiantes. Las versiones anteriores de esta clase procesan un conjunto de calificaciones introducidas por el usuario, pero *no* mantienen los valores de las calificaciones individuales en los miembros de datos de la clase. Por ende, los cálculos repetidos requieren que el usuario vuelva a introducir las calificaciones. Una manera de resolver este problema sería almacenar cada calificación introducida por el usuario en un miembro de datos individual de la clase. Por ejemplo, podríamos crear los miembros de datos `calificacion1`, `calificacion2`, …, `calificacion10` en la clase `LibroCalificaciones` para almacenar 10 calificaciones de estudiantes. No obstante, el código para totalizar las calificaciones y determinar el promedio de la clase sería voluminoso. En esta sección resolvemos este problema, almacenando las calificaciones en un arreglo.

Almacenar las calificaciones de los estudiantes en un arreglo en la clase `LibroCalificaciones`

La figura 7.14 muestra la salida que sintetiza las 10 calificaciones que almacenamos en un objeto de la siguiente versión de la clase `LibroCalificaciones` (figuras 7.15 y 7.16) que utiliza un arreglo de enteros para almacenar las calificaciones de 10 estudiantes en un solo examen. Esto elimina la necesidad de introducir varias veces el mismo conjunto de calificaciones. El arreglo `calificaciones` se declara como miembro de datos en la línea 28 de la figura 7.15; por lo tanto, cada objeto `LibroCalificaciones` mantiene su propio conjunto de calificaciones.

```
Bienvenido al libro de calificaciones para
CS101 Introduccion a la programacion en C++!
```

Fig. 7.14 | Salida del ejemplo de `LibroCalificaciones` que almacena las calificaciones en un arreglo (parte 1 de 2).

```
Las calificaciones son:

Estudiante  1:  87
Estudiante  2:  68
Estudiante  3:  94
Estudiante  4: 100
Estudiante  5:  83
Estudiante  6:  78
Estudiante  7:  85
Estudiante  8:  91
Estudiante  9:  76
Estudiante 10:  87

El promedio de la clase es 84.90
La calificacion mas baja es 68
La calificacion mas alta es 100

Distribucion de calificaciones:
  0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Fig. 7.14 | Salida del ejemplo de `LibroCalificaciones` que almacena las calificaciones en un arreglo (parte 2 de 2).

```
1  // Fig. 7.15: LibroCalificaciones.h
2  // Definición de la clase LibroCalificaciones que usa un arreglo para almacenar
      calificaciones de una prueba.
3  // Las funciones miembro se definen en LibroCalificaciones.cpp
4  #include <string>
5  #include <array>
6
7  // definición de la clase LibroCalificaciones
8  class LibroCalificaciones
9  {
10 public:
11    // constante -- número de estudiantes que tomaron la prueba
12    static const size_t estudiantes = 10; // observe los datos públicos
13
14    // el constructor inicializa el nombre del curso y el arreglo de calificaciones
15    LibroCalificaciones( const std::string &, const std::array< int, estudiantes
      > & );
16
17    void establecerNombreCurso( const std::string & ); // establece el nombre del
                                                             curso
18    string obtenerNombreCurso() const; // obtiene el nombre del curso
19    void mostrarMensaje() const; // muestra un mensaje de bienvenida
20    void procesarCalificaciones() const; // realiza varias operaciones con los
                                              datos de las calificaciones
21    int obtenerMinimo() const; // busca la calificación mínima para la prueba
22    int obtenerMaximo() const; // busca la calificación máxima para la prueba
```

Fig. 7.15 | Definición de la clase `LibroCalificaciones` que usa un arreglo para almacenar calificaciones de una prueba (parte 1 de 2).

```
23    double obtenerPromedio() const; // determina la calificación promedio para la
                                      prueba
24    void imprimirGraficoBarras() const; // imprime gráfico de barras de la
                                          distribución de calificaciones
25    void imprimirCalificaciones() const; // imprime el contenido del arreglo
                                           calificaciones
26 private:
27    std::string nombreCurso; // nombre del curso para este libro de calificaciones
28    std::array< int, estudiantes > calificaciones; // arreglo de calificaciones
                                                     de estudiantes
29 }; // fin de la clase LibroCalificaciones
```

Fig. 7.15 | Definición de la clase `LibroCalificaciones` que usa un arreglo para almacenar calificaciones de una prueba (parte 2 de 2).

```
1  // Fig. 7.16: LibroCalificaciones.cpp
2  // Funciones miembro para la clase LibroCalificaciones
3  // que manipulan un arreglo de calificaciones.
4  #include <iostream>
5  #include <iomanip>
6  #include "LibroCalificaciones.h" // definición de la clase LibroCalificaciones
7  using namespace std;
8
9  // el constructor inicializa nombreCurso y el arreglo calificaciones
10 LibroCalificaciones::LibroCalificaciones( const string &nombre,
11    const array< int, estudiantes > &arregloCalificaciones )
12    : nombreCurso( nombre ), calificaciones( arregloCalificaciones )
13 {
14 } // fin del constructor de LibroCalificaciones
15
16 // función para establecer el nombre del curso
17 void LibroCalificaciones::establecerNombreCurso( const string &nombre )
18 {
19    nombreCurso = nombre; // almacena el nombre del curso
20 } // fin de la función establecerNombreCurso
21
22 // función para obtener el nombre del curso
23 string LibroCalificaciones::obtenerNombreCurso() const
24 {
25    return nombreCurso;
26 } // fin de la función obtenerNombreCurso
27
28 // muestra un mensaje de bienvenida para el usuario de LibroCalificaciones
29 void LibroCalificaciones::mostrarMensaje() const
30 {
31    // esta instrucción llama a obtenerNombreCurso para obtener el
32    // nombre del curso que representa este LibroCalificaciones
33    cout << "Bienvenido al libro de calificaciones para\n"
       << obtenerNombreCurso() << "!"
34       << endl;
35 } // fin de la función mostrarMensaje
36
37 // realiza varias operaciones con los datos
38 void LibroCalificaciones::procesarCalificaciones() const
39 {
40    // imprime el arreglo calificaciones
41    imprimirCalificaciones();
42
```

Fig. 7.16 | Funciones miembro de la clase `LibroCalificaciones` que manipulan un arreglo de calificaciones (parte 1 de 3).

```
43     // llama a la función obtenerPromedio para calcular la calificacion promedio
44     cout << setprecision( 2 ) << fixed;
45     cout << "\nEl promedio de la clase es " << obtenerPromedio() << endl;
46
47     // llama a las funciones obtenerMinimo y obtenerMaximo
48     cout << "La calificacion mas baja es " << obtenerMinimo()
       << "\nLa calificacion mas alta es "
49        << obtenerMaximo() << endl;
50
51     // llama a la función imprimirGraficoBarras para imprimir el gráfico de
          ditribución de calificaciones
52     imprimirGraficoBarras();
53   } // fin de la función procesarCalificaciones
54
55   // busca la calificación mínima
56   int LibroCalificaciones::obtenerMinimo() const
57   {
58     int calificacionInf = 100; // asume que la calificación más baja es 100
59
60     // itera a través del arreglo calificaciones
61     for ( int calificacion : calificaciones )
62     {
63       // si la calificación actual es menor que calificacionInf, la asigna a
            calificacionInf
64       if ( calificacion < calificacionInf )
65         calificacionInf = calificacion; // nueva calificación más baja
66     } // fin de for
67
68     return calificacionInf; // devuelve la calificación más baja
69   } // fin de la función obtenerMinimo
70
71   // busca la calificación máxima
72   int LibroCalificaciones::obtenerMaximo() const
73   {
74     int calificacionSup = 0; // asume que la calificación más alta es 0
75
76     // itera a través del arreglo calificaciones
77     for ( int calificacion : calificaciones )
78     {
79       // si la calificación actual es mayor que calificacionSup, la asigna a
            calificacionSup
80       if ( calificacion > calificacionSup )
81         calificacionSup = calificacion; // nueva calificación más alta
82     } // fin de for
83
84     return calificacionSup; // devuelve la calificación más alta
85   } // fin de la función obtenerMaximo
86
87   // determina la calificación promedio para la prueba
88   double LibroCalificaciones::obtenerPromedio() const
89   {
90     int total = 0; // inicializa el total
91
92     // suma las calificaciones en el arreglo
93     for ( int calificacion : calificaciones )
94       total += calificacion;
95
```

Fig. 7.16 | Funciones miembro de la clase `LibroCalificaciones` que manipulan un arreglo de calificaciones (parte 2 de 3).

```
96     // devuelve el promedio de las calificaciones
97     return static_cast< double >( total ) / calificaciones.size();
98   } // fin de la función obtenerPromedio
99
100  // imprime gráfico de barras que muestra la distribución de las calificaciones
101  void LibroCalificaciones::imprimirGraficoBarras() const
102  {
103    cout << "\nDistribucion de calificaciones:" << endl;
104
105    // almacena la frecuencia de calificaciones en cada rango de 10 calificaciones
106    const size_t tamanioFrecuencia = 11;
107    array< unsigned int, tamanioFrecuencia > frecuencia = {};
       // inicializa elementos con 0
108
109    // para cada calificación, incrementa la frecuencia apropiada
110    for ( int calificacion : calificaciones )
111      ++frecuencia[ calificacion / 10 ];
112
113    // para cada frecuencia de calificación, imprime barra en el gráfico
114    for ( size_t cuenta = 0; cuenta < tamanioFrecuencia; ++cuenta )
115    {
116      // imprime etiquetas de las barras ("0-9:", ..., "90-99:", "100:" )
117      if ( 0 == cuenta )
118        cout << "  0-9: ";
119      else if ( 10 == cuenta )
120        cout << "  100: ";
121      else
122        cout << cuenta * 10 << "-" << ( cuenta * 10 ) + 9 << ": ";
123
124      // imprime barra de asteriscos
125      for ( unsigned int estrellas = 0; estrellas < frecuencia[ cuenta ];
         ++estrellas )
126        cout << '*';
127
128      cout << endl; // empieza una nueva línea de salida
129    } // fin de for exterior
130  } // fin de la función imprimirGraficoBarras
131
132  // imprime el contenido del arreglo calificaciones
133  void LibroCalificaciones::imprimirCalificaciones() const
134  {
135    cout << "\nLas calificaciones son:\n\n";
136
137    // imprime la calificación de cada estudiante
138    for ( size_t estudiante = 0; estudiante < calificaciones.size(); ++estudiante )
139      cout << "Estudiante " << setw( 2 ) << estudiante + 1 << ": " << setw( 3 )
140        << calificaciones[ estudiante ] << endl;
141  } // fin de la función imprimirCalificaciones
```

Fig. 7.16 | Funciones miembro de la clase `LibroCalificaciones` que manipulan un arreglo de calificaciones (parte 3 de 3).

El tamaño del arreglo en la línea 28 de la figura 7.15 se especifica mediante el miembro de datos `public static const` llamado `estudiantes` (declarado en la línea 12), el cual es `public`, de manera que sea accesible para los clientes de la clase. Pronto veremos un ejemplo de un programa cliente que utiliza esta constante. Al declarar a `estudiantes` con el calificador `const`, indicamos que este miembro de datos es constante; su valor no se puede modificar después de inicializarlo. La palabra clave `static` en esta declaración de variable indica que el miembro de datos *es compartido por todos los objetos de la*

clase; así que, en esta implementación específica de la clase `LibroCalificaciones`, todos los objetos `LibroCalificaciones` almacenan calificaciones para el mismo número de estudiantes. En la sección 3.4 vimos que cuando cada objeto de una clase mantiene su propia copia de un atributo, la variable que representa a ese atributo se conoce como miembro de datos; cada objeto (instancia) de la clase tiene una *copia separada* de la variable en memoria. Hay variables para las que cada objeto de una clase *no* tiene una copia separada. Éste es el caso con los **miembros de datos `static`**, que también se conocen como **variables de clase**. Cuando se crean los objetos de una clase que contiene miembros de datos `static`, todos los objetos comparten una copia de los miembros de datos `static` de la clase. Se puede acceder a un miembro de datos `static` dentro de la definición de la clase y de las definiciones de las funciones miembro al igual que con cualquier otro miembro de datos. Como veremos más adelante, también se puede acceder a un miembro de datos `public static` desde el exterior de la clase, *aun cuando no existan objetos de la misma*; para ello se utiliza el nombre de la clase, seguido del operador de resolución de ámbito (`::`) y el nombre del miembro de datos. En el capítulo 9 aprenderá más acerca de los miembros de datos `static`.

Constructor

El constructor de la clase (declarado en la línea 15 de la figura 7.15 y definido en las líneas 10 a 14 de la figura 7.16) tiene dos parámetros: el nombre del curso y una referencia a un arreglo de calificaciones. Cuando un programa crea un objeto `LibroCalificaciones` (por ejemplo, en la línea 15 de la figura 7.17), el programa pasa un arreglo `int` existente al constructor, el cual copia los valores del arreglo en el miembro de datos `calificaciones` (línea 12 de la figura 7.16). Los valores de las calificaciones en el arreglo que se pasa podrían haberse recibido de un usuario, o de un archivo en disco (como veremos en el capítulo 14, Procesamiento de archivos). En nuestro programa de prueba, simplemente inicializamos un arreglo con un conjunto de valores de calificaciones (figura 7.17, líneas 11 y 12). Una vez que las calificaciones se almacenan en el miembro de datos `calificaciones` de la clase `LibroCalificaciones`, todas las funciones miembro de la clase pueden acceder al arreglo `calificaciones` según sea necesario, para realizar varios cálculos. Cabe mencionar que el constructor recibe tanto el parámetro `string` como el arreglo por referencia; esto es más eficiente que recibir copias del objeto `string` original y del arreglo originales. El constructor no necesita modificar el objeto `string` ni el arreglo originales, por lo que también declaramos cada parámetro como `const`, para asegurar que el constructor no modifique accidentalmente los datos originales en la función que hizo la llamada. También modificamos la función `establecerNombreCurso` para que reciba su argumento `string` por referencia.

La función miembro `procesarCalificaciones`

La función miembro `procesarCalificaciones` (declarada en la línea 20 de la figura 7.15 y definida en las líneas 38 a 53 de la figura 7.16) contiene una serie de llamadas a funciones miembro que produce un reporte en el que se resumen las calificaciones. La línea 41 llama a la función miembro `imprimirCalificaciones` para imprimir el contenido del arreglo `calificaciones`. Las líneas 138 a 140 en la función miembro `imprimirCalificaciones` utilizan una instrucción `for` para imprimir la calificación de cada estudiante. Aunque los subíndices de los arreglos empiezan en 0, lo común es que el profesor enumere a los estudiantes empezando desde 1. Por ende, las líneas 139 y 140 imprimen `estudiante + 1` como el número de estudiante para producir las etiquetas `"Estudiante 1: "`, `"Estudiante 2: "`, y así en lo sucesivo.

La función miembro `obtenerPromedio`

A continuación, la función miembro `procesarCalificaciones` llama a la función miembro `obtenerPromedio` (línea 45) para obtener el promedio de las calificaciones. La función miembro `obtenerPromedio` (declarada en la línea 23 de la figura 7.15 y definida en las líneas 88 a 98 de la figura 7.16) totaliza los valores en el arreglo `calificaciones` antes de calcular el promedio. El cálculo del promedio en la línea 97 utiliza `calificaciones.size()` para determinar el número de calificaciones que se van a promediar.

Las funciones miembro `obtenerMinimo` *y* `obtenerMaximo`

Las líneas 48 y 49 en `procesarCalificaciones` llaman a las funciones miembro `obtenerMinimo` y `obtenerMaximo` para determinar las calificaciones más baja y más alta de cualquier estudiante en el examen. Vamos a examinar la forma en que la función miembro `obtenerMinimo` encuentra la calificación *más baja*. Como la calificación más alta permitida es 100, empezamos por suponer que 100 es la calificación más baja (línea 58). Después comparamos cada uno de los elementos en el arreglo con la calificación más baja, buscando valores más pequeños. En las líneas 61 a 66 de la función miembro `obtenerMinimo` se itera a través del arreglo, y en la línea 64 se compara cada calificación con `calificacionInf`. Si una calificación es menor que `calificacionInf`, a `calificacionInf` se le asigna esa calificación. Cuando se ejecuta la línea 68, `calificacionInf` contiene la calificación más baja en el arreglo. La función miembro `obtenerMaximo` (líneas 72 a 85) funciona de manera similar a la función miembro `obtenerMinimo`.

La función miembro `imprimirGraficoBarras`

Por último, la línea 52 en la función miembro `procesarCalificaciones` llama a la función miembro `imprimirGraficoBarras` para imprimir un gráfico de distribución de los datos de las calificaciones, usando una técnica similar a la de la figura 7.9. En ese ejemplo, calculamos en forma manual el número de calificaciones en cada categoría (es decir, 0-9, 10-19, …, 90-99 y 100), para lo cual simplemente analizamos un conjunto de calificaciones. En las líneas 110 y 111, de este ejemplo, se utiliza una técnica similar a la de las figuras 7.10 y 7.11 para calcular la frecuencia de calificaciones en cada categoría. En la línea 107 se declara y crea el arreglo `frecuencia` de 11 valores del tipo `unsigned int` para almacenar la frecuencia de calificaciones en cada categoría. Para cada `calificacion` en el arreglo `calificaciones`, en las líneas 110 y 111 se incrementa el elemento apropiado del arreglo `frecuencia`. Para determinar qué elemento se debe incrementar, en la línea 111 se divide la `calificacion` actual entre 10, usando la división entera. Por ejemplo, si `calificacion` es `85`, en la línea 111 se incrementa `frecuencia[8]` para actualizar la cuenta de calificaciones en el rango de 80 a 89. Después, en las líneas 114 a 129 se imprime el gráfico de barras (vea la figura 7.17) con base en los valores en el arreglo `frecuencia`. Al igual que en las líneas 28 y 29 de la figura 7.9, en las líneas 125 y 126 de la figura 7.16 se utiliza un valor en el arreglo `frecuencia` para determinar el número de asteriscos a mostrar en cada barra.

Prueba de la clase `LibroCalificaciones`

El programa de la figura 7.17 crea un objeto de la clase `LibroCalificaciones` (figuras 7.15 y 7.16) mediante el uso del arreglo `int calificaciones` (que se declara y se inicializa en las líneas 11 y 12). Usamos el operador de resolución de ámbito (`::`) en la expresión "`LibroCalificaciones::estudiantes`" (línea 11) para acceder a la constante `static` llamada `estudiantes`, de la clase `LibroCalificaciones`. Utilizamos aquí esta constante para crear un arreglo que sea del mismo tamaño que el arreglo que se almacena como miembro de datos en la clase `LibroCalificaciones`. En la línea 13 se declara un objeto `string` que representa el nombre de un curso. En la línea 15 se pasa el nombre del curso y el arreglo de `calificaciones` al constructor de `LibroCalificaciones`. En la línea 16 se imprime un mensaje de bienvenida, y en la línea 17 se invoca la función miembro `procesarCalificaciones` del objeto `LibroCalificaciones`.

```
1  // Fig. 7.17: fig07_17.cpp
2  // Crea un objeto LibroCalificaciones usando un arreglo de calificaciones.
3  #include <array>
4  #include "LibroCalificaciones.h" // definición de la clase LibroCalificaciones
5  using namespace std;
6
7  // la función main empieza la ejecución del programa
8  int main()
9  {
```

Fig. 7.17 | Crea un objeto `LibroCalificaciones` usando un arreglo de calificaciones, y después invoca a la función miembro `procesarCalificaciones` para analizarlas (parte 1 de 2).

```
10    // arreglo de calificaciones de estudiantes
11    const array< int, LibroCalificaciones::estudiantes > calificaciones =
12       { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
13    string nombreCurso = "CS101 Introduccion a la programacion en C++";
14
15    LibroCalificaciones miLibroCalificaciones( nombreCurso, calificaciones );
16    miLibroCalificaciones.mostrarMensaje();
17    miLibroCalificaciones.procesarCalificaciones();
18 } // fin de main
```

Fig. 7.17 | Crea un objeto `LibroCalificaciones` usando un arreglo de calificaciones, y después invoca a la función miembro `procesarCalificaciones` para analizarlas (parte 2 de 2).

7.7 Búsqueda y ordenamiento de datos en arreglos

En esta sección vamos a usar la función **`sort`** de la Biblioteca estándar de C++ para ordenar los elementos de un arreglo en forma ascendente, y la función **`binary_search`** integrada para determinar si un valor se encuentra o no en el arreglo.

Ordenamiento

El **ordenamiento** de los datos (colocarlos en orden ascendente o descendente) es una de las aplicaciones de cómputo más importantes. Un banco ordena todos los cheques por número de cuenta, para poder preparar los estados bancarios individuales al final de cada mes. Las compañías telefónicas ordenan sus directorios por apellido; y cuando las entradas contienen el *mismo* apellido, las ordenan por nombre de pila para facilitar la búsqueda de números telefónicos. Casi todas las empresas deben ordenar ciertos datos y, en algunos casos, cantidades masivas de ellos. El ordenamiento de datos es un problema intrigante que ha atraído algunos de los esfuerzos de investigación más intensos en el campo de las ciencias computacionales. En el capítulo 20 (en inglés en el sitio web) vamos a investigar e implementar algunos esquemas de ordenamiento, analizaremos su rendimiento y presentaremos la notación Big O para caracterizar lo duro que debe trabajar cada esquema para realizar su tarea.

Búsqueda

A menudo puede ser necesario determinar si un arreglo contiene un valor que concuerde con cierto **valor clave**. Al proceso de buscar un elemento específico de un arreglo se le llama **búsqueda**. En el capítulo 20 (en inglés en el sitio web) vamos a investigar e implementar dos algoritmos de búsqueda: la *búsqueda* lineal, que es simple pero lenta y se usa para buscar en un arreglo *desordenado*, y la *búsqueda binaria*, que es más compleja pero mucho más rápida y se usa para buscar en un arreglo *ordenado*.

Demostración de las funciones `sort` *y* `binary_search`

La figura 7.18 comienza por crear un arreglo desordenado de objetos `string` (líneas 13 y 14) y muestra el contenido del arreglo (líneas 17 a 19). A continuación, en la línea 21 se usa la función `sort` de la Biblioteca estándar de C++ para ordenar los elementos del arreglo `colores` en forma ascendente. Los argumentos de la función `sort` especifican el rango de elementos que deben ordenarse; en este caso, todo el arreglo. En capítulos posteriores hablaremos sobre los detalles completos de las funciones `begin` y `end` de la plantilla de clase `array`. Como veremos más adelante, la función `sort` puede usarse para ordenar los elementos de varios tipos diferentes de estructuras de datos. Las líneas 24 a 26 muestran el contenido del arreglo ordenado.

Las líneas 29 y 34 demuestran el uso de `binary_search` para determinar si un valor está en el arreglo. La secuencia de valores debe ordenarse primero en forma ascendente; `binary_search` *no* verifica esto por nosotros. Los primeros dos argumentos de la función representan el rango de elementos a

buscar y el tercero es la *clave de búsqueda*: el valor a localizar en el arreglo. La función devuelve un valor `bool` para indicar si se encontró el valor o no. En el capítulo 16 usaremos la función `find` estándar de C++ para obtener la ubicación de la clave de búsqueda en un arreglo.

```
1  // Fig. 7.18: fig07_18.cpp
2  // Búsqueda y ordenamiento de arreglos.
3  #include <iostream>
4  #include <iomanip>
5  #include <array>
6  #include <string>
7  #include <algorithm> // contiene sort y binary_search
8  using namespace std;
9
10 int main()
11 {
12    const size_t tamanioArreglo = 7; // tamaño del arreglo colores
13    array< string, tamanioArreglo > colores = { "rojo", "naranja", "amarillo",
14       "verde", "azul", "indigo", "violeta" };
15
16    // imprime el arreglo original
17    cout << "Arreglo desordenado:\n";
18    for ( string color : colores )
19       cout << color << " ";
20
21    sort( colores.begin(), colores.end() ); // ordena el contenido de colores
22
23    // imprime el arreglo ordenado
24    cout << "\nArreglo ordenado:\n";
25    for ( string elemento : colores )
26       cout << elemento << " ";
27
28    // busca "indigo" en colores
29    bool encontro = binary_search( colores.begin(), colores.end(), "indigo" );
30    cout << "\n\n\"indigo\" " << ( encontro ? "se" : "no se" )
31       << " encuentra en colores" << endl;
32
33    // busca "cian" en colores
34    encontro = binary_search( colores.begin(), colores.end(), "cyan" );
35    cout << "\"cian\" " << ( encontro ? "se" : "no se" )
36       << " encuentra en colores" << endl;
37 } // fin de main
```

```
Arreglo desordenado:
rojo naranja amarillo verde azul indigo violeta
Arreglo ordenado:
amarillo azul indigo naranja rojo verde violeta

"indigo" se encuentra en colores
"cian" no se encuentra en colores
```

Fig. 7.18 | Búsqueda y ordenamiento de arreglos.

7.8 Arreglos multidimensionales

Es posible usar arreglos de dos dimensiones (subíndices) para representar **tablas de valores**, las cuales consisten en información ordenada en **filas** y **columnas**. Para identificar un elemento específico de una tabla, debemos especificar dos subíndices. Por convención, el primero identifica la *fila* del elemento y el segundo su *columna*. Los arreglos que requieren dos subíndices para identificar un elemento específico se llaman **arreglos bidimensionales** o **arreglos 2-D**. Los arreglos con dos o más dimensiones se conocen como **arreglos multidimensionales** y pueden tener más de dos dimensiones. La figura 7.19 ilustra un arreglo bidimensional a, que contiene tres filas y cuatro columnas (es decir, un arreglo de tres por cuatro). En general, a un arreglo con *m* filas y *n* columnas se le llama **arreglo de *m* por *n***.

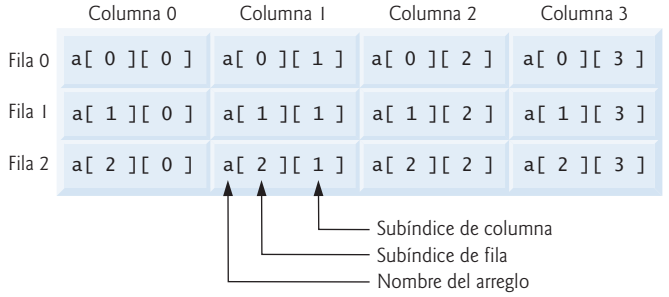


Fig. 7.19 | Arreglo bidimensional con tres filas y cuatro columnas.

Cada elemento en el arreglo a se identifica en la figura 7.19 mediante el nombre de un elemento de la forma `a[i][j]`, donde a es el nombre del arreglo, `i` y `j` son los subíndices que identifican en forma única a cada elemento en el arreglo a. Observe que los nombres de los elementos en la fila 0 tienen todos un primer subíndice de 0, y los nombres de los elementos en la columna 3 tienen un segundo subíndice de 3.

Error común de programación 7.5

Es un error hacer referencia al elemento `a[x][y]` de un arreglo bidimensional en forma incorrecta como `a[x, y]`. En realidad, `a[x, y]` se trata como `a[y]`, ya que C++ evalúa la expresión `x, y` (que contiene un operador coma) simplemente como `y` (la última de las expresiones separadas por coma).

La figura 7.20 demuestra cómo inicializar arreglos bidimensionales en las declaraciones. Las líneas 13 y 14 declaran cada una un arreglo de arreglos, con dos filas y tres columnas. Observe la declaración de tipo `array` anidada. En cada `array` se especifica el tipo de sus elementos como:

```
array< int, columnas >
```

para indicar que cada `array` contiene como elementos arreglos de tres elementos de valores `int`; la constante `columnas` tiene el valor de 3.

```
1  // Fig. 7.20: fig07_20.cpp
2  // Inicialización de arreglos multidimensionales.
3  #include <iostream>
```

Fig. 7.20 | Inicialización de arreglos multidimensionales (parte 1 de 2).

```
4  #include <array>
5  using namespace std;
6
7  const size_t filas = 2;
8  const size_t columnas = 3;
9  void imprimirAreglo( const array< array< int, columnas >, filas> & );
10
11 int main()
12 {
13    array< array< int, columnas >, filas > arreglo1 = { 1, 2, 3, 4, 5, 6 };
14    array< array< int, columnas >, filas > arreglo2 = { 1, 2, 3, 4, 5 };
15
16    cout << "Los valores en el arreglo1 por fila son:" << endl;
17    imprimirArreglo( arreglo1 );
18
19    cout << "\nLos valores en el arreglo2 por fila son:" << endl;
20    imprimirArreglo( arreglo2 );
21 } // fin de main
22
23 // imprime arreglo con dos filas y tres columnas
24 void imprimirArreglo( const array< array< int, columnas >, filas> & a )
25 {
26    // itera a través de las filas del arreglo
27    for ( auto const &fila : a )
28    {
29       // itera por las columnas de la fila actual
30       for ( auto const &elemento : fila )
31          cout << elemento << ' ';
32
33       cout << endl; // inicia nueva línea de salida
34    } // fin de for externo
35 } // fin de la función imprimirArreglo
```

```
Los valores en el arreglo1 por fila son:
1 2 3
4 5 6
Los valores en el arreglo2 por fila son:
1 2 3
4 5 0
```

Fig. 7.20 | Inicialización de arreglos multidimensionales (parte 2 de 2).

La declaración de `arreglo1` (línea 13) proporciona seis inicializadores. El compilador inicializa los elementos de la fila 0 seguidos de los elementos de la fila 1. Así, los primeros tres valores inicializan los elementos de la fila 0 con 1, 2 y 3, y los últimos tres inicializan los elementos de la fila 1 con 4, 5 y 6. La declaración de `arreglo2` (línea 14) proporciona sólo cinco inicializadores. Éstos se asignan a la fila 0, después a la fila 1. Cualquier elemento que no tenga un inicializador explícito se inicializa con *cero*, por lo que `arreglo2[1][2]` es 0.

El programa llama a la función `imprimirArreglo` para imprimir cada uno de los elementos del arreglo. Observe que el prototipo de la función (línea 9) y la definición (líneas 24 a 35) especifican que la función recibe un arreglo de dos filas y tres columnas. El parámetro recibe el arreglo por referencia y se declara como `const`, ya que la función no modifica los elementos del arreglo.

Instrucciones `for` *anidadas basadas en el rango*

Para procesar los elementos de un arreglo bidimensional, usamos un ciclo anidado en donde el ciclo *externo* itera a través de las *filas* y el ciclo *interno* itera a través de las *columnas* de una fila dada. El ciclo anidado

de la función `imprimirArreglo` se implementa con instrucciones `for` basadas en el rango. Las líneas 27 y 30 introducen la palabra clave **`auto`** de C++, la cual indica al compilador que infiera (determine) el tipo de datos de una variable con base en el valor inicializador de ésta. La variable de rango `fila` del ciclo inferior se inicializa con un elemento del parámetro `a`. Si analizamos la declaración del arreglo, podemos ver que contiene elementos del tipo

```
array< int, columnas >
```

de modo que el compilador infiera que `fila` se refiere a un arreglo de tres elementos de valores `int` (de nuevo, `columnas` es 3). La parte `const &` en la declaración de `fila` indica que la referencia *no puede* usarse para modificar las filas y evita que cada fila se *copie* a la variable de rango. La variable de rango `elemento` del ciclo interno se inicializa con un elemento del arreglo representado por `fila`, por lo que el compilador infiere que `elemento` hace referencia a un `int` debido a que cada fila contiene tres valores `int`. En un IDE, por lo general pasamos el ratón sobre una variable declarada con `auto` y el IDE muestra el tipo inferido por la variable. La línea 31 muestra el valor de una fila y columna dadas.

Instrucciones `for` *anidadas, controladas por contador*

Podríamos haber implementado el ciclo anidado con una repetición controlada por contador de la siguiente manera:

```
for ( size_t fila = 0; fila < a.size(); ++fila )
{
   for ( size_t columna = 0; columna < a[ fila ].size(); ++columna )
      cout << a[ fila ][ columna ] << ' ';

   cout << endl;
} // fin de for externo
```

Otras manipulaciones comunes de arreglos

Muchas manipulaciones comunes en los arreglos utilizan instrucciones de repetición `for`. Por ejemplo, la siguiente instrucción `for` establece todos los elementos en la fila 2 del arreglo `a` de la figura 7.19 con cero:

```
for ( size_t columna = 0; columna < 4; ++columna )
   a[ 2 ][ columna ] = 0;
```

La instrucción `for` sólo varía el segundo subíndice (es decir, el subíndice de la columna). La instrucción `for` anterior es equivalente a las siguientes instrucciones de asignación:

```
a[ 2 ][ 0 ] = 0;
a[ 2 ][ 1 ] = 0;
a[ 2 ][ 2 ] = 0;
a[ 2 ][ 3 ] = 0;
```

La siguiente instrucción `for` controlada por contador anidada determina el total de *todos* los elementos en el arreglo a de la figura 7.19:

```
total = 0;

for ( size_t fila = 0; fila < a.size(); ++fila )
   for ( size_t columna = 0; columna < a[ fila ].size(); ++columna )
      total += a[ fila ][ columna ];
```

La instrucción `for` calcula el total de los elementos del arreglo, una fila a la vez. La instrucción `for` exterior empieza estableciendo `fila` (es decir, el subíndice de la fila) en `0`, de manera que la instrucción `for` interior pueda calcular el total de los elementos de la fila 0. Después, la instrucción `for` exterior incrementa `fila` a `1`, de manera que se pueda obtener el total de los elementos de la fila 1. Luego, la instrucción `for` exterior incrementa `fila` a `2`, de manera que pueda obtenerse el total de los elementos de la fila 2. Cuando termina la instrucción `for` anidada, `total` contiene la suma de todos los elementos del arreglo. Este ciclo anidado puede implementarse con instrucciones `for` basadas en el rango, como:

```
total = 0;
for ( auto fila : a ) // por cada fila
   for ( auto columna : fila ) // por cada columna en la fila
      total += columna;
```

7.9 Caso de estudio: la clase `LibroCalificaciones` que usa un arreglo bidimensional

En la sección 7.6 presentamos la clase `LibroCalificaciones` (figuras 7.15 y 7.16), la cual utilizó un arreglo unidimensional para almacenar las calificaciones de los estudiantes de un solo examen. En la mayoría de los semestres, los estudiantes presentan varios exámenes. Es probable que los profesores quieran analizar las calificaciones a lo largo de todo el semestre, tanto para un solo estudiante como para la clase en general.

Cómo almacenar las calificaciones de los estudiantes en un arreglo bidimensional en la clase `LibroCalificaciones`

La figura 7.21 muestra el resultado en donde se sintetizan las calificaciones de 10 estudiantes de tres exámenes. Almacenamos las calificaciones como un arreglo bidimensional en un objeto de la siguiente versión de la clase `LibroCalificaciones` de las figuras 7.22 y 7.23. Cada fila del arreglo representa las calificaciones de un solo estudiante durante todo el curso, y cada columna representa todas las calificaciones que obtuvieron los estudiantes para un examen específico. Un programa cliente, como el de la figura 7.24, pasa el arreglo como argumento para el constructor de `LibroCalificaciones`. Como hay 10 estudiantes y tres exámenes, usamos un arreglo de tres por diez para almacenar las calificaciones.

```
Bienvenido al libro de calificaciones para
CS101 Introduccion a la programacion en C++!

Las calificaciones son:

            Prueba 1  Prueba 2  Prueba 3  Promedio
Estudiante  1    87        96        70     84.33
Estudiante  2    68        87        90     81.67
Estudiante  3    94       100        90     94.67
Estudiante  4   100        81        82     87.67
Estudiante  5    83        65        85     77.67
Estudiante  6    78        87        65     76.67
Estudiante  7    85        75        83     81.00
Estudiante  8    91        94       100     95.00
Estudiante  9    76        72        84     77.33
Estudiante 10    87        93        73     84.33
```

Fig. 7.21 | Salida de `LibroCalificaciones` que usa dos arreglos bidimensionales (parte 1 de 2).

```
La calificacion mas baja en el libro de calificaciones es 65
La calificacion mas alta en el libro de calificaciones es 100

Distribucion general de calificaciones:
  0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: ***
70-79: ******
80-89: ***********
90-99: *******
  100: ***
```

Fig. 7.21 | Salida de `LibroCalificaciones` que usa dos arreglos bidimensionales (parte 2 de 2).

```
1  // Fig. 7.22: LibroCalificaciones.h
2  // Definición de la clase LibroCalificaciones que utiliza un
3  // arreglo bidimensional para almacenar calificaciones de una prueba.
4  // Las funciones miembro se definen en LibroCalificaciones.cpp
5  #include <array>
6  #include <string>
7
8  // definición de la clase LibroCalificaciones
9  class LibroCalificaciones
10 {
11 public:
12    // constantes
13    static const size_t estudiantes = 10; // número de estudiantes
14    static const size_t pruebas = 3; // número de pruebas
15
16    // el constructor inicializa el nombre del curso y el arreglo de calificaciones
17    LibroCalificaciones( const std::string &,
18       std::array< std::array< int, pruebas, estudiantes > & );
19
20    void establecerNombreCurso( const std::string & ); // establece el nombre del
                                                          curso
21    std::string obtenerNombreCurso() const; // obtiene el nombre del curso
22    void mostrarMensaje() const; // muestra un mensaje de bienvenida
23    void procesarCalificaciones() const; // realiza varias operaciones en los
                                              datos de las calificaciones
24    int obtenerMinimo() const; // encuentra el valor mínimo en el libro de
                                 calificaciones
25    int obtenerMaximo() const; // encuentra el valor máximo en el libro de
                                 calificaciones
26    double obtenerPromedio( const std::array< int, pruebas > & ) const;
27    void imprimirGraficoBarras() const; // imprime gráfico de barras de la
                                          distribución de calificaciones
28    void imprimirCalificaciones() const; // imprime el contenido del arreglo
                                           calificaciones
29 private:
30    std::string nombreCurso; // nombre del curso para este libro de calificaciones
31    std::array< std::array< int, pruebas >, estudiantes > calificaciones;
      // arreglo bidimensional de calificaciones
32 }; // fin de la clase LibroCalificaciones
```

Fig. 7.22 | Definición de la clase `LibroCalificaciones` que utiliza un arreglo bidimensional para almacenar calificaciones de una prueba.

```
1  // Fig. 7.23: LibroCalificaciones.cpp
2  // Definiciones de las funciones miembro de LibroCalificaciones,
3  // que utiliza un arreglo bidimensional para almacenar calificaciones.
4  #include <iostream>
5  #include <iomanip> // manipuladores de flujo parametrizados
6  using namespace std;
7
8  // incluye la definición de la clase LibroCalificaciones de LibroCalificaciones.h
9  #include "LibroCalificaciones.h" // definición de la clase LibroCalificaciones
10
11 // constructor con dos argumentos que inicializa nombreCurso y el arreglo
      calificaciones
12 LibroCalificaciones::LibroCalificaciones( const string &nombre,
13    std::array< std::array< int, pruebas >, estudiantes > &arregloCalificaciones )
14    : nombreCurso( nombre ), calificaciones( arregloCalificaciones )
15 {
16 } // fin del constructor de LibroCalificaciones con dos argumentos
17
18 // función para establecer el nombre del curso
19 void LibroCalificaciones::establecerNombreCurso( const string &nombre )
20 {
21    nombreCurso = nombre; // almacena el nombre del curso
22 } // fin de la función establecerNombreCurso
23
24 // función para obtener el nombre del curso
25 string LibroCalificaciones::obtenerNombreCurso() const
26 {
27    return nombreCurso;
28 } // fin de la función obtenerNombreCurso
29
30 // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
31 void LibroCalificaciones::mostrarMensaje() const
32 {
33    // esta instrucción llama a obtenerNombreCurso para obtener el
34    // nombre del curso que representa este LibroCalificaciones
35    cout << "Bienvenido al libro de calificaciones para\n"
       << obtenerNombreCurso() << "!"
36       << endl;
37 } // fin de la función mostrarMensaje
38
39 // realiza varias operaciones con los datos
40 void LibroCalificaciones::procesarCalificaciones() const
41 {
42    // imprime el arreglo calificaciones
43    imprimirCalificaciones();
44
45    // llama a las funciones obtenerMinimo y obtenerMaximo
46    cout << "\nLa calificacion mas baja en el libro de calificaciones es "
       << obtenerMinimo()
47       << "\nLa calificacion mas alta en el libro de calificaciones es "
         << obtenerMaximo() << endl;
48
49    // imprime gráfico de distribución de todas las calificaciones en todas las
        pruebas
50    imprimirGraficoBarras();
51 } // fin de la función procesarCalificaciones
```

Fig. 7.23 | Definiciones de las funciones miembro de `LibroCalificaciones`, que utiliza un arreglo bidimensional para almacenar calificaciones (parte 1 de 4).

```
52
53  // encuentra la calificación más baja en todo el libro de calificaciones
54  int LibroCalificaciones::obtenerMinimo() const
55  {
56     int calificacionInf = 100; // asume que la calificación más baja es 100
57
58     // itera a través de las filas del arreglo calificaciones
59     for ( auto const &estudiante : calificaciones )
60     {
61        // itera a través de las columnas de la fila actual
62        for ( auto const &calificacion : estudiante )
63        {
64           // si la calificación actual es menor que calificacionInf, la asigna a
              calificacionInf
65           if ( calificacion < calificacionInf )
66              calificacionInf = calificacion; // nueva calificación más baja
67        } // fin de for interior
68     } // fin de for exterior
69
70     return calificacionInf; // devuelve la calificación más baja
71  } // fin de la función obtenerMinimo
72
73  // busca la calificación más alta en todo el libro de calificaciones
74  int LibroCalificaciones::obtenerMaximo() const
75  {
76     int calificacionSup = 0; // asume que la calificación más alta es 0
77
78     // itera a través de las filas del arreglo calificaciones
79     for ( auto const &estudiante : calificaciones )
80     {
81        // itera a través de las columnas de la fila actual
82        for ( auto const &calificacion : estudiante  )
83        {
84           // si la calificación actual es mayor que calificacionSup, la asigna a
              calificacionSup
85           if ( calificacion > calificacionSup )
86              calificacionSup = calificacion; // nueva calificación más alta
87        } // fin de for interior
88     } // fin de for exterior
89
90     return calificacionSup; // devuelve la calificación más alta
91  } // fin de la función obtenerMaximo
92
93  // determina la calificación promedio para un conjunto específico de
     calificaciones
94  double LibroCalificaciones::obtenerPromedio( const array<int, pruebas>
    &conjuntoDeCalificaciones ) const
95  {
96     int total = 0; // inicializa el total
97
98     // suma las calificaciones en el arreglo
99     for ( int calificacion : conjuntoDeCalificaciones )
100       total += calificacion;
101
```

Fig. 7.23 | Definiciones de las funciones miembro de `LibroCalificaciones`, que utiliza un arreglo bidimensional para almacenar calificaciones (parte 2 de 4).

```
102    // devuelve el promedio de las calificaciones
103    return static_cast< double >( total ) / conjuntoDeCalificaciones.size();
104 } // fin de la función obtenerPromedio
105
106 // imprime gráfico de barras que muestra la distribución de las calificaciones
107 void LibroCalificaciones::imprimirGraficoBarras() const
108 {
109    cout << "\nDistribucion general de calificaciones:"  << endl;
110
111    // almacena la frecuencia de las calificaciones en cada rango
          de 10 calificaciones
112    const size_t tamanioFrecuencia = 11;
113    array< unsigned int, tamanioFrecuencia > frecuencia = {}; // inicializa con ceros
114
115    // para cada calificación, incrementa la frecuencia apropiada
116    for ( auto const &estudiante : calificaciones )
117       for ( auto const &prueba : estudiante )
118          ++frecuencia[ prueba / 10 ];
119
120    // para cada frecuencia de calificaciones, imprime la barra en el gráfico
121    for ( size_t cuenta = 0; cuenta < tamanioFrecuencia; ++cuenta )
122    {
123       // imprime las etiquetas de las barras ("0-9:", ..., "90-99:", "100:" )
124       if ( 0 == cuenta )
125          cout << "  0-9: ";
126       else if ( 10 == cuenta )
127          cout << "  100: ";
128       else
129          cout << cuenta * 10 << "-" << ( cuenta * 10 ) + 9 << ": ";
130
131       // imprime barra de asteriscos
132       for ( unsigned int estrellas = 0; estrellas < frecuencia[ cuenta ];
          estrellas++ )
133          cout << '*';
134
135       cout << endl; // empieza una nueva línea de salida
136    } // fin de for exterior
137 } // fin de la función imprimirGraficoBarras
138
139 // imprime el contenido del arreglo calificaciones
140 void LibroCalificaciones::imprimirCalificaciones() const
141 {
142    cout << "\nLas calificaciones son:\n\n";
143    cout << "            "; // alinea los encabezados de las columnas
144
145    // crea un encabezado de columna para cada una de las pruebas
146    for ( size_t prueba = 0; prueba < pruebas; ++prueba )
147       cout << "Prueba " << prueba + 1 << "  ";
148
149    cout << "Promedio" << endl; // encabezado de la columna de promedio de
                                   estudiantes
150
```

Fig. 7.23 | Definiciones de las funciones miembro de `LibroCalificaciones`, que utiliza un arreglo bidimensional para almacenar calificaciones (parte 3 de 4).

```
151    // crea filas/columnas de texto que representan el arreglo calificaciones
152    for ( size_t estudiante = 0; estudiante < calificaciones.size(); ++estudiante )
153    {
154       cout << "Estudiante " << setw( 2 ) << estudiante + 1;
155
156       // imprime las calificaciones del estudiante
157       for ( size_t prueba = 0; prueba < calificaciones[ estudiante ].size();
          ++prueba )
158          cout << setw( 8 ) << calificaciones[ estudiante ][ prueba ];
159
160       // llama a la función miembro obtenerPromedio para calcular el promedio del
161       // estudiante; pasa la fila de calificaciones como argumento
162       double promedio = obtenerPromedio( calificaciones[ estudiante ] );
163       cout << setw( 9 ) << setprecision( 2 ) << fixed << promedio << endl;
164    } // fin de for exterior
165 } // fin de la función imprimirCalificaciones
```

Fig. 7.23 | Definiciones de las funciones miembro de `LibroCalificaciones`, que utiliza un arreglo bidimensional para almacenar calificaciones (parte 4 de 4).

Generalidades de las funciones de la clase `LibroCalificaciones`

Cinco funciones miembro (declaradas en las líneas 24 a 28 de la figura 7.22) realizan manipulaciones de arreglos para procesar las calificaciones. Cada una de estas funciones miembro es similar a su contraparte en la versión anterior de la clase `LibroCalificaciones` con un arreglo unidimensional (figuras 7.15 y 7.16). La función miembro `obtenerMinimo` (definida en las líneas 54 a 71 de la figura 7.23) determina la calificación más baja de cualquier estudiante durante el semestre. La función miembro `obtenerMaximo` (definida en las líneas 74 a 91 de la figura 7.23) determina la calificación más alta de cualquier estudiante durante el semestre. La función miembro `obtenerPromedio` (líneas 94 a 104 de la figura 7.23) determina el promedio semestral de un estudiante específico. La función miembro `imprimirGraficoBarras` (líneas 107 a 137 de la figura 7.23) imprime un gráfico de barras de la distribución de todas las calificaciones de los estudiantes durante el semestre. La función miembro `imprimirCalificaciones` (líneas 140 a 165 de la figura 7.23) imprime el arreglo bidimensional en formato tabular, junto con el promedio semestral de cada estudiante.

Las funciones `obtenerMinimo` *y* `obtenerMaximo`

Cada una de las funciones miembro `obtenerMinimo`, `obtenerMaximo`, `imprimirGraficoBarras` e `imprimirCalificaciones` iteran a través del arreglo `calificaciones` mediante el uso de instrucciones `for` anidadas basadas en rango, o de instrucciones `for` controladas por contador. Por ejemplo, considere la instrucción `for` anidada en la función miembro `obtenerMinimo` (líneas 59 a 68). La instrucción `for` exterior itera por las filas que representan a cada `estudiante`, y la instrucción `for` interior itera a través de las calificaciones de un estudiante específico. Cada calificación se compara con la variable `calificacionInf` en el cuerpo de la instrucción `for` interior. Si una calificación es menor que `calificacionInf`, a `calificacionInf` se le asigna esa calificación. Esto se repite hasta que se hayan recorrido todas las filas y columnas de `calificaciones`. Cuando se completa la ejecución de la instrucción anidada, `calificacionInf` contiene la calificación más baja de todo el arreglo bidimensional. La función miembro `obtenerMaximo` funciona de manera similar a la función miembro `obtenerMinimo`.

La función miembro `imprimirGraficoBarras`

La función miembro `imprimirGraficoBarras` en la figura 7.23 es casi idéntica a la de la figura 7.16. Sin embargo, para imprimir la distribución de calificaciones en general durante todo un semestre, la función utiliza una instrucción `for` anidada (líneas 116 a 118) para incrementar los elementos del arreglo uni-

dimensional `frecuencia`, con base en todas las calificaciones en el arreglo bidimensional. El resto del código en cada una de las dos funciones miembro `imprimirGraficoBarras` que muestran el gráfico es idéntico.

La función `imprimirCalificaciones`

La función miembro `imprimirCalificaciones` (líneas 140 a 165) utiliza instrucciones `for` anidadas, controladas por contador, para imprimir valores del arreglo `calificaciones`, además del promedio semestral de cada estudiante. La salida en la figura 7.21 muestra el resultado, el cual se asemeja al formato tabular del libro de calificaciones real de un profesor. Las líneas 146 y 147 imprimen los encabezados de columna para cada prueba. Aquí utilizamos una instrucción `for` controlada por contador, para poder identificar cada prueba con un número. De manera similar, la instrucción `for` en las líneas 152 a 164 imprime primero una etiqueta de fila mediante el uso de una variable contador para identificar a cada estudiante (línea 154). Aunque los subíndices de los arreglos empiezan en 0, en las líneas 147 y 154 se imprimen `prueba + 1` y `estudiante + 1` en forma respectiva, para producir números de prueba y estudiante que empiecen en 1 (vea la figura 7.21) La instrucción `for` interior en las líneas 157 y 158 utiliza la variable contador `estudiante` de la instrucción `for` exterior para iterar a través de una fila específica del arreglo `calificaciones`, e imprime la calificación de la prueba de cada estudiante. Por último, en la línea 162 se obtiene el promedio semestral de cada estudiante, para lo cual se pasa la fila actual de `calificaciones` (es decir, `calificaciones[estudiante]`) a la función miembro `obtenerPromedio`.

La función `obtenerPromedio`

La función miembro `obtenerPromedio` (líneas 94 a 104) toma como argumento un arreglo unidimensional de resultados de la prueba para un estudiante específico. Cuando la línea 162 llama a `obtenerPromedio`, el primer argumento es `calificaciones[ estudiante ]`, el cual especifica que debe pasarse una fila específica del arreglo bidimensional `calificaciones` a `obtenerPromedio`. Por ejemplo, con base en el arreglo creado en la figura 7.24, el argumento `calificaciones[1]` representa los tres valores (un arreglo unidimensional de calificaciones) almacenados en la fila `1` del arreglo bidimensional `calificaciones`. Los elementos de un arreglo bidimensional son arreglos unidimensionales. La función miembro `obtenerPromedio` calcula la suma de los elementos del arreglo, divide el total entre el número de resultados de la prueba y devuelve el resultado de punto flotante como un valor `double` (línea 103).

Prueba de la clase `LibroCalificaciones`

El programa de la figura 7.24 crea un objeto de la clase `LibroCalificaciones` (figuras 7.22 y 7.23) mediante el uso del arreglo bidimensional de valores `int` llamado `arregloCalif` (el cual se declara y se inicializa en las líneas 11 a 21). En la línea 11 se accede a las constantes `static` llamadas `estudiantes` y `pruebas` de la clase `LibroCalificaciones` para indicar el tamaño de cada dimensión del arreglo `arregloCalificaciones`. En las líneas 23 y 24 se pasan el nombre de un curso y `calificaciones` al constructor de `LibroCalificaciones`. Después, en las líneas 25 y 26 se invocan las funciones miembro `mostrarMensaje` y `procesarCalificaciones` de `miLibroCalificaciones`, para mostrar un mensaje de bienvenida y obtener un informe que sintetice las calificaciones de los estudiantes para el semestre, respectivamente.

```
1  // Fig. 7.24: fig07_24.cpp
2  // Crea un objeto LibroCalificaciones mediante el uso de un arreglo bidimensional
   //    de calificaciones.
3  #include <array>
4  #include "LibroCalificaciones.h" // definición de la clase LibroCalificaciones
5  using namespace std;
6
```

Fig. 7.24 | Crea un objeto `LibroCalificaciones` mediante el uso de un arreglo bidimensional de calificaciones, y después invoca a la función miembro `procesarCalificaciones` para analizarlas (parte 1 de 2).

```
// la función main empieza la ejecución del programa
int main()
{
   // arreglo bidimensional de calificaciones de estudiantes
   array< array< int, LibroCalificaciones::pruebas >,
   LibroCalificaciones::estudiantes > calificaciones =
      { 87, 96, 70,
        68, 87, 90,
        94, 100, 90,
        100, 81, 82,
        83, 65, 85,
        78, 87, 65,
        85, 75, 83,
        91, 94, 100,
        76, 72, 84,
        87, 93, 73 };

   LibroCalificaciones miLibroCalificaciones(
      "CS101 Introduccion a la programacion en C++", calificaciones );
   miLibroCalificaciones.mostrarMensaje();
   miLibroCalificaciones.procesarCalificaciones();
} // fin de main
```

Fig. 7.24 | Crea un objeto `LibroCalificaciones` mediante el uso de un arreglo bidimensional de calificaciones, y después invoca a la función miembro `procesarCalificaciones` para analizarlas (parte 2 de 2).

7.10 Introducción a la plantilla de clase vector de la Biblioteca estándar de C++

Ahora introduciremos la plantilla de clase **`vector`** de la Biblioteca estándar de C++, que es similar a la plantilla de clase `array`, pero también soporta el ajuste de tamaño dinámico. Con la excepción de las características que modifican a un vector, las demás características que se muestran en la figura 7.25 también funcionan en los arreglos. La plantilla de clase estándar `vector` está definida en el encabezado `<vector>` (línea 5) y pertenece al espacio de nombres `std`. En el capítulo 15 hablaremos sobre la funcionalidad completa de `vector`. Al final de esta sección demostraremos las herramientas de comprobación de límites de la clase `vector` e introduciremos el mecanismo de manejo de excepciones de C++, que puede usarse para detectar y manejar el índice fuera de límites de un objeto `vector`.

```
// Fig. 7.25: fig07_25.cpp
// Demostración de la plantilla de clase vector de la Biblioteca estándar de C++.
#include <iostream>
#include <iomanip>
#include <vector>
#include <stdexcept>
using namespace std;

void imprimirVector( const vector< int > & ); // muestra el vector
void recibirVector( vector< int > & ); // introduce los valores en el vector

int main()
{
```

Fig. 7.25 | Demostración de la plantilla de clase `vector` de la Biblioteca estándar de C++ (parte 1 de 4).

```
14    vector< int > enteros1( 7 ); // vector de 7 elementos< int >
15    vector< int > enteros2( 10 ); // vector de 10 elementos< int >
16
17    // imprime el tamaño y el contenido de enteros1
18    cout << "El tamanio del vector enteros1 es " << enteros1.size()
19       << "\nvector despues de la inicializacion:" << endl;
20    imprimirVector( enteros1 );
21
22    // imprime el tamaño y el contenido de enteros2
23    cout << "\nEl tamanio del vector enteros2 es " << enteros2.size()
24       << "\nvector despues de la inicializacion:" << endl;
25    imprimirVector( enteros2 );
26
27    // recibe e imprime enteros1 y enteros2
28    cout << "\nEscriba 17 enteros:" << endl;
29    recibirVector( enteros1 );
30    recibirVector( enteros2 );
31
32    cout << "\nDespues de la entrada, los vectores contienen:\n"
33       << "enteros1:" << endl;
34    imprimirVector( enteros1 );
35    cout << "enteros2:" << endl;
36    imprimirVector( enteros2 );
37
38    // usa el operador de desigualdad (!=) con objetos vector
39    cout << "\nEvaluacion: enteros1 != enteros2" << endl;
40
41    if ( enteros1 != enteros2 )
42       cout << "enteros1 y enteros2 no son iguales" << endl;
43
44    // crea el vector enteros3 usando enteros1 como un
45    // inicializador; imprime el tamaño y el contenido
46    vector< int > enteros3( enteros1 ); // constructor de copia
47
48    cout << "\nEl tamanio del vector enteros3 es " << enteros3.size()
49       << "\nvector despues de la inicializacion:" << endl;
50    imprimirVector( enteros3 );
51
52    // usa el operador de asignacion (=) sobrecargado
53    cout << "\nAsignacion de enteros2 a enteros1:" << endl;
54    enteros1 = enteros2; // asigna enteros2 a enteros1
55
56    cout << "enteros1:" << endl;
57    imprimirVector( enteros1 );
58    cout << "enteros2:" << endl;
59    imprimirVector( enteros2 );
60
61    // usa el operador de igualdad (==) con objetos vector
62    cout << "\nEvaluacion: enteros1 == enteros2" << endl;
63
64    if ( enteros1 == enteros2 )
65       cout << "enteros1 y enteros2 son iguales" << endl;
66
```

Fig. 7.25 | Demostración de la plantilla de clase `vector` de la Biblioteca estándar de C++ (parte 2 de 4).

```
67    // usa corchetes para asignar el valor en la ubicación 5 como un rvalue
68    cout << "\nenteros1[5] es " << enteros1[ 5 ];
69
70    // usa corchetes para crear lvalue
71    cout << "\n\nAsignacion de 1000 a enteros1[5]" << endl;
72    enteros1[ 5 ] = 1000;
73    cout << "enteros1:" << endl;
74    imprimirVector( enteros1 );
75
76    // intenta usar subíndice fuera de rango
77    try
78    {
79       cout << "\nIntento de mostrar enteros1.at( 15 )" << endl;
80       cout << enteros1.at( 15 ) << endl; // ERROR: fuera de rango
81    } // fin de try
82    catch ( out_of_range &ex )
83    {
84       cerr << "Ocurrio una excepcion: " << ex.what() << endl;
85    } // fin de catch
86
87    // cambiar el tamaño de un vector
88    cout << "\nEl tamanio actual de enteros3 es: " << enteros3.size() << endl;
89    enteros3.push_back( 1000 ); // agrega 1000 al final del vector
90    cout << "El tamanio nuevo de enteros3 es: " << enteros3.size() << endl;
91    cout << "Ahora enteros3 contiene: ";
92    imprimirVector( enteros3 );
93 } // fin de main
94
95 // imprime el contenido del vector
96 void imprimirVector( const vector< int > &arreglo )
97 {
98    for ( int elemento : elementos )
99       cout << elemento << " ";
100
101   cout << endl;
102 } // fin de la función imprimirVector
103
104 // recibe el contenido del vector
105 void recibirVector( vector< int > &arreglo )
106 {
107   for ( int &elemento : elementos )
108      cin >> elemento;
109 } // fin de la función recibirVector
```

```
El tamanio del vector enteros1 es 7
vector despues de la inicializacion:
0 0 0 0 0 0 0

El tamanio del vector enteros2 es 10
vector despues de la inicializacion:
0 0 0 0 0 0 0 0 0 0
```

Fig. 7.25 | Demostración de la plantilla de clase `vector` de la Biblioteca estándar de C++ (parte 3 de 4).

```
Escriba 17 enteros:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

Despues de la entrada, los vectores contienen:
enteros1:
1 2 3 4 5 6 7
enteros2:
8 9 10 11 12 13 14 15 16 17

Evaluacion: enteros1 != enteros2
enteros1 y enteros2 no son iguales

El tamanio del vector enteros3 es 7
vector despues de la inicializacion:
1 2 3 4 5 6 7

Asignacion de enteros2 a enteros1:
enteros1:
8 9 10 11 12 13 14 15 16 17
enteros2:
8 9 10 11 12 13 14 15 16 17

Evaluacion: enteros1 == enteros2
enteros1 y enteros2 son iguales

enteros1[5] es 13

Asignacion de 1000 a enteros1[5]
enteros1:
8 9 10 11 12 1000 14 15 16 17

Intento de mostrar enteros1.at( 15 )
An exception ocurred: invalid vector<T> subscript

El tamanio actual de enteros3 es: 7
El tamanio nuevo de enteros3 es: 8
Ahora enteros3 contiene: 1 2 3 4 5 6 7 1000
```

Fig. 7.25 | Demostración de la plantilla de clase `vector` de la Biblioteca estándar de C++ (parte 4 de 4).

Creación de objetos `vector`

En las líneas 14 y 15 se crean dos objetos `vector` que almacenan valores de tipo `int`: `enteros1` contiene siete elementos, y `enteros2` contiene 10 elementos. De manera predeterminada, todos los elementos de cada objeto `vector` se establecen en 0. Al igual que los arreglos, pueden definirse objetos `vector` para almacenar la mayoría de los tipos de datos, al sustituir `int` en `vector< int >` con el tipo de datos apropiado.

Función miembro `size` *de* `vector`*; función* `imprimirVector`

En la línea 18 se utiliza la función miembro `size` de `vector` para obtener el tamaño (es decir, el número de elementos) de `enteros1`. En la línea 20 se pasa `enteros1` a la función `imprimirVector` (líneas 96 a 102), la cual usa una instrucción `for` basada en rango para obtener el valor en cada elemento del `vector` para imprimirlo. Al igual que con la plantilla de clase `array`, es posible hacer esto mediante un ciclo controlado por contador y el operador de subíndice (`[]`). En las líneas 23 y 25 se realizan las mismas tareas para `enteros2`.

Función `recibirVector`

En las líneas 29 y 30 se pasan los objetos `enteros1` y `enteros2` a la función `recibirVector` (líneas 105 a 109) para que el usuario introduzca los valores de los elementos de cada `vector`. La función utiliza una instrucción `for` basada en rango, con una variable de rango que es una referencia a un `int` para formar *lvalues* que se utilizan para almacenar los valores de entrada en cada elemento `vector`.

Comparación de desigualdad de objetos `vector`

En la línea 41 se demuestra que los objetos `vector` se pueden comparar entre sí mediante el operador `!=`. Si el contenido de dos objetos `vector` no es igual, el operador devuelve `true`; en caso contrario devuelve `false`.

Inicialización de un `vector` *con el contenido de otro*

La plantilla de clase `vector` de la Biblioteca estándar de C++ nos permite crear un nuevo objeto `vector` que se inicializa con el contenido de un `vector` existente. En la línea 46 se crea un objeto `vector` llamado `enteros3` y se inicializa con una copia de `enteros1`. Esto invoca al *constructor de copia* de `vector` para realizar la operación de copia. En el capítulo 10 aprenderá con detalle acerca de los constructores de copia. En las líneas 48 a 50 se imprime el tamaño y el contenido de `enteros3` para demostrar que se inicializó en forma correcta.

Asignar objetos `vector` *y comparar si dos objetos* `vector` *son iguales*

En la línea 54 se asigna `enteros2` a `enteros1`, lo cual demuestra que el operador de asignación (=) se puede usar con objetos `vector`. En las líneas 56 a 59 se imprime el contenido de ambos objetos para mostrar que ahora contienen valores idénticos. Después, en la línea 64 se compara `enteros1` con `enteros2` mediante el operador de igualdad (==) para determinar si el contenido de los dos objetos es el mismo después de la asignación en la línea 54 (lo cual es cierto).

Uso del operador `[]` *para acceder a los elementos del* `vector` *y modificarlos*

En las líneas 68 y 70 se utilizan corchetes (`[]`) para obtener un elemento de `vector` y usarlo como *rvalue* y *lvalue*, respectivamente. En la sección 5.9 vimos que un *rvalue* no se puede modificar, pero un *lvalue* sí. Como es el caso con los arreglos, C++ *no realiza comprobación de límites cuando se accede a elementos de un objeto* `vector` *mediante el uso de corchetes*.[1] Por lo tanto, el programador debe asegurar que las operaciones en las que se utilizan los corchetes (`[]`) no traten accidentalmente de manipular elementos fuera de los límites del `vector`. Sin embargo, la plantilla de clase estándar `vector` cuenta con la capacidad de comprobar los límites en su función miembro `at` (al igual que la plantilla de clase `array`), que usamos en la línea 80 y describiremos en breve.

Manejo de excepciones: procesamiento de un subíndice fuera de rango

Una **excepción** indica un problema que ocurre mientras se ejecuta un programa. El nombre "excepción" sugiere que el problema ocurre con poca frecuencia; si la "regla" es que una instrucción se ejecuta normalmente en forma correcta, entonces el problema representa la "excepción a la regla". El **manejo de excepciones** nos permite crear **programas tolerantes a errores** que puedan resolver (o manejar) excepciones. En muchos casos, esto permite a un programa seguirse ejecutando como si no hubiera encontrado problemas. Por ejemplo, la figura 7.25 se ejecuta hasta completarse, aun cuando se trató de acceder a un subíndice fuera de rango. Los problemas más severos podrían evitar que un programa continúe su ejecución normal y tenga que notificar al usuario sobre el problema, para después terminar. Cuando una

1 Algunos compiladores tienen opciones para comprobación de límites y ayudar a evitar desbordamientos de búfer.

función detecta un problema, como el subíndice inválido de un arreglo o un argumento inválido, lanza una excepción; es decir, ocurre una excepción. Aquí presentamos brevemente el manejo de excepciones. Hablaremos con detalle en el capítulo 17 (en el sitio web), Manejo de excepciones: un análisis más detallado.

La instrucción `try`

Para manejar una excepción, coloque el código que pudiera lanzar una excepción en una **instrucción `try`** (líneas 77 a 85). El **bloque `try`** (líneas 77 a 81) contiene el código que podría *lanzar* una excepción, y el **bloque `catch`** (líneas 82 a 85) contiene el código que *maneja* la excepción, en caso de que ocurra. Como veremos en el capítulo 17, puede tener varios bloques `catch` para manejar diferentes tipos de excepciones que podrían lanzarse en el bloque try correspondiente. Si el código en el bloque `try` se ejecuta con éxito, se ignoran las líneas 82 a 85. Las llaves que delimitan los cuerpos de los bloques `try` y `catch` son obligatorias.

La función miembro **`at`** de `vector` cuenta con comprobación de límites y lanza una excepción si su argumento es un subíndice inválido. De manera predeterminada, esto hace que un programa de C++ termine. Si el subíndice es válido, la función `at` devuelve el elemento en la ubicación especificada como un *lvalue* modificable o un *lvalue* no modificable. Un *lvalue* no modificable es una expresión que identifica a un objeto en memoria (como un elemento en un `vector`), pero no puede usarse para modificar ese objeto. Si se hace una llamada a `at` en un arreglo `const` o mediante una referencia declarada como `const`, la función devuelve un *lvalue* no modificable.

Ejecución del bloque `catch`

Cuando el programa llama a la función miembro `at` de `vector` con el argumento `15` (línea 80), la función intenta acceder al elemento en la ubicación 15, que se encuentra *fuera* de los límites del `vector`; `enteros1` tiene sólo 10 elementos en este punto. Puesto que la comprobación de límites se realiza en tiempo de ejecución, la función miembro `at` de `vector` genera una excepción; específicamente, la línea 80 lanza una excepción **`out_of_range`** (del encabezado `<stdexcept>`) para notificar al programa sobre este problema. En este punto, el bloque `try` termina de inmediato y el bloque `catch` comienza a ejecutarse; si declaró variables en el bloque `try`, ahora están fuera de alcance y no pueden accederse en el bloque `catch`.

El bloque `catch` declara un tipo (`out_of_range`) y un parámetro de excepción (`ex`) que recibe como referencia. El bloque `catch` puede manejar excepciones del tipo especificado. Dentro del bloque puede usar el identificador del parámetro para interactuar con un objeto de excepción capturado.

Función miembro `what` *del parámetro de excepción*

Cuando las líneas 82 a 85 *atrapan* la excepción, el programa muestra un mensaje para indicar el problema que ocurrió. La línea 84 llama a la función miembro **`what`** del objeto excepción para obtener el mensaje de error que está almacenado en este objeto y mostrarlo. Una vez que se muestra el mensaje en este ejemplo, se considera que se manejó la excepción y el programa continúa con la siguiente instrucción después de la llave de cierre del bloque `catch`. En este ejemplo, las líneas 88 a 92 se ejecutan a continuación. Usaremos el manejo de excepciones de nuevo en los capítulos 9 a 12; el capítulo 17 presenta un análisis más detallado sobre el manejo de excepciones.

Cambiar el tamaño de un `vector`

Una de las diferencias clave entre un `vector` y un `array` es que un `vector` puede crecer en forma dinámica para dar cabida a más elementos. Para demostrar esto, en la línea 88 se muestra el tamaño actual de `enteros3`, en la línea 89 se llama a la función miembro **`push_back`** de `vector` para agregar un nuevo elemento que contiene 1000 al final del `vector` y en la línea 90 se muestra el nuevo tamaño de `enteros3`. Después en la línea 92 se muestra el nuevo contenido de `enteros3`.

C++11: Lista para inicializar un **vector**

Muchos de los ejemplos con `array` en este capítulo utilizan incializadores de listas para especificar los valores iniciales de los elementos de un arreglo. C++11 también permite esto para los objetos `vector` (y otras estructuras de datos de la Biblioteca estándar de C++). Al momento de escribir este libro, no había soporte para los inicializadores de listas para objetos `vector` en Visual C++.

7.11 Conclusión

En este capítulo empezó nuestra introducción a las estructuras de datos, explorando el uso de las plantillas de clase `array` y `vector` para almacenar datos y obtenerlos de listas y tablas de valores. Los ejemplos de este capítulo demostraron cómo declarar un arreglo, inicializarlo y hacer referencia a los elementos individuales de un arreglo. Pasamos arreglos a las funciones por referencia y utilizamos el calificador `const` para evitar que la función a la que se llamó modificara los elementos del arreglo, para hacer valer el principio del menor privilegio. Aprendió a usar la nueva instrucción `for` basada en rango para manipular todos los elementos de un arreglo. También le mostramos cómo usar las funciones `sort` y `binary_search` de la Biblioteca estándar de C++ para ordenar y buscar en un arreglo, respectivamente. Aprendió a declarar y manipular arreglos multidimensionales de arreglos. Utilizamos instrucciones `for` anidadas, controladas por contador y basadas en rango, para iterar a través de todas las filas y columnas de un arreglo bidimensional. Además le mostramos cómo usar `auto` para inferir el tipo de una variable con base en su valor inicializador. Finalmente, demostramos las herramientas de la plantilla de clase `vector` de la Biblioteca estándar de C++. En ese ejemplo, vimos cómo acceder a los elementos de objetos `array` y `vector` con la comprobación de límites y demostramos los conceptos básicos sobre manejo de excepciones. En capítulos posteriores continuaremos nuestra cobertura de las estructuras de datos.

Ya le hemos presentado los conceptos básicos de las clases, los objetos, las instrucciones de control, las funciones y los objetos `array`. En el capítulo 8 presentaremos una de las herramientas más poderosas de C++: el apuntador. Los apuntadores llevan la cuenta de la ubicación en donde se almacenan los datos y las funciones en memoria, lo cual nos permite manipular esos elementos en formas interesantes. Como veremos más adelante, C++ también cuenta con un elemento del lenguaje conocido como arreglo (diferente de la plantilla de clase `array`), que está muy relacionado con los apuntadores. En el código de C++ contemporáneo, se considera una mejor práctica usar la plantilla de clase array de C++11 que los arreglos tradicionales.

Resumen

Sección 7.1 Introducción

- Las estructuras de datos (pág. 279) son colecciones de elementos de datos relacionados. Los arreglos (pág. 279) son estructuras de datos que consisten en elementos de datos relacionados del mismo tipo. Los arreglos son entidades "estáticas", en cuanto a que permanecen del mismo tamaño a lo largo de la ejecución del programa.

Sección 7.2 Arreglos

- Un arreglo es un grupo consecutivo de localidades de memoria que comparten el mismo tipo.
- Cada arreglo conoce su propio tamaño, el cual puede determinarse mediante una llamada a su función miembro `size` (pág. 280).
- Para hacer referencia a una ubicación específica o elemento en un arreglo, especificamos el nombre del arreglo y el número de posición del elemento específico en el arreglo.
- Para hacer referencia a cualquiera de los elementos de un arreglo, un programa proporciona el nombre del arreglo seguido del índice (pág. 279) del elemento específico entre corchetes (`[]`).

- El primer elemento en cada arreglo tiene el índice cero (pág. 279), y algunas veces se le llama el elemento cero.
- Un índice debe ser un entero o una expresión entera (que utilice cualquier tipo integral).
- Los corchetes que se utilizan para encerrar el subíndice de un arreglo son un operador con la misma precedencia que los paréntesis.

Sección 7.3 Declaración de arreglos

- Los arreglos ocupan espacio en memoria. El programador especifica el tipo de cada elemento y el número de elementos requeridos de la siguiente manera:

  `array<` *tipo, tamañoArreglo* `>` *nombreArreglo*;

  y el compilador reserva la cantidad de memoria apropiada.
- Los arreglos se pueden declarar de manera que contengan cualquier tipo de datos. Por ejemplo, un arreglo de tipo `char` se puede utilizar para almacenar una cadena de caracteres.

Sección 7.4 Ejemplos acerca del uso de arreglos

- Los elementos de un arreglo se pueden inicializar en la declaración de arreglos, seguida del nombre del arreglo con un signo de igual y una lista inicializadora (pág. 282): una lista separada por comas (encerrada entre llaves) de inicializadores constantes (pág. 282).
- Al inicializar un arreglo con una lista inicializadora, si hay menos inicializadores que elementos en el arreglo, el resto de los elementos se inicializa con cero. El número de inicializadores debe ser menor o igual que el tamaño del arreglo.
- Una variable constante que se utiliza para especificar el tamaño de un arreglo debe inicializarse con una expresión constante al momento de declararse, y no puede modificarse en lo sucesivo.
- C++ no cuenta con comprobación de límites de los arreglos (pág. 291). Hay que asegurar que todas las referencias al arreglo permanezcan dentro de los límites del mismo.
- Una variable local `static` en la definición de una función existe durante el tiempo que se ejecute el programa, pero sólo es visible en el cuerpo de la función.
- Un programa inicializa los arreglos locales `static` la primera vez que encuentra sus declaraciones. Si el programador no inicializa en forma explícita un arreglo `static`, el compilador inicializa con cero cada elemento de ese arreglo a la hora de crearlo.

Sección 7.5 Instrucción `for` *basada en rango*

- La nueva instrucción `for` basada en rango de C++ (pág. 293) nos permite manipular todos los elementos de un arreglo sin usar un contador, con lo cual se evita la posibilidad de "salirse" del arreglo y se elimina la necesidad de que implementemos nuestra propia comprobación de límites.
- La sintaxis de una instrucción `for` basada en rango es:

  `for (` *declaracionVariableRango* `:` *expresion* `)`
  instruccion

  en donde *declaracionVariableRango* tiene un tipo y un identificador, y *expresión* es el arreglo a través del cual se va a iterar. El tipo en la *declaracionVariableRango* debe ser consistente con el tipo de los elementos del arreglo. El identificador representa los elementos sucesivos de un arreglo en iteraciones sucesivas del ciclo. Puede usar la instrucción `for` basada en rango con la mayoría de las estructuras de datos preconstruidas de la Biblioteca estándar de C++ (las que se conocen comúnmente como contenedores), incluyendo las clases `array` y `vector`.
- Podemos usar una instrucción `for` basada en rango para modificar cada elemento, convirtiendo a *definicionVariableRango* en una referencia.
- La instrucción `for` basada en rango puede usarse en vez de la instrucción `for` controlada por contador, cada vez que la iteración del código a través de un arreglo no requiera acceso al subíndice del elemento.

Sección 7.6 Caso de estudio: la clase `LibroCalificaciones` *que usa un arreglo para almacenar las calificaciones*

- Las variables de clase (miembros de datos `static`, pág. 300) son compartidas por todos los objetos de la clase en la que se declaran las variables.
- Se puede acceder a un miembro de datos `static` dentro de la definición de la clase y de las definiciones de las funciones miembro, al igual que con cualquier otro miembro de datos.
- También se puede acceder a un miembro de datos `public static` desde el exterior de la clase, aún y cuando no existan objetos de la misma; para ello se utiliza el nombre de la clase, seguido del operador de resolución de ámbito binario (`::`) y el nombre del miembro de datos.

Sección 7.7 Búsqueda y ordenamiento de datos en arreglos

- Ordenar los datos (colocarlos en orden ascendente o descendente) es una de las aplicaciones de cómputo más importantes.
- Al proceso de buscar un elemento específico de un arreglo se le denomina búsqueda.
- La funcion `sort` de la Biblioteca estándar de C++ ordena los elementos de un arreglo en forma ascendente. Los argumentos de la función especifican el rango de elementos que deben ordenarse. Más adelante verá que la función `sort` puede usarse en otros tipos de contenedores también.
- La función `binary_search` de la Biblioteca estándar de C++ determina si un valor está en un arreglo. La secuencia de valores debe ordenarse en forma ascendente primero. Los primeros dos argumentos de la función representan el rango de elementos a buscar y el tercero es la clave de búsqueda: el valor a localizar. La función devuelve un `bool` para indicar si se encontró el valor o no.

Sección 7.8 Arreglos multidimensionales

- Los arreglos multidimensionales (pág. 304) de dos dimensiones se utilizan con frecuencia para representar tablas de valores (pág. 304), las cuales consisten en información ordenada en filas y columnas.
- Los arreglos que requieren dos subíndices para identificar a un elemento específico se llaman arreglos bidimensionales (pág. 304). Un arreglo con *m* filas y *n* columnas se llama arreglo de *m* por *n* (pág. 304).

Sección 7.9 Caso de estudio: la clase `LibroCalificaciones` *que usa un arreglo bidimensional*

- En una declaración de variable, puede usarse la palabra clave `auto` (pág. 306) en vez de un nombre de tipo, para inferir el tipo de la variable con base en el valor inicializador de la misma.

Sección 7.10 Introducción a la plantilla de clase `vector` *de la Biblioteca estándar de C++*

- La plantilla de clase `vector` (pág. 314) de la Biblioteca estándar de C++ representa una alternativa más completa a los arreglos, ya que cuenta con muchas capacidades que no se proporcionan para los arreglos basados en apuntador estilo C.
- De manera predeterminada, todos los elementos de un objeto `vector` entero se establecen en 0.
- Un `vector` se puede definir de manera que almacene cualquier tipo de datos, mediante el uso de una declaración como la siguiente:

  `vector<` *tipo* `>` *nombre*`(` *tamaño* `);`
- La función miembro `size` (pág. 317) de la plantilla de clase `vector` devuelve el número de elementos en el `vector` en el que se invoca.
- Para acceder al valor de un elemento de un `vector` (o para modificarlo), se utilizan corchetes (`[]`).
- Los objetos de la plantilla de clase estándar `vector` se pueden comparar de manera directa con los operadores de igualdad (`==`) y desigualdad (`!=`). El operador de asignación (`=`) también se puede usar con objetos `vector`.
- Un *lvalue* no modificable es una expresión que identifica a un objeto en memoria (como un elemento en un `vector`), pero no se puede utilizar para modificar ese objeto. Un *lvalue* modificable también identifica a un objeto en memoria, pero se puede usar para modificar el objeto.
- Una excepción (pág. 318) indica un problema que ocurre mientras se ejecuta un programa. El nombre "excepción" sugiere que el problema ocurre con poca frecuencia; si la "regla" es que una instrucción se ejecute normalmente en forma correcta, entonces el problema representa la "excepción a la regla".

- El manejo de excepciones (pág. 318) nos permite crear programas tolerantes a errores (pág. 318) que pueden resolver excepciones.
- Para manejar una excepción, coloque cualquier código que pueda lanzar una excepción (pág. 319) en una instrucción `try`.
- El bloque `try` (pág. 319) contiene el código que podría lanzar una excepción, y el bloque `catch` (pág. 319) contiene el código que maneja la excepción, en caso de que ocurra.
- Cuando termina un bloque `try`, las variables declaradas en ese bloque quedan fuera de alcance.
- Un bloque `catch` declara un tipo y un parámetro de excepción. Dentro del bloque `catch` es posible usar el identificador del parámetro para interactuar con un objeto excepción atrapado.
- El método `what` de un objeto excepción (pág. 319) devuelve el mensaje de error de la excepción.

Ejercicios de autoevaluación

7.1 ***(Llene los espacios en blanco)*** Complete las siguientes oraciones:

a) Las listas y tablas de valores pueden guardarse en _____ o _____.
b) Los elementos de un arreglo están relacionados por el hecho de que tienen el mismo _____ y _____.
c) El número utilizado para referirse a un elemento específico de un arreglo se conoce como el _____ de ese elemento.
d) Un(a) _____ debe usarse para declarar el tamaño de un arreglo, ya que elimina los números mágicos.
e) Al proceso de colocar los elementos de un arreglo en orden se le conoce como _____ el arreglo.
f) Al proceso de determinar si un arreglo contiene un valor clave específico se le conoce como _____ el arreglo.
g) Un arreglo que usa dos subíndices se conoce como un arreglo _____.

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

a) Un arreglo puede guardar muchos tipos distintos de valores.
b) El subíndice de un arreglo debe ser generalmente de tipo `float`.
c) Si hay menos inicializadores en una lista inicializadora que el número de elementos en el arreglo, el resto de los elementos se inicializa con el último valor en la lista inicializadora.
d) Es un error si una lista inicializadora contiene más inicializadores que elementos en el arreglo.

7.3 ***(Escriba instrucciones de C++)*** Escriba una o más instrucciones que realicen las siguientes tareas para un arreglo llamado `fracciones`:

a) Defina una variable constante entera llamada `tamanioArreglo` para representar el tamaño de un arreglo y que se inicialice con `10`.
b) Declare un arreglo con `tamanioArreglo` elementos de tipo `double`, e inicialice los elementos con `0`.
c) Nombre el cuarto elemento del arreglo.
d) Haga referencia al elemento `4` del arreglo.
e) Asigne el valor `1.667` al elemento `9` del arreglo.
f) Asigne el valor `3.333` al séptimo elemento del arreglo.
g) Imprima los elementos 6 y 9 del arreglo con dos dígitos de precisión a la derecha del punto decimal, y muestre los resultados que aparecen realmente en la pantalla.
h) Imprima todos los elementos del arreglo usando una instrucción `for`. Defina la variable entera `i` como variable de control para el ciclo. Muestre la salida.
i) Imprima todos los elementos del arreglo separados por espacios, usando una instrucción `for` basada en rango.

7.4 ***(Preguntas sobre arreglos bidimensionales)*** Responda a las siguientes preguntas en relación con un arreglo llamado `tabla`:

a) Declare el arreglo para que almacene valores `int` y cuente con tres filas y tres columnas. Suponga que se ha declarado la variable `tamanioArreglo` con el valor de `3`.
b) ¿Cuántos elementos contiene el arreglo?

c) Utilice una instrucción `for` controlada por contador para inicializar cada elemento del arreglo con la suma de sus subíndices.
d) Escriba una instrucción `for` anidada que muestre los valores de cada elemento del arreglo `tabla` en formato tabular, con 3 filas y 3 columnas. Cada fila y columna deben etiquetarse con el número de fila o columna correspondiente. Suponga que el arreglo se inicializó con una lista inicializadora que contiene los valores del 1 al 9 en orden. Muestre la salida.

7.5 *(Encuentre el error)* Encuentre y corrija el error en cada uno de los siguientes segmentos de programa:

a)
```
#include <iostream>;
```
b)
```
tamanioArreglo = 10; // tamanioArreglo se declaró como const
```
c) Suponga que `array< int, 10 > b = {};`
```
for ( size_t i = 0; i <= b.size(); ++i )
   b[ i ] = 1;
```
d) Suponga que a es un arreglo bidimensional de valores `int` con dos filas y dos columnas:
```
a[ 1, 1 ] = 5;
```

Respuestas a los ejercicios de autoevaluación

7.1 a) arreglos, objetos `vector`. b) nombre de arreglo, tipo. c) subíndice o índice. d) variable constante. e) ordenamiento. f) búsqueda. g) bidimensional.

7.2 a) Falso. Un arreglo sólo puede guardar valores del mismo tipo.
b) Falso. El subíndice de un arreglo debe ser un entero o una expresión entera.
c) Falso. El resto de los elementos se inicializa con cero.
d) Verdadero.

7.3 a)
```
const size_t tamanioArreglo = 10;
```
b)
```
array< double, tamanioArreglo > fracciones = { 0.0 };
```
c)
```
fracciones[ 3 ]
```
d)
```
fracciones[ 4 ]
```
e)
```
fracciones[ 9 ] = 1.667;
```
f)
```
fracciones[ 6 ] = 3.333;
```
g)
```
cout << fixed << setprecision( 2 );
cout << fracciones[ 6 ] << ' ' << fracciones[ 9 ] << endl;
```
Salida:
```
3.33 1.67
```
h)
```
for ( size_t i = 0; i < fracciones.size(); ++i )
   cout << "fracciones[" << i << "] = " << fracciones[ i ] << endl;
```
Salida:
```
fracciones[ 0 ] = 0.0
fracciones[ 1 ] = 0.0
fracciones[ 2 ] = 0.0
fracciones[ 3 ] = 0.0
fracciones[ 4 ] = 0.0
fracciones[ 5 ] = 0.0
fracciones[ 6 ] = 3.333
fracciones[ 7 ] = 0.0
fracciones[ 8 ] = 0.0
fracciones[ 9 ] = 1.667
```
i)
```
for ( double element : fracciones )
   cout << element << ' ';
```

7.4 a)
```
array< array< int, tamanioArreglo >, tamanio Arreglo  > tabla;
```
b) Nueve.

c)
```
for ( size_t fila = 0; fila < tamanioArreglo(); ++fila )
   for ( size_t columna = 0; columna < tabla[ fila ].size(); ++columna )
      tabla[ fila ][ columna ] = fila + columna;
```
d)
```
cout << "    [0]  [1]  [2]" << endl;

for ( size_t i = 0; i < tamanioArreglo; ++i ) {
   cout << '[' << i << "] ";

   for ( size_t j = 0; j < tamanioArreglo; ++j )
      cout << setw( 3 ) << tabla[ i ][ j ] << "  ";
   cout << endl;
}
```
Salida:
```
    [0]  [1]  [2]
[0]   1    8    0
[1]   2    4    6
[2]   5    0    0
```

7.5 a) *Error:* punto y coma al final de la directiva del preprocesador `#include`.
Corrección: elimina el punto y coma.
b) *Error:* asignar un valor a una variable constante que utiliza una instrucción de asignación.
Corrección: inicializa la variable constante en una declaración `const size_t tamanioArreglo`.
c) *Error:* hacer referencia a un elemento del arreglo fuera de sus límites (`b[10]`).
Corrección: cambie la condición de continuación de ciclo para usar < en vez de <=.
d) *Error:* el subíndice del arreglo se escribió en forma incorrecta.
Corrección: cambie la instrucción por `a[ 1 ][ 1 ] = 5;`.

Ejercicios

7.6 ***(Llene los espacios en blanco)*** Complete las siguientes oraciones:
a) Los nombres de los cuatro elementos del arreglo `p` son _____, _____, _____ y _____.
b) Al proceso de nombrar un arreglo, declarar su tipo y especificar el número de elementos se le conoce como _____ el arreglo.
c) Por convención, el primer subíndice en un arreglo bidimensional identifica el(la) _____ de un elemento y el segundo índice identifica el(la) _____ del elemento.
d) Un arreglo de *m* por *n* contiene _____ filas, _____ columnas y _____ elementos.
e) El nombre del elemento en la fila 3 y la columna 5 del arreglo `d` es _____.

7.7 ***(Verdadero o falso)*** Conteste con *verdadero* o *falso* a cada una de las siguientes proposiciones; en caso de ser *falso*, explique por qué.
a) Para referirnos a una ubicación o elemento específico dentro de un arreglo, especificamos el nombre del arreglo y el valor del elemento específico.
b) La definición de un arreglo reserva espacio para el mismo.
c) Para indicar que deben reservarse 100 ubicaciones para el arreglo entero `p`, el programador escribe la declaración
```
p[ 100 ];
```
d) Hay que usar una instrucción `for` para inicializar con cero los elementos de un arreglo de 15 elementos.
e) Hay que usar instrucciones `for` anidadas para sumar el total de los elementos de un arreglo bidimensional.

7.8 ***(Escriba instrucciones en C++)*** Escriba instrucciones en C++ que realicen cada una de las siguientes tareas:
a) Mostrar el valor del elemento `6` del arreglo de caracteres `alfabeto`.
b) Recibir un valor y colocarlo en el elemento 4 del arreglo de punto flotante unidimensional llamado `calificaciones`.
c) Inicializar con `8` cada uno de los 5 elementos del arreglo entero unidimensional.

d) Sumar el total e imprimir los elementos del arreglo `temperaturas` de punto flotante con 100 elementos.
e) Copiar el arreglo `a` en la primera parte del arreglo `b`. Suponga que ambos arreglos contienen valores `double` y que los arreglos `a` y `b` tienen 11 y 34 elementos, respectivamente.
f) Determinar e imprimir los valores menor y mayor contenidos en el arreglo `w` con 99 elementos de punto flotante.

7.9 *(Preguntas sobre arreglos bidimensionales)* Considere un arreglo entero `t` de 2 por 3.
a) Escriba una declaración para `t`.
b) ¿Cuántas filas tiene `t`?
c) ¿Cuántas columnas tiene `t`?
d) ¿Cuántos elementos tiene `t`?
e) Escriba los nombres de todos los elementos en la fila 1 de `t`.
f) Escriba los nombres de todos los elementos en la columna 2 de `t`.
g) Escriba una instrucción que asigne cero al elemento de `t` en la primera fila y la segunda columna.
h) Escriba una serie de instrucciones que inicialice cada elemento de `t` con cero. No utilice un ciclo.
i) Escriba una instrucción `for` anidada y controlada por contador, que inicialice cada elemento de `t` con cero.
j) Escriba una instrucción `for` anidada y basada en rango, que inicialice cada elemento de `t` con cero.
k) Escriba una instrucción que reciba como entrada los valores para los elementos de `t` desde el teclado.
l) Escriba una serie de instrucciones que determine e imprima el valor más pequeño en el arreglo `t`.
m) Escriba una instrucción que muestre los elementos en la fila 0 de `t`.
n) Escriba una instrucción que totalice los elementos de la columna 2 de `t`.
o) Escriba una serie de instrucciones para imprimir el contenido de `t` en formato tabular ordenado. Enliste los subíndices de columna como encabezados a lo largo de la parte superior, y enliste los subíndices de fila a la izquierda de cada fila.

7.10 *(Rangos de salarios de vendedores)* Utilice un arreglo unidimensional para resolver el siguiente problema. Una compañía paga a sus vendedores por comisión. Los vendedores reciben $200 por semana más el 9% de sus ventas totales de esa semana. Por ejemplo, un vendedor que acumule $5000 en ventas en una semana, recibirá $200 más el 9% de $5000, o un total de $650. Escriba un programa (utilizando un arreglo de contadores) que determine cuántos vendedores recibieron salarios en cada uno de los siguientes rangos (suponga que el salario de cada vendedor se trunca a una cantidad entera):
a) $200–299
b) $300–399
c) $400–499
d) $500–599
e) $600–699
f) $700–799
g) $800–899
h) $900–999
i) $1000 en adelante

7.11 *(Preguntas sobre arreglos unidimensionales)* Escriba instrucciones individuales que realicen las siguientes operaciones con arreglos unidimensionales:
a) Inicializar con cero los 10 elementos del arreglo `cuentas` de tipo entero.
b) Sumar uno a cada uno de los 15 elementos del arreglo `bono` de tipo entero.
c) Leer 12 valores para el arreglo de valores `double` llamado `temperaturasMensuales` mediante el teclado.
d) Imprimir los 5 valores del arreglo entero `mejoresPuntuaciones` en formato de columnas.

7.12 *(Encontrar los errores)* Encuentre el (los) error(es) en cada una de las siguientes instrucciones:
a) Asuma que `a` es un arreglo de tres valores `int`:

```
cout << a[ 1 ] << " " << a[ 2 ] << " " << a[ 3 ] << endl;
```

b)
```
array< double, 3 > f = { 1.1, 10.01, 100.001, 1000.0001 };
```

c) Asuma que `d` es un arreglo de valores `double` con dos filas y 10 columnas.

```
d[ 1, 9 ] = 2.345;
```

7.13 *(Eliminación de duplicados con `array`)* Use un arreglo unidimensional para resolver el siguiente problema. Recibir como entrada 20 números, cada uno de los cuales debe estar entre 10 y 100, inclusive. A medida que se lea cada número, validarlo y almacenarlo en el arreglo, sólo si no es un duplicado de un número ya leído. Después de leer todos los valores, mostrar sólo los valores únicos que el usuario introdujo. Prepárese para el "peor caso", en el que los 20 números son diferentes. Use el arreglo más pequeño que sea posible para resolver este problema.

7.14 *(Eliminación de duplicados con `vector`)* Reimplemente el ejercicio 7.13, usando ahora un `vector`. Comience con un vector vacío y use su función `push_back` para agregar cada valor único al `vector`.

7.15 *(Inicialización de arreglos bidimensionales)* Etiquete los elementos del arreglo bidimensional `ventas` de 3 por 5, para indicar el orden en el que se establecen en cero, mediante el siguiente fragmento de programa:

```
for ( size_t fila = 0; fila < ventas.size(); ++fila )
   for ( size_t columna = 0; columna < sales[ fila ].size(); ++columna )
      ventas[ fila ][ columna ] = 0;
```

7.16 *(Tirar dados)* Escriba un programa para simular el tiro de dos dados. Después debe calcularse la suma de los dos valores. [*Nota:* cada dado puede mostrar un valor entero del 1 al 6, por lo que la suma de los valores variará del 2 al 12, siendo 7 la suma más frecuente, mientras que 2 y 12 serán las sumas menos frecuentes]. En la figura 7.26 se muestran las 36 posibles combinaciones de los dos dados. Su programa debe tirar los dados 36 000 veces. Utilice un arreglo unidimensional para registrar el número de veces que aparezca cada una de las posibles sumas. Imprima los resultados en formato tabular. Determine además si los totales son razonables (es decir, hay seis formas de tirar un 7, por lo que aproximadamente una sexta parte de los tiros deben ser 7).

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Fig. 7.26 | Los 36 posibles resultados de tirar dos dados.

7.17 *(¿Qué hace este código)* ¿Qué hace el siguiente programa?

```
1  // Ej. 7.17: ej07_17.cpp
2  // ¿Qué hace este programa?
3  #include <iostream>
4  #include <array>
5  using namespace std;
6
7  const size_t tamanioArreglo = 10;
8  int queEsEsto( const array< int, tamanioArreglo > &, size_t ); // prototipo
9
10 int main()
11 {
12    array< int, tamanioArreglo > a = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

```
13
14    int resultado = queEsEsto( a, tamanioArreglo );
15
16    cout << "El resultado es " << resultado << endl;
17 } // fin de main
18
19 // ¿Qué hace esta función?
20 int queEsEsto( const array< int, tamanioArreglo > &b, size_t tamanio )
21 {
22    if ( tamanio == 1 ) // caso base
23       return b[ 0 ];
24    else // paso recursivo
25       return b[ tamanio - 1 ] + queEsEsto( b, tamanio - 1 );
26 } // fin de la función queEsEsto
```

7.18 *(Modificación al juego de Craps)* Modifique el programa de la figura 6.11 para ejecutar 1000 juegos de craps. El programa debe mantener un registro de las estadísticas y responder a las siguientes preguntas:

a) ¿Cuántos juegos se ganan en el primer tiro, en el segundo tiro, …, en el vigésimo tiro y después de éste?
b) ¿Cuántos juegos se pierden en el primer tiro, en el segundo tiro, …, en el vigésimo tiro y después de éste?
c) ¿Cuáles son las probabilidades de ganar en craps? [*Nota*: con el tiempo descubrirá que craps es uno de los juegos de casino más justos. ¿Qué cree usted que significa esto?]
d) ¿Cuál es la duración promedio de un juego de craps?
e) ¿Las probabilidades de ganar mejoran con la duración del juego?

7.19 *(Conversión del ejemplo del* **vector** *de la sección 7.10 a un arreglo)* Convierta el ejemplo de un vector de la figura 7.26 para que use arreglos. Elimine las características que sean sólo para vectores.

7.20 *(¿Qué hace este código)* ¿Qué hace el siguiente programa?

```
1  // Ej. 7.20: ej07_20.cpp
2  // ¿Qué hace este programa?
3  #include <iostream>
4  #include <array>
5  using namespace std;
6
7  const size_t tamanioArreglo = 10;
8  void unaFuncion( const array< int, tamanioArreglo > &, size_t ); // prototipo
9
10 int main()
11 {
12    array< int, tamanioArreglo > a = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
13
14    cout << "Los valores en el arreglo son:" << endl;
15    unaFuncion( a, 0 );
16    cout << endl;
17 } // fin de main
18
19 // ¿Qué hace esta función?
20 void unaFuncion( const array< int, tamanioArreglo > &b, size_t actual )
21 {
22    if ( actual < b.size() )
23    {
24       unaFuncion( b, actual + 1 );
25       cout << b[ actual ] << "  ";
26    } // fin de if
27 } // fin de la función unaFuncion
```

7.21 ***(Resumen de ventas)*** Use un arreglo bidimensional para resolver el siguiente problema: una compañía tiene cuatro vendedores (1 a 4) que venden cinco productos distintos (1 a 5). Una vez al día, cada vendedor pasa una nota por cada tipo de producto vendido. Cada nota contiene lo siguiente:

a) El número del vendedor.
b) El número del producto.
c) El valor total en dólares de ese producto vendido en ese día.

Así, cada vendedor pasa entre 0 y 5 notas de venta por día. Suponga que está disponible la información sobre todas las notas del mes pasado. Escriba un programa que lea toda esta información para las ventas del último mes (los datos de un vendedor a la vez) y que resuma las ventas totales por vendedor, por producto. Todos los totales deben guardarse en el arreglo bidimensional `ventas`. Después de procesar toda la información del mes pasado, muestre los resultados en formato tabular, en donde cada columna represente a un vendedor específico y cada fila represente a un producto. Saque el total de cada fila para obtener las ventas totales de cada producto durante el último mes. Saque el total de cada columna para obtener las ventas totales por cada vendedor durante el último mes. Su impresión tabular debe incluir estos totales cruzados a la derecha de las filas totalizadas, y en la parte inferior de las columnas totalizadas.

7.22 ***(Paseo del caballo)*** Uno de los enigmas más interesantes para los entusiastas del ajedrez es el problema del Paseo del caballo. La pregunta es: ¿Puede la pieza de ajedrez, conocida como caballo, moverse alrededor de un tablero de ajedrez vacío y tocar cada una de las 64 posiciones una y sólo una vez? A continuación estudiaremos detalladamente este intrigante problema.

El caballo realiza solamente movimientos en forma de L (dos espacios en una dirección y un espacio en una dirección perpendicular). Por lo tanto, como se muestra en la figura 7.27, desde una posición cerca del centro de un tablero de ajedrez vacío, el caballo puede hacer ocho movimientos distintos (numerados del 0 al 7).

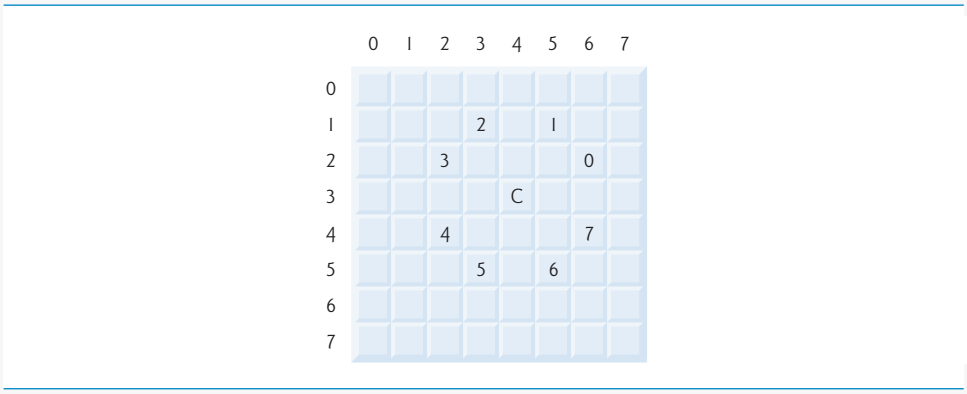


Fig. 7.27 | Los ocho posibles movimientos del caballo.

a) Dibuje un tablero de ajedrez de 8 por 8 en una hoja de papel, e intente realizar un Paseo del caballo en forma manual. Ponga un `1` en la posición inicial, un `2` en la segunda posición, un `3` en la tercera, etc. Antes de empezar el paseo, estime qué tan lejos podrá avanzar, recordando que un paseo completo consta de 64 movimientos. ¿Qué tan lejos llegó? ¿Estuvo esto cerca de su estimación?
b) Ahora desarrollaremos un programa para mover el caballo alrededor de un tablero de ajedrez. El tablero estará representado por un arreglo bidimensional llamado `tablero`, de ocho por ocho. Cada posición se inicializará con cero. Describiremos cada uno de los ocho posibles movimientos en términos de sus componentes horizontales y verticales. Por ejemplo, un movimiento de tipo 0, como se muestra en la figura 7.27, consiste en mover dos posiciones horizontalmente a la derecha y una posición verticalmente hacia arriba. Un movimiento de tipo 2 consiste en mover una posición horizontalmente a la izquierda y dos posiciones verticalmente hacia arriba. Los movimientos horizontal a la izquierda y vertical hacia arriba se indican con números negativos. Los ocho movimientos

pueden describirse mediante dos arreglos unidimensionales llamados `horizontal` y `vertical`, de la siguiente manera:

```
horizontal[ 0 ] = 2     vertical[ 0 ] = -1
horizontal[ 1 ] = 1     vertical[ 1 ] = -2
horizontal[ 2 ] = -1    vertical[ 2 ] = -2
horizontal[ 3 ] = -2    vertical[ 3 ] = -1
horizontal[ 4 ] = -2    vertical[ 4 ] = 1
horizontal[ 5 ] = -1    vertical[ 5 ] = 2
horizontal[ 6 ] = 1     vertical[ 6 ] = 2
horizontal[ 7 ] = 2     vertical[ 7 ] = 1
```

Haga que las variables `filaActual` y `columnaActual` indiquen la fila y columna, respectivamente, de la posición actual del caballo. Para hacer un movimiento de tipo `numeroMovimiento`, en donde `numeroMovimiento` puede estar entre 0 y 7, su programa debe utilizar las instrucciones

```
filaActual += vertical[ numeroMovimiento ];
columnaActual += horizontal[ numeroMovimiento ];
```

Utilice un contador que varíe de `1` a `64`. Registre la última cuenta en cada posición a la que se mueva el caballo. Evalúe cada movimiento potencial para ver si el caballo ya visitó esa posición y, desde luego, pruebe cada movimiento potencial para asegurarse que el caballo no se salga del tablero de ajedrez. Ahora escriba un programa para desplazar el caballo por el tablero. Ejecute el programa. ¿Cuántos movimientos hizo el caballo?

c) Después de intentar escribir y ejecutar un programa de Paseo del caballo, probablemente haya desarrollado algunas ideas valiosas. Utilizaremos estas ideas para desarrollar una **heurística** (o estrategia) para mover el caballo. La heurística no garantiza el éxito, pero una heurística cuidadosamente desarrollada mejora considerablemente la probabilidad de tener éxito. Probablemente usted ya observó que las posiciones externas son más difíciles que las posiciones cercanas al centro del tablero. De hecho, las posiciones más difíciles o inaccesibles son las cuatro esquinas.

La intuición sugiere que usted debe intentar mover primero el caballo a las posiciones más problemáticas y dejar pendientes aquellas a las que sea más fácil llegar, de manera que cuando el tablero se congestione cerca del final del paseo, habrá una mayor probabilidad de éxito.

Podríamos desarrollar una "heurística de accesibilidad" clasificando cada una de las posiciones de acuerdo a qué tan accesibles son y luego mover siempre el caballo (usando los movimientos en L del caballo) a la posición más inaccesible. Etiquetaremos un arreglo bidimensional llamado `accesibilidad` con números que indiquen desde cuántas posiciones es accesible una posición determinada. En un tablero de ajedrez en blanco, cada una de las 16 posiciones más cercanas al centro se clasifican con `8`; cada posición en la esquina se clasifica con `2`; y las demás posiciones tienen números de accesibilidad `3`, `4` o `6`, de la siguiente manera:

```
2 3 4 4 4 4 3 2
3 4 6 6 6 6 4 3
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
3 4 6 6 6 6 4 3
2 3 4 4 4 4 3 2
```

Escriba una nueva versión del programa del Paseo del caballo, utilizando la heurística de accesibilidad. El caballo deberá moverse siempre a la posición con el número de accesibilidad más bajo. En caso de un empate, el caballo podrá moverse a cualquiera de las posiciones empatadas. Por lo tanto, el paseo puede empezar en cualquiera de las cuatro esquinas. [*Nota:* al ir moviendo el caballo alrededor del tablero, su aplicación deberá reducir los números de accesibilidad a medida que se vayan ocupando más posiciones. De esta manera y en cualquier momento dado durante el paseo, el número de accesibilidad de cada una de las posiciones disponibles seguirá siendo igual al número preciso de posiciones desde las que se puede llegar a esa posición]. Ejecute esta versión de su progra-

ma. ¿Logró completar el paseo? Ahora modifique el programa para realizar 64 paseos, en donde cada uno empiece desde una posición distinta en el tablero. ¿Cuántos paseos completos logró realizar?

d) Escriba una versión del programa del Paseo del caballo que, al encontrarse con un empate entre dos o más posiciones, decida qué posición elegir buscando más adelante aquellas posiciones que se puedan alcanzar desde las posiciones "empatadas". Su aplicación debe mover el caballo a la posición empatada para la cual el siguiente movimiento lo lleve a una posición con el número de accesibilidad más bajo.

7.23 ***(Paseo del caballo: métodos de fuerza bruta)*** En el ejercicio 7.22, desarrollamos una solución al problema del Paseo del caballo. El método utilizado, llamado "heurística de accesibilidad", genera muchas soluciones y se ejecuta con eficiencia.

A medida que se incremente de manera continua la potencia de las computadoras, seremos capaces de resolver más problemas con menos potencia y algoritmos relativamente menos sofisticados. A éste le podemos llamar el método de la "fuerza bruta" para resolver problemas.

a) Utilice la generación de números aleatorios para permitir que el caballo se desplace a lo largo del tablero (mediante sus movimientos legítimos en L) en forma aleatoria. Su programa debe ejecutar un paseo e imprimir el tablero final. ¿Qué tan lejos llegó el caballo?

b) La mayoría de las veces, el programa anterior produce un paseo relativamente corto. Ahora modifique su aplicación para intentar 1000 paseos. Use un arreglo unidimensional para llevar el registro del número de paseos de cada longitud. Cuando su programa termine de intentar los 1000 paseos, deberá imprimir esta información en un formato tabular ordenado. ¿Cuál fue el mejor resultado?

c) Es muy probable que el programa anterior le haya brindado algunos paseos "respetables", pero no completos. Ahora deje que su aplicación se ejecute hasta que produzca un paseo completo. [*Precaución:* esta versión del programa podría ejecutarse durante horas en una computadora poderosa]. Una vez más, mantenga una tabla del número de paseos de cada longitud e imprímala cuando se encuentre el primer paseo completo. ¿Cuántos paseos intentó su programa antes de producir uno completo? ¿Cuánto tiempo se tomó?

d) Compare la versión de la fuerza bruta del Paseo del caballo con la versión heurística de accesibilidad. ¿Cuál requirió un estudio más cuidadoso del problema? ¿Qué algoritmo fue más difícil de desarrollar? ¿Cuál requirió más poder de cómputo? ¿Podríamos tener la certeza (por adelantado) de obtener un paseo completo mediante el método de la heurística de accesibilidad? ¿Podríamos tener la certeza (por adelantado) de obtener un paseo completo mediante el método de la fuerza bruta? Argumente las ventajas y desventajas de solucionar el problema mediante la fuerza bruta en general.

7.24 ***(Ocho reinas)*** Otro enigma para los entusiastas del ajedrez es el problema de las Ocho reinas, el cual pregunta lo siguiente: ¿es posible colocar ocho reinas en un tablero de ajedrez vacío, de tal manera que ninguna reina "ataque" a cualquier otra (es decir, que no haya dos reinas en la misma fila, en la misma columna o a lo largo de la misma diagonal)? Use la idea desarrollada en el ejercicio 7.22 para formular una heurística y resolver el problema de las Ocho reinas. Ejecute su programa. [*Sugerencia:* es posible asignar un valor a cada una de las posiciones en el tablero de ajedrez, para indicar cuántas posiciones de un tablero vacío se "eliminan" si una reina se coloca en esa posición. A cada una de las esquinas se le asignaría el valor 22, como se demuestra en la figura 7.28. Una vez que estos "números de eliminación" se coloquen en las 64 posiciones, una heurística apropiada podría ser la siguiente: coloque la siguiente reina en la posición con el número de eliminación más pequeño. ¿Por qué esta estrategia es intuitivamente atractiva?].

7.25 ***(Ocho reinas: métodos de fuerza bruta)*** En este ejercicio usted desarrollará varios métodos de fuerza bruta para resolver el problema de las Ocho reinas que presentamos en el ejercicio 7.24.

a) Utilice la técnica de la fuerza bruta aleatoria desarrollada en el ejercicio 7.23, para resolver el problema de las Ocho reinas.

b) Utilice una técnica exhaustiva (es decir, pruebe todas las combinaciones posibles de las ocho reinas en el tablero).

c) ¿Por qué supone que el método de la fuerza bruta exhaustiva podría no ser apropiado para resolver el problema del Paseo del caballo?

d) Compare y contraste el método de la fuerza bruta aleatoria con el de la fuerza bruta exhaustiva en general.

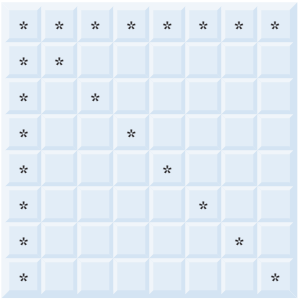


Fig. 7.28 | Las 22 posiciones eliminadas al colocar una reina en la esquina superior izquierda.

7.26 ***(Paseo del caballo: prueba del paseo cerrado)*** En el Paseo del caballo se lleva a cabo un paseo completo cuando el caballo hace 64 movimientos, en los que toca cada esquina del tablero una sola vez. Un paseo cerrado ocurre cuando el movimiento 64 se encuentra a un movimiento de distancia de la posición en la que el caballo empezó el paseo. Modifique el programa del Paseo del caballo que escribió en el ejercicio 7.22 para probar si el paseo ha sido completo, y si se trató de un paseo cerrado.

7.27 ***(La criba de Eratóstenes)*** Un entero primo es cualquier entero divisible sólo por sí mismo y por el número 1. La Criba de Eratóstenes es un método para encontrar números primos, el cual opera de la siguiente manera:

a) Cree un arreglo con todos los elementos inicializados en 1 (verdadero). Los elementos del arreglo con subíndices primos permanecerán como 1. Cualquier otro elemento del arreglo eventualmente cambiará a cero. En este ejercicio, ignoraremos los elementos 0 y 1.

b) Empezando con el subíndice 2 del arreglo, cada vez que se encuentre un elemento del arreglo cuyo valor sea 1, itere a través del resto del arreglo y asigne cero a todo elemento cuyo subíndice sea múltiplo del subíndice del elemento que tiene el valor 1. Para el subíndice 2 del arreglo, todos los elementos más allá del elemento 2 en el arreglo que tengan subíndices múltiplos de 2 (los índices 4, 6, 8, 10, etcétera) se establecerán en cero; para el subíndice 3 del arreglo, todos los elementos más allá del elemento 3 en el arreglo que sean múltiplos de 3 (los índices 6, 9, 12, 15, etcétera) se establecerán en cero; y así sucesivamente.

Cuando este proceso termine, los elementos del arreglo que aún sean uno indicarán que el subíndice es un número primo. Estos subíndices pueden entonces imprimirse. Escriba un programa que utilice un arreglo de 1000 elementos para determinar e imprimir los números primos entre 2 y 999. Ignore el elemento 0 del arreglo.

Ejercicios de recursividad

7.28 ***(Palíndromos)*** Un palíndromo es una cadena que se escribe de la misma forma tanto al derecho como al revés. Algunos ejemplos de palíndromos son "radar", "reconocer" y (si se ignoran los espacios) "anita lava la tina". Escriba una función recursiva llamada `probarPalindromo`, que devuelva `true` si la cadena almacenada en el arreglo es un palíndromo, y `false` en caso contrario. Cabe mencionar que, al igual que un arreglo, es posible usar el operador de corchetes (`[]`) para iterar a través de los caracteres en un objeto `string`.

7.29 ***(Ocho reinas)*** Modifique el programa de las Ocho reinas que creó en el ejercicio 7.24 para resolver el problema en forma recursiva.

7.30 ***(Imprimir un arreglo)*** Escriba una función recursiva llamada `imprimirArreglo` que reciba un arreglo, un subíndice inicial y un subíndice final como argumentos, que no devuelva nada y que imprima el arreglo. La función deberá dejar de procesar y deberá regresar cuando el subíndice inicial sea igual al subíndice final.

7.31 ***(Imprimir una cadena en forma inversa)*** Escriba una función recursiva llamada `cadenaInversa`, que reciba un arreglo de caracteres que contenga una cadena y un subíndice inicial como argumentos, imprima la cadena en forma inversa y no devuelva nada. La función deberá dejar de procesar y deberá regresar al encontrar la cadena nula de terminación. Cabe mencionar que, al igual que un arreglo, es posible usar el operador de corchetes (`[]`) para iterar a través de los caracteres en un objeto `string`.

7.32 *(Buscar el valor mínimo en un arreglo)* Escriba una función recursiva llamada `minimoRecursivo` que reciba un arreglo de enteros, un subíndice inicial y un subíndice final como argumentos, y que devuelva el elemento más pequeño del arreglo. La función deberá dejar de procesar y deberá regresar al encontrar la cadena nula de terminación.

7.33 *(Recorrido de laberinto)* La cuadrícula de signos # y puntos (.) en la figura 7.29 es la representación de un arreglo bidimensional integrado de un laberinto. En este arreglo bidimensional integrado, los signos # representan las paredes del laberinto y los puntos representan posiciones en las posibles rutas por el laberinto. Sólo pueden hacerse movimientos hacia una ubicación en el arreglo integrado que contenga un punto.

Hay un algoritmo simple para recorrer un laberinto que garantiza encontrar la salida (suponiendo que la haya). Si no hay una salida, volverá a la misma ubicación inicial de nuevo. Coloque su mano derecha en la pared a su derecha y comience a caminar hacia delante. Nunca quite su mano de la pared. Si el laberinto da vuelta a la derecha, siga la pared a su derecha. Mientras no quite su mano de la pared, finalmente llegará a la salida del laberinto. Puede haber una ruta más corta que la que usted eligió, pero se garantiza que saldrá del laberinto si sigue el algoritmo.

```
# # # # # # # # # # # #
# . . . # . . . . . . #
. . # . # . # # # # . #
# # # . # . . . . # . #
# . . . . # # # . # . .
# # # # . # . # . # . #
# . . # . # . # . # . #
# # . # . # . # . # . #
# . . . . . . . . # . #
# # # # # # . # # # . #
# . . . . . . # . . . #
# # # # # # # # # # # #
```

Fig. 7.29 | Representación de un laberinto en un arreglo integrado bidimensional.

Escriba la función recursiva `recorrerLaberinto` para caminar por el laberinto. La función debe recibir argumentos que incluyan un arreglo integrado de 12 por 12 de valores `char`, que representan el laberinto y la ubicación inicial del mismo. A medida que `recorrerLaberinto` intente localizar la salida del laberinto, debe colocar el carácter `X` en cada posición en la ruta. La función debe mostrar el laberinto después de cada movimiento, para que el usuario pueda observar a medida que se vaya resolviendo.

7.34 *(Generación de laberintos al azar)* Escriba una función llamada `generadorLaberinto` que produzca un laberinto al azar. La función debe recibir como argumentos un arreglo bidimensional integrado de 12 por 12 de valores `char` y apuntadoras a las variables `int` que representan la fila y la columna del punto de entrada del laberinto. Pruebe su función `recorrerLaberinto` del ejercicio 7.33, usando varios laberintos generados al azar.

Hacer la diferencia

7.35 *(Votaciones)* Internet y Web permiten que cada vez haya más personas conectadas en red, unidas por una causa, expresen sus opiniones, etcétera. En 2012, los candidatos presidenciales usaron Internet para expresar sus mensajes y recaudar dinero para sus campañas. En este ejercicio, usted escribirá un pequeño programa de votaciones que permite a los usuarios calificar cinco asuntos de conciencia social, desde 1 (menos importante) hasta 10 (más importante). Elija cinco causas (por ejemplo, asuntos políticos, asuntos sobre el entorno global). Use un arreglo `string` unidimensional llamado `temas` para almacenar las causas. Para sintetizar las respuestas de la encuesta, use un arreglo bidimensional de 5 filas y 10 columnas llamado `respuestas` (de tipo `int`), en donde cada fila corresponda a un elemento del arreglo `temas`. Cuando se ejecute el programa, debe pedir al usuario que califique cada asunto. Haga que sus amigos y familiares respondan a la encuesta. Después haga que el programa muestre un resumen de los resultados, incluyendo:

a) Un informe tabular con los cinco temas del lado izquierdo y las 10 calificaciones a lo largo de la parte superior, listando en cada columna el número de calificaciones recibidas para cada tema.
b) A la derecha de cada fila, muestre el promedio de las calificaciones para cada asunto específico.
c) ¿Qué asunto recibió la mayor puntuación total? Muestre tanto el asunto como la puntuación total.
d) ¿Qué asunto recibió la menor puntuación total? Muestre tanto el asunto como la puntuación total.

8

Apuntadores

Recibimos direcciones para ocultar nuestro paradero.
—Saki (H. H. Munro)

Averigua la dirección mediante la indirección.
—William Shakespeare

Muchas cosas, teniendo referencia completa
A un consentimiento, pueden trabajar en forma contraria
—William Shakespeare

¡Descubrirá que es una muy buena práctica verificar siempre sus referencias, señor!
—Dr. Routh

Objetivos

En este capítulo aprenderá a:

- Distinguir qué son los apuntadores.
- Conocer las similitudes y diferencias entre los apuntadores y las referencias.
- Utilizar apuntadores para pasar argumentos a las funciones por referencia.
- Conocer las estrechas relaciones entre los apuntadores y los arreglos integrados.
- Utilizar cadenas basadas en apuntador.
- Utilizar los arreglos integrados.
- Utilizar las herramientas de C++11, incluyendo `nullptr` y las funciones `begin` y `end` de la Biblioteca estándar.

Plan general

8.1 Introducción

En este capítulo hablaremos sobre los *apuntadores*: una de las características más poderosas y desafiantes de usar del lenguaje de programación C++. Nuestros objetivos aquí son ayudarle a determinar cuándo es apropiado usar apuntadores y mostrarle cómo usarlos de manera *correcta* y *responsable*.

En el capítulo 6, vimos que se pueden utilizar referencias para realizar el paso por referencia. Los apuntadores también permiten el paso por referencia, y se pueden utilizar para crear y manipular estructuras dinámicas de datos que pueden crecer y reducirse, como las listas enlazadas, colas, pilas y árboles. En este capítulo explicaremos los conceptos básicos sobre los apuntadores. En el capítulo 19 (en inglés en el sitio web) presentaremos ejemplos de cómo crear y usar estructuras de datos dinámicas que se implementan con apuntadores.

También le mostraremos la estrecha relación entre los *arreglos integrados* y los apuntadores. C++ heredó los arreglos integrados del lenguaje de programación C. Como vimos en el capítulo 7, las clases `array` y `vector` de la Biblioteca estándar de C++ proporcionan implementaciones de los arreglos como objetos completos; de hecho, tanto `array` como `vector` almacenan sus elementos en arreglos integrados. *En los proyectos nuevos de desarrollo de software, es mejor usar objetos* `array` *y* `vector` *en vez de los arreglos integrados.*

De manera similar, C++ en realidad ofrece dos tipos de cadenas: objetos de la clase `string` (que hemos estado usando desde el capítulo 3) y *cadenas basadas en apuntador*, *estilo C* (*cadenas de C*). Este capítulo introduce brevemente las cadenas C para que el lector amplíe su conocimiento sobre los apuntadores y los arreglos integrados. Las cadenas de C se utilizaban ampliamente en software antiguo de C y C++. Hablaremos con detalle sobre ellas en el apéndice F. *En los proyectos nuevos de desarrollo de software, es mejor usar objetos de la clase* `string`.

En el capítulo 12 examinaremos el uso de los apuntadores con los objetos de clases, en donde veremos que lo que se denomina "procesamiento polimórfico" de la programación orientada a objetos se lleva a cabo mediante apuntadores y referencias.

8.2 Declaraciones e inicialización de variables apuntadores

Indirección

Las variables apuntadores contienen *direcciones de memoria* como sus valores. Por lo general, una variable contiene *directamente* un valor específico. Un apuntador contiene la *dirección de memoria* de una variable que, a su vez, contiene un valor específico. En este sentido, el nombre de una variable **hace referencia directa a un valor**, y un apuntador **hace referencia indirecta a un valor** (figura 8.1). Al

proceso de hacer referencia a un valor a través de un apuntador se le conoce comúnmente como **indirección**. Por lo general, los diagramas representan un apuntador en forma de una *flecha* que parte de la *variable que contiene una dirección*, hasta la *variable ubicada en esa dirección* de memoria.

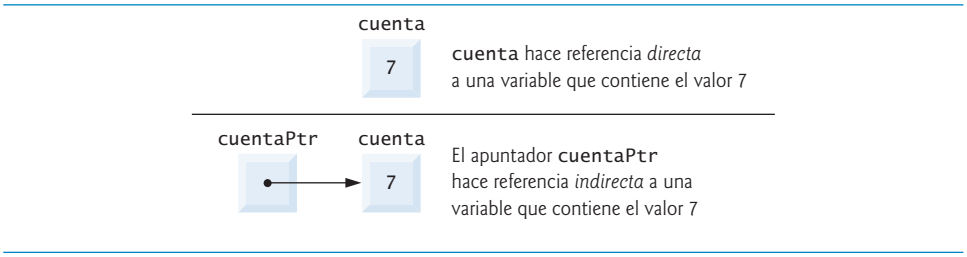


Fig. 8.1 | Referencia directa e indirecta a una variable.

Declaración de apuntadores

Al igual que las demás variables, los apuntadores se deben declarar *antes* de poder usarlos. Por ejemplo, para el apuntador en la figura 8.1, la declaración

```
int *cuentaPtr, cuenta;
```

declara la variable `cuentaPtr` como de tipo `int *` (es decir, un apuntador a un valor `int`) y se lee así (*de derecha a izquierda*): "`cuentaPtr` es un apuntador a un `int`". Además, la variable `cuenta` en la declaración anterior se declara como un `int`, y *no* como un apuntador a un `int`. El `*` en la declaración se aplica *sólo* a `cuentaPtr`. A cada variable que se declara como apuntador se le *debe* anteponer un asterisco (`*`). Por ejemplo, la declaración

```
double *xPtr, *yPtr;
```

indica que tanto `xPtr` como `yPtr` son apuntadores a valores `double`. Cuando el `*` aparece en una declaración, *no* es un operador; en vez de ello, indica que la variable que se está declarando es un apuntador. Los apuntadores se pueden declarar de manera que apunten a objetos de *cualquier* tipo de datos.

Error común de programación 8.1

*Asumir que el * que se utiliza para declarar a un apuntador se distribuye a todos los nombres de variables en la lista separada por comas de variables de una declaración puede provocar errores. Cada apuntador se debe declarar con el * antepuesto al nombre (ya sea con o sin un espacio entre ellos). Si declaramos sólo una variable por cada declaración, evitamos estos tipos de errores y mejoramos la legibilidad de los programas.*

Buena práctica de programación 8.1

Aunque no es un requerimiento, incluir las letras `Ptr` en los nombres de las variables apuntadores deja claro que estas variables son apuntadores, y que deben tratarse de manera acorde.

Inicialización de apuntadores

Los apuntadores se deben inicializar con **`nullptr`** (nuevo en C++11) o con una dirección del tipo correspondiente, ya sea cuando se declaran o en una asignación. Un apuntador con el valor `nullptr` "no apunta a nada", y se conoce como **apuntador nulo**. De aquí en adelante, cuando hagamos referencia a un "apuntador nulo" nos estaremos refiriendo a un apuntador con el valor `nullptr`.

Tip para prevenir errores 8.1

Inicialice todos los apuntadores para evitar que apunten hacia áreas desconocidas o no inicializadas de la memoria.

Apuntadores nulos anteriores a C++11
En versiones anteriores de C++, el valor especificado para un apuntador nulo era `0` o `NULL`. Este último se define en varios encabezados de la biblioteca estándar para representar el valor `0`. Inicializar un apuntador a `NULL` es equivalente a inicializar un apuntador a `0`, pero antes de C++11, se utilizaba `0` por convención. El `0` es el *único* valor entero que puede asignarse directamente a una variable apuntador sin primero *convertir* el entero a un tipo apuntador.

8.3 Operadores de apuntadores

Operador dirección (&)
El **operador dirección** (`&`) es un operador unario que *obtiene la dirección de memoria de su operando*. Por ejemplo, teniendo en cuenta las siguientes declaraciones:

```
int y = 5; // declara la variable y
int *yPtr = nullptr; // declara la variable apuntador yPtr
```

la instrucción

```
yPtr = &y; // asigna la dirección de y a yPtr
```

asigna la dirección de la variable `y` a la variable apuntador `yPtr`. Entonces, se dice que la variable `yPtr` "apunta a" `y`. Ahora, `yPtr` hace referencia *indirecta* al valor de la variable `y`. Observe que el uso del signo `&` en la instrucción anterior *no* es el mismo que el uso del `&` en la *declaración de una variable de referencia*, a la cual *siempre* se le antepone el nombre de un tipo de datos. Al declarar una referencia, el `&` forma parte del tipo. En una expresión como `&y`, el `&` es el *operador dirección*.

La figura 8.2 muestra una representación de la memoria después de la asignación anterior. La "relación de señalamiento" se indica mediante el dibujo de una flecha desde el cuadro que representa al apuntador `yPtr` en la memoria, hasta el cuadro que representa a la variable `y` en la memoria.

La figura 8.3 muestra otra representación del apuntador en memoria, con la variable entera `y` almacenada en la ubicación de memoria `600000` y la variable apuntador `yPtr` almacenada en la ubicación de memoria `500000`. El operando del operador dirección debe ser un *lvalue*; el operador dirección *no se puede* aplicar a constantes o expresiones que produzcan valores temporales (como los resultados de los cálculos).

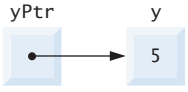


Fig. 8.2 | Representación gráfica de un apuntador que apunta a una variable en memoria.

Fig. 8.3 | Representación de `y` y `yPtr` en memoria.

Operador de indirección (*)
El **operador** `*` unario, que se conoce comúnmente como el **operador de indirección** u **operador de desreferencia**, *devuelve un lvalue que representa al objeto al que apunta su operando apuntador*. Por ejemplo (haciendo referencia otra vez a la figura 8.2), la instrucción

```
cout << *yPtr << endl;
```

imprime el valor de la variable y (en este caso, 5), al igual que la instrucción.

```
cout << y << endl;
```

Al proceso de utilizar el * de esta manera, se le conoce como **desreferenciar un apuntador**. Un *apuntador desreferenciado* también se puede usar en el lado *izquierdo* de una instrucción de asignación, como en

```
*yPtr = 9;
```

lo cual asignaría 9 a y en la figura 8.3. El *apuntador desreferenciado* también se puede utilizar para recibir un valor de entrada, como en

```
cin >> *yPtr;
```

lo cual coloca el valor de entrada en y.

Error común de programación 8.2

Al desreferenciar un apuntador que no se haya inicializado se produce un comportamiento indefinido, el cual podría producir un error fatal en tiempo de ejecución. Esto también podría conducir a la modificación accidental de datos importantes y permitir que el programa se ejecutara por completo, lo que posiblemente produzca resultados incorrectos.

Tip para prevenir errores 8.2

Desreferenciar un apuntador nulo produce un comportamiento indefinido y, por lo general, es un error en tiempo fatal de ejecución, por lo que debe asegurarse de que un apuntador no sea nulo antes *de desreferenciarlo.*

Uso de los operadores dirección (&) e indirección (*)

El programa de la figura 8.4 demuestra los operadores & y * para apuntadores. Las ubicaciones de memoria se imprimen mediante << en este ejemplo como enteros *hexadecimales* (base 16) (en el apéndice D, Sistemas numéricos, podrá obtener más información acerca de los enteros hexadecimales). Las direcciones de memoria que imprime este programa son *dependientes de la plataforma*, por lo que tal vez usted obtenga distintos resultados cuando ejecute el programa. La dirección de a (línea 11) y el valor de aPtr (línea 12) son idénticos en la salida, lo cual confirma que la dirección de a se asigna en definitiva a la variable apuntador aPtr.

```
1  // Fig. 8.4: fig08_04.cpp
2  // Los operadores & y * de los apuntadores.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int a = 7; // se asigna un 7 a la variable a
9     int *aPtr = &a; // aPtr se inicializa con la dirección de la variable int a
10
11    cout << "La direccion de a es " << &a
12       << "\nEl valor de aPtr es " << aPtr;
13    cout << "\n\nEl valor de a es " << a
14       << "\nEl valor de *aPtr es " << *aPtr << endl;
15 } // fin de main
```

Fig. 8.4 | Los operadores & y * de los apuntadores (parte 1 de 2).

```
La direccion de a es 002DFD80
El valor de aPtr es 002DFD80

El valor de a es 7
El valor de *aPtr es 77
```

Fig. 8.4 | Los operadores & y * de los apuntadores (parte 2 de 2).

Precedencia y asociatividad de los operadores presentados hasta ahora
La figura 8.5 lista la precedencia y asociatividad de los operadores presentados hasta ahora. El operador dirección (&) y el operador desreferencia (*) son *operadores unarios* en el cuarto nivel.

Operadores	Asociatividad	Tipo
`::` `()`	izquierda a derecha *[Vea la precaución en la figura 2.10 con respecto al agrupamiento de paréntesis].*	primario
`()` `[]` `++` `--` `static_cast<`*type*`>(`*operand*`)`	izquierda a derecha	postfijo
`++` `--` `+` `-` `!` `&` `*`	derecha a izquierda	unario (prefijo)
`*` `/` `%`	izquierda a derecha	multiplicativa
`+` `-`	izquierda a derecha	aditiva
`<<` `>>`	izquierda a derecha	inserción/extracción
`<` `<=` `>` `>=`	izquierda a derecha	relacional
`==` `!=`	izquierda a derecha	igualdad
`&&`	izquierda a derecha	AND lógico
`||`	izquierda a derecha	OR lógico
`?:`	derecha a izquierda	condicional
`=` `+=` `-=` `*=` `/=` `%=`	derecha a izquierda	asignación
`,`	izquierda a derecha	coma

Fig. 8.5 | Precedencia y asociatividad de los operadores vistos hasta ahora.

8.4 Paso por referencia mediante apuntadores

En C++ hay tres formas de pasar argumentos a una función: paso por valor, paso por referencia con argumentos tipo referencia y **paso por referencia con argumentos tipo apuntador**. En el capítulo 6 comparamos y contrastamos el paso por referencia con argumentos tipo referencia y el paso por valor. En esta sección, explicaremos el paso por referencia con argumentos tipo apuntador.

Como vimos en el capítulo 6, se puede utilizar `return` para devolver *un valor* de una función a la que se llamó, o simplemente devolver el *control*. También vimos que se pueden pasar argumentos a una función mediante el uso de parámetros de referencia, los cuales permiten a la función llamada *modificar los valores originales de los argumentos en la función que hizo la llamada*. Los parámetros de referencia también permiten a los programas pasar *objetos de datos grandes* a una función, y evitar la sobrecarga de pasar los objetos por valor (que, desde luego, copia el objeto). Al igual que las referencias, los apuntadores también se pueden usar para modificar una o más variables en la función que hace la llamada, o pasar apuntadores a objetos de datos grandes para evitar la sobrecarga de pasar los objetos por valor.

Podemos usar apuntadores y el operador indirección (*) para realizar el paso por referencia (en forma idéntica al paso por referencia en los programas en C, ya que éste no tiene referencias). Cuando se llama a una función con un argumento que se debe modificar, se pasa la *dirección* del argumento. Por lo general, para realizar esto se aplica el operador dirección (&) al nombre de la variable cuyo valor se va a modificar.

Un ejemplo de paso por valor

Las figuras 8.6 y 8.7 presentan dos versiones de una función que eleva un entero al cubo. La figura 8.6 pasa la variable `numero` *por valor* (línea 14) a la función `cuboPorValor` (líneas 19 a 22), que eleva su argumento al cubo y pasa el nuevo valor de vuelta a `main`, usando una instrucción `return` (línea 21). El nuevo valor se asigna a `numero` (línea 14) en `main`. La función que llama tiene la oportunidad de examinar el resultado de la llamada a la función *antes* de modificar el valor de la variable `numero`. Por ejemplo, podríamos haber almacenado el resultado de `cuboPorValor` en otra variable, para después examinar su valor y asignar el resultado a `numero`, sólo después de determinar que el valor devuelto era razonable.

```
1  // Fig. 8.6: fig08_06.cpp
2  // Uso del paso por valor para elevar al cubo el valor de una variable.
3  #include <iostream>
4  using namespace std;
5
6  int cuboPorValor( int ); // prototipo
7
8  int main()
9  {
10    int numero = 5;
11
12    cout << "El valor original de numero es " << numero;
13
14    numero = cuboPorValor( numero ); // pasa el numero por valor a cuboPorValor
15    cout << "\nEl nuevo valor de numero es " << numero << endl;
16 } // fin de main
17
18 // calcula y devuelve el cubo del argumento entero
19 int cuboPorValor( int n )
20 {
21    return n * n * n; // eleva al cubo la variable local n y devuelve el resultado
22 } // fin de la función cuboPorValor
```

```
El valor original de numero es 5
El nuevo valor de numero es 125
```

Fig. 8.6 | Uso del paso por valor para elevar al cubo el valor de una variable.

Un ejemplo de paso por referencia con apuntadores

En la figura 8.7 se pasa la variable `numero` a la función `cuboPorReferencia` mediante el uso del *paso por referencia con un argumento tipo apuntador* (línea 15); la *dirección* de `numero` se pasa a la función. La función `cuboPorReferencia` (líneas 21 a 24) especifica el parámetro `nPtr` (un apuntador a `int`) para recibir su argumento. La función *usa el apuntador desreferenciado* para elevar al cubo el valor al que apunta `nPtr` (línea 23). Esto modifica *directamente* el valor de `numero` en `main` (línea 11). La línea 23 es equivalente a

```
*nPtr = (*nPtr) * (*nPtr) * (*nPtr); // eleva *nPtr al cubo
```

```
1  // Fig. 8.7: fig08_07.cpp
2  // Uso del paso por referencia con un argumento apuntador para elevar
3  // al cubo el valor de una variable.
4  #include <iostream>
5  using namespace std;
6
7  void cuboPorReferencia( int * ); // prototipo
8
9  int main()
10 {
11    int numero = 5;
12
13    cout << "El valor original de numero es " << numero;
14
15    cuboPorReferencia( &numero ); // pasa la dirección de numero
                                    a cuboPorReferencia
16
17    cout << "\nEl nuevo valor de numero es " << numero << endl;
18 } // fin de main
19
20 // calcula el cubo de *nPtr; modifica la variable numero en main
21 void cuboPorReferencia( int *nPtr )
22 {
23    *nPtr = *nPtr * *nPtr * *nPtr; // eleva *nPtr al cubo
24 } // fin de la función cuboPorReferencia
```

```
El valor original de numero es 5
El nuevo valor de numero es 125
```

Fig. 8.7 | Uso del paso por referencia con un argumento tipo apuntador para elevar al cubo el valor de una variable.

Una función que recibe una *dirección* como argumento debe definir un *parámetro tipo apuntador* para *recibir* la dirección. Por ejemplo, el encabezado para la función `cuboPorReferencia` (línea 21) especifica que `cuboPorReferencia` debe recibir la dirección de una variable `int` (es decir, un apuntador a un `int`) como argumento, debe almacenar la dirección en forma local en `nPtr` y *no* debe devolver un valor.

El prototipo de función para `cuboPorReferencia` (línea 7) contiene `int *` entre paréntesis. Al igual que con otros tipos de variables, no es necesario incluir los *nombres* de los parámetros tipo apuntador en los prototipos. El compilador *ignora* los nombres de los parámetros incluidos para fines de documentación.

Perspectiva: todos los argumentos se pasan por valor

En C++, *todos* los argumentos *siempre* se pasan por valor. Al pasar una variable por referencia con un apuntador *en realidad no se pasa algo por referencia*: se *pasa* un apuntador a esa variable *por valor* y se *copia* al parámetro del apuntador correspondiente de esa función. Así, la función a la que se llamó puede acceder a esa variable en la función que hizo la llamada, con sólo desreferenciar el apuntador, logrando así el *paso por referencia*.

Análisis gráfico del paso por valor y del paso por referencia

Las figuras 8.8 y 8.9 analizan gráficamente la ejecución de los programas de las figuras 8.6 y 8.7, respectivamente. En los diagramas, los valores en los rectángulos sobre una expresión o variable dada representan el valor de esa expresión o variable. La columna derecha de cada diagrama muestra las funciones `cuboPorValor` (figura 8.6) y `cuboPorReferencia` (figura 8.7) *sólo* cuando se están ejecutando.

Paso 1: antes de que `main` llame a `cuboPorValor`:

```
int main()
{
   int numero = 5;

   numero = cuboPorValor( numero );
}
```

numero: 5

Paso 2: después de que `cuboPorValor` recibe la llamada:

```
int main()
{
   int numero = 5;

   numero = cuboPorValor( numero );
}
```

numero: 5

```
int cuboPorValor( int n )
{
   return n * n * n;
}
```

n: 5

Paso 3: después de que `cuboPorValor` eleva al cubo el parámetro `n` y antes de que `cuboPorValor` regrese a `main`:

```
int main()
{
   int numero = 5;

   numero = cuboPorValor( numero );
}
```

numero: 5

```
int cuboPorValor( int n )
{
   125
   return n * n * n;
}
```

n: 5

Paso 4: después de que `cuboPorValor` regresa a `main` y antes de asignar el resultado a `numero`:

```
int main()
{
   int numero = 5;
   125
   numero = cuboPorValor( numero );
}
```

numero: 5

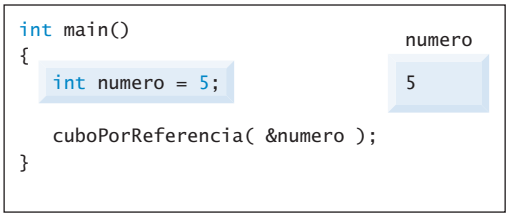
Paso 5: después de que `main` completa la asignación para `numero`:

```
int main()
{
   int numero = 5;
   125
   number = cuboPorValor( numero );
}
```

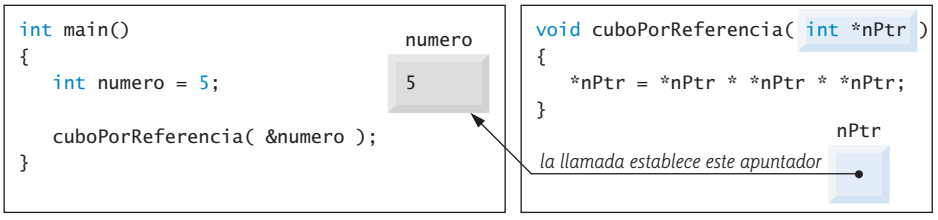
numero: 125

Fig. 8.8 | Análisis del paso por valor del programa de la figura 8.6.

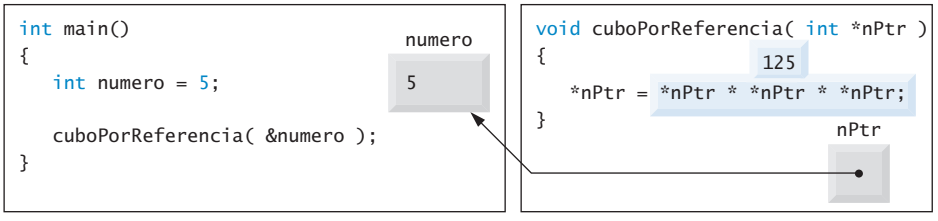
Paso 1: antes de que `main` llame a `cuboPorReferencia`:



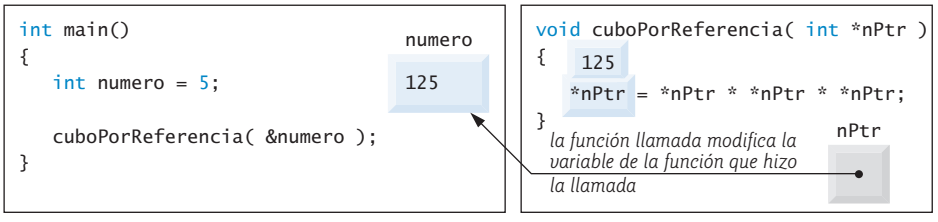
Paso 2: después de que `cuboPorReferencia` recibe la llamada y antes de que `*nPtr` se eleve al cubo:



Paso 3: antes de que se asigne a `*nPtr` el resultado del cálculo `5 * 5 * 5`:



Paso 4: después de asignar 125 a `*nPtr` y antes de que el control del programa regrese a `main`:



Paso 5: después de que `cuboPorReferencia` regresa a `main`:

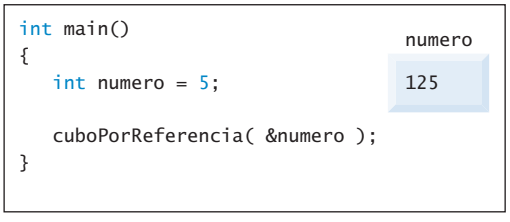


Fig. 8.9 | Análisis del paso por referencia (con un argumento tipo apuntador) del programa de la figura 8.7.

8.5 Arreglos integrados

En el capítulo 7, usamos la plantilla de clase `array` para representar listas y tablas de valores de *tamaño fijo*. También usamos la plantilla de clase `vector`, que es similar a `array`, sólo que también puede crecer (o encogerse como veremos en el capítulo 15) en forma dinámica para dar cabida a más o menos elementos. En esta sección presentamos los *arreglos integrados*, que también son estructuras de datos de *tamaño fijo*.

Declaración de un arreglo integrado

Para especificar el tipo y el número de elementos requeridos por un arreglo integrado, use una declaración de la forma:

```
tipo nombreArreglo[ tamañoArreglo ];
```

El compilador reserva la cantidad de memoria apropiada. El *tamañoArreglo* debe ser una constante entera mayor que cero. Por ejemplo, para indicar al compilador que reserve 12 elementos para el arreglo integrado de valores `int` llamado `c`, use la declaración:

```
int c[ 12 ]; // c es un arreglo integrado de 12 enteros
```

Acceso a los elementos de un arreglo integrado

Al igual que con los objetos `array`, se utiliza el operador subíndice (`[]`) para acceder a los elementos individuales de un arreglo integrado. En el capítulo 7 vimos que el operador subíndice (`[]`) *no* cuenta con comprobación de límites para los objetos `array`; esto también aplica para los arreglos integrados.

Inicialización de arreglos integrados

Podemos inicializar los elementos de un arreglo integrado mediante el uso de una *lista inicializadora*. Por ejemplo,

```
int n[ 5 ] = { 50, 20, 30, 10, 40 };
```

crea un arreglo integrado de cinco valores `int` y los inicializa con los valores en la lista inicializadora. Si proporciona menos inicializadores que el número de elementos, los elementos restantes se *inicializan por valor*; los tipos numéricos fundamentales se establecen con `0`, los valores `bool` se establecen con `false`, los apuntadores se establecen con `nullptr` y los objetos de clases se inicializan mediante sus constructores predeterminados. Si proporciona demasiados inicializadores, se produce un error de compilación. La nueva sintaxis de inicialización de listas de C++11 que presentamos en el capítulo 4 se basa en la sintaxis de listas inicializadoras de los arreglos integrados.

Si se *omite* el tamaño de un arreglo integrado de una declaración con una lista inicializadora, el compilador ajusta el tamaño del arreglo integrado con el número de elementos en la lista inicializadora. Por ejemplo,

```
int n[] = { 50, 20, 30, 10, 40 };
```

crea un arreglo de cinco elementos.

Tip para prevenir errores 8.3

Especifique siempre el tamaño de un arreglo integrado, incluso cuando se proporcione una lista inicializadora. Esto permite al compilador asegurar que no se proporcionen demasiados inicializadores.

Paso de arreglos integrados a funciones

El valor del nombre de un arreglo integrado se convierte de manera implícita a la dirección del primer elemento del arreglo integrado. Así, `nombreArreglo` se convierte implícitamente en `&nombreArreglo[0]`.

Por esta razón, no es necesario tomar la dirección (&) de un arreglo integrado para pasarla a una función; sólo hay que pasar el nombre del arreglo integrado. Como vimos en la sección 8.4, una función que recibe un apuntador a una variable en la función que hizo la llamada puede *modificar* esa variable en la función que hizo la llamada. Para los arreglos integrados, esto significa que la función que hizo la llamada puede modificar *todos* los elementos de un arreglo integrado en la función que hizo la llamada; a menos que la función anteponga la palabra `const` al parámetro del arreglo integrado correspondiente, para indicar que los elementos *no* deben modificarse.

Observación de Ingeniería de Software 8.1

Aplicar el calificador de tipo `const` *a un parámetro de arreglo integrado en una definición de función para evitar que el arreglo integrado original se modifique en el cuerpo de la función es otro ejemplo del principio de menor privilegio. Las funciones no deben tener la capacidad de modificar un arreglo integrado, a menos que sea absolutamente necesario.*

Declaración de parámetros de arreglos integrados

Podemos declarar un parámetro de arreglo integrado en el encabezado de una función, como se muestra a continuación:

```
int sumaElementos( const int valores[], const size_t numeroDeElementos )
```

lo cual indica que el primer argumento de la función debe ser un arreglo integrado unidimensional de valores `int` que la función *no* debe modificar. A diferencia de los objetos `array`, los arreglos integrados no conocen su propio tamaño, por lo que una función que procesa un arreglo integrado debería tener parámetros para recibir el arreglo integrado *y* su tamaño.

El encabezado anterior también puede escribirse así:

```
int sumaElementos( const int *valores, const size_t numeroDeElementos )
```

El compilador no distingue entre una función que recibe un apuntador y una que recibe un arreglo integrado. Desde luego, esto significa que la función debe "saber" cuando está recibiendo un arreglo integrado o simplemente una variable individual que se está pasando por referencia. Cuando el compilador encuentra el parámetro de una función para un arreglo integrado unidimensional de la forma `const int valores[]`, convierte el parámetro a la notación de apuntador `const int *valores` (es decir, "`valores` es un apuntador a una constante entera"). Estas formas de declarar un parámetro de arreglo integrado unidimensional son intercambiables: por cuestión de *claridad*, es mejor usar la notación `[]` cuando la función espera un argumento de arreglo integrado.

C++11: Las funciones **`begin`** *y* **`end`** *de la Biblioteca estándar*

En la sección 7.7, le mostramos cómo ordenar un objeto `array` con la función `sort` de la Biblioteca estándar de C++. Ordenamos un arreglo de objetos `string` llamado `colores` de la siguiente manera:

```
sort( colores.begin(), colores.end() ); // ordena el contenido de colores
```

Las funciones `begin` y `end` de la clase `array` especifican que hay que ordenar todo el arreglo. La función `sort` (y muchas otras funciones de la Biblioteca estándar de C++) también puede aplicarse a los arreglos integrados. Por ejemplo, para ordenar el arreglo integrado `n` que se mostró anteriormente en esta sección, podemos escribir:

```
sort( begin( n ), end( n ) ); // ordena el contenido del arreglo integrado n
```

Las nuevas funciones **`begin`** y **`end`** de C++11 (del encabezado `<iterator>`) reciben cada una un arreglo integrado como argumento, y devuelven un apuntador que puede usarse para representar rangos de elementos para procesar en funciones de la Biblioteca estándar de C++, como `sort`.

Limitaciones de los arreglos integrados
Los arreglos integrados tienen varias limitaciones:

- *No pueden compararse* mediante el uso de los operadores relacionales y de igualdad; debemos usar un ciclo para comparar dos arreglos integrados, elemento por elemento.
- *No pueden asignarse* unos a otros.
- *No conocen su propio tamaño*; por lo general, una función que procesa un arreglo integrado recibe como argumentos *tanto* el *nombre* del arreglo integrado *como* su *tamaño*.
- *No cuentan con comprobación de límites automática*: debemos asegurarnos de que las expresiones que accedan a los arreglos usen subíndices que se encuentren dentro de los límites del arreglo integrado.

Los objetos de las plantillas de clases `array` y `vector` son más seguros, robustos y proporcionan más herramientas que los arreglos integrados.

Algunas veces se requieren arreglos integrados
En el código de C++ contemporáneo, es más conveniente usar los objetos `array` (o `vector`), que son más robustos, para representar listas y tablas de valores. Sin embargo, existen casos en los que se *deben* usar los arreglos integrados, como al procesar los **argumentos de línea de comandos** de un programa. Para suministrar argumentos de línea de comandos a un programa, hay que colocarlos después del nombre del programa cuando se vaya a ejecutar desde la línea de comandos. Por lo general, dichos argumentos pasan opciones a un programa. Por ejemplo, en una computadora Windows el comando

```
dir /p
```

usa el argumento `/p` para listar el contenido del directorio actual, haciendo pausa después de cada pantalla de información. De manera similar, en OS X o Linux, el siguiente comando usa el argumento `-la` para listar el contenido del directorio actual, con detalles acerca de cada archivo y directorio:

```
ls -la
```

Los argumentos de línea de comandos se pasan a `main` como un arreglo integrado de cadenas basadas en apuntador (sección 8.10). El apéndice F muestra cómo procesar argumentos de la línea de comandos.

8.6 Uso de `const` con apuntadores

Recuerde que el calificador `const` nos permite informar al compilador que el valor de una variable específica *no* se debe modificar. Existen muchas posibilidades para usar (o *no* usar) `const` con los parámetros de funciones, así que ¿cómo elegir la más apropiada de estas posibilidades? Hay que dejar que el *principio del menor privilegio* sea nuestro guía. Siempre debemos otorgar a una función el *suficiente* acceso a los datos en sus parámetros para que pueda realizar su tarea especificada, pero *nada más*. En esta sección veremos cómo combinar `const` con las declaraciones de apuntadores para hacer valer el principio del menor privilegio.

En el capítulo 6 explicamos que, cuando se llama a una función mediante el paso por valor, se pasa una *copia* del argumento a la función. Si la copia se *modifica* en la función a la que *se llamó*, el valor original se mantiene *sin cambios* en la función que hizo la llamada. En muchos casos, incluso la copia del valor del argumento *no* debe alterarse en la función a la que se llamó.

Considere una función que recibe un apuntador al elemento inicial de un arreglo integrado y su tamaño como argumentos, y por consiguiente imprime los elementos del arreglo integrado. Dicha función debería iterar a través de los elementos e imprimir cada uno por separado. El tamaño del arreglo integrado se utiliza en el cuerpo de la función para determinar el subíndice más alto del arreglo, de ma-

nera que el ciclo pueda terminar cuando se complete la impresión. El tamaño del arreglo no necesita cambiar en el cuerpo de la función, por lo que debe declararse como `const` para *asegurar* que no cambie. Desde luego, como el arreglo integrado sólo se va a imprimir, también debe declararse como `const`. En especial, esto es importante ya que los arreglos integrados *siempre* se pasan por referencia, y podrían modificarse fácilmente en la función a la que se llama. Un intento de modificar un valor `const` es un error de *compilación*.

Observación de Ingeniería de Software 8.2

Si un valor no cambia (o no debe cambiar) en el cuerpo de una función que lo recibe, el parámetro se debe declarar `const`.

Tip para prevenir errores 8.4

Antes de usar una función, compruebe su prototipo para determinar los parámetros que puede y no puede modificar.

Hay cuatro maneras de pasar un apuntador a una función: un *apuntador no constante a datos no constantes*, un *apuntador no constante a datos constantes* (figura 8.10), un *apuntador constante a datos no constantes* (figura 8.11) y un *apuntador constante a datos constantes* (figura 8.12). Cada combinación proporciona un nivel distinto de privilegios de acceso.

8.6.1 Apuntador no constante a datos no constantes

El mayor nivel de acceso se otorga mediante un **apuntador no constante a datos no constantes**; los *datos se pueden modificar* a través del apuntador desreferenciado, y el *apuntador se puede modificar* para que apunte a otros datos. La declaración para dicho apuntador (por ejemplo, `int *cuentaPtr`) *no* incluye `const`.

8.6.2 Apuntador no constante a datos constantes

Un **apuntador no constante a datos constantes** es un apuntador que se puede modificar para apuntar a *cualquier* elemento de datos del tipo apropiado, pero los datos a los que apunta *no se pueden* modificar a través de ese apuntador. Dicho apuntador podría usarse para *recibir* un argumento tipo arreglo integrado para una función que pueda leer los elementos, pero *no* modificarlos. Cualquier intento por modificar los datos en la función resulta en un error de compilación. La declaración para dicho apuntador coloca `const` a la *izquierda* del tipo del apuntador, como en:

```
const int *cuentaPtr;
```

La declaración se lee de *derecha a izquierda* como "`cuentaPtr` es un apuntador a una *constante* entera" o, en forma más precisa, "`cuentaPtr` es un apuntador *no constante* a una *constante* entera".

En la figura 8.10 se muestra el mensaje de error de compilación de GNU C++ que se produce al tratar de compilar una función que recibe un *apuntador no constante* a *datos constantes*, y después trata de usar ese apuntador para modificar los datos.

```
1  // Fig. 8.10: fig08_10.cpp
2  // Intento de modificar los datos a través de un
3  // apuntador no constante a datos constantes.
4
5  void f( const int * ); // prototipo
6
7  int main()
8  {
```

Fig. 8.10 | Intento de modificar datos a través de un apuntador no constante a datos `const` (parte 1 de 2).

```
 9     int y = 0;
10
11     f( &y ); // f intentará realizar una modificación ilegal
12  } // fin de main
13
14  // una variable constante no se puede modificar a través de xPtr
15  void f( const int *xPtr )
16  {
17     *xPtr = 100; // error: no se puede modificar un objeto const
18  } // fin de la función f
```

Mensaje de error del compilador GNU C++:

```
fig08_10.cpp: In function 'void f(const int*)':
fig08_10.cpp:17:12: error: assignment of read-only location '* xPtr'
```

Fig. 8.10 | Intento de modificar datos a través de un apuntador no constante a datos `const` (parte 2 de 2).

Cuando se llama a una función con un arreglo integrado como argumento, su contenido se pasa efectivamente por referencia debido a que el nombre del arreglo integrado puede convertirse de manera implícita en la dirección del primer elemento del arreglo integrado. Sin embargo y *de manera predeterminada, los objetos como arreglos y vectores se pasan por valor; se pasa una copia del objeto completo*. Para ello se requiere la sobrecarga en tiempo de ejecución de crear una *copia* de cada elemento de datos en el objeto y almacenarla en la pila de llamadas a funciones. Cuando se pasa un apuntador a un objeto, sólo debe realizarse una copia de la *dirección* del objeto; el *objeto en sí no se copia*.

Tip de rendimiento 8.1

Si no es necesario modificarlos por la función a la que se llamó, pase objetos grandes utilizando apuntadores a datos constantes o referencias a datos constantes, para obtener los beneficios de rendimiento del paso por referencia y evitar la sobrecarga de la copia del paso por valor.

Observación de Ingeniería de Software 8.3

Hay que pasar objetos grandes usando apuntadores a datos constantes, o referencias a datos constantes, para obtener la seguridad del paso por valor.

Observación de Ingeniería de Software 8.4

Use el paso por valor para pasar argumentos de tipos fundamentales (como `int`*,* `double`*, etc.) a una función, a menos que la función que hace la llamada requiera de manera* explícita *que la función a la que se llamó pueda modificar directamente el valor en la función que hizo la llamada. Éste es otro ejemplo del principio de menor privilegio.*

8.6.3 Apuntador constante a datos no constantes

Un **apuntador constante a datos no constantes** es un apuntador que siempre apunta a la misma ubicación de memoria; los datos en esa ubicación se *pueden* modificar a través del apuntador. Los apuntadores que se declaran como `const` *se deben inicializar a la hora de declararse*, pero si el apuntador es un parámetro de función, se *inicializa con el apuntador que se pasa a la función*.

El programa de la figura 8.11 trata de modificar un apuntador constante. En la línea 11 se declara el apuntador `ptr` de tipo `int * const`. La declaración se lee de *derecha a izquierda* como "`ptr` es un

apuntador constante a un entero no constante". El apuntador se *inicializa* con la dirección de la variable entera `x`. En la línea 14 se hace un intento por *asignar* la dirección de `y` a `ptr`, pero el compilador genera un mensaje de error. No ocurre ningún error cuando en la línea 13 se asigna el valor `7` a `*ptr`; el valor no constante al que apunta `ptr` se *puede* modificar mediante el uso de `ptr` desreferenciado, aun cuando el mismo `ptr` se haya declarado como `const`.

```
1  // Fig. 8.11: fig08_11.cpp
2  // Intento de modificar un apuntador constante a datos no constantes.
3
4  int main()
5  {
6     int x, y;
7
8     // ptr es un apuntador constante a un entero que se puede
9     // modificar a través de ptr, pero ptr siempre apunta a la
10    // misma ubicación de memoria.
11    int * const ptr = &x; // el apuntador const se debe inicializar
12
13    *ptr = 7; // se permite: *ptr no es const
14    ptr = &y; // error: ptr es const; no se puede asignar a una nueva dirección
15 } // fin de main
```

Mensaje de error del compilador Microsoft Visual C++:

```
you cannot assign to a variable that is const
```

Fig. 8.11 | Intento de modificar un apuntador constante a datos no constantes.

8.6.4 Apuntador constante a datos constantes

El *mínimo* privilegio de acceso se otorga mediante un **apuntador constante a datos constantes**. Dicho apuntador *siempre* apunta a la *misma* ubicación de memoria, y los datos en esa ubicación de memoria *no se pueden* modificar mediante el uso del apuntador. Así es como se debe pasar un arreglo integrado a una función que *sólo lea* el arreglo integrado, usando la notación de subíndices de arreglos, y *no se modifica* el arreglo integrado. El programa de la figura 8.12 declara la variable apuntador `ptr` de tipo `const int * const` (línea 13). Esta declaración se lee de *derecha a izquierda* como "`ptr` es un *apuntador constante a una constante entera*". La figura muestra los mensajes de error del compilador LLVM de Xcode que se generan cuando se hace un intento de modificar los datos a los que `ptr` apunta (línea 17) y cuando se hace un intento de modificar la dirección almacenada en la variable apuntador (línea 18); esto aparece en las líneas de código con los errores en el editor de texto de Xcode. En la línea 15 no ocurren errores cuando el programa intenta desreferenciar a `ptr`, o cuando el programa trata de imprimir el valor al que `ptr` apunta, ya que *ni* el apuntador *ni* los datos a los que apunta se están modificando en la instrucción.

```
1  // Fig. 8.12: fig08_12.cpp
2  // Intento de modificar un apuntador constante a datos constantes.
3  #include <iostream>
```

Fig. 8.12 | Intento de modificar un apuntador constante a datos constantes (parte 1 de 2).

```
4  using namespace std;
5
6  int main()
7  {
8     int x = 5, y;
9
10    // ptr es un apuntador constante a un entero constante.
11    // ptr siempre apunta a la misma ubicación; el entero
12    // en esa ubicación no se puede modificar.
13    const int *const ptr = &x;
14
15    cout << *ptr << endl;
16
17    *ptr = 7; // error: *ptr es const; no se puede asignar un nuevo valor
18    ptr = &y; // error: ptr es const; no se puede asignar una nueva dirección
19 } // fin de main
```

Mensaje de error del compilador LLVM de Xcode:

```
Read-only variable is not assignable
Read-only variable is not assignable
```

Fig. 8.12 | Intento de modificar un apuntador constante a datos constantes (parte 2 de 2).

8.7 Operador `sizeof`

El operador unario `sizeof` en *tiempo de compilación* determina el tamaño en bytes de un arreglo integrado o de cualquier otro tipo de datos, variable o constante *durante la compilación de un programa*. Cuando se aplica al *nombre* de un arreglo integrado, como en la figura 8.13 (línea 13), el operador `sizeof` devuelve el *número total de bytes en el arreglo integrado* como un valor de tipo `size_t`. La computadora que utilizamos para compilar este programa almacena variables de tipo `double` en 8 bytes de memoria, y `numeros` se declara de forma que tenga 20 elementos (línea 11), por lo que usa 160 bytes en memoria. Cuando se aplica a un *parámetro apuntador* (línea 22) en una función que *recibe un arreglo integrado como argumento*, el operador `sizeof` devuelve el tamaño del *apuntador* en bytes (4 en el sistema que utilizamos); *no* el tamaño del arreglo.

Error común de programación 8.3

Al utilizar el operador `sizeof` en una función para buscar el tamaño en bytes de un parámetro tipo arreglo, se obtiene el tamaño en bytes de un apuntador, no el tamaño en bytes del arreglo integrado.

```
1  // Fig. 8.13: fig08_13.cpp
2  // Al aplicar el operador sizeof al nombre de un arreglo integrado
3  // se devuelve el número de bytes en el arreglo integrado.
4  #include <iostream>
5  using namespace std;
6
```

Fig. 8.13 | Al aplicar el operador `sizeof` al nombre de un arreglo, se devuelve el número de bytes en el mismo (parte 1 de 2).

```
7  size_t getSize( double * ); // prototipo
8
9  int main()
10 {
11    double numeros[ 20 ]; // 20 valores double; ocupa 160 bytes en nuestro sistema
12
13    cout << "El numero de bytes en el arreglo es " << sizeof( numeros );
14
15    cout << "\nEl numero de bytes devueltos por getSize es "
16       << getSize( numeros ) << endl;
17 } // fin de main
18
19 // devuelve el tamaño de ptr
20 size_t getSize( double *ptr )
21 {
22    return sizeof( ptr );
23 } // fin de la función getSize
```

```
El numero de bytes en el arreglo es 160
El numero de bytes devueltos por getSize es 4
```

Fig. 8.13 | Al aplicar el operador `sizeof` al nombre de un arreglo, se devuelve el número de bytes en el mismo (parte 2 de 2).

El número de *elementos* en un arreglo integrado se puede determinar mediante el uso de los resultados de dos operaciones `sizeof`. Por ejemplo, para determinar el número de elementos en el arreglo integrado `numeros`, use la siguiente expresión (que se evalúa en *tiempo de compilación*):

```
sizeof numeros / sizeof( numeros[ 0 ] )
```

La expresión divide el número de bytes en `numeros` (160, suponiendo valores `double` de 8 bytes) entre el número de bytes en el elemento cero del arreglo integrado (8); el resultado es el número de elementos en `numeros` (20).

Cómo determinar el tamaño de los tipos fundamentales, un arreglo integrado y un apuntador

En la figura 8.14 se utiliza `sizeof` para calcular el número de bytes utilizados para almacenar la mayoría de los tipos de datos estándar. La salida se produjo usando la configuración predeterminada en Visual C++ 2012, en una computadora con Windows 7. Los tamaños de los tipos son *dependientes de la plataforma*. Por ejemplo, en otro sistema, `double` y `long double` pueden ser de distintos tamaños.

```
1  // Fig. 8.14: fig08_14.cpp
2  // Uso del operador sizeof para determinar los tamaños de los tipos de datos
      estándar.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     char c; // variable de tipo char
9     short s; // variable de tipo short
```

Fig. 8.14 | Uso del operador `sizeof` para determinar tamaños estándar de los tipos de datos (parte 1 de 2).

```
10    int i; // variable de tipo int
11    long l; // variable de tipo long
12    long ll; // variable de tipo long long
13    float f; // variable de tipo float
14    double d; // variable de tipo double
15    long double ld; // variable de tipo long double
16    int array[ 20 ]; // arreglo integrado de int
17    int *ptr = array; // variable de tipo int *
18
19    cout << "sizeof c = " << sizeof c
20       << "\tsizeof(char) = " << sizeof( char )
21       << "\nsizeof s = " << sizeof s
22       << "\tsizeof(short) = " << sizeof( short )
23       << "\nsizeof i = " << sizeof i
24       << "\tsizeof(int) = " << sizeof( int )
25       << "\nsizeof l = " << sizeof l
26       << "\tsizeof(long) = " << sizeof( long )
27       << "\nsizeof ll = " << sizeof ll
28       << "\tsizeof(long long) = " << sizeof( long long )
29       << "\nsizeof f = " << sizeof f
30       << "\tsizeof(float) = " << sizeof( float )
31       << "\nsizeof d = " << sizeof d
32       << "\tsizeof(double) = " << sizeof( double )
33       << "\nsizeof ld = " << sizeof ld
34       << "\tsizeof(long double) = " << sizeof( long double )
35       << "\nsizeof arreglo = " << sizeof arreglo
36       << "\nsizeof ptr = " << sizeof ptr << endl;
37 } // fin de main
```

```
sizeof c = 1    sizeof(char) = 1
sizeof s = 2    sizeof(short) = 2
sizeof i = 4    sizeof(int) = 4
sizeof l = 4    sizeof(long) = 4
sizeof ll = 8   sizeof(long long) = 8
sizeof f = 4    sizeof(float) = 4
sizeof d = 8    sizeof(double) = 8
sizeof ld = 8   sizeof(long double) = 8
sizeof arreglo = 80
sizeof ptr = 4
```

Fig. 8.14 | Uso del operador `sizeof` para determinar tamaños estándar de los tipos de datos (parte 2 de 2).

Tip de portabilidad 8.1

El número de bytes utilizados para almacenar un tipo de datos específico puede variar de un sistema a otro. Al escribir programas que dependan de los tamaños de los tipos de datos, use siempre `sizeof` *para determinar el número de bytes utilizados para almacenar los tipos de datos.*

El operador `sizeof` se puede aplicar a cualquier expresión o nombre de tipo. Cuando se aplica `sizeof` al nombre de una variable (que no sea el nombre de un arreglo integrado) u otra expresión, se devuelve el número de bytes utilizados para almacenar el tipo específico del valor de la expresión. Los paréntesis utilizados con `sizeof` *sólo* son requeridos si se suministra el nombre de un tipo (por ejemplo, `int`) como su operando. Los paréntesis utilizados con `sizeof` *no* son requeridos cuando el operando de `sizeof` es una expresión. Recuerde que `sizeof` es un operador en *tiempo de compilación*, por lo que no se evalúa su operando.

8.8 Expresiones y aritmética de apuntadores

En esta sección se describen los operadores que pueden tener *apuntadores* como operandos, y cómo se utilizan estos operadores con los apuntadores. C++ permite la **aritmética de apuntadores**: es posible realizar unas cuantas operaciones aritméticas con apuntadoras. *La aritmética de apuntadores es apropiada sólo para apuntadores que apuntan a los elementos de arreglos integrados.*

Un apuntador se puede incrementar (++) o decrementar (--), se puede sumar un entero a un apuntador (+ o +=), se puede restar un entero de un apuntador (- o -=), o se puede restar un apuntador de otro apuntador del mismo tipo; esta operación en particular es apropiada sólo para dos apuntadores que apuntan a elementos del *mismo* arreglo integrado.

Tip de portabilidad 8.2

La mayoría de las computadoras de la actualidad tienen enteros de cuatro o de ocho bytes. Debido a que los resultados de la aritmética de apuntadores dependen del tamaño de los objetos a los que apunta un apuntador, la aritmética de apuntadores es dependiente del equipo.

Suponga que se ha declarado el arreglo `int v[5]` y que su primer elemento se encuentra en la ubicación de memoria 3000. Suponga que se ha inicializado el apuntador `vPtr` para que apunte a `v[0]` (es decir, el valor de `vPtr` es 3000). En la figura 8.15 se muestra un diagrama de esta situación para un equipo con enteros de cuatro bytes. Observe que `vPtr` se puede inicializar para que apunte al arreglo `v` con cualquiera de las siguientes instrucciones (debido a que el nombre de un arreglo integrado es equivalente a la dirección de su elemento cero):

```
int *vPtr = v;
int *vPtr = &v[ 0 ];
```

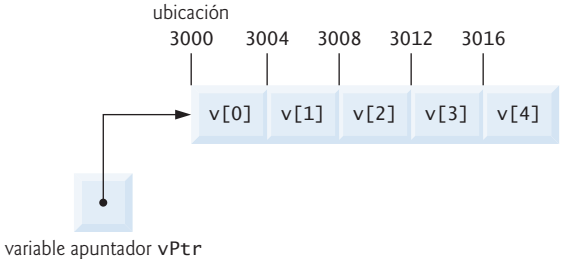


Fig. 8.15 | El arreglo integrado `v` y una variable apuntador `int *vPtr` que apunta a `v`.

Sumar enteros a, y restar enteros de apuntadores

En la aritmética convencional, la suma `3000 + 2` produce el valor `3002`. Por lo general éste *no* es el caso con la aritmética de apuntadores. Cuando se suma (o se resta) un entero a un apuntador, éste *no* simplemente se incrementa o decrementa debido a ese entero, sino por ese entero *multiplicado por el tamaño del objeto al que el apuntador hace referencia*. El número de bytes depende del tipo de datos del objeto. Por ejemplo, la instrucción

```
vPtr += 2;
```

produciría `3008` (del cálculo `3000 + 2 * 4`), suponiendo que un `int` se almacena en cuatro bytes de memoria. En el arreglo integrado `v`, `vPtr` apuntaría ahora a `v[2]` (figura 8.16). Si se almacena un entero en ocho bytes de memoria, entonces el cálculo anterior produciría la ubicación de memoria `3016` (`3000 + 2 * 8`).

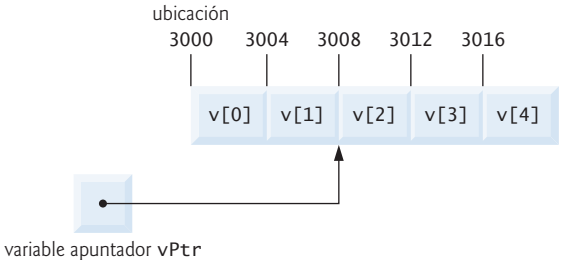


Fig. 8.16 | El apuntador `vPtr` después de la aritmética de apuntadores.

Si `vPtr` se hubiera incrementado a `3016`, lo cual apunta a `v[4]`, la instrucción

```
vPtr -= 4;
```

establecería a `vPtr` de vuelta en `3000`; el inicio del arreglo integrado. Si un apuntador se va a incrementar o decrementar por uno, se pueden utilizar los operadores de incremento (`++`) y decremento (`--`). Cada una de las instrucciones

```
++vPtr;
vPtr++;
```

incrementa el apuntador para que apunte al *siguiente* elemento del arreglo integrado. Cada una de las instrucciones

```
--vPtr;
vPtr--;
```

decrementa el apuntador para que apunte al elemento *anterior* del arreglo integrado.

Tip para prevenir errores 8.5

En la aritmética de apuntadores no hay comprobación de límites. El programador debe asegurarse de que cada operación aritmética con apuntadores que sume o reste un entero a un apuntador dé como resultado otro apuntador que haga referencia a un elemento dentro de los límites del arreglo integrado.

Resta de apuntadores

Las variables apuntador que apuntan al *mismo* arreglo integrado se pueden restar entre sí. Por ejemplo, si `vPtr` contiene la dirección `3000` y `v2Ptr` contiene la dirección `3008`, la instrucción

```
x = v2Ptr - vPtr;
```

asignaría a `x` el *número de elementos del arreglo integrado* de `vPtr` a `v2Ptr`; en este caso, `2`. *La aritmética de apuntadores no tiene significado, a menos que se realice en un apuntador que apunte a un arreglo integrado*. No podemos asumir que dos variables del mismo tipo se almacenan en forma contigua en la memoria, a menos que sean elementos adyacentes de un arreglo integrado.

Error común de programación 8.4

Restar o comparar dos apuntadores que no hagan referencia a los elementos del mismo arreglo integrado es un error lógico.

Asignación de apuntadores

Un apuntador se puede asignar a otro, si ambos apuntadores son del *mismo* tipo. En caso contrario, se debe usar un operador de conversión de tipos (por lo general, `reinterpret_cast`; lo veremos en la sección 14.7) para convertir el valor del apuntador, que está a la derecha de la asignación, al tipo del apuntador que está a la izquierda de la asignación. La excepción a esta regla es el **apuntador a `void`** (es decir, **`void *`**), el cual es un apuntador genérico capaz de representar *cualquier* tipo de apuntador. *Cualquier apuntador a un tipo fundamental o tipo de clase puede asignarse a un apuntador de tipo* `void *` *sin necesidad de conversión de tipos*. Sin embargo, un apuntador de tipo `void *` *no se puede* asignar directamente a un apuntador de otro tipo; el apuntador de tipo `void *` debe *convertirse* primero en el tipo apropiado de apuntador.

*No se puede desreferenciar un void **

No se puede desreferenciar un apuntador `void *`. Por ejemplo, el compilador "sabe" que un apuntador a `int` hace referencia a cuatro bytes de memoria en un equipo con enteros de cuatro bytes, pero un apuntador a `void` simplemente contiene una dirección de memoria para un tipo de datos *desconocido*; el compilador *no* conoce el número preciso de bytes a los que hace referencia el apuntador y el tipo de los datos. El compilador debe conocer el tipo de datos para determinar el número de bytes a desreferenciar para un apuntador específico; para un apuntador a `void`, este número de bytes *no se puede* determinar.

Error común de programación 8.5

Asignar un apuntador de un tipo a un apuntador de otro (que no sea `void *`*) sin usar una conversión de tipos (por lo general,* `reinterpret_cast`*) es un error de compilación.*

Error común de programación 8.6

Las operaciones permitidas en apuntadores `void *` *son: comparar apuntadores* `void *` *con otros apuntadores, convertir apuntadores* `void *` *a otros tipos de apuntadores y asignar direcciones a apuntadores* `void *`*. Todas las demás operaciones en apuntadores* `void *` *son errores de compilación.*

Comparación de apuntadores

Los apuntadores se pueden comparar mediante el uso de los operadores de igualdad y relacionales. Las comparaciones en las que se utilizan operadores relacionales no tienen significado, a menos que los apuntadores apunten a miembros del *mismo* arreglo integrado. Las comparaciones con apuntadores comparan las *direcciones* almacenadas en los apuntadores. Una comparación de dos apuntadores que apunten al mismo arreglo integrado podría mostrar, por ejemplo, que un apuntador apunta a un elemento de mayor numeración del arreglo integrado que el otro apuntador. Un uso común de la comparación de apuntadores es determinar si un apuntador tiene el valor `nullptr`, `0` o `NULL` (es decir, si el apuntador no apunta a nada).

8.9 Relación entre apuntadores y arreglos integrados

Los arreglos integrados y los apuntadores están estrechamente relacionados en C++ y se pueden utilizar de manera *casi* intercambiable. Los apuntadores se pueden utilizar para realizar cualquier operación en la que se involucren los subíndices de arreglos.

Suponga las siguientes declaraciones:

```
int b[ 5 ]; // crea el arreglo int b de 5 elementos; b es un apuntador const
int *bPtr; // crea el apuntador int bPtr, que no es un apuntador const
```

Podemos establecer `bPtr` a la dirección del primer elemento en el arreglo integrado `b` con la instrucción

```
bPtr = b; // asigna la dirección del arreglo integrado b a bPtr
```

Esto es equivalente a asignar la dirección del primer elemento del arreglo, como se muestra a continuación:

```
bPtr = &b[ 0 ]; // también asigna la dirección del arreglo integrado b a bPtr
```

Notación apuntador/desplazamiento

El elemento `b[3]` del arreglo integrado se puede referenciar de manera alternativa con la siguiente expresión de apuntador:

```
*( bPtr + 3 )
```

El `3` en la expresión anterior es el **desplazamiento** para el apuntador. Cuando el apuntador apunta al inicio de un arreglo integrado, el desplazamiento indica a cuál elemento del arreglo integrado se debe hacer referencia, y el valor del desplazamiento es idéntico al subíndice. La notación anterior se conoce como **notación apuntador/desplazamiento**. Los paréntesis son necesarios, debido a que la precedencia de `*` es mayor que la precedencia de `+`. Sin los paréntesis, la expresión anterior sumaría `3` a una copia del valor de `*bPtr` (es decir, se sumaría `3` a `b[0]`, suponiendo que `bPtr` apunta al inicio del arreglo integrado).

Así como se puede hacer referencia al elemento del arreglo integrado con una expresión de apuntador, la *dirección*

```
&b[ 3 ]
```

se puede escribir con la expresión de apuntador

```
bPtr + 3
```

Notación apuntador/desplazamiento con el nombre del arreglo integrado como el apuntador

El nombre del arreglo integrado se puede tratar como apuntador y se puede utilizar en la aritmética de apuntadores. Por ejemplo, la expresión

```
*( b + 3 )
```

también se refiere al elemento `b[3]` del arreglo. En general, todas las expresiones de arreglos integrados con subíndice se pueden escribir con un apuntador y un desplazamiento. En este caso, la notación apuntador/desplazamiento se utilizó con el nombre del arreglo integrado como un apuntador. La expresión anterior *no* modifica el nombre del arreglo integrado; `b` sigue apuntando al primer elemento en el arreglo integrado.

Notación apuntador/subíndice

Los apuntadores pueden usar subíndices de la misma forma que los arreglos integrados. Por ejemplo, la expresión

```
bPtr[ 1 ]
```

hace referencia al elemento `b[1]` del arreglo; esta expresión utiliza la **notación apuntador/subíndice**.

El nombre de un arreglo integrado no se puede modificar

La expresión

```
b += 3
```

produce un error de compilación, ya que trata de *modificar* el valor del nombre del arreglo integrado con la aritmética de apuntadores.

Buena práctica de programación 8.3

Por cuestión de claridad, usamos la notación de arreglos integrados en vez de la notación de apuntadores al manipular arreglos integrados.

Demostración de la relación entre apuntadores y arreglos integrados

En la figura 8.17 se utilizan las cuatro notaciones descritas en esta sección para hacer referencia a los elementos de un arreglo integrado (*notación de subíndice de arreglo, notación apuntador/desplazamiento con el nombre del arreglo integrado como apuntador, notación de subíndice de apuntador y notación apuntador/desplazamiento con un apuntador*) para realizar la misma tarea, a saber, imprimir los cuatro elementos del arreglo integrado de valores `int`, llamado `b`.

```
1  // Fig. 8.17: fig08_17.cpp
2  // Uso de notaciones de subíndice y apuntador con arreglos integrados.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int b[] = { 10, 20, 30, 40 }; // crea el arreglo integrado b de 4 elementos
9     int *bPtr = b; // establece bPtr para que apunte al arreglo integrado b
10
11    // imprime el arreglo integrado b usando la notación de subíndice de arreglo
12    cout << "Se imprime el arreglo b con:\n\nNotacion de subindice de arreglo\n";
13
14    for ( size_t i = 0; i < 4; ++i )
15       cout << "b[" << i << "] = " << b[ i ] << '\n';
16
17    // imprime el arreglo integrado b usando el nombre del arreglo y la notación
         apuntador/desplazamiento
18    cout << "\nNotacion apuntador/desplazamiento en donde "
19       << "el apuntador es el nombre del arreglo\n";
20
21    for ( size_t desplazamiento1 = 0; desplazamiento1 < 4; ++desplazamiento1 )
22       cout << "*(b + " << desplazamiento1 << ") = " << *( b + desplazamiento1 )
         << '\n';
23
24    // imprime el arreglo integrado b usando bPtr y la notación de subíndice de
         arreglo
25    cout << "\nNotacion de subindice de apuntador\n";
26
27    for ( size_t j = 0; j < 4; ++j )
28       cout << "bPtr[" << j << "] = " << bPtr[ j ] << '\n';
29
30    cout << "\nNotacion apuntador/desplazamiento\n";
31
32    // imprime el arreglo integrado b usando bPtr y la notacion apuntador/
         desplazamiento
33    for ( size_t desplazamiento2 = 0; desplazamiento2 < 4; ++desplazamiento2 )
34       cout << "*(bPtr + " << desplazamiento2 << ") = "
35          << *( bPtr + desplazamiento2 ) << '\n';
36 } // fin de main
```

```
Se imprime el arreglo b con:

Notacion de subindice de arreglo
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40
```

Fig. 8.17 | Uso de notaciones de subíndice y apuntador con los arreglos integrados (parte 1 de 2).

```
Notacion apuntador/desplazamiento en donde el apuntador es el nombre del arreglo
*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

Notacion de subindice de apuntador
bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

Notacion apuntador/desplazamiento
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

Fig. 8.17 | Uso de notaciones de subíndice y apuntador con los arreglos integrados (parte 2 de 2).

8.10 Cadenas basadas en apuntador

Ya utilizamos la clase `string` de la Biblioteca estándar de C++ para representar las cadenas como objetos completos. Por ejemplo, el caso de estudio de la clase `LibroCalificaciones` en los capítulos 3 a 7 representa el nombre de un curso que usa un objeto `string`. En el capítulo 21 (en inglés en el sitio web) se presenta la clase `string` con detalle. En esta sección introduciremos las cadenas basadas en apuntador estilo C (como se definen mediante el lenguaje de programación C), a las que simplemente llamaremos **cadenas de C**. *La clase `string` de C++ es preferida para usarse en nuevos programas, ya que elimina muchos de los problemas de seguridad y errores que pueden ocurrir al manipular cadenas de C.* Aquí cubriremos las cadenas de C para una comprensión más detallada de los apuntadores y los arreglos integrados. Además, si trabaja con programas heredados de C y C++, es probable que se encuentre con cadenas basadas en apuntador. En el apéndice F cubriremos con detalle las cadenas de C.

Caracteres y constantes tipo carácter

Los caracteres son los bloques de construcción fundamentales de los programas de código fuente de C++. Todo programa está compuesto de una secuencia de caracteres que (al agruparse de una manera significativa) el compilador interpreta como una serie de instrucciones que se utilizan para realizar una tarea. Un programa puede contener **constantes tipo carácter**. Una constante carácter es un valor entero que se representa como un carácter entre comillas sencillas. El *valor* de una constante carácter es el valor entero del carácter en el conjunto de caracteres del equipo. Por ejemplo, 'z' representa el valor entero de z (122 en el conjunto de caracteres ASCII; vea el apéndice B), y '\n' representa el valor entero de nueva línea (10 en el conjunto de caracteres ASCII).

Cadenas de caracteres

Una cadena es una serie de caracteres que se trata como una sola unidad. Una cadena puede incluir letras, dígitos y diversos **caracteres especiales** tales como +, -, *, / y $. Las **literales de cadena**, o **constantes de cadena** en C++ se escriben entre dobles comillas, como se muestra a continuación:

```
"John Q. Doe"              (un nombre)
"9999 Main Street"         (una calle)
"Maynard, Massachusetts"   (una ciudad y un estado)
"(201) 555-1212"           (un número telefónico)
```

Cadenas basadas en apuntador

Una cadena basada en apuntador es un arreglo integrado de caracteres que termina con el **carácter nulo** (`'\0'`), el cual indica en dónde termina la cadena en memoria. Una cadena se utiliza a través de un apuntador a su primer carácter. El valor de (`sizeof`) una literal de cadena es la longitud de la cadena, *incluyendo* el carácter nulo de terminación. Las cadenas basadas en apuntador son como los arreglos integrados: el nombre de un arreglo es también un apuntador a su primer elemento.

Literales de cadena como inicializadores

Una literal de cadena se puede utilizar como inicializador en la declaración de un arreglo integrado de caracteres, o de una variable de tipo `const char *`. Cada una de las declaraciones

```
char color[] = "azul";
const char *colorPtr = "azul";
```

inicializa una variable con la cadena `"azul"`. La primera declaración crea un arreglo integrado de *cinco elementos* llamado `color`, el cual contiene los caracteres `'a'`, `'z'`, `'u'`, `'l'` *y* `'\0'`. La segunda declaración crea la variable apuntador `colorPtr` que apunta a la letra `b` en la cadena `"azul"` (que termina en `'\0'`) en alguna parte de la memoria. Las literales de cadena tienen una *duración de almacenamiento estática* (existen durante la ejecución del programa) y pueden o no *compartirse*, si se hace referencia a la misma literal de cadena desde varias ubicaciones en un programa.

Tip para prevenir errores 8.6

Si necesita modificar el contenido de una literal de cadena, almacénela primero en un arreglo integrado de valores `char`.

Constantes tipo carácter como inicializadores

La declaración `char color[] = "azul";` también se podría escribir como

```
char color[] = { 'a', 'z', 'u', 'l', '\0' };
```

que utiliza constantes tipo carácter entre comillas sencillas (') como inicializadores para cada elemento del arreglo integrado. Al declarar un arreglo integrado de caracteres para que contenga una cadena, el arreglo integrado debe ser lo bastante grande como para almacenar la cadena *y* su carácter nulo de terminación. El compilador determina el tamaño del arreglo integrado en la declaración anterior, con base en el *número de inicializadores proporcionados en la lista inicializadora*.

Error común de programación 8.7

Si no se asigna suficiente espacio en un arreglo integrado de caracteres para almacenar el carácter nulo que termina una cadena, se produce un error.

Error común de programación 8.8

Crear o usar una cadena estilo C que no contiene un carácter nulo de terminación puede producir errores lógicos.

Tip para prevenir errores 8.7

Al almacenar una cadena de caracteres en un arreglo integrado de caracteres, asegúrese de que el arreglo sea lo bastante grande como para contener la cadena más larga que se vaya a almacenar. C++ permite almacenar cadenas de cualquier longitud. Si una cadena es mayor que el arreglo integrado de caracteres en la que se va a almacenar, los caracteres que estén más allá del final del arreglo integrado sobrescribirán datos en las ubicaciones de memoria que sigan después del arreglo integrado, lo cual produce errores lógicos y fugas de seguridad potenciales.

Acceder a los caracteres en una cadena estilo C

Como una cadena estilo C es un arreglo integrado de caracteres, podemos acceder a los caracteres individuales en una cadena directamente con la notación de subíndice de arreglo. Por ejemplo, en la declaración anterior, `color[0]` es el carácter `'a'`, `color[2]` es `'u'` y `color[4]` es el carácter nulo.

Leer cadenas de caracteres y colocarlas en arreglos integrados tipo `char` *mediante* `cin`

Una cadena se puede leer y colocar en un arreglo integrado de caracteres mediante la extracción de flujos con `cin`. Por ejemplo, la siguiente instrucción se puede utilizar para colocar una cadena en el arreglo integrado de 20 elementos tipo `char`, llamado `palabra`:

```
cin >> palabra;
```

La cadena introducida por el usuario se almacena en `palabra`. La anterior instrucción lee caracteres hasta encontrar un carácter de espacio en blanco o un indicador de fin de archivo. La cadena no debe ser mayor de 19 caracteres, para dejar espacio al carácter nulo de terminación. Se puede utilizar el manipulador de flujo `setw` para *asegurar* que la cadena que se coloque en `palabra` *no excede al tamaño del arreglo integrado*. Por ejemplo, la instrucción

```
cin >> setw( 20 ) >> palabra;
```

especifica que `cin` debe leer un máximo de 19 caracteres en el arreglo `palabra` y guardar la ubicación número 20 para almacenar el carácter nulo de terminación para la cadena. El manipulador de flujo `setw` no es una opción pegajosa; *sólo* se aplica al siguiente valor que se va a recibir como entrada. Si se introducen más de 19 caracteres, el resto de los mismos no se almacena en `palabra`, *sino que estarán en el flujo de entrada y podrán ser leídos por la siguiente operación de entrada*.[1] Desde luego que también puede fallar cualquier operación de entrada. En la sección 13.8 le mostraremos cómo detectar fallas de entrada.

Leer líneas de texto y colocarlas en arreglos integrados tipo `char` *mediante* `cin.getline`

En algunos casos es conveniente recibir como entrada una *línea completa de texto* y colocarla en un arreglo integrado de caracteres. Para este fin, el objeto `cin` cuenta con la función miembro **`getline`**, la cual recibe tres argumentos (un *arreglo integrado de caracteres* en el que se va a almacenar la línea de texto, una *longitud* y un *carácter delimitador*. Por ejemplo, las instrucciones

```
char enunciado[ 80 ];
cin.getline( enunciado, 80, '\n' );
```

declaran a `enunciado` como un arreglo integrado de 80 caracteres y leen una línea de texto del teclado, para colocarla en el arreglo integrado. La función deja de leer caracteres al momento de encontrar el carácter delimitador `'\n'`, cuando se introduce el *indicador de fin de archivo* o cuando el número de caracteres leídos hasta ese momento sea uno menos que la longitud especificada en el segundo argumento. El último carácter en el arreglo integrado se reserva para el *carácter nulo de terminación*. Al encontrar el carácter delimitador, éste se lee y se *descarta*. El tercer argumento para `cin.getline` tiene `'\n'` como valor predeterminado, por lo que la llamada a la función anterior podría escribirse de la siguiente manera:

```
cin.getline( enunciado, 80 );
```

El capítulo 13, Entrada/salida de flujos: un análisis más detallado, proporciona una discusión detallada sobre `cin.getline` y otras funciones de entrada/salida.

---

1 Para aprender cómo ignorar los caracteres adicionales en el flujo de entrada, consulte el artículo en: `www.daniweb.com/software-development/cpp/threads/90228/flushing-the-input-stream`.

Mostrar cadenas estilo C en pantalla
Un arreglo integrado de caracteres que representa una cadena con terminación nula puede mostrarse en pantalla mediante `cout` y <<. La instrucción

```
cout << enunciado;
```

muestra en pantalla el arreglo integrado llamado `enunciado`. Al igual que `cin`, `cout` no toma en cuenta qué tan grande es el arreglo integrado de caracteres. Los caracteres se imprimen hasta encontrar un *carácter nulo de terminación*; el carácter nulo *no* se muestra. [*Nota:* `cin` y `cout` asumen que el arreglo integrado de caracteres debe procesarse como cadenas terminadas por caracteres nulos; `cin` y `cout` no proporcionan herramientas similares de procesamiento de entrada y salida para otros tipos de arreglos integrados].

8.11 Conclusión

En este capítulo proporcionamos una introducción detallada a los apuntadores: variables que contienen direcciones de memoria como sus valores. Empezamos por demostrar cómo declarar e inicializar apuntadores. Vimos cómo utilizar el operador dirección (&) para asignar la dirección de una variable a un apuntador, y el operador indirección (*) para acceder a los datos almacenados en la variable a la que un apuntador hace referencia indirecta. Hablamos sobre cómo pasar argumentos por referencia, usando argumentos tipo apuntador.

Vimos cómo declarar y usar arreglos integrados, que C++ heredó del lenguaje de programación C. Aprendió a usar `const` con apuntadores para hacer valer el principio del menor privilegio. Demostramos cómo usar apuntadores no constantes a datos no constantes, apuntadores no constantes a datos constantes, apuntadores constantes a datos no constantes y apuntadores constantes a datos constantes. Hablamos sobre el operador `sizeof`, que se puede usar para determinar los tamaños de los tipos de datos y las variables en bytes, durante la compilación de un programa.

Demostramos cómo usar apuntadores en aritmética y expresiones de comparación. Vimos cómo se puede utilizar la aritmética de apuntadores para saltar de un elemento de un arreglo integrado a otro. Dimos una breve introducción sobre las cadenas basadas en apuntador.

En el siguiente capítulo, comenzaremos con nuestro tratamiento más detallado sobre las clases. Aprenderá acerca del alcance de los miembros de una clase, y cómo mantener los objetos en un estado consistente. También aprenderá acerca del uso de funciones miembro especiales, llamadas constructores y destructores, las cuales se ejecutan cuando un objeto se crea y se destruye, respectivamente; después hablaremos sobre cuándo se hacen las llamadas a los constructores y destructores. Además, demostraremos el uso de argumentos predeterminados con constructores, y el uso de la asignación a nivel de miembros para asignar un objeto de una clase con otro objeto de la misma clase. También hablaremos sobre el peligro de devolver una referencia a un miembro de datos `private` de una clase.

Resumen

Sección 8.2 Declaraciones e inicialización de variables apuntadores

- Los apuntadores son variables que contienen direcciones de memoria de otras variables como sus valores.
- La declaración

```
int *ptr;
```

declara a `ptr` como un apuntador a una variable de tipo `int` y se lee así: "`ptr` es un apuntador a un valor `int`". El uso que se da aquí al carácter * en una declaración indica que la variable es un apuntador.

- Es posible inicializar un apuntador con la dirección de un objeto del mismo tipo, o con `nullptr` (pág. 336).
- El único entero que se puede asignar a un apuntador sin conversión de tipos es `0`.

Sección 8.3 Operadores de apuntadores

- El operador `&` (dirección) (pág. 337) obtiene la dirección de memoria de su operando.
- El operando del operador dirección debe ser el nombre de una variable (o de otro valor *lvalue*); el operador dirección no se puede aplicar a constantes o expresiones que produzcan valores temporales (como los resultados de cálculos).
- El operador de indirección (o desreferencia) `*` (pág. 337) devuelve un sinónimo para el nombre del objeto al que apunta su operando en la memoria. A esto se le conoce como desreferenciar el apuntador (pág. 338).

Sección 8.4 Paso por referencia mediante apuntadores

- Al llamar a una función con un argumento que la función que hace la llamada desea que la función llamada modifique, se puede pasar la dirección del argumento. Así, la función llamada utiliza el operador indirección (`*`) para desreferenciar el apuntador y modificar el valor del argumento en la función que hace la llamada.
- Una función que recibe una dirección como argumento debe tener un apuntador como su correspondiente parámetro.

Sección 8.5 Arreglos integrados

- Los arreglos integrados (al igual que los objetos `array`) son estructuras de datos de tamaño fijo.
- Para especificar el tipo y número de los elementos requeridos por un arreglo integrado, use una declaración de la forma:

  tipo nombreArreglo`[` *tamañoArreglo* `]`;

  El compilador reservará la cantidad apropiada de memoria. El *tamañoArreglo* debe ser una constante entera mayor que cero.
- Al igual que con los objetos `array`, utilizamos el operador subíndice (`[]`) para acceder a los elementos individuales de un arreglo integrado.
- El operador subíndice (`[]`) no proporciona comprobación de límites para los objetos `array` o los arreglos integrados.
- Podemos inicializar los elementos de un arreglo integrado mediante una lista inicializadora. Si proporciona menos inicializadores que el número de elementos del arreglo integrado, los elementos restantes se inicializan con 0. Si proporciona demasiados inicializadores, se produce un error de compilación.
- Si se omite el tamaño del arreglo integrado de una declaración con una lista inicializadora, el compilador ajusta el tamaño del arreglo integrado al número de elementos en la lista inicializadora.
- El valor del nombre del arreglo integrado puede convertirse de manera implícita en la dirección en memoria del primer elemento del arreglo integrado.
- Para pasar un arreglo integrado a una función, sólo hay que pasar el nombre del arreglo integrado. La función a la que se llamó puede modificar todos los elementos de un arreglo integrado en la función que hizo la llamada (a menos que la función coloque la palabra `const` antes del parámetro de arreglo integrado correspondiente, para indicar que no deben modificarse los elementos del arreglo integrado).
- Los arreglos integrados no conocen su propio tamaño, por lo que una función que procesa un arreglo integrado debe tener parámetros para recibir tanto el arreglo integrado como su tamaño.
- El compilador no distingue entre una función que recibe un apuntador y una función que recibe un arreglo integrado unidimensional. Una función debe "saber" cuando recibe un arreglo integrado, o simplemente una variable individual que se pasa por referencia.
- El compilador convierte un parámetro de función para un arreglo integrado unidimensional como `const int valores[]` en la notación de apuntador `const int *valores`. Estas formas son intercambiables; por claridad es mejor usar los corchetes (`[]`) cuando la función espera un argumento tipo arreglo integrado.
- La función `sort` (y muchas otras funciones de la biblioteca) puede aplicarse también a los arreglos integrados.

- Las nuevas funciones `begin` y `end` de C++11 (del encabezado `<iterator>`; pág. 345) reciben cada una un arreglo integrado como argumento, y devuelven un apuntador que puede usarse con las funciones de la Biblioteca estándar de C++ como `sort`, para representar el rango de elementos del arreglo integrado a procesar.
- Los arreglos integrados no pueden compararse entre sí mediante los operadores relacionales y de igualdad.
- Los arreglos integrados no pueden asignarse unos a otros; los nombres de estos arreglos integrados son apuntadores `const`.
- Los arreglos integrados no conocen su propio tamaño.
- Los arreglos integrados no cuentan con comprobación de límites automática.
- En el código contemporáneo de C++, hay que usar objetos de las plantillas de clase `array` y `vector` (que son más robustas) para representar listas y tablas de valores.

Sección 8.6 Uso de `const` con apuntadores

- El calificador `const` nos permite informar al compilador que el valor de una variable específica no se debe modificar a través del identificador especificado.
- Hay cuatro formas de pasar un apuntador a una función: un apuntador no constante a datos no constantes (pág. 347), un apuntador no constante a datos constantes (pág. 347), un apuntador constante a datos no constantes (pág. 348) y un apuntador constante a datos constantes (pág. 349).
- Para pasar un solo elemento de un arreglo integrado por referencia mediante el uso de apuntadores, se debe pasar la dirección de ese elemento.

Sección 8.7 Operador `sizeof`

- El operador `sizeof` (pág. 350) determina el tamaño en bytes de un tipo, variable o constante en tiempo de compilación.
- Cuando se aplica al nombre de un arreglo integrado, el operador `sizeof` devuelve el número total de bytes en el arreglo integrado. Cuando se aplica a un parámetro de arreglo integrado, `sizeof` devuelve el tamaño de un apuntador.

Sección 8.8 Expresiones y aritmética de apuntadores

- C++ permite la aritmética de apuntadores (pág. 353): operaciones aritméticas que se pueden realizar con apuntadores.
- La aritmética de apuntadores es apropiada sólo para los que apuntan a elementos de arreglos integrados.
- Las operaciones aritméticas que se pueden realizar con los apuntadores son: incrementar (`++`) o decrementar (`--`) un apuntador, sumar (`+` o `+=`) un entero a un apuntador, restar (`-` o `-=`) un entero de un apuntador, y restar un apuntador de otro; esta operación específica es apropiada sólo para dos apuntadores que apuntan a elementos del mismo arreglo integrado.
- Cuando se suma o resta un entero a un apuntador, éste se incrementa o decrementa en base a ese entero multiplicado por el tamaño del objeto al que hace referencia el apuntador.
- Un apuntador se puede asignar a otro, si ambos son del mismo tipo. En caso contrario, debe usarse una conversión de tipos. La excepción a esto es el apuntador `void *`, el cual es un tipo de apuntador genérico que puede contener valores de apuntadores de cualquier tipo.
- Las únicas operaciones válidas en un apuntador `void *` son: comparar apuntadores `void *` con otros apuntadores, asignar direcciones a apuntadores `void *` y convertir apuntadores `void *` a tipos de apuntadores válidos.
- Los apuntadores se pueden comparar mediante el uso de los operadores de igualdad y relacionales. Las comparaciones mediante operadores relacionales sólo tienen significado si los apuntadores apuntan a miembros del mismo arreglo integrado.

Sección 8.9 Relación entre apuntadores y arreglos integrados

- Los apuntadores que apuntan a arreglos integrados pueden usar subíndices de la misma forma que los nombres de arreglos integrados.
- En la notación apuntador/desplazamiento (pág. 356) si el apuntador apunta al primer elemento de un arreglo integrado, el desplazamiento es el mismo que el de un subíndice de arreglo.

- Todas las expresiones de arreglos integrados con subíndice se pueden escribir con un apuntador y un desplazamiento (pág. 356), ya sea utilizando el nombre del arreglo integrado como apuntador o un apuntador separado que apunte al arreglo integrado.

Sección 8.10 Cadenas basadas en apuntador

- Una constante tipo carácter (pág. 358) es un valor entero que se representa como carácter entre comillas sencillas. Este valor de una constante tipo carácter es el valor entero del carácter en el conjunto de caracteres del equipo.
- Una cadena es una serie de caracteres que se tratan como una sola unidad. Una cadena puede incluir letras, dígitos y varios caracteres especiales como +, -, *, / y $.
- En C++, las literales de cadena (o constantes de cadena) se escriben entre signos de comillas dobles (pág. 358).
- Una cadena basada en apuntador es un arreglo integrado de caracteres que termina con un carácter nulo (`'\0'`; pág. 359), que indica en dónde termina la cadena en la memoria. Se accede a una cadena mediante un apuntador a su primer carácter.
- El tamaño de (`sizeof`) una literal de cadena es la longitud de ésta, incluyendo el carácter nulo de terminación.
- Es posible usar una literal de cadena como inicializador para un arreglo integrado de caracteres o una variable de tipo `const char*`.
- Las literales de cadena tienen duración de almacenamiento estático y tal vez puedan compartirse o no, si se hace referencia a la misma literal de cadena desde varias ubicaciones en un programa.
- El efecto de modificar una literal de cadena es indefinido; por ende, debemos declarar siempre un apuntador a una literal de cadena como `const char*`.
- Al declarar un arreglo integrado de caracteres para contener una cadena, el arreglo integrado debe ser lo suficientemente grande como para almacenar la cadena y su carácter nulo de terminación.
- Si una cadena es más larga que el arreglo integrado de caracteres en donde se va a almacenar, los caracteres más allá del final del arreglo integrado sobrescribirán los datos en memoria después del arreglo integrado, lo cual producirá errores lógicos.
- Es posible acceder a los caracteres individuales en una cadena en forma directa, mediante la notación de subíndices de arreglos.
- Se puede colocar una cadena en un arreglo integrado de caracteres mediante la extracción de flujos con `cin`. Se leen caracteres hasta encontrar un carácter de espacio en blanco o el indicador de fin de archivo.
- El manipulador de flujo `setw` puede usarse para asegurar que la cadena que se lee y se coloca en un arreglo integrado de caracteres no exceda el tamaño del arreglo integrado.

El objeto `cin` cuenta con la función miembro `getline` (pág. 360) para recibir como entrada una línea completa de texto y colocarla en un arreglo integrado de caracteres. La función recibe tres argumentos: un arreglo integrado de caracteres en donde se va a almacenar la línea de texto, una longitud y un carácter delimitador. El tercer argumento tiene `'\n'` como valor predeterminado.

- Es posible imprimir un arreglo integrado de caracteres para representar una cadena con terminación nula mediante `cout` y `<<`. Los caracteres de la cadena se imprimen hasta encontrar el carácter nulo de terminación.

Ejercicios de autoevaluación

8.1 Complete los siguientes enunciados:

a) Un apuntador es una variable que contiene como valor la _____ de otra variable.
b) Un apuntador se debe inicializar con _____ o _____.
c) El único entero que se puede asignar directamente a un apuntador es _____.

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

a) El operador dirección `&` se puede aplicar sólo a constantes y expresiones.

b) Un apuntador que se declara de tipo `void *` se puede desreferenciar.
c) Un apuntador de un tipo no se puede asignar a uno de otro tipo sin una operación de conversión de tipos.

8.3 Para cada uno de los siguientes enunciados, escriba instrucciones en C++ que realicen la tarea especificada. Suponga que los números de punto flotante con precisión doble se almacenan en ocho bytes, y que la dirección inicial del arreglo está en la ubicación 1002500 en la memoria. Cada parte del ejercicio debe usar los resultados de incisos anteriores, en donde sea apropiado.

a) Declarar un arreglo de tipo `double` llamado `numeros` con 10 elementos, e inicializar los elementos con los valores `0.0, 1.1, 2.2, ..., 9.9`. Suponga que se ha definido la constante `tamanio` como `10`.
b) Declarar un apuntador `nPtr` que apunte a una variable de tipo `double`.
c) Usar una instrucción `for` para imprimir los elementos del arreglo integrado `numeros` mediante el uso de notación de subíndice. Imprima cada número con un dígito a la derecha del punto decimal.
d) Escribir dos instrucciones separadas, cada una de las cuales debe asignar la dirección inicial del arreglo integrado `numeros` a la variable apuntador `nPtr`.
e) Usar una instrucción `for` para imprimir los elementos del arreglo integrado `numeros`, usando la notación apuntador/desplazamiento con el apuntador `nPtr`.
f) Usar una instrucción `for` para imprimir los elementos del arreglo `numeros`, usando la notación apuntador/desplazamiento con el nombre del arreglo como el apuntador.
g) Usar una instrucción `for` para imprimir los elementos del arreglo `numeros`, usando la notación apuntador/subíndice con el apuntador `nPtr`.
h) Hacer referencia al cuarto elemento del arreglo `numeros`, usando la notación de subíndice de arreglo, la notación apuntador/desplazamiento con el nombre del arreglo integrado como apuntador, la notación de subíndice de apuntador con `nPtr` y la notación apuntador/desplazamiento con `nPtr`.
i) Suponiendo que `nPtr` apunta al inicio del arreglo integrado `numeros`, ¿qué dirección se desreferencia mediante `nPtr + 8`? ¿Qué valor se almacena en esa ubicación?
j) Suponiendo que `nPtr` apunta a `numeros[5]`, ¿qué dirección se referencia mediante `nPtr` después de ejecutar `nPtr -= 4`? ¿Cuál es el valor almacenado en esa ubicación?

8.4 Para cada uno de los siguientes enunciados, escriba una sola instrucción que realice la tarea especificada. Suponga que se han declarado las variables de punto flotante `numero1` y `numero2`, y que `numero1` se ha inicializado con `7.3`.

a) Declarar la variable `fPtr` para que sea un apuntador a un objeto de tipo `double` e inicializar el apuntador con `nullptr`.
b) Asignar la dirección de la variable `numero1` a la variable apuntador `fPtr`.
c) Imprimir el valor del objeto al que apunta `fPtr`.
d) Asignar a la variable `numero2` el valor del objeto al que apunta `fPtr`.
e) Imprimir el valor de `numero2`.
f) Imprimir la dirección de `numero1`.
g) Imprimir la dirección almacenada en `fPtr`. ¿La dirección que se imprime es igual que la de `numero1`?

8.5 Realice la tarea especificada por cada uno de los siguientes enunciados:

a) Escribir el encabezado para una función llamada `intercambiar`, que reciba dos apuntadores a los números de punto flotante con precisión doble `x` y `y` como parámetros, y no devuelva un valor.
b) Escribir el prototipo para la función en la parte (a).
c) Escribir dos instrucciones, cada una de las cuales debe inicializar el arreglo de caracteres `vocal` con la cadena de vocales `"AEIOU"`.

8.6 Busque el error en cada uno de los siguientes segmentos de programa. Suponga las siguientes declaraciones e instrucciones:

```
int *zPtr; // zPtr hará referencia al arreglo z
void *sPtr = nullptr;
int numero;
int z[ 5 ] = { 1, 2, 3, 4, 5 };
```

a)
```
++zPtr;
```
b)
```
// usa el apuntador para obtener el primer valor del arreglo integrado
numero = zPtr;
```

c)
```
// asigna el elemento 2 del arreglo (el valor 3) a numero
numero = *zPtr[ 2 ];
```
d)
```
// imprime el arreglo integrado z completo
for ( size_t i = 0; i <= 5; ++i )
   cout << zPtr[ i ] << endl;
```
e)
```
// asigna a numero el valor al que apunta sPtr
numero = *sPtr;
```
f)
```
++z;
```

Respuestas a los ejercicios de autoevaluación

8.1 a) dirección. b) `nullptr`, una dirección. c) `0`.

8.2 a) Falso. El operando del operador dirección debe ser un *lvalue*; el operador dirección no se puede aplicar a constantes o expresiones que no den referencias como resultado.
b) Falso. Un apuntador a `void` no se puede desreferenciar. Dicho apuntador no tiene un tipo que permita al compilador determinar el número de bytes de memoria a desreferenciar, y el tipo de datos a los que apunta el apuntador.
c) Falso. Se pueden asignar apuntadores de cualquier tipo a apuntadores `void`. Los apuntadores de tipo `void` se pueden asignar a apuntadores de otros tipos sólo con una conversión de tipos explícita.

8.3 a)
```
double numeros[ tamanio ] = { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9 };
```
b)
```
double *nPtr;
```
c)
```
cout << fixed << showpoint << setprecision( 1 );
for ( size_t i = 0; i < tamanio; ++i )
   cout << numeros[ i ] << ' ';
```
d)
```
nPtr = numeros;
nPtr = &numeros[ 0 ];
```
e)
```
cout << fixed << showpoint << setprecision( 1 );
for ( size_t j = 0; j < tamanio; ++j )
   cout << *( nPtr + j ) << ' ';
```
f)
```
cout << fixed << showpoint << setprecision( 1 );
for ( size_t k = 0; k < tamanio; ++k )
   cout << *( numeros + k ) << ' ';
```
g)
```
cout << fixed << showpoint << setprecision( 1 );
for ( size_t m = 0; m < tamanio; ++m )
   cout << nPtr[ m ] << ' ';
```
h)
```
numeros[ 3 ]
*( numeros + 3 )
nPtr[ 3 ]
*( nPtr + 3 )
```
i)
```
La dirección es 1002500 + 8 * 8 = 1002564. El valor es 8.8.
```
j)
```
La dirección de numeros[ 5 ] es 1002500 + 5 * 8 = 1002540.
La dirección de nPtr -= 4 es 1002540 - 4 * 8 = 1002508.
El valor en esa ubicación es 1.1.
```

8.4 a)
```
double *fPtr = nullptr;
```
b)
```
fPtr = &numero1;
```
c)
```
cout << "El valor de *fPtr es " << *fPtr << endl;
```
d)
```
numero2 = *fPtr;
```
e)
```
cout << "El valor de numero2 es " << numero2 << endl;
```
f)
```
cout << "La direccion de numero1 es " << &numero1 << endl;
```
g)
```
cout << "La direccion almacenada en fPtr es " << fPtr << endl;
```
Sí, el valor es el mismo.

8.5 a) `void intercambiar( double *x, double *y )`
b) `void intercambiar( double *, double * );`
c) `char vocal[] = "AEIOU";`
`char vocal[] = { 'A', 'E', 'I', 'O', 'U', '\0' };`

8.6 a) *Error:* no se ha inicializado `zPtr`.
Corrección: inicializar `zPtr` con `zPtr = z;`
b) *Error:* el apuntador no se desreferencia.
Corrección: cambiar la instrucción a `numero = *zPtr;`
c) *Error:* `zPtr[ 2 ]` no es un apuntador y no se debe desreferenciar.
Corrección: cambiar `*zPtr[ 2 ]` a `zPtr[ 2 ]`.
d) *Error:* se hace referencia a un elemento del arreglo integrado fuera de los límites de éste, con subíndice de apuntador.
Corrección: para evitar esto, cambie el operador relacional en la instrucción `for` a `<` o cambie el `5` a `4`.
e) *Error:* se desreferencia un apuntador `void`.
Corrección: para desreferenciar el apuntador `void`, primero se debe convertir en un apuntador entero. Cambie la instrucción a `numero = *static_cast< int * >(sPtr);`
f) *Error:* tratar de modificar el nombre de un arreglo integrado con la aritmética de apuntadores.
Corrección: usar una variable apuntador en vez del nombre del arreglo integrado para realizar la aritmética de apuntadores, o usar un subíndice con el nombre del arreglo para hacer referencia a un elemento específico.

Ejercicios

8.7 ***(Verdadero o falso)*** Indique si cada uno de los siguientes enunciados es *verdadero* o *falso*. Si es *falso*, explique por qué.

a) Dos apuntadores que apuntan a distintos arreglos integrados no se pueden comparar de una forma que tenga sentido.
b) Como el nombre de un arreglo integrado puede convertirse de manera explícita en un apuntador al primer elemento del arreglo integrado, los nombres de los arreglos integrados se pueden manipular de la misma forma que los apuntadores.

8.8 ***(Escriba instrucciones de C++)*** Para cada uno de los siguientes enunciados, escriba instrucciones de C++ que realicen la tarea especificada. Suponga que los enteros sin signo se almacenan en dos bytes y que la dirección inicial del arreglo integrado está en la ubicación 1002500 en memoria.

a) Declarar un arreglo integrado de tipo `unsigned int` llamado `valores` con cinco elementos, e inicializar los elementos con los enteros pares del 2 al 10. Suponga que la constante simbólica `TAMANIO` se ha definido como `5`.
b) Declarar un apuntador `vPtr` que apunte a un objeto del tipo `unisgned int`.
c) Usar una instrucción `for` para imprimir los elementos del arreglo integrado `valores` mediante el uso de la notación de subíndices.
d) Escribir dos instrucciones separadas que asignen la dirección inicial del arreglo integrado `valores` a la variable apuntador `vPtr`.
e) Usar una instrucción `for` para imprimir los elementos del arreglo integrado `valores` usando la notación apuntador/desplazamiento.
f) Usar una instrucción `for` para imprimir los elementos del arreglo integrado `valores` usando la notación apuntador/desplazamiento, con el nombre del arreglo integrado como apuntador.
g) Usar una instrucción `for` para imprimir los elementos del arreglo integrado `valores`, mediante el uso de subíndices con el apuntador al arreglo.
h) Hacer referencia al quinto elemento de `valores` mediante el uso de la notación de subíndices de arreglo, la notación apuntador/desplazamiento con el nombre del arreglo integrado como apuntador, la notación de subíndice de apuntador y la notación apuntador/desplazamiento.

i) ¿Qué dirección se referencia mediante `vPtr + 3`? ¿Qué valor se almacena en esa ubicación?
j) Suponiendo que `vPtr` apunta a `valores[ 4 ]`, ¿qué dirección se referencia mediante `vPtr -= 4`? ¿Qué valor se almacena en esa ubicación?

8.9 *(Escriba instrucciones de C++)* Para cada uno de los siguientes enunciados, escriba una sola instrucción que realice la tarea especificada. Suponga que se declararon las variables `long` llamadas `valor1` y `valor2`, y que se inicializó `valor1` con `200000`.

a) Declarar la variable `longPtr` para que sea un apuntador a un objeto de tipo `long`.
b) Asignar la dirección de la variable `valor1` a la variable apuntador `longPtr`.
c) Imprimir el valor del objeto al que apunta `longPtr`.
d) Asignar a la variable `valor2` el valor del objeto al que apunta `longPtr`.
e) Imprimir el valor de `valor2`.
f) Imprimir la dirección de `valor1`.
g) Imprimir la dirección almacenada en `longPtr`. ¿El valor que se imprimió es igual que la dirección de `valor1`?

8.10 *(Encabezados y prototipos de funciones)* Realice la tarea especificada en cada uno de los siguientes enunciados:

a) Escribir el encabezado para la función `cero`, que reciba un parámetro tipo arreglo integrado de enteros `long` llamado `enterosGrandes`, y que no devuelva un valor.
b) Escriba el prototipo para la función en la parte (a).
c) Escriba el encabezado para la función `sumar1YSumar`, que reciba un parámetro tipo arreglo integrado entero `unoDemasiadoChico` y devuelva un entero.
d) Escribir el prototipo para la función descrita en la parte (c).

8.11 *(Busque los errores en el código)* Busque el error en cada uno de los siguientes segmentos. Si puede corregirse el error, explique cómo.

a)
```
int *numero;
cout << numero << endl;
```
b)
```
double *realPtr;
long *enteroPtr;
enteroPtr = realPtr;
```
c)
```
int * x, y;
x = y;
```
d)
```
char s[] = "este es un arreglo de caracteres";
for ( ; *s != '\0'; ++s)
   cout << *s << ' ';
```
e)
```
short *numPtr, resultado;
void *genericoPtr = numPtr;
resultado = *genericoPtr + 7;
```
f)
```
double x = 19.34;
double xPtr = &x;
cout << xPtr << endl;
```

8.12 *(Simulación: La tortuga y la liebre)* En este ejercicio usted recreará la clásica carrera de la tortuga y la liebre. Utilizará la generación de números aleatorios para desarrollar una simulación de este memorable suceso.

Nuestros competidores empezarán la carrera en la "posición 1" de 70 posiciones. Cada posición representa a una posible posición a lo largo del curso de la carrera. La línea de meta se encuentra en la posición 70. El primer competidor en llegar a la posición 70 recibirá una cubeta llena con zanahorias y lechuga frescas. El recorrido se abre paso hasta la cima de una resbalosa montaña, por lo que ocasionalmente los competidores pierden terreno.

Un reloj hace tictac una vez por segundo. Con cada tic del reloj, su programa debe usar las funciones `moverTortuga` y `moverLiebre` para ajustar la posición de los animales, de acuerdo con las reglas de la figura 8.18. Estas funciones deben usar el paso por referencia basado en apuntador para modificar la posición de la tortuga y la liebre.

Use variables para llevar el registro de las posiciones de los animales (los números de las posiciones son del 1 al 70). Empiece con cada animal en la posición 1 (la "puerta de inicio"). Si un animal se resbala hacia la izquierda antes de la posición 1, regréselo a la posición 1.

Genere los porcentajes que se muestran en la siguiente tabla, produciendo un entero aleatorio i en el rango $1 \leq i \leq 10$. Para la tortuga, realice un "paso pesado rápido" cuando $1 \leq i \leq 5$, un "resbalón" cuando $6 \leq i \leq 7$ o un "paso pesado lento" cuando $8 \leq i \leq 10$. Utilice una técnica similar para mover a la liebre.

Empiece la carrera imprimiendo el mensaje

```
PUM  !!!
Y ARRANCAN  !!!
```

Para cada tic del reloj (es decir, cada repetición de un ciclo) imprima una línea de 70 posiciones, mostrando la letra `T` en la posición de la tortuga y la letra `H` en la posición de la liebre. En ocasiones los competidores se encontrarán en la misma posición. En este caso, la tortuga muerde a la liebre y su programa debe imprimir `OUCH!!!` empezando en esa posición. Todas las posiciones de impresión distintas de la `T`, la `H` o el mensaje `OUCH!!!` (en caso de un empate) deben estar en blanco.

Después de imprimir cada línea, compruebe si uno de los animales ha llegado o se ha pasado de la posición 70. De ser así, imprima quién fue el ganador y termine la simulación. Si la tortuga gana, imprima `LA TORTUGA GANA!!! YAY!!!` Si la liebre gana, imprima `La liebre gana. Que mal.` Si ambos animales ganan en el mismo tic del reloj, tal vez usted quiera favorecer a la tortuga (la más débil) o quizás quiera imprimir `Es un empate.` Si ninguno de los dos animales gana, ejecute el ciclo de nuevo para simular el siguiente tic del reloj.

Animal	Tipo de movimiento	Porcentaje del tiempo	Movimiento actual
Tortuga	Paso pesado rápido	50%	3 posiciones a la derecha
	Resbalón	20%	6 posiciones a la izquierda
	Paso pesado lento	30%	1 posición a la derecha
Liebre	Dormir	20%	Ningún movimiento
	Gran salto	20%	9 posiciones a la derecha
	Gran resbalón	10%	12 posiciones a la izquierda
	Pequeño salto	30%	1 posición a la derecha
	Pequeño resbalón	20%	2 posiciones a la izquierda

Fig. 8.18 | Reglas para ajustar las posiciones de la tortuga y la liebre.

8.13 *(¿Qué hace este código?)* ¿Qué es lo que hace este programa?

```
1  // Ej. 8.13: ej08_13.cpp
2  // ¿Qué hace este programa?
3  #include <iostream>
4  using namespace std;
5
6  void misterio1( char *, const char * ); // prototipo
7
8  int main()
9  {
10    char cadena1[ 80 ];
11    char cadena2[ 80 ];
12
```

```
   cout << "Escriba dos cadenas: ";
   cin >> cadena1 >> cadena2;
   misterio1( cadena1, cadena2 );
   cout << cadena1 << endl;
} // fin de main

// ¿Qué hace esta función?
void misterio1( char *s1, const char *s2 )
{
   while ( *s1 != '\0' )
      ++s1;

   for ( ; ( *s1 = *s2 ); ++s1, ++s2 )
      ; // instrucción vacía
} // fin de la función misterio1
```

8.14 *(¿Qué hace este código?)* ¿Qué es lo que hace este programa?

```
// Ej. 8.14: ej08_14.cpp
// ¿Qué hace este programa?
#include <iostream>
using namespace std;

int misterio2( const char * ); // prototipo

int main()
{
  char cadena1[ 80 ];

   cout << "Escriba una cadena: ";
   cin >> cadena1;
   cout << misterio2( cadena1 ) << endl;
} // fin de main

// ¿Qué hace esta función?
int misterio2( const char *s )
{
   unsigned int x;

   for ( x = 0; *s != '\0'; ++s )
      ++x;

   return x;
} // fin de la función misterio2
```

Sección especial: construya su propia computadora

En los siguientes problemas nos desviaremos temporalmente del mundo de la programación en lenguajes de alto nivel. Vamos a "abrir de par en par" una computadora y ver su estructura interna. Presentaremos la programación en lenguaje máquina y escribiremos varios programas en este lenguaje. Para que ésta sea una experiencia valiosa, crearemos también una computadora (mediante la técnica de la *simulación* basada en software) en la que pueda ejecutar sus programas en lenguaje máquina.

8.15 *(Programación en lenguaje máquina)* Vamos a crear una computadora a la que llamaremos Simpletron. Como su nombre lo implica, es una máquina simple pero, como veremos pronto, también es poderosa. Simpletron sólo ejecuta programas escritos en el único lenguaje que entiende directamente: el lenguaje máquina de Simpletron, o LMS.

Simpletron contiene un *acumulador*, un "registro especial" en el cual se coloca la información antes de que Simpletron la utilice en los cálculos, o que la analice de distintas maneras. Toda la información dentro de Simpletron se manipula en términos de *palabras*. Una palabra es un número decimal con signo de cuatro dígitos, tal como `+3364`, `-1293`, `+0007`, `-0001`, etc. Simpletron está equipada con una memoria de 100 palabras, y se hace referencia a estas palabras mediante sus números de ubicación `00`, `01`, ..., `99`.

Antes de ejecutar un programa LMS debemos *cargar*, o colocar, el programa en memoria. La primera instrucción de cada programa LMS se coloca siempre en la ubicación `00`. El simulador empezará a ejecutarse en esta ubicación.

Cada instrucción escrita en LMS ocupa una palabra de la memoria de Simpletron; por lo tanto, las instrucciones son números decimales de cuatro dígitos con signo. Vamos a suponer que el signo de una instrucción LMS siempre será positivo, pero el signo de una palabra de información puede ser positivo o negativo. Cada una de las ubicaciones en la memoria de Simpletron puede contener una instrucción, un valor de datos utilizado por un programa o un área no utilizada (y por lo tanto indefinida) de memoria. Los primeros dos dígitos de cada instrucción LMS son el *código de operación* que especifica la operación a realizar. Los códigos de operación de LMS se sintetizan en la figura 8.19.

Código de operación	Significado
Operaciones de entrada/salida:	
`const int lee = 10;`	Lee una palabra desde el teclado y la introduce en una ubicación específica de memoria.
`const int escribe = 11;`	Escribe una palabra de una ubicación específica de memoria y la imprime en la pantalla.
Operaciones de carga/almacenamiento:	
`const int carga = 20;`	Carga una palabra de una ubicación específica de memoria y la coloca en el acumulador.
`final int almacena = 21;`	Almacena una palabra del acumulador dentro de una ubicación específica de memoria.
Operaciones aritméticas:	
`const int suma = 30;`	Suma una palabra de una ubicación específica de memoria a la palabra en el acumulador (deja el resultado en el acumulador).
`const int resta = 31;`	Resta una palabra de una ubicación específica de memoria a la palabra en el acumulador (deja el resultado en el acumulador).
`final int divide = 32;`	Divide una palabra de una ubicación específica de memoria entre la palabra en el acumulador (deja el resultado en el acumulador).
`const int multiplica = 33;`	Multiplica una palabra de una ubicación específica de memoria por la palabra en el acumulador (deja el resultado en el acumulador).
Operaciones de transferencia de control:	
`final int bifurca = 40;`	Bifurca hacia una ubicación específica de memoria.
`final int bifurcaneg = 41;`	Bifurca hacia una ubicación específica de memoria si el acumulador es negativo.
`final int bifurcacero = 42;`	Bifurca hacia una ubicación específica de memoria si el acumulador es cero.
`const int alto = 43;`	Alto. El programa completó su tarea.

Fig. 8.19 | Códigos de operación del Lenguaje máquina Simpletron (LMS).

Los últimos dos dígitos de una instrucción LMS son el *operando* (la dirección de la ubicación en memoria que contiene la palabra a la cual se aplica la operación).

Ahora consideremos dos programas simples en LMS. El primero (figura 8.20) lee dos números del teclado, calcula e imprime su suma. La instrucción `+1007` lee el primer número del teclado y lo coloca en la ubicación `07` (que se inicializó con cero). La instrucción `+1008` lee el siguiente número y lo coloca en la ubicación `08`. La instrucción *carga*, `+2007`, coloca (copia) el primer número en el acumulador y la instrucción *suma*, `+3008`, suma el segundo número al número en el acumulador. *Todas las instrucciones LMS aritméticas dejan sus resultados en el acumulador*. La instrucción *almacena*, `+2109`, coloca (copia) el resultado de vuelta en la ubicación de memoria `09`. Después la instrucción *escribe*, `+1109`, toma el número y lo imprime (como un número decimal de cuatro dígitos con signo). La instrucción *alto*, `+4300`, termina la ejecución.

Ubicación	Número	Instrucción
00	+1007	(Lee A)
01	+1008	(Lee B)
02	+2007	(Carga A)
03	+3008	(Suma B)
04	+2109	(Almacena C)
05	+1109	(Escribe C)
06	+4300	(Alto)
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Resultado C)

Fig. 8.20 | Ejemplo 1 en LMS.

El programa en LMS de la figura 8.21 lee dos números desde el teclado, para luego determinar e imprimir el valor más grande. Observe el uso de la instrucción `+4107` como una transferencia de control condicional, en forma muy similar a la instrucción `if` de C++.

Ubicación	Número	Instrucción
00	+1009	(Lee A)
01	+1010	(Lee B)
02	+2009	(Carga A)
03	+3110	(Resta B)
04	+4107	(Bifurcación negativa a 07)
05	+1109	(Escribe A)
06	+4300	(Alto)
07	+1110	(Escribe B)
08	+4300	(Alto)
09	+0000	(Variable A)
10	+0000	(Variable B)

Fig. 8.21 | Ejemplo 2 de LMS.

Ahora escriba programas en LMS para realizar cada una de las siguientes tareas:

a) Usar un ciclo controlado por centinela para leer números positivos, calcular e imprimir la suma. Terminar la entrada cuando se introduzca un número negativo.

b) Usar un ciclo controlado por contador para leer siete números, algunos positivos y otros negativos, y calcular e imprimir su promedio.
c) Leer una serie de números, determinar e imprimir el número más grande. El primer número leído indica cuántos números deben procesarse.

8.16 ***(Un simulador de computadora)*** Tal vez a primera instancia parezca extravagante, pero en este problema usted va a crear su propia computadora. No, no va a soldar componentes, sino que utilizará la poderosa técnica de la *simulación basada en software* para crear un *modelo de software* de Simpletron. Su simulador Simpletron convertirá la computadora que usted utiliza en Simpletron, y será capaz de ejecutar, probar y depurar los programas LMS que escribió en el ejercicio 8.15.

Cuando ejecute su simulador Simpletron, debe empezar mostrando lo siguiente:

```
*** Bienvenido a Simpletron! ***

*** Por favor, introduzca en su programa una instruccion    ***
*** (o palabra de datos) a la vez . Yo le mostrare el       ***
*** numero de ubicacion y un signo de interrogacion (?).    ***
*** Entonces usted escribira la palabra para esa ubicacion. ***
*** Escriba el valor centinela -99999 para dejar de         ***
*** introducir su programa. ***
```

Su programa debe simular la memoria del Simpletron con un arreglo de un solo subíndice llamado `memoria`, que cuente con 100 elementos. Ahora suponga que el simulador se está ejecutando y examinaremos el diálogo a medida que introduzcamos el programa del segundo ejemplo del ejercicio 8.15:

```
00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999

*** Se completo la carga del programa ***
*** Empieza la ejecucion del programa ***
```

Los números a la derecha de cada `?` en el diálogo anterior representan las instrucciones del programa de LMS introducidas por el usuario.

Ahora el programa en LMS se ha colocado (o cargado) en el arreglo `memoria`. A continuación, Simpletron debe ejecutar el programa en LMS. La ejecución comienza con la instrucción en la ubicación `00` y, como en C++, continúa secuencialmente a menos que se lleve a otra parte del programa mediante una transferencia de control.

Use la variable `acumulador` para representar el registro acumulador. Use la variable `contadorInstrucciones` para llevar el registro de la ubicación en memoria que contiene la instrucción que se está ejecutando. Use la variable `codigoDeOperacion` para indicar la operación que se esté realizando actualmente (es decir, los dos dígitos a la izquierda en la palabra de instrucción). Use la variable `operando` para indicar la ubicación de memoria en la que va a operar la instrucción actual. Por lo tanto, `operando` está compuesta por los dos dígitos más a la derecha de la instrucción que se esté ejecutando en esos momentos. No ejecute las instrucciones directamente desde la memoria. En vez de eso, transfiera la siguiente instrucción a ejecutar desde la memoria hasta una variable llamada `registroDeInstruccion`. Luego "recoja" los dos dígitos a la izquierda y colóquelos en `codigoDeOperacion`, después "recoja" los dos dígitos a la derecha y colóquelos en `operando`. Cuando Simpletron comience con la ejecución, todos los registros especiales se deben inicializar con cero.

Ahora vamos a "dar un paseo" por la ejecución de la primera instrucción LMS, `+1009` en la ubicación de memoria `00`. A este procedimiento se le conoce como *ciclo de ejecución de una instrucción*.

El `contadorInstrucciones` nos indica la ubicación de la siguiente instrucción a ejecutar. Nosotros *obtenemos* el contenido de esa ubicación de `memoria`, utilizando la siguiente instrucción de C++:

```
registroDeInstruccion = memoria[ contadorInstrucciones ];
```

El código de operación y el operando se extraen del registro de instrucción, mediante las instrucciones

```
codigoDeOperacion = registroDeInstruccion / 100;
operando = registroDeInstruccion % 100;
```

Ahora, Simpletron debe determinar que el código de operación es en realidad un *lee* (en comparación con un *escribe*, *carga*, etcétera). Una instrucción `switch` establece la diferencia entre las 12 operaciones de LMS. En la instrucción `switch` se simula el comportamiento de varias instrucciones LMS, como se muestra en la figura 8.22 (dejaremos las otras a usted).

lee:	`cin >> memoria[ operando ];`
carga:	`acumulador = memoria[ operando ];`
suma:	`acumulador += memoria[ operando ];`
bifurca:	En breve hablaremos sobre las instrucciones de bifurcación.
alto:	Esta instrucción imprime el mensaje `*** Termino la ejecucion de Simpletron ***`

Fig. 8.22 | Comportamiento de las instrucciones de LMS.

La instrucción *alto* también hace que Simpletron imprima el nombre y contenido de cada registro, así como el contenido completo de la memoria. Por lo general, a este tipo de impresión se le denomina *vaciado de memoria y registro*. Para ayudarlo a programar su método de vaciado, en la figura 8.23 contiene un formato de vaciado de muestra. Observe que un vaciado, después de la ejecución de un programa de Simpletron, muestra los valores actuales de las instrucciones y los valores de los datos al momento en que se terminó la ejecución. Para dar formato a los números con su signo como se muestra en el vaciado, use el manipulador de flujo **showpos**. Para deshabilitar la visualización del signo, use el manipulador de flujo **noshowpos**. Para los números que tienen menos de cuatro dígitos, se puede dar formato a éstos con ceros a la izquierda entre el signo y el valor, usando la siguiente instrucción antes de imprimir el valor:

```
cout << setfill( '0' ) << internal;
```

```
REGISTROS:
acumulador              +0000
contadorInstrucciones     00
registroDeInstruccion   +0000
codigoDeOperacion         00
operando                  00

MEMORIA:
       0     1     2     3     4     5     6     7     8     9
 0 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
10 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
20 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
30 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
40 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
50 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
60 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
70 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
80 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
90 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
```

Fig. 8.23 | Ejemplo de un vaciado de registro y memoria.

El manipulador de flujo parametrizado `setfill` (del encabezado `<iomanip>`) especifica el carácter de relleno que debe aparecer entre el signo y el valor, cuando un número se muestra con una anchura de campo de cinco caracteres, pero no tiene cuatro dígitos. (Se reserva una posición en la anchura de campo para el signo). El manipulador de flujo `internal` indica que los caracteres de relleno deben aparecer entre el signo y el valor numérico.

Procedamos ahora con la ejecución de la primera instrucción de nuestro programa, `+1009` en la ubicación `00`. Como lo hemos indicado, la instrucción `switch` simula esta tarea ejecutando la siguiente instrucción de C++:

```
cin >> memoria[ operando ];
```

Se debe mostrar un signo de interrogación (`?`) en la pantalla, antes de que se ejecute la instrucción `cin` para pedir la entrada al usuario. Simpletron espera a que el usuario introduzca un valor y oprima la clave *Intro*. Después, el valor se lee en la ubicación `09`.

En este punto se ha completado la simulación de la primera instrucción. Todo lo que resta es preparar a Simpletron para que ejecute la siguiente instrucción. Como la instrucción que acaba de ejecutarse no es una transferencia de control, sólo necesitamos incrementar el registro contador de instrucciones de la siguiente manera:

```
++contadorInstrucciones;
```

Esta acción completa la ejecución simulada de la primera instrucción. Todo el proceso (es decir, el ciclo de ejecución de una instrucción) empieza de nuevo, con la búsqueda de la siguiente instrucción a ejecutar.

Ahora veremos cómo se simulan las instrucciones de bifurcación (las transferencias de control). Todo lo que necesitamos hacer es ajustar el valor en el `contadorInstrucciones` de manera apropiada. Por lo tanto, la instrucción de bifurcación incondicional (`40`) se simula dentro de la instrucción `switch` como

```
contadorInstrucciones = operando;
```

La instrucción condicional "bifurcar si el acumulador es cero" se simula como

```
if ( acumulador == 0 )
   contadorInstrucciones = operando;
```

En este punto, usted debe implementar su simulador Simpletron y ejecutar cada uno de los programas que escribió en el ejercicio 8.15. Las variables que representan la memoria y los registros del simulador Simpletron deben definirse en `main` y pasarse a otras funciones por valor o por referencia, según sea apropiado.

Su simulador debe comprobar diversos tipos de errores. Por ejemplo, durante la fase de carga del programa, cada número que el usuario escribe en la `memoria` de Simpletron debe encontrarse dentro del rango de `-9999` a `+9999`. Su simulador debe usar un ciclo `while` para probar que cada número introducido se encuentre dentro de este rango y, en caso contrario, seguir pidiendo al usuario que vuelva a introducir el número hasta que introduzca un número correcto.

Durante la fase de ejecución, su simulador debe comprobar varios errores graves, como los intentos de dividir entre cero, intentos de ejecutar códigos de operación inválidos, desbordamientos del acumulador (es decir, las operaciones aritméticas que den como resultado valores mayores que `+9999` o menores que `-9999`) y demás. Dichos errores graves se conocen como **errores fatales**. Al detectar un error fatal, su simulador deberá imprimir un mensaje de error tal como

```
*** Intento de dividir entre cero ***
*** La ejecucion de Simpletron se termino en forma anormal ***
```

y deberá imprimir un vaciado de registro y memoria completo en el formato que vimos anteriormente. Este análisis ayudará al usuario a localizar el error en el programa.

8.17 ***(Proyecto: modificaciones al simulador Simpletron)*** En el ejercicio 8.16 usted escribió una simulación de software de una computadora que ejecuta programas escritos en el Lenguaje máquina Simpletron (LMS). En este ejercicio proponemos varias modificaciones y mejoras al simulador Simpletron. En los ejercicios 18.31 a 18.35 proponemos la creación de un compilador que convierta los programas escritos en un lenguaje de programación de alto nivel (una variación de BASIC) a LMS. Algunas de las siguientes modificaciones y mejoras pueden requerirse para ejecutar los programas producidos por el compilador. [*Nota:* algunas modificaciones pueden estar en conflicto con otras, y por lo tanto deberán realizarse por separado].

a) Extienda la memoria del simulador Simpletron, de manera que contenga 1000 ubicaciones de memoria para permitir a Simpletron manejar programas más grandes.
b) Permita al simulador realizar cálculos de módulo. Esta modificación requiere de una instrucción adicional en lenguaje máquina Simpletron.
c) Permita al simulador realizar cálculos de exponenciación. Esta modificación requiere una instrucción adicional en lenguaje máquina Simpletron.
d) Modifique el simulador para que pueda utilizar valores hexadecimales, en vez de valores enteros para representar instrucciones en lenguaje máquina Simpletron.
e) Modifique el simulador para permitir la impresión de una nueva línea. Esta modificación requiere una instrucción adicional en lenguaje máquina Simpletron.
f) Modifique el simulador para procesar valores de punto flotante además de valores enteros.
g) Modifique el simulador para manejar la introducción de cadenas. [*Sugerencia:* cada palabra de Simpletron puede dividirse en dos grupos, cada una de las cuales guarda un entero de dos dígitos. Cada entero de dos dígitos representa el equivalente decimal de código ASCII de un carácter. Agregue una instrucción de lenguaje máquina que reciba como entrada una cadena y la almacene, empezando en una ubicación de memoria específica de Simpletron. La primera mitad de la palabra en esa ubicación será una cuenta del número de caracteres en la cadena (es decir, la longitud de la cadena). Cada media palabra subsiguiente contiene un carácter ASCII, expresado como dos dígitos decimales. La instrucción en lenguaje máquina convierte cada carácter en su equivalente ASCII y lo asigna a una media palabra].
h) Modifique el simulador para manejar la impresión de cadenas almacenadas en el formato de la parte (g). [*Sugerencia:* agregue una instrucción en lenguaje máquina que imprima una cadena, empezando en cierta ubicación de memoria de Simpletron. La primera mitad de la palabra en esa ubicación es una cuenta del número de caracteres en la cadena (es decir, la longitud de la misma). Cada media palabra subsiguiente contiene un carácter ASCII expresado como dos dígitos decimales. La instrucción en lenguaje máquina comprueba la longitud e imprime la cadena, traduciendo cada número de dos dígitos en su carácter equivalente].
i) Modifique el simulador para incluir la instrucción `LMS_DEPURA` que imprima un vaciado de memoria después de que se ejecute cada instrucción. Proporcione a `LMS_DEPURA` un código de operación de `44`. La palabra `+4401` debe activar el modo de depuración, y `+4400` debe desactivarlo.

Clases, un análisis más detallado: lanzar excepciones

9

Mi objeto, en todo sublime,
lograré con el tiempo.
—W. S. Gilbert

¿Es éste un mundo en el cual
se deben ocultar las virtudes?
—William Shakespeare

No tengas amigos que no sean
iguales a ti.
—Confucio

Objetivos

En este capítulo aprenderá a:

- Usar una guardia de inclusión.
- Acceder a los miembros de una clase a través del nombre de un objeto, una referencia a un objeto o un apuntador a un objeto.
- Usar los destructores para realizar "tareas de mantenimiento de terminación".
- Descubrir el orden de las llamadas de los constructores y destructores.
- Conocer los peligros de devolver una referencia a datos `private`.
- Asignar los miembros de datos de un objeto a los de otro objeto.
- Crear objetos compuestos de otros objetos.
- Usar las funciones y las clases `friend`.
- Usar el apuntador `this` en una función miembro para acceder a un miembro no `static` de la clase.
- Usar los miembros de datos y las funciones miembro `static`.

Plan general

9.1 Introducción

En este capítulo analizaremos las clases de forma más detallada. Usaremos un caso de estudio integrado con la clase `Tiempo` además de otros ejemplos para demostrar varias herramientas de construcción de clases. Empezaremos con una clase llamada `Tiempo`, que repasa varias de las características presentadas en los capítulos anteriores. El ejemplo también demuestra el uso de una *guardia de inclusión* en los encabezados, para evitar que el código en el encabezado se incluya en el mismo archivo de código fuente más de una vez.

También demostraremos cómo el código cliente puede acceder a los miembros `public` de una clase a través del nombre de un objeto, una referencia a un objeto o un apuntador a un objeto. Como veremos más adelante, los nombres de objetos y las referencias se pueden usar con el operador punto (`.`) de selección de miembros para acceder a un miembro `public`, y los apuntadores se pueden usar con el operador flecha (`->`) de selección de miembros.

Hablaremos sobre las funciones de acceso que pueden leer o escribir en los miembros de datos de un objeto. Un uso común de las funciones de acceso es evaluar la veracidad o falsedad de las condiciones; dichas funciones se conocen como *funciones predicado*. También demostraremos la noción de una *función utilitaria* (también conocida como *función auxiliar*): una función miembro `private` que soporta la operación de las funciones miembro `public` de la clase, pero *no* está diseñada para que los clientes de la clase la utilicen.

Demostraremos cómo pasar argumentos a los constructores y cómo se pueden usar los argumentos predeterminados en un constructor, para permitir que el código cliente inicialice objetos mediante el uso de una variedad de argumentos. Después, hablaremos sobre una función miembro especial llamada *destructor*, la cual forma parte de toda clase y se utiliza para realizar "tareas de mantenimiento de terminación" en un objeto antes de destruirlo. Luego demostraremos el *orden* en el que se hacen las llamadas a los constructores y destructores.

Mostraremos que al devolver una referencia o apuntador a datos `private` *se quebranta el encapsulamiento de una clase*, lo cual permite que el código cliente acceda directamente a los datos de un objeto. Usaremos la asignación predeterminada a nivel de miembros, para asignar un objeto de una clase a otro objeto de la misma clase.

Usaremos objetos `const` y funciones miembro `const` para evitar modificaciones en los objetos y hacer valer el principio del menor privilegio. Hablaremos sobre la *composición* (una forma de reutilización en donde una clase puede tener objetos de otras clases como miembros. A continuación usaremos la *amistad* (*Friendship*) para especificar que una función no miembro también puede acceder a los

miembros no `public`: una técnica que se utiliza con frecuencia en la sobrecarga de operadores (capítulo 10) por cuestiones de rendimiento. Hablaremos sobre el apuntador `this`, que es un argumento *implícito* en todas las llamadas a las funciones miembro no `static` de una clase, lo que les permite acceder a los miembros de datos y a las funciones miembro no `static` del objeto correcto. Motivaremos la necesidad de miembros de clase `static` y le mostraremos cómo usarlos en sus propias clases.

9.2 Caso de estudio con la clase `Tiempo`

Nuestro primer ejemplo crea y prueba la clase `Tiempo`. Demostraremos un importante concepto de ingeniería de software de C++: el uso de una guardia de inclusión en los encabezados, para evitar que el código del encabezado se incluya en el mismo archivo de código fuente más de una vez. Como una clase sólo se puede definir una vez, al usar estas directivas del preprocesador evitamos los errores por múltiples definiciones.

Definición de la clase `Tiempo`

La definición de la clase (figura 9.1) contiene prototipos (líneas 13 a 16) para las funciones miembro `Tiempo`, `establecerTiempo`, `imprimirUniversal` e `imprimirEstandar`, e incluye los miembros `private unsigned int hora`, `minuto` y `segundo` (líneas 18 a 20). *Solo* se puede acceder a los miembros de datos `private` de la clase `Tiempo` a través de sus funciones miembro. En el capítulo 11 introduciremos un tercer especificador de acceso llamado `protected`, a medida que estudiemos la herencia y el papel que desempeña en la programación orientada a objetos.

Buena práctica de programación 9.1

Por cuestión de claridad y legibilidad, use cada especificador de acceso sólo una vez en una definición de clase. Primero coloque los miembros `public`, en donde sean fáciles de localizar.

Observación de Ingeniería de Software 9.1

Cada miembro de una clase debe tener visibilidad `private`, a menos que pueda demostrarse que el elemento necesita visibilidad `public`. Éste es otro ejemplo del principio del menor privilegio.

```
1  // Fig. 9.1: Tiempo.h
2  // Definición de la clase Tiempo.
3  // Las funciones miembro están definidas en Tiempo.cpp
4
5  // evita múltiples inclusiones del encabezado
6  #ifndef TIEMPO_H
7  #define TIEMPO_H
8
9  // definición de la clase Tiempo
10 class Tiempo
11 {
12 public:
13    Tiempo(); // constructor
14    void establecerTiempo( int, int, int ); // establece hora, minuto y segundo
15    void imprimirUniversal() const; // imprime la hora en formato universal
16    void imprimirEstandar() const; // imprime la hora en formato estándar
17 private:
18    unsigned int hora; // 0 - 23 (formato de reloj de 24 horas)
19    unsigned int minuto; // 0 - 59
20    unsigned int segundo; // 0 - 59
21 }; // fin de la clase Tiempo
22
23 #endif
```

Fig. 9.1 | Definición de la clase `Tiempo`.

En la figura 9.1, la definición de la clase va encerrada en la siguiente **guardia de inclusión** (líneas 6, 7 y 23):

```
// evita múltiples inclusiones del archivo de encabezado
#ifndef TIEMPO_H
#define TIEMPO_H
   ...
#endif
```

Cuando construyamos programas más grandes, las demás definiciones y declaraciones también se colocarán en encabezados. La guardia de inclusión anterior evita que el código entre `#ifndef` (que significa "si no está definido") y `#endif` se incluya, si ya se ha definido el nombre `TIEMPO_H`. Si el encabezado *no* se ha incluido antes en un archivo, el nombre `TIEMPO_H` se define mediante la directiva `#define` y *se* incluyen las instrucciones del archivo de encabezado. Si el encabezado ya se incluyó, `TIEMPO_H` se encuentra *definido* de antemano y el archivo de encabezado *no* se vuelve a incluir. Por lo general, los intentos de incluir un encabezado varias veces (inadvertidamente) ocurren en programas extensos con muchos encabezados, que a su vez pueden incluir otros encabezados.

Tip para prevenir errores 9.1

Use las directivas del preprocesador `#ifndef`, `#define` y `#endif` para formar una guardia de inclusión que evite incluir los encabezados más de una vez en un archivo de código fuente.

Buena práctica de programación 9.2

Por convención, use el nombre del encabezado en mayúsculas, sustituyendo el punto por un guión bajo en las directivas del preprocesador `#ifndef` y `#define` de un encabezado.

Funciones miembro de la clase `Tiempo`

En la figura 9.2, el constructor de `Tiempo` (líneas 11 a 14) inicializa los miembros de datos con 0, el equivalente en tiempo universal de 12 AM. No se pueden almacenar valores inválidos en los miembros de datos de un objeto `Tiempo`, ya que se hace una llamada al constructor cuando se crea el objeto `Tiempo`, y todos los intentos subsiguientes de un cliente por modificar los miembros de datos son escudriñados por la función `establecerTiempo` (que veremos en breve). Por último, es importante observar que podemos definir *constructores sobrecargados* para una clase; en la sección 6.18 estudiamos las funciones sobrecargadas.

```
1  // Fig. 9.2: Tiempo.cpp
2  // definiciones de las funciones miembro para la clase Tiempo.
3  #include <iostream>
4  #include <iomanip>
5  #include <stdexcept> // para la clase de excepción invalid_argument
6  #include "Tiempo.h" // incluye la definición de la clase Tiempo de Tiempo.h
7
8  using namespace std;
9
10 // el constructor de Tiempo inicializa cada miembro de datos con cero.
11 Tiempo::Tiempo()
12    : hora( 0 ), minuto( 0 ), segundo( 0 )
13 {
14 } // fin del constructor de Tiempo
```

Fig. 9.2 | Definiciones de las funciones miembro de la clase `Tiempo` (parte 1 de 2).

```
15
16 // establece el nuevo valor de Tiempo usando la hora universal
17 void Tiempo::establecerTiempo( int h, int m, int s )
18 {
19    // validando hora, minuto y segundo
20    if ( ( h >= 0 && h < 24 ) && ( m >= 0 && m < 60 ) &&
21       ( s >= 0 && s < 60 ) )
22    {
23       hora = h;
24       minuto = m;
25       segundo = s;
26    } // fin de if
27    else
28       throw invalid_argument(
29          "hora, minuto y/o segundo estaban fuera de rango" );
30 } // fin de la función establecerTiempo
31
32 // imprime el Tiempo en formato de hora universal (HH:MM:SS)
33 void Tiempo::imprimirUniversal() const
34 {
35    cout << setfill( '0' ) << setw( 2 ) << hora << ":"
36       << setw( 2 ) << minuto << ":" << setw( 2 ) << segundo;
37 } // fin de la función imprimirUniversal
38
39 // imprime el Tiempo en formato de hora estándar (HH:MM:SS AM o PM)
40 void Tiempo::imprimirEstandar() const
41 {
42    cout << ( ( hora == 0 || hora == 12 ) ? 12 : hora % 12 ) << ":"
43       << setfill( '0' ) << setw( 2 ) << minuto << ":" << setw( 2 )
44       << segundo << ( hora < 12 ? " AM" : " PM" );
45 } // fin de la función imprimirEstandar
```

Fig. 9.2 | Definiciones de las funciones miembro de la clase `Tiempo` (parte 2 de 2).

Antes de C++11, sólo los miembros de datos `static const int` (que vimos en el capítulo 7) podían inicializarse en donde se declaraban en el cuerpo de la clase. Por esta razón, lo usual es que los miembros de datos se inicialicen mediante el constructor de la clase, ya que *no hay inicialización predeterminada para los miembros de datos de tipos fundamentales*. A partir de C++11, ahora es posible usar un *inicializador dentro de la clase* para inicializar cualquier miembro de datos en donde se declare en la definición de la clase.

La función miembro `establecerHora` *de la clase* `Tiempo` *y el lanzamiento de excepciones*

La función `establecerHora` (líneas 17 a 30) es una función `public` que declara tres parámetros `int` y los utiliza para establecer la hora. Las líneas 20 y 21 evalúan cada argumento para determinar si el valor se encuentra en el rango y, de ser así, las líneas 23 a 25 asignan los valores a los miembros de datos `hora`, `minuto` y `segundo`. El valor `hora` debe ser mayor o igual que `0` y menor que `24`, debido a que el formato de hora universal representa las horas como enteros del 0 al 23 (por ejemplo, 1 PM es la hora 13 y 11 PM es la hora 23; medianoche es la hora 0 y mediodía es la hora 12). De manera similar, los valores de `minuto` y `segundo` deben ser mayores o iguales que `0`, y menores que `60`. Para cualquier valor fuera de estos rangos, `establecerHora` **lanza una excepción** (líneas 28 y 29) de tipo **`invalid_argument`** (del encabezado `<stdexcept>`), la cual notifica al código cliente que se recibió un argumento inválido. Como vimos en la sección 7.10, puede usar `try...catch` para atrapar excepciones e intentar recupe-

rarse de ellas, lo cual haremos en la figura 9.3. La **instrucción `throw`** (líneas 28 y 29) crea un nuevo objeto de tipo `invalid_argument`. Los paréntesis que van después del nombre de la clase indican una llamada al constructor `invalid_argument`, el cual nos permite especificar una cadena de mensaje de error personalizada. Después de crear la excepción, la instrucción `throw` termina de inmediato la función `establecerHora` y la excepción se regresa al código que intentó establecer la hora.

La función miembro `imprimirUniversal` de la clase `Tiempo`

La función `imprimirUniversal` (líneas 33 a 37 de la figura 9.2) no recibe argumentos e imprime el tiempo en formato universal, el cual consiste de tres pares de dígitos separados por dos puntos. Si el tiempo es 1:30:07 PM, la función `imprimirUniversal` devuelve `13:30:07`. En la línea 35 se utiliza el manipulador de flujo parametrizado **`setfill`** para especificar el **carácter de relleno** que se muestra cuando se imprime un entero en un campo *más ancho* que el número de dígitos en el valor. Los caracteres de relleno aparecen a la *izquierda* de los dígitos en el número, ya que éste se encuentra *alineado a la derecha* de manera predeterminada—en los valores *alineados a la izquierda*, los caracteres de relleno aparecerían a la derecha—. En este ejemplo, si el valor de `minuto` es 2, se mostrará como 02 ya que el carácter de relleno se estableció en cero (`'0'`). Si el número que se va a imprimir rellena el campo especificado, *no* se mostrará el carácter de relleno. Una vez que se especifica el carácter de relleno mediante `setfill`, se aplica para *todos* los valores subsiguientes que se muestran en campos más amplios que el valor que se va a mostrar (`setfill` es una opción "adherible"). Esto es en contraste con `setw`, que *sólo* se aplica al siguiente valor mostrado (`setw` es una opción "no adherible").

Tip para prevenir errores 9.2

Cada opción adherible (como un carácter de relleno o una precisión de punto flotante) se debe restaurar *a su opción anterior cuando ya no se necesite. Si no se hace esto, se puede producir una salida con formato incorrecto posteriormente en un programa. En el capítulo 13, Entrada/salida de flujos: un análisis más detallado, veremos cómo restablecer el carácter de relleno y la precisión.*

La función miembro `imprimirEstandar` de la clase `Tiempo`

La función `imprimirEstandar` (líneas 40 a 45) no recibe argumentos e imprime la fecha en formato de tiempo estándar, el cual consiste en los valores de `hora`, `minuto` y `segundo` separados por dos puntos, y va seguido de un indicador AM o PM (por ejemplo, `1:27:06 PM`). Al igual que la función `imprimirUniversal`, la función `imprimirEstandar` usa `setfill('0')` para dar formato a `minuto` y `segundo` como valores de dos dígitos con ceros a la izquierda, en caso de ser necesario. En la línea 42 se utiliza el operador condicional (`?:`) para determinar el valor de `hora` a mostrar; si `hora` es `0` o `12` (AM o PM), aparece como 12; en caso contrario, `hora` aparece como un valor de 1 a 11. El operador condicional en la línea 44 determina si se va a mostrar AM o PM.

Definición de funciones miembro fuera de la definición de la clase: alcance de las clases

Aun cuando una función miembro declarada en una definición de clase puede definirse fuera de esa definición de clase (y "enlazarse" a la clase mediante el *operador de resolución de ámbito binario*), todavía está dentro del **alcance de esa clase**; su nombre es conocido sólo para los otros miembros de la clase a los que se hace referencia a través de un objeto de la clase, una referencia a un objeto de la clase, un apuntador a un objeto de la clase o del operador de resolución de ámbito. En breve hablaremos más acerca del alcance de las clases.

Si una función miembro se define en el cuerpo de una clase, se declara de manera implícita como *en línea*. Recuerde que el compilador se reserva el derecho de no poner en línea cualquier función.

Tip de rendimiento 9.1

Al definir una función miembro dentro de la definición de clase, se pone en línea la función miembro (si el compilador opta por hacer esto). Esto puede mejorar el rendimiento.

Observación de Ingeniería de Software 9.2

Sólo las funciones miembro más simples y estables (es decir, aquellas cuyas implementaciones tengan pocas probabilidades de cambiar) deben definirse en el encabezado de la clase.

Comparación entre funciones miembro y funciones globales (también conocidas como funciones libres)

Las funciones miembro `imprimirUniversal` e `imprimirEstandar` no reciben argumentos, ya que estas funciones saben de manera implícita que deben imprimir los miembros de datos del objeto `Tiempo` específico para el cual se invocaron. Esto puede hacer que las llamadas a las funciones miembro sean más concisas que las llamadas a las funciones convencionales en la programación por procedimientos.

Observación de Ingeniería de Software 9.3

Por lo general, el uso de una metodología de programación orientada a objetos puede simplificar las llamadas a las funciones, al reducir el número de parámetros. Este beneficio se deriva del hecho de que al encapsular los miembros de datos y las funciones miembro dentro de una clase, se proporciona a las funciones miembro el derecho de acceder a los miembros de datos.

Observación de Ingeniería de Software 9.4

Por lo general, las funciones miembro son más cortas que las funciones en los programas no orientados a objetos, ya que los datos almacenados en los miembros de datos se han validado idealmente mediante un constructor, o mediante funciones miembro que almacenan nuevos datos. Debido a que los datos ya se encuentran en el objeto, comúnmente las llamadas a funciones miembro no tienen argumentos, o tienen menos argumentos que las llamadas a funciones comunes en los lenguajes no orientados a objetos. Por ende, las llamadas, las definiciones de las funciones y los prototipos de las funciones son más cortos. Esto mejora muchos aspectos del desarrollo de programas.

Tip para prevenir errores 9.3

El hecho de que las llamadas a funciones miembro generalmente no reciben argumentos, o reciben una cantidad mucho menor de argumentos que las llamadas a funciones convencionales en los lenguajes no orientados a objetos, se reduce la probabilidad de pasar los argumentos incorrectos, los tipos incorrectos de argumentos o el número incorrecto de los mismos.

Uso de la clase `Tiempo`

Una vez que se ha definido la clase `Tiempo`, se puede usar como un tipo en las declaraciones como se muestra a continuación:

```
Tiempo puestaDeSol; // objeto de tipo Tiempo
array< Tiempo, 5 > arregloDeTiempos; // arreglo de 5 objetos Tiempo
Tiempo &horaDeComer = puestaDeSol; // referencia a un objeto Tiempo
Tiempo *tiempoPtr = &horaDeComer; // apuntador a un objeto Tiempo
```

La figura 9.3 usa la clase `Tiempo`. En la línea 11 se crea una instancia de un solo objeto de la clase `Tiempo` llamado `t`. Cuando se crea la instancia del objeto, se hace una llamada al constructor de `Tiempo` para inicializar cada miembro de datos `private` con `0`. Después, en las líneas 15 y 17 se imprime el tiempo en los formatos universal y estándar, respectivamente, para confirmar que los miembros se hayan

inicializado en forma apropiada. En la línea 19 se establece un nuevo tiempo, para lo cual se hace una llamada a la función miembro `establecerTiempo`, y en las líneas 23 y 25 se imprime el tiempo de nuevo en ambos formatos.

```
1  // Fig. 9.3: fig09_03.cpp
2  // Programa para probar la clase Tiempo.
3  // NOTA: Este archivo se debe compilar con Tiempo.cpp.
4  #include <iostream>
5  #include <stdexcept> // para la clase de excepción invalid_argument
6  #include "Tiempo.h" // incluye la definición de la clase Tiempo de Tiempo.h
7  using namespace std;
8
9  int main()
10 {
11    Tiempo t; // instancia un objeto t de la clase Tiempo
12
13    // imprime los valores iniciales del objeto Tiempo t
14    cout << "El tiempo universal inicial es ";
15    t.imprimirUniversal(); // 00:00:00
16    cout << "\nEl tiempo universal estandar es ";
17    t.imprimirEstandar(); // 12:00:00 AM
18
19    establecerTiempo( 13, 27, 6 ); // cambia el tiempo
20
21    // imprime los nuevos valores del objeto t Tiempo
22    cout << "\n\nEl tiempo universal despues de establecerTiempo es ";
23    t.imprimirUniversal(); // 13:27:06
24    cout << "\nEl tiempo estandar despues de establecerTiempo es ";
25    t.imprimirEstandar(); // 1:27:06 PM
26
27    // intenta establecer el tiempo con valores inválidos
28    try
29    {
30       t.establecerHora( 99, 99, 99 ); // todos los valores fuera de rango
31    } // fin de try
32    catch ( invalid_argument &e )
33    {
34       cout << "\n\nExcepcion: " << e.what() << endl;
35    } // fin de catch
36
37    // imprime los valores de t después de especificar valores inválidos
38    cout << "\n\nDespues de intentar ajustes invalidos:"
39       << "\nTiempo universal: ";
40    t.imprimirUniversal(); // 13:27:06
41    cout << "\nTiempo estandar: ";
42    t.imprimirEstandar(); // 1:27:06 PM
43    cout << endl;
44 } // fin de main
```

```
El tiempo universal inicial es 00:00:00
El tiempo universal estandar es 12:00:00 AM

El tiempo universal despues de establecerTiempo es 13:27:06
El tiempo estandar despues de establecerTiempo es 1:27:06 PM

Excepcion: hora, minuto y/o segundo estaban fuera de rango
```

Fig. 9.3 | Programa para probar la clase `Tiempo` (parte 1 de 2).

```
Despues de intentar ajustes invalidos:
Tiempo universal: 13:27:06
Tiempo estandar: 1:27:06 PM
```

Fig. 9.3 | Programa para probar la clase `Tiempo` (parte 2 de 2).

Llamar a **`establecerTiempo`** *con valores inválidos*

Para ilustrar que el método `establecerTiempo` valida sus argumentos, en la línea 30 se hace una llamada a `establecerTiempo` con argumentos inválidos de `99` para `hora`, `minuto` y `segundo`. Esta instrucción se coloca en un bloque `try` (líneas 28 a 31) en caso de que `establecerTiempo` lance una excepción `invalid_argument`, debido a que todos los argumentos son inválidos. Cuando esto ocurre, la excepción se atrapa en las líneas 32 a 35 y en la línea 34 se muestra el mensaje de error de la excepción, mediante una llamada a su función miembro `what`. En las líneas 38 a 42 se imprime el tiempo de nuevo en ambos formatos, para confirmar que `establecerTiempo` no modificó el tiempo cuando se suministraron argumentos inválidos.

Un avance acerca de la composición y la herencia

A menudo, las clases no se tienen que crear "desde cero". En vez de ello, pueden *incluir objetos de otras clases como miembros*, o pueden **derivarse** de otras clases que proporcionen atributos y comportamientos que las nuevas clases puedan usar. Dicha reutilización de software puede mejorar de manera considerable la productividad del programador y simplificar el mantenimiento del código. Al proceso de incluir objetos de clases como miembros de otras clases se le llama **composición** (o **agregación**) y se describe en la sección 9.11. Al proceso de derivar nuevas clases a partir de clases existentes se le llama **herencia** y se describe en el capítulo 11.

Tamaño de los objetos

A menudo, las personas con poca experiencia en la programación orientada a objetos suponen que los objetos deben ser bastante grandes, ya que contienen miembros de datos y funciones miembro. En *sentido lógico*, esto es verdad; podemos considerar que los objetos contienen datos y funciones (y nuestra discusión sin duda ha fomentado este punto de vista); sin embargo, *físicamente* esto no es verdad.

Tip de rendimiento 9.2

Los objetos sólo contienen datos, por lo que son mucho menores que si también contuvieran funciones miembro. El compilador crea una copia (sólo) de las funciones miembro, separada de todos los objetos de la clase. Estos objetos comparten *esta única copia. Desde luego que cada objeto de la clase necesita su propia copia de los datos de la clase, ya que éstos pueden variar entre un objeto y otro. El código de la función no se puede modificar y, por ende, puede compartirse entre todos los objetos de una clase.*

9.3 Alcance de las clases y acceso a los miembros de una clase

Los miembros de datos de una clase y las funciones miembro pertenecen al alcance de esa clase. Las funciones que no son miembro se definen en *alcance de espacio de nombres global*, de manera predeterminada (en la sección 23.4 hablaremos sobre los espacios de nombres con más detalle).

Dentro del alcance de una clase, los miembros de ésta pueden ser utilizados inmediatamente por todas las funciones miembro de esa clase, y se pueden referenciar por nombre. Fuera del alcance de una clase, los miembros `public` de la clase se referencian a través de uno de los manejadores en un objeto: el *nombre de un objeto*, una *referencia* a un objeto, o un *apuntador* a un objeto. El tipo del objeto, referencia o apuntador especifica la interfaz (es decir, las funciones miembro) accesible para el cliente. [En la sección 9.13 veremos que el compilador inserta un *manejador implícito* en cada referencia a un miembro de datos o función miembro, desde el interior de un objeto].

Alcance de clase y alcance de bloque
Las variables que se declaran en una función miembro tienen *alcance de bloque* y sólo esa función las conoce. Si una función miembro define una variable con el mismo nombre que una variable con alcance de clase, la variable con alcance de clase se *oculta* en la función debido a la variable con alcance de bloque. Para acceder a dicha variable oculta, hay que colocar antes de su nombre el nombre de la clase, seguido del operador de resolución de ámbito (`::`). Se puede acceder a las variables globales ocultas con el operador de resolución de ámbito unario (vea el capítulo 6).

Operadores de selección de miembro punto (.) y flecha (->)
Antes del operador punto (`.`) de selección de miembro se coloca el nombre de un objeto o una referencia a un objeto para acceder a los miembros de ese objeto. Antes del **operador flecha (->) de selección de miembros** se coloca un apuntador a un objeto, para acceder a los miembros de ese objeto.

Acceso a los miembros de clases **`public`** *a través de objetos, referencias y apuntadores*
Considere una clase llamada `Cuenta` con una función miembro `public` llamada `establecerSaldo`. Dadas las siguientes declaraciones:

```
Cuenta cuenta; // un objeto Cuenta

// arefCuenta se refiere a un objeto Cuenta
Cuenta &refCuenta = cuenta;

// ptrCuenta apunta a un objeto Cuenta
Cuenta *ptrCuenta = &cuenta;
```

Puede invocar a la función miembro `establecerSaldo` mediante los operadores de selección de miembros punto (`.`) y flecha (`->`) como se muestra a continuación:

```
// llama a establecerSaldo mediante el objeto Cuenta
cuenta.establecerSaldo( 123.45 );

// llama a establecerSaldo mediante una referencia al objeto Cuenta
refCuenta.establecerSaldo( 123.45 );

// llama a establecerSaldo mediante un apuntador al objeto Cuenta
ptrCuenta->establecerSaldo( 123.45 );
```

9.4 Funciones de acceso y funciones utilitarias

Funciones de acceso
Las **funciones de acceso** pueden leer o mostrar datos. Otro uso común para las funciones de acceso es para evaluar la veracidad o falsedad de las condiciones; a menudo, a dichas funciones se les conoce como **funciones predicado**. Un ejemplo de una función predicado sería una función `estaVacio` para cualquier clase contenedora (una clase capaz de contener muchos objetos), tal como `vector`. Un programa podría evaluar a `estaVacio` antes de tratar de leer otro elemento del objeto contenedor. Una función predicado `estaLleno` podría evaluar un objeto de una clase contenedora para determinar si no tiene espacio adicional. Dos funciones predicado útiles para nuestra clase `Tiempo` podrían ser `esAM` y `esPM`.

Funciones utilitarias
Una **función utilitaria** (también conocida como **función auxiliar**) es una función miembro `private` que apoya la operación de las otras funciones miembro de una clase. Las funciones utilitarias se declaran `private` debido a que los clientes de la clase no deben usarlas. Un uso común de una función utilitaria sería colocar cierto código común en una función, que de lo contrario se duplicaría en otras funciones miembro.

9.5 Caso de estudio de la clase Tiempo: constructores con argumentos predeterminados

El programa de las figuras 9.4 a 9.6 mejora la clase `Tiempo` para demostrar cómo se pasan los argumentos implícitamente a un constructor. El constructor definido en la figura 9.2 inicializa `hora`, `minuto` y `segundo` con `0` (es decir, medianoche en el formato de tiempo universal). Al igual que otras funciones, los constructores pueden especificar *argumentos predeterminados*. En la línea 13 de la figura 9.4 se declara el constructor de `Tiempo` para incluir argumentos predeterminados, especificando un valor predeterminado de cero para cada argumento que se pasa al constructor. El constructor se declara `explicit` debido a que puede llamarse con un argumento. En la sección 10.13 hablaremos sobre los constructores `explicit` con detalle.

```
1  // Fig. 9.4: Tiempo.h
2  // Clase Tiempo que contiene un constructor con argumentos predeterminados.
3  // Las funciones miembro se definen en Tiempo.cpp.
4
5  // evita múltiples inclusiones del archivo de encabezado
6  #ifndef TIEMPO_H
7  #define TIEMPO_H
8
9  // Definición de la clase Tiempo
10 class Tiempo
11 {
12 public:
13    explicit Tiempo( int = 0, int = 0, int = 0 ); // constructor predeterminado
14
15    // funciones "establecer"
16    void establecerTiempo( int, int, int ); // establece hora, minuto, segundo
17    void establecerHora( int ); // establece la hora (después de la validación)
18    void establecerMinuto( int ); // establece el minuto (después de la validación)
19    void establecerSegundo( int ); // establece el segundo (después de la validación)
20
21    // funciones "obtener"
22    unsigned int obtenerHora() const; // devuelve la hora
23    unsigned int obtenerMinuto() const; // devuelve el minuto
24    unsigned int obtenerSegundo() const; // devuelve el segundo
25
26    void imprimirUniversal() const; // imprime el tiempo en formato universal
27    void imprimirEstandar() const; // imprime el tiempo en formato estándar
28 private:
29    unsigned int hora; // 0 - 23 (formato de reloj de 24 horas)
30    unsigned int minuto; // 0 - 59
31    unsigned int segundo; // 0 - 59
32 }; // fin de la clase Tiempo
33
34 #endif
```

Fig. 9.4 | Clase `Tiempo` que contiene un constructor con argumentos predeterminados.

En la figura 9.5, en las líneas 10 a 13 se define la nueva versión del constructor de `Tiempo` que recibe valores para los parámetros `hora`, `minuto` y `segundo` que se utilizarán para inicializar los miembros de datos `private` `hora`, `minuto` y `segundo`, respectivamente. Los argumentos predeterminados para el constructor aseguran que, aun si no se proporcionan valores en la llamada a un constructor, éste de todas formas inicializa los miembros de datos. *Un constructor que utiliza valores predeterminados para todos sus argumentos también es un constructor predeterminado; es decir, un constructor que se*

puede invocar sin argumentos. Puede haber como máximo un constructor predeterminado por clase. La versión de la clase `Tiempo` en este ejemplo proporciona funciones establecer y obtener para cada miembro de datos. Ahora el constructor de `Tiempo` llama a `establecerTiempo`, que a su vez llama a las funciones `establecerHora`, `establecerMinuto` y `establecerSegundo` para validar y asignar valores a los miembros de datos.

Observación de Ingeniería de Software 9.5

Cualquier cambio en los valores de los argumentos predeterminados de una función requiere que se recompile el código cliente (para asegurar que el programa aún funcione correctamente).

```
1  // Fig. 9.5: Tiempo.cpp
2  // Definiciones de las funciones miembro para la clase Tiempo.
3  #include <iostream>
4  #include <iomanip>
5  #include <stdexcept>
6  #include "Tiempo.h" // incluye la definición de la clase Tiempo de Tiempo.h
7  using namespace std;
8
9  // el constructor de Tiempo inicializa cada miembro de datos
10 Tiempo::Tiempo( int hora, int minuto, int segundo )
11 {
12    establecerTiempo( hora, minuto, segundo ); // valida y establece el tiempo
13 } // fin del constructor de Tiempo
14
15 // establece nuevo valor de Tiempo usando el formato universal
16 void Tiempo::establecerTiempo( int h, int m, int s )
17 {
18    establecerHora( h ); // establece el campo private hora
19    establecerMinuto( m ); // establece el campo private minuto
20    establecerSegundo( s ); // establece el campo private segundo
21 } // fin de la función establecerTiempo
22
23 // establece el valor de hora
24 void Tiempo::establecerHora( int h )
25 {
26    if ( h >= 0 && h < 24 )
27       hora = h;
28    else
29       throw invalid_argument( "hora debe estar entre 0 y 23" );
30 } // fin de la función establecerHora
31
32 // establece el valor de minuto
33 void Tiempo::establecerMinuto( int m )
34 {
35    if ( m >= 0 && m < 60 )
36       minuto = m;
37    else
38       throw invalid_argument( "minuto debe estar entre 0 y 59" );
39 } // fin de la función establecerMinuto
```

Fig. 9.5 | Definiciones de las funciones miembro de la clase `Tiempo` (parte 1 de 2).

```
40
41  // establece el valor de segundo
42  void Tiempo::establecerSegundo( int s )
43  {
44     if ( s >= 0 && s < 60 )
45        segundo = s;
46     else
47        throw invalid_argument( "segundo debe estar entre 0 y 59" );
48  } // fin de la función establecerSegundo
49
50  // devuelve el valor de la hora
51  unsigned int Tiempo::obtenerHora() const
52  {
53     return hora;
54  } // fin de la función obtenerHora
55
56  // devuelve el valor del minuto
57  unsigned Tiempo::obtenerMinuto() const
58  {
59     return minuto;
60  } // fin de la función obtenerMinuto
61
62  // devuelve el valor del segundo
63  unsigned Tiempo::obtenerSegundo() const
64  {
65     return segundo;
66  } // fin de la función obtenerSegundo
67
68  // imprime el Tiempo en formato universal (HH:MM:SS)
69  void Tiempo::imprimirUniversal() const
70  {
71     cout << setfill( '0' ) << setw( 2 ) << obtenerHora() << ":"
72        << setw( 2 ) << obtenerMinuto() << ":" << setw( 2 ) << obtenerSegundo();
73  } // fin de la función imprimirUniversal
74
75  // imprime el Tiempo en formato estándar (HH:MM:SS AM o PM)
76  void Tiempo::imprimirEstandar() const
77  {
78     cout << ( ( obtenerHora() == 0 || obtenerHora() == 12 )
       ? 12 : obtenerHora() % 12 )
79        << ":" << setfill( '0' ) << setw( 2 ) << obtenerMinuto()
80        << ":" << setw( 2 ) << obtenerSegundo() << ( hora < 12 ? " AM" : " PM" );
81  } // fin de la función imprimirEstandar
```

Fig. 9.5 | Definiciones de las funciones miembro de la clase `Tiempo` (parte 2 de 2).

En la figura 9.5, en la línea 12 del constructor se hace una llamada a la función miembro `establecerTiempo` con los valores que se pasan al constructor (o a los valores predeterminados). La función `establecerTiempo` llama a `establecerHora` para asegurar que el valor suministrado para `hora` esté en el rango de 0 a 23, y después llama a `establecerMinuto` y `establecerSegundo` para asegurar que los valores para `minuto` y `segundo` se encuentren en el rango de 0 a 59. Las funciones `establecerHora` (líneas 24 a 30), `establecerMinuto` (líneas 33 39) y `establecerSegundo` (líneas 42 a 48) lanzan una excepción si se recibe un argumento fuera de rango.

La función `main` en la figura 9.6 inicializa cinco objetos `Tiempo`: uno con los tres argumentos predeterminados en la llamada implícita al constructor (línea 10), uno con un argumento especificado (línea 11), uno con dos argumentos especificados (línea 12), uno con tres argumentos especificados (línea 13) y uno con tres argumentos inválidos especificados (línea 38). Así, el programa muestra cada objeto en formatos de tiempo universal y tiempo estándar. Para el objeto `Tiempo` llamado `t5` (línea 38), el programa muestra un mensaje de error debido a que los argumentos del constructor están fuera de rango.

```
1  // Fig. 9.6: fig09_06.cpp
2  // Constructor con argumentos predeterminados.
3  #include <iostream>
4  #include <stdexcept>
5  #include "Tiempo.h" // incluye la definición de la clase Tiempo de Tiempo.h
6  using namespace std;
7
8  int main()
9  {
10    Tiempo t1; // valor predeterminado en todos los argumentos
11    Tiempo t2( 2 ); // se especifica hora; valores predeterminados para minuto y
                      segundo
12    Tiempo t3( 21, 34 ); // se especifican hora y minuto; valor predeterminado
                           para segundo
13    Tiempo t4( 12, 25, 42 ); // se especifican hora, minuto y segundo
14
15    cout << "Se construyo con:\n\nt1: todos los argumentos predeterminados\n  ";
16    t1.imprimirUniversal(); // 00:00:00
17    cout << "\n  ";
18    t1.imprimirEstandar(); // 12:00:00 AM
19
20    cout << "\n\nt2: se especifico hora; minuto y segundo predeterminados\n  ";
21    t2.imprimirUniversal(); // 02:00:00
22    cout << "\n  ";
23    t2.imprimirEstandar(); // 2:00:00 AM
24
25    cout << "\n\nt3: se especificaron hora y minuto; segundo predeterminado\n  ";
26    t3.imprimirUniversal(); // 21:34:00
27    cout << "\n  ";
28    t3.imprimirEstandar(); // 9:34:00 PM
29
30    cout << "\n\nt4: se especificaron hora, minuto y segundo\n  ";
31    t4.imprimirUniversal(); // 12:25:42
32    cout << "\n  ";
33    t4.imprimirEstandar(); // 12:25:42 PM
34
35    // intenta inicializar t6 con valores inválidos
36    try
37    {
38       Tiempo t5( 27, 74, 99 ); // se especifican valores incorrectos
39    } // fin de try
40    catch ( invalid_argument &e )
41    {
42       cerr << "\n\nExcepcion al inicializar t5: " << e.what() << endl;
43    } // fin de catch
44 } // fin de main
```

Fig. 9.6 | Constructor con argumentos predeterminados (parte 1 de 2).

```
Se construyo con:

t1: todos los argumentos predeterminados
  00:00:00
  12:00:00 AM

t2: se especifico hora; minuto y segundo predeterminados
  02:00:00
  2:00:00 AM

t3: se especificaron hora y minuto; segundo predeterminado
  21:34:00
  9:34:00 PM

t4: se especificaron hora, minuto y segundo
  12:25:42
  12:25:42 PM

Excepcion al inicializar t5: hora debe estar entre 0 y 23
```

Fig. 9.6 | Constructor con argumentos predeterminados (parte 2 de 2).

Observaciones en relación con el constructor y las funciones Establecer *y* Obtener *de la clase* `Tiempo`

Las funciones *establecer* y *obtener* de `Tiempo` se llaman a través del cuerpo de la clase. En especial, la función `establecerTiempo` (líneas 16 a 21 de la figura 9.5) llama a las funciones `establecerHora`, `establecerMinuto` y `establecerSegundo`, y las funciones `imprimirUniversal` e `imprimirEstandar` llaman a las funciones `establecerHora`, `establecerMinuto` y `establecerSegundo` en las líneas 71 a 72 y 78 a 80. En cada caso, estas funciones podrían haber accedido a los datos `private` de la clase directamente. Sin embargo, considere el modificar la representación del tiempo, de tres valores `int` (que requieren 12 bytes de memoria en sistemas con valores `int` de cuatro bytes) a un solo valor `int` que represente el número total de segundos transcurridos desde medianoche (que sólo requiere cuatro bytes de memoria). Si realizáramos dicha modificación, sólo tendrían que cambiar los cuerpos de las funcion es que acceden directamente a los datos `private`; en especial, las funciones individuales *establecer* y *obtener* para `hora`, `minuto` y `segundo`. No habría necesidad de modificar los cuerpos de las funciones `establecerTiempo`, `imprimirUniversal` o `imprimirEstandar`, ya que *no* acceden directamente a los datos. Al diseñar la clase de esta forma se reduce la probabilidad de que se produzcan errores de programación al alterar la implementación de la clase.

De manera similar, el constructor de `Tiempo` podría escribirse de manera que incluya una copia de las instrucciones apropiadas de la función `establecerTiempo`. Esto puede ser ligeramente más eficiente, debido a que se elimina la llamada adicional a `establecerTiempo`. Sin embargo, al duplicar instrucciones en varias funciones o constructores se dificulta más el proceso de modificar la representación interna de datos de la clase. Al hacer que el constructor de `Tiempo` llame a `establecerTiempo` y que a su vez, `establecerTiempo` llame a `establecerHora`, `establecerMinuto` y `establecerSegundo`, nos permite limitar los cambios al código que valida la `hora`, `minuto` o `segundo` con la correspondiente función *establecer*. Esto reduce la probabilidad de errores cuando se altera la implementación de la clase.

Observación de Ingeniería de Software 9.6

Si una función miembro de una clase ya proporciona toda o una parte de la funcionalidad requerida por un constructor (u otra función miembro) de la clase, hay que llamar a esa función miembro desde el constructor (u otra función miembro). Esto simplifica el mantenimiento del código y reduce la probabilidad de un error si se modifica la implementación del código. Como regla general: evite repetir código.

Error común de programación 9.1

Un constructor puede llamar a otras funciones miembro de la clase, como las funciones establecer *u* obtener*, pero debido a que el constructor está inicializando el objeto, los miembros de datos tal vez no se encuentren todavía inicializados. Si se utilizan los miembros de datos antes de que se hayan inicializado en forma apropiada, se pueden producir errores lógicos.*

C++11: uso de inicializadores de listas para llamar a los constructores

En la sección 4.10 vimos que ahora C++ proporciona una sintaxis de inicialización uniforme conocida como inicializadores de listas, los cuales pueden usarse para inicializar cualquier variable. Las líneas 11 a 13 de la figura 9.6 pueden escribirse mediante el uso de inicializadores de listas, como se muestra a continuación:

```
Tiempo t2{ 2 }; // se especifica la hora; valores predeterminados para minuto y
                   segundo
Tiempo t3{ 21, 34 }; // se especifican hora y minuto; valor predeterminado para
                        segundo
Tiempo t4{ 12, 25, 42 }; // se especifican hora, minuto y segundo
```

o

```
Tiempo t2 = { 2 }; // se especifica la hora; valores predeterminados para minuto y
                      segundo
Tiempo t3 = { 21, 34 }; // se especifican hora y minuto; valor predeterminado para
                           segundo
Tiempo t4 = { 12, 25, 42 }; // se especifican hora, minuto y segundo
```

Es preferible la forma sin el signo =.

C++11: constructores sobrecargados y delegación de constructores

En la sección 6.18 vimos cómo sobrecargar funciones. Los constructores de una clase y las funciones miembro también pueden sobrecargarse. Por lo general, los constructores sobrecargados permiten inicializar objetos con distintos tipos y/o números de argumentos. Para sobrecargar un constructor, hay que proporcionar en la definición de la clase un prototipo para cada versión del constructor y proporcionar la definición de un constructor separado por cada versión sobrecargada. Esto también se aplica a las funciones miembro de la clase.

En las figuras 9.4 a 9.6, el constructor de `Tiempo` con tres parámetros tenía un argumento predeterminado para cada parámetro. Podríamos haber definido el constructor en vez de cuatro constructores sobrecargados con los siguientes prototipos:

```
Tiempo(); // valor predeterminado de 0 para hora, minuto y segundo
Tiempo( int ); // inicializa hora; valor predeterminado de 0 para minuto y segundo
Tiempo( int, int ); // inicializa hora y minuto; valor predeterminado de 0 para segundo
Tiempo( int, int, int ); // inicializa hora, minuto y segundo
```

Así como un constructor puede llamar a las otras funciones miembro de una clase para realizar tareas, C++11 ahora permite que los constructores llamen a otros constructores en la misma clase. El constructor que hace la llamada se conoce como **constructor de delegación**: *delega* su trabajo a otro constructor. Esto es útil cuando los constructores sobrecargados tienen código común que podría haberse definido antes en una función utilitaria `private` que todos los constructores pudieran llamar.

Los primeros tres de los cuatro constructores de `Tiempo` declarados anteriormente pueden delegar el trabajo a uno que tenga tres argumentos `int`, pasando 0 como el valor predeterminado para los parámetros adicionales. Para ello, hay que usar un inicializador de miembros con el nombre de la clase, como se indica a continuación:

```
Tiempo::Tiempo()
   : Tiempo( 0, 0, 0 ) // delega a Tiempo( int, int, int )
{
} // fin de constructor sin argumentos

Tiempo::Tiempo( int hora )
   : Tiempo( hora, 0, 0 ) // delega a Tiempo( int, int, int )
{
} // fin de constructor con un argumento
Tiempo::Tiempo( int hora, int minuto )
   : Tiempo( hora, minuto, 0 ) // delega a Tiempo( int, int, int )
{
} // fin de constructor con dos argumentos
```

9.6 Destructores

Un **destructor** es otro tipo de función miembro especial. El nombre del destructor para una clase es el **carácter tilde** (~) seguido del nombre de la clase. Esta convención de nomenclatura tiene un atractivo intuitivo, ya que como veremos en un capítulo posterior, el operador tilde es el operador de complemento a nivel de bits, y en cierto sentido, el destructor es el complemento del constructor. Un destructor no tiene que especificar parámetros o un tipo de valor de retorno.

El destructor de una clase se llama de manera *implícita* cuando se destruye un objeto. Por ejemplo, esto ocurre a medida que un objeto automático se destruye cuando la ejecución del programa sale del alcance en el que se instanció el objeto. *El destructor en sí no libera la memoria del objeto*; realiza **tareas de mantenimiento de terminación** antes de que se reclame la memoria del objeto, de forma que ésta se pueda reutilizar para contener nuevos objetos.

Aun cuando no se han definido destructores para las clases presentadas hasta ahora, *toda clase tiene un destructor*. Si el programador no especifica un destructor de manera *explícita*, el compilador define un "destructor" vacío. [*Nota*: más adelante veremos que un destructor creado de manera *implícita* desempeña, de hecho, operaciones importantes sobre los objetos de tipos de clases que se crean a través de la composición (sección 9.11) y la herencia (capítulo 11)]. En el capítulo 10 crearemos destructores apropiados para las clases cuyos objetos contienen memoria asignada en forma dinámica (por ejemplo, para arreglos y cadenas) o que utilizan otros recursos del sistema (por ejemplo, archivos en disco, que veremos en el capítulo 14). En el capítulo 10 veremos cómo asignar y desasignar la memoria en forma dinámica.

9.7 Cuándo se hacen llamadas a los constructores y destructores

El compilador llama de manera *implícita* a los constructores y destructores. El orden en el que ocurren estas llamadas a funciones depende del orden en el que la ejecución entra y sale de los alcances en los que se instancian los objetos. Por lo general, las llamadas a los destructores se realizan en *orden inverso* a las llamadas correspondientes a los constructores, pero como veremos en las figuras 9.7 a 9.9, las clases de almacenamiento de los objetos pueden alterar el orden en el que se llama a los destructores.

Constructores y destructores para objetos en alcance global

Los constructores se llaman para los objetos que se definen en alcance global (también conocido como alcance de espacio de nombres global) *antes* de que cualquier otra función (incluyendo a `main`), en ese programa, empiece a ejecutarse (aunque *no* se garantiza el orden de ejecución de los constructores de objetos globales entre los archivos). Los destructores correspondientes se llaman cuando termina `main`. La función `exit` obliga a un programa a terminar de inmediato y *no* ejecuta a los destructores de objetos locales. A menudo, la función `exit` se utiliza para terminar un programa cuando ocurre un error fatal

irrecuperable. La función **abort** se ejecuta de manera similar a la función `exit`, pero obliga al programa a terminar *de inmediato*, sin permitir que se llamen los destructores de ningún objeto. La función `abort` se utiliza comúnmente para indicar una *terminación anormal* del programa (vea el apéndice F para obtener más información acerca de las funciones `exit` y `abort`).

Constructores y destructores para objetos locales

El constructor para un objeto local se llama cuando la ejecución llega al punto en el que se define ese objeto; el correspondiente destructor se llama cuando la ejecución sale del alcance del objeto (es decir, el bloque en el que se define el objeto ha terminado de ejecutarse). Los constructores y destructores para los objetos locales se llaman cada vez que la ejecución entra y sale del alcance del objeto. Los destructores no se llaman para los objetos automáticos si el programa termina con una llamada a la función `exit` o a la función `abort`.

Constructores y destructores para objetos locales `static`

El constructor para un objeto local `static` se llama sólo *una vez*, cuando la ejecución llega por primera vez al punto en el que se define el objeto; el destructor correspondiente se llama cuando termina `main`, o cuando el programa llama a la función `exit`. Los objetos globales y `static` se destruyen en el orden *inverso* de su creación. Los destructores *no* se llaman para los objetos `static` si el programa termina con una llamada a la función `abort`.

Demostración de cuándo se hacen llamadas a los constructores y destructores

El programa de las figuras 9.7 a 9.9 demuestra el orden en el que se llaman los constructores y destructores para los objetos de la clase `CrearYDestruir` (figuras 9.7 y 9.8) de varias clases de almacenamiento en varios alcances. Cada objeto de la clase `CrearYDestruir` contiene un entero (`idObjeto`) y una cadena (`mensaje`), los cuales se utilizan en la salida del programa para identificar al objeto (figura 9.7, líneas 16 y 17). Este ejemplo mecánico es sólo para fines pedagógicos. Por esta razón, la línea 19 del destructor en la figura 9.8 determina si el objeto que se va a destruir tiene un valor `idObjeto` de `1` o `6` (línea 19) y, de ser así, imprime un carácter de nueva línea. Esta línea facilita más el seguimiento de la salida del programa.

```
1  // Fig. 9.7: CrearYDestruir.h
2  // Definición de la clase CrearYDestruir.
3  // Las funciones miembro se definen en CrearYDestruir.cpp.
4  #include <string>
5  using namespace std;
6
7  #ifndef CREAR_H
8  #define CREAR_H
9
10 class CrearYDestruir
11 {
12 public:
13    CrearYDestruir( int, string ); // constructor
14    ~CrearYDestruir(); // destructor
15 private:
16    int idObjeto; // número de ID para el objeto
17    string mensaje; // mensaje que describe al objeto
18 }; // fin de la clase CrearYDestruir
19
20 #endif
```

Fig. 9.7 | Definición de la clase `CrearYDestruir`.

```
1  // Fig. 9.8: CrearYDestruir.cpp
2  // Definiciones de las funciones miembro de CrearYDestruir.
3  #include <iostream>
4  #include "CrearYDestruir.h"// incluye la definición de la clase CrearYDestruir
5  using namespace std;
6
7  // el constructor establece el número de ID del objeto y el mensaje descriptivo
8  CrearYDestruir::CrearYDestruir( int ID, string cadenaMensaje )
9     : idObjeto( ID ), mensaje( cadenaMensaje )
10 {
11    cout << "El constructor del objeto " << idObjeto << "   se ejecuta   "
12       << mensaje << endl;
13 } // fin del constructor de CrearYDestruir
14
15 // destructor
16 CrearYDestruir::~CrearYDestruir()
17 {
18    // imprime nueva línea para ciertos objetos; mejora la legibilidad
19    cout << ( idObjeto == 1 || idObjeto == 6 ? "\n" : "" );
20
21    cout << "El destructor del objeto " << idObjeto << "   se ejecuta   "
22       << mensaje << endl;
23 } // fin del destructor ~CrearYDestruir
```

Fig. 9.8 | Definiciones de las funciones miembro de la clase `CrearYDestruir`.

En la figura 9.9 se define el objeto `primero` (línea 10) en alcance global. Su constructor se llama *antes* de que se ejecute cualquier instrucción en `main`, y su destructor se llama al momento en que termina el programa, *después* de que se hayan ejecutado los destructores para todos los objetos con duración de almacenamiento automático.

```
1  // Fig. 9.9: fig09_09.cpp
2  // Orden en el que se llama a los
3  // constructores y destructores.
4  #include <iostream>
5  #include "CrearYDestruir.h" // incluye la definición de la clase CrearYDestruir
6  using namespace std;
7
8  void crear( void ); // prototipo
9
10 CrearYDestruir primero( 1, "(global antes de main)" ); // objeto global
11
12 int main()
13 {
14    cout << "\nFUNCION MAIN: EMPIEZA LA EJECUCION" << endl;
15    CrearYDestruir segundo( 2, "(local automatic in main)" );
16    static CreateAndDestroy third( 3, "(local static en main)" );
17
18    crear(); // llama a la función para crear objetos
19
20    cout << "\nFUNCION MAIN: CONTINUA LA EJECUCION" << endl;
21    CrearYDestruir cuarto( 4, "(local automatico en main)" );
```

Fig. 9.9 | Orden en el que se llaman los constructores y los destructores (parte 1 de 2).

```
22    cout << "\nFUNCION MAIN: TERMINA LA EJECUCION" << endl;
23    } // fin de main
24
25    // función para crear objetos
26    void crear( void )
27    {
28       cout << "\nFUNCION CREAR: EMPIEZA LA EJECUCION" << endl;
29       CrearYDestruir quinto( 5, "(local automatico en crear)" );
30       static CrearYDestruir sexto( 6, "(local static en crear)" );
31       CrearYDestruir septimo( 7, "(local automatico en crear)" );
32       cout << "\nFUNCION CREAR: TERMINA LA EJECUCION" << endl;
33    } // fin de la función crear
```

```
El constructor del objeto 1   se ejecuta   (global antes de main)

FUNCION MAIN: EMPIEZA LA EJECUCION
El constructor del objeto 2   se ejecuta   (local automatico en main)
El constructor del objeto 3   se ejecuta   (local static en main)

FUNCION CREAR: EMPIEZA LA EJECUCION
El constructor del objeto 5   se ejecuta   (local automatico en crear)
El constructor del objeto 6   se ejecuta   (local static en crear)
El constructor del objeto 7   se ejecuta   (local automatico en crear)

FUNCION CREAR: TERMINA LA EJECUCION
El destructor del objeto 7   se ejecuta   (local automatico en crear)
El destructor del objeto 5   se ejecuta   (local automatico en crear)

FUNCION MAIN: CONTINUA LA EJECUCION
El constructor del objeto 4   se ejecuta   (local automatico en main)

FUNCION MAIN: TERMINA LA EJECUCION
El destructor del objeto 4   se ejecuta   (local automatico en main)
El destructor del objeto 2   se ejecuta   (local automatico en main)

El destructor del objeto 6   se ejecuta   (local static en crear)
El destructor del objeto 3   se ejecuta   (local static en main)

El destructor del objeto 1   se ejecuta   (global antes de main)
```

Fig. 9.9 | Orden en el que se llaman los constructores y los destructores (parte 2 de 2).

La función `main` (líneas 12 a 23) declara tres objetos. Los objetos `segundo` (línea 15) y `cuarto` (línea 21) son objetos locales, y el objeto `tercero` (línea 16) es un objeto local `static`. El constructor para cada uno de estos objetos se llama cuando la ejecución llega al punto en el que se declara ese objeto. Los destructores para los objetos `cuarto` y después `segundo` se llaman (en el orden inverso en que se llamaron sus constructores) cuando la ejecución llega al final de `main`. Como el objeto `tercero` es `static`, existe hasta que el programa termina su ejecución. El destructor para el objeto `tercero` se llama *antes* del destructor para el objeto global `primero`, pero *después* de que se destruyen todos los demás objetos.

La función `crear` (líneas 26 a 33) declara tres objetos: `quinto` (línea 29) y `séptimo` (línea 31) como objetos locales automáticos, y `sexto` (línea 30) como objeto local `static`. Los destructores para los objetos `séptimo` y después `quinto` se llaman (en el orden *inverso* en que se llamaron sus constructores) cuando

crear termina. Como sexto es static, existe hasta que el programa termina su ejecución. El destructor de sexto se llama *antes* de los destructores para tercero y primero, pero *después* de que se destruyen todos los demás objetos.

9.8 Caso de estudio con la clase Tiempo: una trampa sutil (devolver una referencia o un apuntador a un miembro de datos private)

Una referencia a un objeto es un alias para el nombre del objeto y, por ende, puede usarse del lado izquierdo de una instrucción de asignación. En este contexto, la referencia se convierte en un *lvalue* perfectamente aceptable que puede recibir un valor. Una manera de usar esta herramienta es hacer que una función miembro public de una clase devuelva una referencia a un miembro de datos private de esa clase. Si una función devuelve una referencia declarada como const, esa referencia es un *lvalue* no modificable y no puede usarse para modificar los datos.

El programa de las figuras 9.10 a 9.12 utiliza una clase Tiempo simplificada (figuras 9.10 y 9.11) para demostrar cómo devolver una referencia a un miembro de datos private con la función miembro establecerHoraIncorrecta (declarada en la línea 15 de la figura 9.10 y definida en las líneas 37 a 45 de la figura 9.11). Dicha devolución de referencia en realidad hace una llamada a la función miembro establecerHoraIncorrecta, ¡un alias para el miembro de datos private hora! La llamada a la función se puede utilizar en cualquier forma que se pueda usar el miembro de datos private, incluso como un *lvalue* en una instrucción de asignación, ¡con lo cual *se permite a los clientes de la clase hacer lo que quieran con los datos private de ésta*! Ocurriría el mismo problema si la función devolviera un apuntador a los datos private.

```
1  // Fig. 9.10: Tiempo.h
2  // Declaración de la clase Tiempo.
3  // Las funciones miembro se definen en Tiempo.cpp
4
5  // evita múltiples inclusiones del encabezado
6  #ifndef TIEMPO_H
7  #define TIEMPO_H
8
9  class Tiempo
10 {
11 public:
12    explicit Tiempo( int = 0, int = 0, int = 0 );
13    void establecerTiempo( int, int, int );
14    unsigned int obtenerHora() const;
15    unsigned int &establecerHoraIncorrecta( int ); // devolución de referencia
                                                     peligrosa
16 private:
17    unsigned int hora;
18    unsigned int minuto;
19    unsigned int segundo;
20 }; // fin de la clase Tiempo
21
22 #endif
```

Fig. 9.10 | Declaración de la clase Tiempo.

```
1  // Fig. 9.11: Tiempo.cpp
2  // Definiciones de las funciones miembro de la clase Tiempo.
3  #include <stdexcept>
4  #include "Tiempo.h" // incluye la definición de la clase Tiempo
5  using namespace std;
6
7  // función constructor para inicializar los datos privados; llama a la función
      miembro
8  // establecerTiempo para establecer las variables; los valores predeterminados
      son 0 (vea la definición de la clase)
9  Tiempo::Tiempo( int hr, int min, int seg )
10 {
11    establecerTiempo( hr, min, seg );
12 } // fin del constructor de Tiempo
13
14 // establece los valores de hora, minuto y segundo
15 void Tiempo::establecerTiempo( int h, int m, int s )
16 {
17    // valida hora, minuto y segundo
18    if ( ( h >= 0 && h < 24 ) && ( m >= 0 && m < 60 ) &&
19       ( s >= 0 && s < 60 ) )
20    {
21       hora = h;
22       minuto = m;
23       segundo = s;
24    } // fin de if
25    else
26       throw invalid_argument(
27          "hora, minuto y/o segundo estaban fuera de rango" );
28 } // fin de la función establecerTiempo
29
30 // devuelve el valor de hora
31 unsigned int Tiempo::obtenerHora()
32 {
33    return hora;
34 } // fin de la función obtenerHora
35
36 // mala práctica: devolver una referencia a un miembro de datos privado.
37 unsigned int &Tiempo::establecerHoraIncorrecta( int hh )
38 {
39    if ( hh >= 0 && hh < 24 )
40       hora = hh;
41    else
42       throw invalid_argument( "hora debe estar entre 0 y 23" );
43
44    return hora; // devolución de referencia peligrosa
45 } // fin de la función establecerHoraIncorrecta
```

Fig. 9.11 | Definiciones de las funciones miembro de la clase `Tiempo`.

En la figura 9.12 se declaran el objeto `Tiempo` llamado `t` (línea 10) y la referencia `horaRef` (línea 13), que se inicializa con la referencia devuelta por la llamada `t.establecerHoraIncorrecta(20)`. En la línea 15 se muestra el valor del alias `horaRef`. Esto muestra cómo `horaRef` *quebranta el encapsulamiento de la clase*; las instrucciones en `main` no deben tener acceso a los datos `private` de la clase. A continua-

ción, en la línea 16 se usa el alias para establecer el valor de `hora` en `30` (un valor inválido) y en la línea 17 se muestra el valor devuelto por la función `getHora`, para mostrar que al asignar un valor a `horaRef` en realidad se modifican los datos `private` en el objeto `Tiempo t`. Por último, en la línea 21 se usa la misma llamada a la función `establecerHoraIncorrecta` como un *lvalue* y se asigna 74 (otro valor inválido) a la referencia devuelta por la función. En la línea 26 se muestra de nuevo el valor devuelto por la función `getHora`, para mostrar que al asignar un valor al resultado de la llamada a la función en la línea 21 se modifican los datos `private` en el objeto `Tiempo t`.

Observación de Ingeniería de Software 9.7

Al devolver una referencia a un apuntador a un miembro de datos `private` *se quebranta el encapsulamiento de la clase y se hace dependiente el código cliente de la representación de los datos de la clase. Hay casos en donde es apropiado hacer esto; le mostraremos un ejemplo al crear nuestra clase* `Array` *personalizada en la sección 10.10.*

```
1  // Fig. 9.12: fig09_12.cpp
2  // Demostración de una función miembro pública que
3  // devuelve una referencia a un miembro de datos privado.
4  #include <iostream>
5  #include "Tiempo.h" // incluye la definición de la clase Tiempo
6  using namespace std;
7
8  int main()
9  {
10    Tiempo t; // crea un objeto Tiempo
11
12    // inicializa horaRef con la referencia devuelta por establecerHoraIncorrecta
13    int &horaRef = t.establecerHoraIncorrecta( 20 ); // 20 es una hora válida
14
15    cout << "Hora valida antes de la modificacion: " << horaRef;
16    horaRef = 30; // usa horaRef para establecer un valor inválido en el objeto Tiempo t
17    cout << "\nHora invalida despues de la modificacion: " << t.obtenerHora();
18
19    // Peligro: ¡La llamada a una función que devuelve
20    // una referencia se puede usar como un lvalue!
21    t.establecerHoraIncorrecta( 12 ) = 74; // asigna otro valor inválido a hora
22
23    cout << "\n\n*********************************************************\n"
24       << "MALA PRACTICA DE PROGRAMACION!!!!!!!!\n"
25       << "t.establecerHoraIncorrecta( 12 ) como un lvalue, hora invalida: "
26       << t.obtenerHora()
27       << "\n*********************************************************" << endl;
28 } // fin de main
```

```
Hora valida antes de la modificacion: 20
Hora invalida despues de la modificacion: 30

*********************************************************************
MALA PRACTICA DE PROGRAMACION!!!!!!!!
t.establecerHoraIncorrecta( 12 ) como un lvalue, hora invalida: 74
*********************************************************************
```

Fig. 9.12 | Función miembro `public` que devuelve una referencia a un miembro de datos `private`.

9.9 Asignación predeterminada a nivel de miembros

El operador de asignación (=) se puede utilizar para asignar un objeto a otro objeto del mismo tipo. De manera predeterminada, dicha asignación se realiza mediante la **asignación a nivel de miembros** (también conocida como **asignación de copia**): cada miembro de datos del objeto a la *derecha* del operador de asignación se asigna de manera individual al *mismo* miembro de datos en el objeto a la *izquierda* del operador de asignación. En las figuras 9.13 y 9.14 se define una clase `Fecha`. En la línea 18 de la figura 9.15 se usa la **asignación predeterminada a nivel de miembros** para asignar los miembros de datos del objeto `Fecha` llamado `fecha1` a los correspondientes miembros de datos del objeto `Fecha` llamado `fecha2`. En este caso, el miembro `mes` de `fecha1` se asigna al miembro `mes` de `fecha2`, el miembro `dia` de `fecha1` se asigna al miembro `dia` de `fecha2` y el miembro `anio` de `fecha1` se asigna al miembro `anio` de `fecha2`. [*Precaución:* la asignación a nivel de miembros puede producir problemas graves al utilizarse con una clase cuyos miembros de datos contengan apuntadores a la memoria asignada en forma dinámica; veremos estos problemas en el capítulo 10 y le mostraremos cómo lidiar con ellos].

```
1  // Fig. 9.13: Fecha.h
2  // Declaración de la clase Fecha. Las funciones miembro se definen en Fecha.cpp
3
4  // evita múltiples inclusiones del encabezado
5  #ifndef FECHA_H
6  #define FECHA_H
7
8  // definición de la clase Fecha
9  class Fecha
10 {
11 public:
12    explicit Fecha( int = 1, int = 1, int = 2000 ); // constructor predeterminado
13    void imprimir();
14 private:
15    unsigned int mes;
16    unsigned int dia;
17    unsigned int anio;
18 }; // fin de la clase Fecha
19
20 #endif
```

Fig. 9.13 | Declaración de la clase `Fecha`.

```
1  // Fig. 9.14: Fecha.cpp
2  // Definiciones de las funciones miembro de la clase Fecha.
3  #include <iostream>
4  #include "Fecha.h" // incluye la definición de la clase Fecha de Fecha.h
5  using namespace std;
6
7  // Constructor de Fecha (debe realizar comprobación de rangos)
8  Fecha::Fecha( int m, int d, int a )
9     : mes( m ), dia( d ), anio( a ){
10 {
11 } // fin del constructor de Fecha
```

Fig. 9.14 | Definiciones de las funciones miembro de la clase `Fecha` (parte 1 de 2).

```
12
13  // imprime la Fecha en el formato mm/dd/aaaa
14  void Fecha::imprimir()
15  {
16     cout << mes << '/' << dia << '/' << anio;
17  } // fin de la función imprimir
```

Fig. 9.14 | Definiciones de las funciones miembro de la clase `Fecha` (parte 2 de 2).

```
1   // Fig. 9.15: fig09_15.cpp
2   // Demuestra que los objetos de una clase se pueden asignar
3   // unos a otros mediante la asignación predeterminada a nivel de bits.
4   #include <iostream>
5   #include "Fecha.h" // incluye la definición de la clase Fecha de Fecha.h
6   using namespace std;
7
8   int main()
9   {
10     Fecha fecha1( 7, 4, 2004 );
11     Fecha fecha2; // el valor predeterminado de fecha2 es 1/1/2000
12
13     cout << "fecha1 = ";
14     fecha1.imprimir();
15     cout << "\nfecha2 = ";
16     fecha2.imprimir();
17
18     fecha2 = fecha1; // asignación predeterminada a nivel de bits
19
20     cout << "\n\nDespues de la asignacion predeterminada a nivel de bits,
                fecha2 = ";
21     fecha2.imprimir();
22     cout << endl;
23  } // fin de main
```

```
fecha1 = 7/4/2004
fecha2 = 1/1/2000

Despues de la asignacion predeterminada a nivel de bits, fecha2 = 7/4/2004
```

Fig. 9.15 | Se pueden asignar unos objetos a otros mediante el uso de la asignación predeterminada a nivel de bits.

Los objetos se pueden pasar como argumentos para funciones y se pueden devolver de las funciones. Dichos procesos de pasar y devolver valores se llevan a cabo usando el paso por valor de manera predeterminada; se pasa o devuelve una *copia* del objeto. En tales casos, C++ crea un nuevo objeto y utiliza un **constructor de copia** para copiar los valores del objeto original en el nuevo objeto. Para cada clase, el compilador proporciona un constructor de copia predeterminado que copie cada miembro del objeto original en el miembro correspondiente del nuevo objeto. Al igual que la asignación a nivel de miembros, los constructores de copia pueden provocar problemas graves cuando se utilizan con una clase cuyos miembros de datos contienen apuntadores a memoria asignada en forma dinámica. En el capítulo 10 veremos cómo definir constructores de copia personalizados, que copien de manera apropiada los objetos que contengan apuntadores a memoria asignada en forma dinámica.

9.10 Objetos const y funciones miembro const

Ahora veamos cómo se aplica el principio del menor privilegio a los objetos. Algunos objetos necesitan ser modificables y otros no. Podemos usar la palabra clave `const` para especificar que un objeto *no es* modificable, y que cualquier intento por modificar el objeto debe producir un error de compilación. La instrucción

```
const Tiempo mediodia ( 12, 0, 0 );
```

declara un objeto `const mediodia` de la clase `Tiempo` y lo inicializa con las 12 del mediodía. Es posible instanciar objetos `const` y no `const` de la misma clase.

Observación de Ingeniería de Software 9.8

Los intentos de modificar un objeto `const` se atrapan en tiempo de compilación, en vez de producir errores en tiempo de ejecución.

Tip de rendimiento 9.3

Declarar variables y objetos `const` cuando sea apropiado puede mejorar el rendimiento; los compiladores pueden realizar ciertas optimizaciones sobre constantes que no se pueden llevar a cabo sobre variables que no son `const`.

C++ no permite llamadas a funciones miembro para objetos `const`, a menos que las mismas funciones miembro también se declaren como `const`. Esto se aplica incluso para las funciones miembro *obtener* que *no* modifican el objeto. *Ésta es también una razón clave de que hayamos declarado como `const` todas las funciones miembro que no modifican a los objetos sobre los cuales se llaman.*

Como vimos a partir de la clase `LibroCalificaciones` en el capítulo 3, una función miembro se especifica como `const` *tanto* en su prototipo mediante la inserción de la palabra clave `const` *después* de la lista de parámetros de la función y, en el caso de la definición de la función, antes de la llave izquierda que comienza el *cuerpo* de la función.

Error común de programación 9.2

Definir como `const` una función miembro que modifica un miembro de datos de un objeto es un error de compilación.

Error común de programación 9.3

Definir como `const` una función miembro que llama a una función miembro no `const` de la clase en la misma instancia de ésta, es un error de compilación.

Error común de programación 9.4

Invocar a una función miembro no `const` en un objeto `const` es un error de compilación.

Hay un interesante problema que surge para los constructores y destructores, cada uno de los cuales por lo general modifica objetos. Un constructor *debe* tener la capacidad de modificar un objeto, para que éste se pueda inicializar de manera apropiada. Un destructor debe tener la capacidad de realizar sus tareas de mantenimiento de terminación antes de que el sistema reclame la memoria ocupada por el objeto. Tratar de declarar un constructor o un destructor como `const` es un error de compilación. Lo "constante" de un objeto `const` se hace valer desde el momento en que el constructor *completa* la inicialización del objeto, hasta que se llama al destructor de ese objeto.

Uso de funciones miembro `const` *y no* `const`

El programa de las figura 9.16 usa la clase `Tiempo` de las figuras 9.4 y 9.5, pero elimina la palabra clave `const` del prototipo y la definición de la función `imprimirEstandar`, para que podamos mostrar un error de compilación. Instanciamos dos objetos `Tiempo`: el objeto no `const` `despertar` (línea 7) y el objeto `const` `mediodia` (línea 8). El programa trata de invocar a las funciones miembro no `const` `establecerHora` (línea 13) e `imprimirEstandar` (línea 20) en el objeto `const` `mediodia`. En cada caso, el compilador genera un mensaje de error. El programa también ilustra las otras tres combinaciones de llamadas a funciones miembro en los objetos: una función miembro no `const` en un objeto no `const` (línea 11), una función miembro `const` en un objeto no `const` (línea 15) y una función miembro `const` en un objeto `const` (líneas 17 a 18). Los mensajes de error generados para las funciones miembro no `const` que se llaman desde un objeto `const` se muestran en la ventana de resultados.

```
1  // Fig. 9.16: fig09_16.cpp
2  // Objetos const y funciones miembro const.
3  #include "Tiempo.h" // incluye la definición de la clase Tiempo
4
5  int main()
6  {
7     Tiempo despertar( 6, 45, 0 ); // objeto no constante
8     const Tiempo mediodia( 12, 0, 0 ); // objeto constante
9
10                                    // OBJETO     FUNCIÓN MIEMBRO
11    despertar.establecerHora( 18 ); // no const   no const
12
13    mediodia.establecerHora( 12 );  // const      no const
14
15    despertar.obtenerHora();        // no const   const
16
17    mediodia.obtenerMinuto();       // const      const
18    mediodia.imprimirUniversall();  // const      const
19
20    mediodia.imprimirEstandar();    // const      no const
21 } // fin de main
```

Microsoft Visual C++ compiler error messages:

```
C:\ejemplos\cap09\Fig09_16_18\fig09_18.cpp(13) : error C2662:
   'Tiempo::establecerHora' : cannot convert 'this' pointer from 'const Tiempo' to
   'Tiempo &'
         Conversion loses qualifiers
C:\ejemplos\cap09\Fig09_16_18\fig09_18.cpp(20) : error C2662:
   'Tiempo::imprimirEstandar' : cannot convert 'this' pointer from 'const Tiempo' to
   'Tiempo &'
       Conversion loses qualifiers
```

Fig. 9.16 | Objetos `const` y funciones miembro `const`.

Un constructor debe ser una función miembro no `const`, pero de todas formas se puede utilizar para inicializar un objeto `const` (figura 9.16, línea 8). En la figura 9.5 vimos que la definición del constructor de `Tiempo` llama a otra función miembro no `const` (`establecerTiempo`) para realizar la inicialización de un objeto `Tiempo`. Se permite la invocación de una función miembro no `const` desde la llamada al constructor como parte de la inicialización de un objeto `const`.

La línea 20 en la figura 9.16 genera un error de compilación, aun cuando la función miembro `imprimirEstandar` de la clase `Tiempo` no modifica el objeto en el que se le invoca. El hecho de que una función miembro no modifique un objeto *no* es suficiente; la función se debe declarar *explícitamente* como `const`.

9.11 Composición: objetos como miembros de clases

Un objeto `RelojDespertador` necesita saber cuándo se supone que debe sonar su alarma, así que ¿por qué no incluir un objeto `Tiempo` como miembro de la clase `RelojDespertador`? Dicha capacidad se conoce como **composición** y algunas veces como **relación "*tiene un*"**; *una clase puede tener objetos de otras clases como miembros.*

Observación de Ingeniería de Software 9.9

Una forma común de reutilización de software es la composición, en la cual una clase tiene objetos de otros tipos como miembros.

Anteriormente vimos cómo pasar argumentos al constructor de un objeto que creamos en `main`. En esta sección veremos cómo *el constructor de una clase puede pasar argumentos a los constructores de objetos miembro mediante inicializadores de miembros.*

Observación de Ingeniería de Software 9.10

*Los miembros de datos se construyen en el orden en el que se declaran en la definición de la clase (no en el orden en el que se listan en la lista de inicializadores de miembros del constructor) y antes de que se construyan los objetos de la clase que los encierra (algunas veces conocidos como **objetos anfitriones**).*

El siguiente programa utiliza las clases `Fecha` (figuras 9.17 y 9.18) y `Empleado` (figuras 9.19 y 9.20) para demostrar la composición. La definición de la clase `Empleado` (figura 9.19) contiene los miembros de datos `private` `primerNombre`, `apellidoPaterno`, `fechaNacimiento` y `fechaContratacion`. Los miembros `fechaNacimiento` y `fechaContratacion` son objetos `const` de la clase `Fecha`, que contiene los datos miembro `private` `mes`, `dia` y `anio`. El encabezado del constructor de `Empleado` (figura 9.20, líneas 10 y 11) especifica que el constructor tiene cuatro parámetros (`primero`, `ultimo`, `fechaDeNacimiento` y `fechaDeContratacion`). Los primeros dos parámetros se pasan mediante inicializadores de miembros al constructor de la clase `string` para los miembros de datos `primerNombre` y `apellidoPaterno`. Los últimos dos parámetros se pasan mediante inicializadores de miembros al constructor de la clase `Fecha` para los miembros de datos `fechaNacimiento` y `fechaContratacion`.

```
1  // Fig. 9.17: Fecha.h
2  // Definición de la clase Fecha; las funciones miembro se definen en Fecha.cpp
3  #ifndef FECHA_H
4  #define FECHA_H
5
6  class Fecha
7  {
8  public:
9     static const unsigned int mesesPorAnio = 12; // meses en un año
10    explicit Fecha( int = 1, int = 1, int = 1900 ); // constructor predeterminado
11    void imprimir() const; // imprime la fecha en formato mes/día/año
12    ~Fecha(); // se proporciona para confirmar el orden de destrucción
```

Fig. 9.17 | Definición de la clase `Fecha` (parte 1 de 2).

```
13   private:
14      unsigned int mes; // 1-12 (Enero-Diciembre)
15      unsigned int dia; // 1-31 con base en el mes
16      unsigned int anio; // cualquier año
17
18      // función utilitaria para comprobar si dia es apropiado para mes y anio
19      unsigned int comprobarDia( int ) const;
20   }; // fin de la clase Fecha
21
22   #endif
```

Fig. 9.17 | Definición de la clase `Fecha` (parte 2 de 2).

```
1    // Fig. 9.18: Fecha.cpp
2    // Definiciones de las funciones miembro de la clase Fecha.
3    #include <array>
4    #include <iostream>
5    #include <stdexcept>
6    #include "Fecha.h" // incluye la definición de la clase Fecha
7    using namespace std;
8
9    // el constructor confirma el valor apropiado para el mes; llama
10   // a la función utilitaria comprobarDia para confirmar un valor apropiado para dia
11   Fecha::Fecha( int mn, int dd, int aa )
12   {
13      if ( mn > 0 && mn <= mesesPorAnio ) // valida el mes
14         mes = mn;
15      else
16         throw invalid_argument( "mes debe estar entre 1 y 12" );
17
18      anio = aa; // se pudo validar aa
19      dia = comprobarDia( dd ); // valida dia
20
21      // imprime objeto Fecha para mostrar cuándo se llama a su constructor
22      cout << "Constructor del objeto Fecha para fecha ";
23      imprimir();
24      cout << endl;
25   } // fin del constructor de Fecha
26
27   // imprime objeto Fecha en el formato mes/dia/anio
28   void Fecha::imprimir() const
29   {
30      cout << mes << '/'  << dia << '/'  << anio;
31   } // fin de la función imprimir
32
33   // imprime objeto Fecha para mostrar cuándo se llama a su destructor
34   Fecha::~Fecha()
35   {
36      cout << "Destructor del objeto Fecha para fecha ";
37      imprimir();
38      cout << endl;
39   } // fin del destructor ~Fecha
```

Fig. 9.18 | Definiciones de las funciones miembro de la clase `Fecha` (parte 1 de 2).

```
40
41 // función utilitaria para confirmar el valor de dia apropiado con base
42 // en mes y anio; maneja años bisiestos también
43 unsigned int Fecha::comprobarDia( int diaPrueba ) const
44 {
45    static const array< int, mesesPorAnio + 1 > diasPorMes =
46       { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
47
48    // determina si diaPrueba es válido para el mes especificado
49    if ( diaPrueba > 0 && diaPrueba <= diasPorMes[ mes ] )
50       return diaPrueba;
51
52    // comprueba 29 de febrero para año bisiesto
53    if ( mes == 2 && diaPrueba == 29 && ( anio % 400 == 0 ||
54       ( anio % 4 == 0 && anio % 100 != 0 ) ) )
55       return diaPrueba;
56
57    throw invalid_argument( "Dia invalido para mes y anio actuales" );
58 } // fin de la función comprobarDia
```

Fig. 9.18 | Definiciones de las funciones miembro de la clase `Fecha` (parte 2 de 2).

```
1  // Fig. 9.19: Empleado.h
2  // Definición de la clase Empleado que muestra la composición.
3  // Las funciones miembro se definen en Empleado.cpp.
4  #ifndef EMPLEADO_H
5  #define EMPLEADO_H
6
7  #include <string>
8  #include "Fecha.h" // incluye la definición de la clase Fecha
9
10 class Empleado
11 {
12 public:
13    Empleado( const std::string &, const std::string &,
14       const Fecha &, const Fecha & );
15    void imprimir() const;
16    ~Empleado(); // se proporciona para confirmar el orden de destrucción
17 private:
18    std::string primerNombre; // composición: objeto miembro
19    std::apellidoPaterno; // composición: objeto miembro
20    const Fecha fechaNacimiento; // composición: objeto miembro
21    const Fecha fechaContratacion; // composición: objeto miembro
22 }; // fin de la clase Empleado
23
24 #endif
```

Fig. 9.19 | Definición de la clase `Empleado` que muestra la composición.

```
1  // Fig. 9.20: Empleado.cpp
2  // Definiciones de las funciones miembro de la clase Empleado.
3  #include <iostream>
4  #include "Empleado.h" // definición de la clase Empleado
5  #include "Fecha.h" // definición de la clase Fecha
6  using namespace std;
7
8  // el constructor usa la lista de inicializadores de miembros para pasar los valores
9  // de los inicializadores a los constructores de los objetos miembro
10 Empleado::Empleado( const string &nombre, const string &apellido,
11    const Fecha &fechaDeNacimiento, const Fecha &fechaDeContratacion )
12    : primerNombre( nombre ), // inicializa primerNombre
13      apellidoPaterno( apellido ), // inicializa apellidoPaterno
14      fechaNacimiento( fechaDeNacimiento ), // inicializa fechaNacimiento
15      fechaContratacion( fechaDeContratacion ) // inicializa fechaContratacion
16 {
17    // imprime objeto Empleado para mostrar cuándo se llama al constructor
18    cout << "Constructor del objeto Empleado: "
19       << primerNombre << ' ' << apellidoPaterno << endl;
20 } // fin del constructor de Empleado
21
22 // imprime objeto Empleado
23 void Empleado::imprimir() const
24 {
25    cout << apellidoPaterno << ", " << primerNombre << "  Contratacion: ";
26    fechaContratacion.imprimir();
27    cout << "  Nacimiento: ";
28    fechaNacimiento.imprimir();
29    cout << endl;
30 } // fin de la función imprimir
31
32 // imprime objeto Empleado para mostrar cuándo se llama a su destructor
33 Empleado::~Empleado()
34 {
35    cout << "Destructor del objeto Empleado: "
36       << apellidoPaterno << ", " << primerNombre << endl;
37 } // fin del constructor ~Empleado
```

Fig. 9.20 | Definiciones de las funciones miembro de la clase `Empleado`.

Lista de inicializadores de miembros del constructor de `Empleado`

El signo de *dos puntos (:)* que sigue después del encabezado del constructor (figura 9.20, línea 12) comienza la *lista de inicializadores de miembros*. Los inicializadores de miembros especifican los parámetros del constructor de `Empleado` que se van a pasar a los constructores de los miembros de datos `string` y `Fecha`. Los parámetros `nombre`, `apellido`, `fechaDeNacimiento` y `fechaDeContratacion` se pasan a los constructores para los objetos `primerNombre` (línea 12), `apellidoPaterno` (línea 13), `fechaNacimiento` (línea 14) y `fechaContratacion` (línea 15), respectivamente. De nuevo, los inicializadores de miembros van separados por comas. El orden de los inicializadores de miembros no importa. Se ejecutan en el orden en el que se declaran los objetos miembro en la clase `Empleado`.

Buena práctica de programación 9.3

Por cuestión de claridad, hay que listar los inicializadores de miembros en el orden en el que se declaran los miembros de datos de la clase.

Constructor de copia predeterminado de la clase `Fecha`

Al estudiar la clase `Fecha` (figura 9.17), el lector observará que la clase *no* proporciona un constructor que reciba un parámetro de tipo `Fecha`. Entonces, ¿cómo puede la lista de inicializadores de miembros en el constructor de la clase `Empleado` inicializar los objetos `fechaNacimiento` y `fechaContratacion` al pasar objetos `Fecha` a sus constructores de `Fecha`? Como mencionamos en la sección 9.9, el compilador proporciona a cada clase un *constructor de copia predeterminado* que copia cada miembro de datos del objeto argumento del constructor en el miembro correspondiente del objeto que se va a inicializar. En el capítulo 10 veremos cómo se pueden definir constructores de copia *personalizados*.

Prueba de las clases `Fecha` *y* `Empleado`

En la figura 9.21 se crean dos objetos `Fecha` (líneas 10 y 11) y se pasan como argumentos al constructor del objeto `Empleado` creado en la línea 12. En la línea 15 se imprimen los datos del objeto `Empleado`. Cuando se crea cada objeto `Fecha` en las líneas 10 y 11, el constructor de `Fecha` definido en las líneas 11 a 25 de la figura 9.18 despliega una línea de salida para mostrar que se llamó al constructor (vea las primeras dos líneas de los resultados de ejemplo). [*Nota:* en la línea 12 de la figura 9.21 se hacen dos llamadas adicionales al constructor de `Fecha` que no aparecen en la salida del programa. Cuando se inicializa cada uno de los objetos miembro `Fecha de Empleado` en la lista de inicializadores de miembros del constructor de `Empleado` (figura 9.20, líneas 14 y 15), se hace una llamada al constructor de copia predeterminado para la clase `Fecha`. Como el compilador define este constructor de manera implícita, no contiene instrucciones de salida para demostrar cuándo se hace la llamada].

```
1  // Fig. 9.21: fig09_21.cpp
2  // Demostración de la composición--un objeto con objetos miembro.
3  #include <iostream>
4  #include "Fecha.h" // definición de la clase Fecha
5  #include "Empleado.h" // definición de la clase Empleado
6  using namespace std;
7
8  int main()
9  {
10    Fecha nacimiento( 7, 24, 1949 );
11    Fecha contratacion( 3, 12, 1988 );
12    Empleado gerente( "Bob", "Blue", nacimiento, contratacion );
13
14    cout << endl;
15    gerente.imprimir();
16 } // fin de main
```

```
Constructor del objeto Fecha para fecha 7/24/1949
Constructor del objeto Fecha para fecha 3/12/1988
Constructor del objeto Empleado: Bob Blue

Blue, Bob  Contratacion: 3/12/1988  Nacimiento: 7/24/1949
Destructor del objeto Empleado: Blue, Bob
Destructor del objeto Fecha para fecha 3/12/1988
Destructor del objeto Fecha para fecha 7/24/1949
Destructor del objeto Fecha para fecha 3/12/1988
Destructor del objeto Fecha para fecha 7/24/1949
```

En realidad se hacen cinco llamadas a constructores cuando se construye un `Empleado`: dos llamadas al constructor de la clase `string` (líneas 12 y 13 de la figura 9.20), dos llamadas al constructor de copia predeterminado de la clase `Fecha` (líneas 14 y 15 de la figura 9.20) y la llamada al constructor de la clase `Empleado`.

Fig. 9.21 | Demostración de la composición: un objeto con objetos miembros.

La clase `Fecha` y la clase `Empleado` incluyen un destructor (líneas 34 a 39 de la figura 9.18 y líneas 33 a 37 de la figura 9.20, respectivamente) que imprime un mensaje cuando se destruye un objeto de su clase. Esto nos permite confirmar en la salida del programa que los objetos se construyen de *adentro hacia afuera* y se destruyen en orden *inverso*, desde *afuera hacia dentro* (es decir, los objetos *miembro* de `Fecha` se destruyen después del objeto `Empleado` que los *contiene*).

Observe las últimas cuatro líneas en la salida de la figura 9.21. Las últimas dos líneas son la salida del destructor de `Fecha` que se ejecuta en los objetos `Fecha contratacion` (figura 9.21, línea 11) y `nacimiento` (figura 9.21, línea 10), respectivamente. Estos resultados confirman que los tres objetos creados en `main` se destruyen en el orden *inverso* al orden en el que se construyeron. La salida del destructor de `Empleado` aparece cinco líneas antes de la última. Las líneas tercera y cuarta de la parte inferior de la ventana de resultados muestran la ejecución de los destructores para los objetos miembro `fechaContratacion` (figura 9.19, línea 21) y `fechaNacimiento` (figura 9.19, línea 20) de `Empleado`. Las últimas dos líneas de la salida corresponden a los objetos `Fecha` creados en las líneas 11 y 10 de la figura 9.21.

Estos resultados confirman que el objeto `Empleado` se destruye desde *afuera hacia dentro*; es decir, el destructor de `Empleado` se ejecuta primero (el resultado se muestra cinco líneas antes de la última de la ventana de resultados), y después los objetos miembro se destruyen en el *orden inverso* al que fueron construidos. El destructor de la clase `string` no contiene instrucciones de salida, por lo que *no* vemos que se destruyen los objetos `primerNombre` y `apellidoPaterno`. De nuevo, los resultados en la figura 9.21 no muestran la ejecución de los constructores para los objetos miembro `fechaNacimiento` y `fechaContratacion`, ya que se inicializaron con los constructores de copia *predeterminados* de la clase `Fecha` que proporciona el compilador.

¿Qué ocurre si no usamos la lista inicializadora de miembros?

Si un objeto miembro *no* se inicializa a través de un inicializador de miembros, se hará una llamada *implícita* al *constructor predeterminado* del objeto miembro. Los valores (si los hay) establecidos por el constructor predeterminado se pueden redefinir mediante funciones *establecer*. Sin embargo, para la inicialización compleja, esta metodología puede requerir una cantidad considerable de trabajo y tiempo adicionales.

Error común de programación 9.5

Si un objeto miembro no se inicializa con un inicializador de miembros y la clase del objeto miembro no proporciona un constructor predeterminado (es decir, que la clase del objeto miembro defina uno o más constructores, pero ninguno sea un constructor predeterminado), se produce un error de compilación.

Tip de rendimiento 9.4

Inicialice los objetos miembro explícitamente a través de los inicializadores de miembros. Esto elimina la sobrecarga de "inicializar dos veces" los objetos miembro: una vez cuando se hace la llamada al constructor predeterminado del objeto miembro, y otra vez cuando se hacen las llamadas a las funciones establecer *en el cuerpo del constructor (o después) para inicializar el objeto miembro.*

Observación de Ingeniería de Software 9.11

Si el miembro de datos es un objeto de otra clase, al hacer ese objeto miembro `public` *no se viola el encapsulamiento ni el ocultamiento de los miembros* `private` *de ese objeto miembro. No obstante, sí se viola el encapsulamiento y el ocultamiento de la implementación de la clase que lo contiene, por lo que los objetos miembro de los tipos de clases deben seguir siendo* `private`.

9.12 Funciones `friend` y clases `friend`

Una **función `friend`** de una clase es una función no miembro que tiene el derecho de acceder a los miembros `public` *y* no `public` de la clase. Se pueden declarar funciones independientes, clases completas o funciones miembro de otras clases como *amigas* de otra clase.

En esta sección presentaremos un ejemplo mecánico acerca de cómo funciona una función `friend`. En el capítulo 10 mostraremos funciones `friend` que sobrecargan operadores para usarlos con objetos de clases; como veremos, algunas veces no es posible usar una función miembro para ciertos operadores sobrecargados.

Declaración de una función `friend`

Para declarar una función como amiga (`friend`) de una clase, hay que anteponer la palabra clave `friend` al prototipo de la función en la definición de la clase. Para declarar todas las funciones miembro de la clase `ClaseDos` como amigas de la clase `ClaseUno`, coloque una declaración de la forma

```
friend class ClaseDos;
```

en la definición de la clase `ClaseUno`.

La amistad se *otorga*, *no se toma*; para que la clase B sea amiga (`friend`) de la clase A, ésta debe declarar *explícitamente* que la clase B es su amiga. La relación de amistad no es *simétrica*: si la clase A es amiga de la clase B, no podemos inferir que la clase B es amiga de la clase A. La amistad tampoco es *transitiva*: si la clase A es amiga de la clase B y ésta es amiga de la clase C, no podemos inferior que la clase A es amiga de la clase C.

Modificación de los datos `private` *de una clase con una función* `friend`

La figura 9.22 es un ejemplo mecánico en el que definimos la función `friend` llamada `establecerX` para establecer el miembro de datos `private` `x` de la clase `Cuenta`. Como convención, colocamos la declaración `friend` (línea 9) *primero* en la definición de la clase, incluso antes de declarar las funciones miembro `public`. De nuevo, esta declaración `friend` puede aparecer *en cualquier parte* de la clase.

La función `establecerX` (líneas 29 a 32) es una función independiente (global); no es una función miembro de la clase `Cuenta`. Por esta razón, cuando `establecerX` se invoca para el objeto `contador`, en la línea 41 se pasa `contador` como argumento a `establecerX`, en vez de usar un manejador (como el nombre del objeto) para llamar a la función, como en

```
contador.establecerX( 8 ); // error: establecerX no es una función miembro
```

Si elimina la declaración `friend` en la línea 9, recibirá mensajes de error para indicar que la función `establecerX` no puede modificar el miembro de datos `private` `x` de la clase `Cuenta`.

```
1  //Fig. 9.22: fig09_22.cpp
2  // Las funciones amigas pueden acceder a los miembros privados de una clase.
3  #include <iostream>
4  using namespace std;
5
6  // definición de la clase Cuenta
7  class Cuenta
8  {
9     friend void establecerX( Cuenta &, int ); // declaración friend
```

Fig. 9.22 | Las funciones `friend` pueden acceder a los miembros `private` de una clase (parte 1 de 2).

```
10   public:
11      // constructor
12      Cuenta()
13         : x( 0 ) // inicializa x en 0
14      {
15         // cuerpo vacío
16      } // fin del constructor Cuenta
17
18      // imprime x
19      void imprimir() const
20      {
21         cout << x << endl;
22      } // fin de la función imprimir
23   private:
24      int x; // miembro de datos
25   }; // fin de la clase Cuenta
26
27   // la función establecerX puede modificar los datos privados de Cuenta
28   // debido a que establecerX se declara como amiga de Cuenta (línea 9)
29   void establecerX( Cuenta &c, int val )
30   {
31      c.x = val; // se permite debido a que establecerX es amiga de Cuenta
32   } // fin de la función establecerX
33
34   int main()
35   {
36      Cuenta contador; // crea objeto Cuenta
37
38      cout << "contador.x despues de crear la instancia: ";
39      contador.imprimir();
40
41      establecerX( contador, 8 ); // establece x usando una función friend
42      cout << "contador.x despues de la llamada a la funcion friend establecerX: ";
43      contador.imprimir();
44   } // fin de main
```

```
contador.x despues de crear la instancia: 0
contador.x despues de la llamada a la funcion friend establecerX: 8
```

Fig. 9.22 | Las funciones `friend` pueden acceder a los miembros `private` de una clase (parte 2 de 2).

Como dijimos antes, la figura 9.22 es un ejemplo mecánico acerca del uso de la instrucción `friend`. Por lo general sería apropiado definir la función `establecerX` como función miembro de la clase `Cuenta`. También sería generalmente apropiado separar el programa de la figura 9.22 en tres archivos:

1. Un encabezado (por ejemplo, `Cuenta.h`) que contenga la definición de la clase `Cuenta`, que a su vez contenga el prototipo de la función `friend` llamada `establecerX`.
2. Un archivo de implementación (por ejemplo, `Cuenta.cpp`) que contenga las definiciones de las funciones miembro de la clase `Cuenta` y la definición de la función `friend` llamada `establecerX`.
3. Un programa de prueba (por ejemplo, `fig09_22.cpp`) con `main`.

Funciones `friend` *sobrecargadas*

Es posible especificar funciones sobrecargadas como amigas de una clase. Cada función sobrecargada destinada para ser `friend` debe declararse explícitamente en la definición de la clase como una amiga de ésta.

Observación de Ingeniería de Software 9.12

Aun cuando los prototipos para las funciones `friend` *aparecen en la definición de la clase, las funciones amigas no son funciones miembro.*

Observación de Ingeniería de Software 9.13

Las nociones `private`*,* `protected` *y* `public` *de acceso a miembros no son relevantes para las declaraciones* `friend`*, por lo que estas declaraciones se pueden colocar en* cualquier *parte de la definición de una clase.*

Buena práctica de programación 9.4

Coloque todas las declaraciones de amistad primero dentro del cuerpo de la definición de una clase, y no coloque un especificador de acceso antes de éstas.

9.13 Uso del apuntador `this`

Hemos visto que las funciones miembro de un objeto pueden manipular los datos de éste. Puede haber *muchos* objetos de una clase, así que ¿cómo saben las funciones miembro *cuáles* miembros de datos del objeto deben manipular? Cada objeto tiene acceso a su propia dirección a través de un apuntador llamado `this` (una palabra clave de C++). El apuntador `this` de un objeto *no* es parte del objeto en sí; es decir, el tamaño de la memoria ocupada por el apuntador `this` no se refleja en el resultado de una operación `sizeof` en el objeto. En vez de ello, el apuntador `this` se pasa (por el compilador) como un argumento *implícito* para cada una de las funciones miembro no `static` del objeto. En la sección 9.14 se introducen los miembros de clase `static` y se explica por qué el apuntador `this` *no* se pasa implícitamente a las funciones miembro `static`.

Uso del apuntador `this` *para evitar conflictos de nombres*

Las funciones miembro utilizan el apuntador `this` de manera *implícita* (como hemos hecho hasta este punto) o *explícitamente* para hacer referencia los miembros de datos de un objeto y a otras funciones miembro. Un uso *explícito* común del apuntador `this` es para evitar *conflictos de nombres* entre los miembros de datos de una clase y los parámetros de las funciones miembro (o de otras variables locales). Considere el miembro de datos `hora` y la función miembro `establecerHora` de la clase `Tiempo` de las figuras 9.4 y 9.5. Podríamos haber definido a `establecerHora` de la siguiente forma:

```
// establecer valor de hora
void Tiempo::establecerHora( int hora )
{
   if ( hora >= 0 && hora < 24 )
      this->hora = hora; // usa el apuntador this para acceder al miembro de datos
   else
      throw invalid_argument( "hora debe estar entre 0 y 23" );
} // fin de la función establecerHora
```

En la definición de esta función, el parámetro de `establecerHora` tiene el *mismo nombre* que el miembro de datos `hora`. En el alcance de `establecerHora`, el parámetro `hora` *oculta* el miembro de datos. Sin embargo, aún es posible acceder al miembro de datos `hora` si se califica su nombre con `this->`. Por ende, la siguiente instrucción asigna el valor del parámetro `hora` al miembro de datos `hora`:

```
this->hora = hora; // usa el apuntador this para acceder al miembro de datos
```

Tip para prevenir errores 9.4

Para que su código sea más legible y fácil de mantener, y para evitar errores, nunca oculte los miembros de datos con nombres de variables locales.

Tipo del apuntador `this`

El tipo del apuntador `this` depende del tipo del objeto y de si la función miembro en la que se utiliza `this` se declara `const`. Por ejemplo, en una función miembro no `const` de la clase `Empleado`, el apuntador `this` tiene el tipo `Empleado *`. En una función miembro `const`, el apuntador `this` tiene el tipo de datos `const Empleado *`.

Uso implícito y explícito del apuntador `this` *para acceder a los miembros de datos de un objeto*

La figura 9.23 demuestra el uso implícito y explícito del apuntador `this` para permitir a una función miembro de la clase `Prueba` imprimir los datos `private x` de un objeto `Prueba`. En el siguiente ejemplo y en el capítulo 10, mostraremos varios ejemplos sustanciales y sutiles del uso de `this`.

```
1  // Fig. 9.23: fig09_23.cpp
2  // Uso del apuntador this para hacer referencia a los miembros de un objeto.
3  #include <iostream>
4  using namespace std;
5
6  class Prueba
7  {
8  public:
9     explicit Prueba( int = 0 ); // constructor predeterminado
10    void imprimir() const;
11 private:
12    int x;
13 }; // fin de la clase Prueba
14
15 // constructor
16 Prueba::Prueba( int valor )
17    : x( valor ) // inicializa x con valor
18 {
19    // cuerpo vacío
20 } // fin del constructor de Prueba
21
22 // imprime x usando los apuntadores this implícito y explícito;
23 // los paréntesis alrededor de *this son obligatorios
24 void Prueba::imprimir() const
25 {
26    // usa de manera implícita el apuntador this para acceder al miembro x
27    cout << "        x = " << x;
28
29    // usa de manera explícita el apuntador this y el operador flecha
30    // para acceder a la x del miembro
31    cout << "\n  this->x = " << this->x;
32
```

Fig. 9.23 | Uso del apuntador `this` para hacer referencia a los miembros de un objeto (parte 1 de 2).

```
33     // usa de manera explícita el apuntador this desreferenciado y
34     // el operador punto para acceder a la x del miembro
35     cout << "\n(*this).x = " << ( *this ).x << endl;
36   } // fin de la función imprimir
37
38   int main()
39   {
40     Prueba objetoPrueba( 12 ); // instancia e inicializa objetoPrueba
41
42     objetoPrueba.imprimir();
43   } // fin de main
```

```
        x = 12
  this->x = 12
(*this).x = 12
```

Fig. 9.23 | Uso del apuntador `this` para hacer referencia a los miembros de un objeto (parte 2 de 2).

Para fines ilustrativos, la función miembro `imprimir` (líneas 24 a 36) primero imprime `x` mediante el uso del apuntador `this` de manera *implícita* (línea 27); sólo se especifica el nombre del miembro de datos. Después, `imprimir` usa dos notaciones distintas para acceder a `x` mediante el apuntador `this`: el operador flecha (->) del apuntador `this` (línea 31) y el operador punto (.) del apuntador `this` desreferenciado (línea 35). Observe los paréntesis alrededor de `*this` (línea 35) cuando se utilizan con el operador punto (.) de selección de miembros. Los paréntesis son obligatorios debido a que el operador punto tiene *mayor* precedencia que el operador *. Sin los paréntesis, la expresión `*this.x` se evaluaría como si estuviera en paréntesis como `*(this.x)`, lo cual es un *error de compilación*, ya que el operador punto no se puede utilizar con un apuntador.

Un uso interesante del apuntador `this` es para evitar que un objeto se asigne a sí mismo. Como veremos en el capítulo 10, la *autoasignación* puede producir errores graves cuando el objeto contiene apuntadores a almacenamiento asignado en forma dinámica.

Uso del apuntador `this` para permitir llamadas en cascada a funciones

Otro uso del apuntador `this` es para permitir las **llamadas en cascada a funciones miembro**; es decir, invocar varias funciones en la misma instrucción (como en la línea 12 de la figura 9.26). El programa de las figuras 9.24 a 9.26 modifica las funciones *establecer:* `establecerTiempo`, `establecerHora`, `establecerMinuto` y `establecerSegundo` de la clase `Tiempo`, de manera que cada una devuelva una referencia a un objeto `Tiempo` para permitir las llamadas en cascada a funciones miembro. Observe en la figura 9.25 que la última instrucción en el cuerpo de cada una de estas funciones miembro devuelve `*this` (líneas 23, 34, 45 y 56) en un tipo de valor de retorno de `Tiempo &`.

El programa de la figura 9.26 crea el objeto `Tiempo` llamado `t` (línea 9), y después lo utiliza en las *llamadas en cascada a funciones miembro* (líneas 12 y 24). ¿Por qué funciona la técnica de devolver `*this` como referencia? El operador punto (.) asocia de izquierda a derecha, por lo que la línea 12 evalúa primero a `t.establecerHora(18)`, y después devuelve una referencia al objeto `t` como el valor de la llamada a esta función. Luego, el resto de la expresión se interpreta de la siguiente manera:

```
t.establecerMinuto( 30 ).establecerSegundo( 22 );
```

La llamada a `t.establecerMinuto(30)` se ejecuta y devuelve una referencia al objeto `t`. El resto de la expresión se interpreta así:

```
t.establecerSegundo( 22 );
```

```
1  // Fig. 9.24: Tiempo.h
2  // Llamadas en cascada a funciones miembro.
3
4  // Definición de la clase Tiempo.
5  // Las funciones miembro se definen en Tiempo.cpp.
6  #ifndef TIEMPO_H
7  #define TIEMPO_H
8
9  class Tiempo
10 {
11 public:
12    explicit Tiempo( int = 0, int = 0, int = 0 ); // constructor predeterminado
13
14    // funciones "establecer" (los tipos de valores de retorno Tiempo & permiten
         las llamadas en cascada)
15    Tiempo &establecerTiempo( int, int, int ); // establece hora, minuto, segundo
16    Tiempo &establecerHora( int ); // establece la hora
17    Tiempo &establecerMinuto( int ); // establece el minuto
18    Tiempo &establecerSegundo( int ); // establece el segundo
19
20    // funciones "obtener" (por lo general se declaran const)
21    unsigned int obtenerHora() const; // devuelve la hora
22    unsigned int obtenerMinuto() const; // devuelve el minuto
23    unsigned int obtenerSegundo() const; // devuelve el segundo
24
25    // funciones para imprimir (por lo general se declaran const)
26    void imprimirUniversal() const; // imprime el tiempo universal
27    void imprimirEstandar() const; // imprime el tiempo estándar
28 private:
29    unsigned int hora; // 0 - 23 (formato de reloj de 24 horas)
30    unsigned int minuto; // 0 - 59
31    unsigned int segundo; // 0 - 59
32 }; // fin de clase Tiempo
33
34 #endif
```

Fig. 9.24 | Definición de la clase `Tiempo` modificada para permitir las llamadas en cascada a funciones miembro.

```
1  // Fig. 9.25: Tiempo.cpp
2  // Definiciones de las funciones miembro de la clase Tiempo.
3  #include <iostream>
4  #include <iomanip>
5  #include <stdexcept>
6  #include "Tiempo.h" // Definición de la clase Tiempo
7  using namespace std;
8
9  // función constructor para inicializar los datos privados;
10 // llama a la función miembro establecerTiempo para establecer las variables;
11 // los valores predeterminados son 0 (vea la definición de la clase)
12 Tiempo::Tiempo( int hr, int min, int seg )
13 {
```

Fig. 9.25 | Definiciones de las funciones miembro de la clase `Tiempo`, modificadas para permitir las llamadas en cascada a funciones miembro (parte 1 de 3).

```
14    establecerTiempo( hr, min, seg );
15 } // fin del constructor de Tiempo
16
17 // establece los valores de hora, minuto y segundo
18 Tiempo &Tiempo::establcerTiempo( int h, int m, int s ) // observe Tiempo & return
19 {
20    establecerHora( h );
21    establecerMinuto( m );
22    establecerSegundo( s );
23    return *this; // permite las llamadas en cascada
24 } // fin de la función establecerTiempo
25
26 // establece el valor de hora
27 Tiempo &Tiempo::establecerHora( int h ) // observe Tiempo & return
28 {
29    if ( h >= 0 && h < 24 )
30       hora = h;
31    else
32       throw invalid_argument( "hora debe estar entre 0 y 23" );
33
34    return *this; // permite las llamadas en cascada
35 } // fin de la función establecerHora
36
37 // establece el valor de minuto
38 Tiempo &Tiempo::establecerMinuto( int m ) // observe Tiempo & return
39 {
40    if ( m >= 0 && m < 60 )
41       minuto = m;
42    else
43       throw invalid_argument( "minuto debe estar entre 0 y 59" );
44
45    return *this; // permite las llamadas en cascada
46 } // fin de la función establecerMinuto
47
48 // establece el valor de segundo
49 Tiempo &Tiempo::establecerSegundo( int s ) // observe Tiempo & return
50 {
51    if ( s >= 0 && s < 60 )
52       segundo = s;
53    else
54       throw invalid_argument( "segundo debe estar entre 0 y 59" );
55
56    return *this; // permite las llamadas en cascada
57 } // fin de la función establecerSegundo
58
59 // obtiene el valor de hora
60 unsigned int Tiempo::obtenerHora() const
61 {
62    return hora;
63 } // fin de la función obtenerHora
64
```

Fig. 9.25 | Definiciones de las funciones miembro de la clase `Tiempo`, modificadas para permitir las llamadas en cascada a funciones miembro (parte 2 de 3).

```
65  // obtiene el valor de minuto
66  unsigned int Tiempo::obtenerMinuto() const
67  {
68     return minuto;
69  } // fin de la función getMinuto
70
71  // obtiene el valor de segundo
72  unsigned int Tiempo::obtenerSegundo() const
73  {
74     return segundo;
75  } // fin de la función obtenerSegundo
76
77  // imprime el Tiempo en formato universal (HH:MM:SS)
78  void Tiempo::imprimirUniversal() const
79  {
80     cout << setfill( '0' ) << setw( 2 ) << hora << ":"
81        << setw( 2 ) << minuto << ":" << setw( 2 ) << segundo;
82  } // fin de la función imprimirUniversal
83
84  // imprime el Tiempo en formato estándar (HH:MM:SS AM o PM)
85  void Tiempo::imprimirEstandar() const
86  {
87     cout << ( ( hora == 0 || hora == 12 ) ? 12 : hora % 12 )
88        << ":" << setfill( '0' ) << setw( 2 ) << minuto
89        << ":" << setw( 2 ) << segundo << ( hora < 12 ? " AM" : " PM" );
90  } // fin de la función imprimirEstandar
```

Fig. 9.25 | Definiciones de las funciones miembro de la clase `Tiempo`, modificadas para permitir las llamadas en cascada a funciones miembro (parte 3 de 3).

En la línea 24 (figura 9.26) también se usan las llamadas en cascada. Cabe mencionar que no podemos encadenar otra llamada a una función miembro después de `imprimirEstandar` aquí, ya que `imprimirEstandar` *no* devuelve una referencia a `t`. Al colocar la llamada a `imprimirEstandar` antes de la llamada a `establecerTiempo` en la línea 24, se produce un error de compilación. En el capítulo 10 presentaremos varios ejemplos prácticos acerca del uso de llamadas en cascada a funciones. Uno de esos ejemplos utiliza varios operadores << con `cout` para imprimir múltiples valores en una sola instrucción.

```
1  // Fig. 9.26: fig09_26.cpp
2  // Llamadas en cascada a funciones miembro con el apuntador this.
3  #include <iostream>
4  #include "Tiempo.h" // definición de la clase Tiempo
5  using namespace std;
6
7  int main()
8  {
9     Tiempo t; // crea un objeto Tiempo
10
11    // llamadas en cascada a funciones
12    t.establecerHora( 18 ).establecerMinuto( 30 ).establecerSegundo( 22 );
13
```

Fig. 9.26 | Llamadas en cascada a funciones miembro con el apuntador `this` (parte 1 de 2).

```
14     // imprime el tiempo en los formatos universal y estándar
15     cout << "Tiempo universal: ";
16     t.imprimirUniversal();
17
18     cout << "\nTiempo estandar: ";
19     t.imprimirEstandar();
20
21     cout << "\n\nNuevo tiempo estandar: ";
22
23     // llamadas en cascada a funciones
24     t.establecerTiempo( 20, 20, 20 ).imprimirEstandar();
25     cout << endl;
26  } // fin de main
```

```
Tiempo universal: 18:30:22
Tiempo estandar: 6:30:22 PM

Nuevo tiempo estandar: 8:20:20 PM
```

Fig. 9.26 | Llamadas en cascada a funciones miembro con el apuntador `this` (parte 2 de 2).

9.14 Miembros de clase `static`

Hay una importante excepción a la regla que establece que cada objeto de una clase tiene su propia copia de todos los miembros de datos de la misma. En ciertos casos, *todos* los objetos de una clase sólo deben *compartir una* copia de una variable. Por ésta y otras razones, se utiliza un **miembro de datos `static`**. Dicha variable representa información "a nivel de clase" (es decir, datos compartidos por *todas* las instancias y que *no* son específicos para alguno de los objetos de la clase). Por ejemplo, recuerde que las versiones de la clase `LibroCalificaciones` en el capítulo 7 utilizan miembros de datos `static` para almacenar constantes que representan el número de calificaciones que pueden contener todos los objetos `LibroCalificaciones`.

Motivación de datos a nivel de clase

Vamos a motivar más la necesidad de datos `static` a nivel de clase con un ejemplo. Suponga que tenemos un videojuego con objetos `Marciano` y otras criaturas espaciales. Cada `Marciano` tiende a ser valiente y deseoso de atacar a otras criaturas espaciales cuando está consciente de que hay por lo menos cinco objetos `Marciano` presentes. Si hay menos de cinco, cada `Marciano` se vuelve cobarde. Por lo tanto, cada `Marciano` necesita conocer la `cuentaDeMarcianos`. Podríamos investir a cada instancia de la clase `Marciano` con `cuentaDeMarcianos` como miembro de datos. Si lo hacemos, cada `Marciano` tendrá una copia *separada* del miembro de datos. Cada vez que creemos un nuevo `Marciano`, tendremos que actualizar el miembro de datos `cuentaDeMarcianos` en todos los objetos `Marciano`. Para ello cada objeto `Marciano` tendría que tener acceso a los manejadores de todos los demás objetos `Marciano` en la memoria. Esto desperdicia espacio con las copias redundantes, y tiempo para actualizar las copias separadas. En vez de ello, declaramos a `cuentaDeMarcianos` como `static`. Esto convierte a `cuentaDeMarcianos` en datos a nivel de clase. Cada `Marciano` puede acceder a `cuentaDeMarcianos` como si fuera un miembro de datos del `Marciano`, pero C++ *sólo* mantiene una copia de la variable `static` `cuentaDeMarcianos`. Esto ahorra espacio. Ahorramos tiempo al hacer que el constructor de `Marciano` incremente la variable `static cuentaDeMarcianos`, y al hacer que el destructor de `Marciano` decremente `cuentaDeMarcianos`. Como sólo hay una copia, no tenemos que incrementar o decrementar copias separadas de `cuentaDeMarcianos` para cada objeto `Marciano`.

Tip de rendimiento 9.5

Use datos miembro `static` *para ahorrar almacenamiento cuando sea suficiente con una sola copia de los datos para todos los objetos de una clase.*

Alcance e inicialización de miembros de datos `static`

Los miembros de datos `static` de una clase tienen *alcance de clase*. Un miembro de datos `static` se *debe* inicializar *sólo* una vez. Los miembros de datos `static` de tipo fundamental se inicializan de manera predeterminada con `0`. Antes de C++11, un miembro de datos `static const` de tipo `int` o `enum` podía inicializarse en su declaración en la definición de la clase, y todos los demás miembros de datos `static` debían definirse en *alcance de espacio de nombres global* (es decir, fuera del cuerpo de la definición de la clase). De nuevo, los inicializadores dentro de las class en C++11 también le permiten inicializar estas variables en donde se declaren en la definición de la clase. Si un miembro de datos `static` es un objeto de una clase que proporciona un constructor predeterminado, el miembro de datos `static` no necesita inicializarse debido a que se llamará a su constructor predeterminado.

Acceso a los miembros de datos `static`

Por lo general, se accede a los miembros `static private` y `protected` de una clase a través de las funciones miembro `public` de la misma, o a través de funciones `friend` de la clase. *Los miembros* `static` *de una clase existen aún y cuando no existan objetos de la clase*. Para acceder al miembro `public static` de una clase cuando no existen objetos de esa clase, simplemente se antepone el nombre de la clase y el operador de resolución de ámbito (`::`) al nombre del miembro de datos. Por ejemplo, si nuestra variable anterior `cuentaDeMarcianos` es `public`, se puede utilizar con la expresión `Marciano::cuentaDeMarcianos`, aun cuando no haya objetos `Marciano`. (Desde luego que no se recomienda el uso de datos `public`.)

Para acceder a un miembro `private` o `protected static` de la clase cuando *no* existen objetos de la misma, el programador debe proporcionar una **función miembro `static` `public`** y debe llamar a la función, anteponiendo a su nombre el nombre de la clase y el operador de resolución de ámbito. Una función miembro `static` es un servicio de la *clase*, *no* un *objeto* específico de la misma.

Observación de Ingeniería de Software 9.14

Los datos miembro `static` *y las funciones miembro* `static` *de una clase existen, y se pueden usar aún si no se han instanciado objetos de esa clase.*

Demostración de los miembros de datos `static`

El programa de las figuras 9.27 a 9.29 demuestra el uso de un miembro de datos `private static` llamado `cuenta` (figura 9.27, línea 24) y una función miembro `public static` llamada `obtenerCuenta` (figura 9.27, línea 18). En la figura 9.28, la línea 8 define e inicializa el miembro de datos `cuenta` con cero en *alcance de espacio de nombres global*, y en las líneas 12 a 15 se define la función miembro `static` llamada `obtenerCuenta`. Observe que ni la línea 8 ni la 12 incluyen la palabra clave `static`, pero de todas formas ambas líneas se refieren a los miembros `static` de la clase. La palabra clave `static` no puede aplicarse a la definición de un miembro que aparezca fuera de la definición de la clase. El miembro de datos `cuenta` mantiene un conteo del número de objetos de la clase `Empleado` que se han instanciado. Cuando existen objetos de la clase `Empleado`, el miembro `cuenta` se puede referenciar a través de *cualquier* función miembro de un objeto `Empleado`; en la figura 9.28, se hace referencia a `cuenta` tanto en la línea 22 en el constructor como en la línea 32 en el destructor.

```
1  // Fig. 9.27: Empleado.h
2  // Definición de la clase Empleado con un miembro de datos static para
3  // rastrear el número de objetos Empleado en la memoria
4  #ifndef EMPLEADO_H
5  #define EMPLEADO_H
6
7  #include <string>
8
9  class Empleado
10 {
11 public:
12    Empleado( const std::string &, const std::string & ); // constructor
13    ~Empleado(); // destructor
14    std::string obtenerPrimerNombre() const; // devuelve el primer nombre
15    std::string obtenerApellidoPaterno() const; // devuelve el apellido paterno
16
17    // función miembro static
18    static unsigned int obtenerCuenta(); // devuelve el número de objetos instanciados
19 private:
20    std::string primerNombre;
21    std::string apellidoPaterno;
22
23    // datos static
24    static unsigned int cuenta; // número de objetos instanciados
25 }; // fin de la clase Empleado
26
27 #endif
```

Fig. 9.27 | Definición de la clase `Empleado` con un miembro de datos `static` para rastrear el número de objetos `Empleado` en la memoria.

```
1  // Fig. 9.28: Empleado.cpp
2  // Definiciones de las funciones miembro de la clase Empleado.
3  #include <iostream>
4  #include "Empleado.h" // definición de la clase Empleado
5  using namespace std;
6
7  // define e inicializa el miembro de datos static en alcance de espacio
   //    de nombres global
8  unsigned int Empleado::cuenta = 0; // no puede incluir la palabra clave static
9
10 // define la función miembro static que devuelve el número de
11 // objetos Empleado instanciados (se declara static en Empleado.h)
12 unsigned int Empleado::obtenerCuenta()
13 {
14    return cuenta;
15 } // fin de la función static obtenerCuenta
16
17 // el constructor inicializa los miembros de datos no static e
18 // incrementa el miembro de datos static cuenta
19 Empleado::Empleado( const string &nombre, const string &apellido )
20    : primerNombre( nombre ), apellidoPaterno( apellido )
21 {
```

Fig. 9.28 | Definiciones de las funciones miembro de la clase `Empleado` (parte 1 de 2).

```
22    ++cuenta; // incrementa la cuenta static de empleados
23    cout << "Se llamo al constructor de Empleado para " << primerNombre
24       << ' ' << apellidoPaterno << "." << endl;
25 } // fin del constructor de Empleado
26
27 // el destructor desasigna la memoria asignada en forma dinámica
28 Empleado::~Empleado()
29 {
30    cout << "Se llamo a ~Empleado() para " << primerNombre
31       << ' ' << apellidoPaterno << endl;
32    --cuenta; // decrementa cuenta static de empleados
33 } // fin del destructor ~Empleado
34
35 // devuelve el primer nombre del empleado
36 string Empleado::obtenerPrimerNombre() const
37 {
38    return primerNombre; // devuelve copia del primer nombre
39 } // fin de la función obtenerPrimerNombre
40
41 // devuelve el apellido paterno del empleado
42 string Empleado::obtenerApellidoPaterno() const
43 {
44    return apellidoPaterno; // devuelve copia del apellido paterno
45 } // fin de la función obtenerApellidoPaterno
```

Fig. 9.28 | Definiciones de las funciones miembro de la clase `Empleado` (parte 2 de 2).

La figura 9.29 utiliza la función miembro `static` `getCuenta` para determinar el número de objetos `Empleado` en memoria en diversos puntos en el programa. El programa llama a `Empleado::obtenerCuenta()` antes de que se hayan creado objetos `Empleado` (línea 12), después de crear dos objetos `Empleado` (línea 23) y después de haber destruido a esos objetos `Empleado` (línea 34). Las líneas 16 a 29 en `main` definen un *alcance anidado*. Recuerde que las variables locales existen hasta que termina el alcance en el que están definidas. En este ejemplo, creamos dos objetos `Empleado` en las líneas 17 y 18, dentro del alcance anidado. A medida que se ejecuta cada constructor, se incrementa el miembro de datos `static` `cuenta` de la clase `Empleado`. Estos objetos `Empleado` se destruyen cuando el programa llega a la línea 29. En ese punto se ejecuta el destructor de cada objeto y se decrementa el miembro de datos `static` `count` de la clase `Empleado`.

```
1  // Fig. 9.29: fig09_29.cpp
2  // Miembro de datos static que rastrea el número de objetos de una clase.
3  #include <iostream>
4  #include "Empleado.h" // definición de la clase Empleado
5  using namespace std;
6
7  int main()
8  {
9     // no existen objetos: usa el nombre de la clase y el operador de resolución de
         ámbito binario
10    // para acceder a la función miembro static obtenerCuenta
11    cout << "El numero de empleados antes de instanciar cualquier objeto es "
12       << Empleado::obtenerCuenta() << endl; // usa el nombre de la clase
```

Fig. 9.29 | Miembro de datos `static` que rastrea el número de objetos de una clase (parte 1 de 2).

```
13
14    // el siguiente alcance crea y destruye
15    // objetos Empleado antes de que termine main
16    {
17       Empleado e1( "Susan", "Baker" );
18       Empleado e2( "Robert", "Jones" );
19
20       // existen dos objetos; llama a la función miembro static obtenerCuenta de
            nuevo
21       // usando el nombre de la clase y el operador de resolución de ámbito
22       cout << "El numero de empleados despues de instanciar los objetos es "
23          << Empleado::obtenerCuenta();
24
25       cout << "\n\nEmpleado 1: "
26          << e1.obtenerPrimerNombre() << " " << e1.obtenerApellidoPaterno()
27          << "\nEmpleado 2: "
28          << e1.obtenerPrimerNombre() << " " << e2.obtenerApellidoPaterno()
            << "\n\n";
29    } // fin del alcance anidado en main
30
31    // no existen objetos, por lo que llama a la función miembro static
         obtenerCuenta de nuevo
32    // usando el nombre de la clase y el operador de resolución de ámbito
33    cout << "El numero de empleados despues de eliminar los objetos es "
34       << Empleado::obtenerCuenta() << endl;
35 } // fin de main
```

```
El número de empleados antes de instanciar cualquier objeto es 0
Se llamo al constructor de Empleado para Susan Baker.
Se llamo al constructor de Empleado para Robert Jones.
El numero de empleados despues de instanciar los objetos es 2

Empleado 1: Susan Baker
Empleado 2: Robert Jones

Se llamo a ~Empleado() para Susan Baker
Se llamo a ~Empleado() para Robert Jones

El numero de empleados despues de eliminar los objetos es 0
```

Fig. 9.29 | Miembro de datos `static` que rastrea el número de objetos de una clase (parte 2 de 2).

Una función miembro debe declararse como `static` si *no* accede a los miembros de datos no `static` o a las funciones miembro no `static` de la clase. A diferencia de las funciones miembro no `static`, *una función miembro* `static` *no tiene un apuntador* `this`, ya que *los miembros de datos* `static` *y las funciones miembro* `static` *existen en forma independiente de cualquier objeto de una clase*. El apuntador `this` *debe* hacer referencia a un *objeto* específico de la clase, y cuando se hace la llamada a una función miembro `static`, tal vez *no* haya ningún objeto de su clase en la memoria.

Error común de programación 9.6

Utilizar el apuntador `this` *en una función miembro* `static` *es un error de compilación.*

Error común de programación 9.7

Declarar `const` *una función miembro* `static` *es un error de compilación. El calificador* `const` *indica que una función no puede modificar el contenido del* objeto *en el que opera, pero las funciones miembro* `static` *existen y operan en forma independiente de los objetos de la clase.*

9.15 Conclusión

En este capítulo profundizamos nuestro conocimiento acerca de las clases, usando un caso de estudio con la clase `Tiempo` para presentar varias nuevas características. Usamos una guardia de inclusión para evitar que el código en un archivo de encabezado (`.h`) se incluyera varias veces en el mismo archivo de código fuente (`.cpp`). Aprendió a usar el operador flecha para acceder a los miembros de un objeto por medio de un apuntador del tipo de la clase del objeto. Aprendió que las funciones miembro tienen alcance de clase: el nombre de la función miembro sólo es conocido para los demás miembros de la clase, a menos que un cliente de la clase haga referencia a la función miembro a través del nombre de un objeto, de una referencia a un objeto de la clase, un apuntador a un objeto de la clase o el operador de resolución de ámbito. También hablamos sobre las funciones de acceso (que se utilizan comúnmente para obtener los valores de los miembros de datos, o para evaluar la veracidad o falsedad de las condiciones) y las funciones utilitarias (funciones miembro `private` que soportan la operación de las funciones miembro `public` de la clase).

Aprendió además que un constructor puede especificar argumentos predeterminados que le permitan ser llamado en una variedad de formas. También aprendió que cualquier constructor que se pueda llamar sin argumentos es un constructor predeterminado, y que puede haber a lo más un constructor predeterminado por clase. Hablamos sobre los destructores para realizar tareas de mantenimiento de terminación en un objeto de una clase, antes de destruir ese objeto, y demostramos el orden en el que se llaman los constructores y destructores de un objeto.

Demostramos los problemas que pueden ocurrir cuando una función miembro devuelve una referencia o un apuntador a un miembro de datos `private`, lo cual quebranta el encapsulamiento de la clase. Mostramos además que los objetos del mismo tipo se pueden asignar entre sí, usando la asignación predeterminada a nivel de miembros. En el capítulo 10 hablaremos sobre cómo puede esto provocar problemas cuando un objeto contiene miembros apuntadores.

Aprendió a especificar objetos `const` y funciones miembro `const` para evitar modificaciones a los objetos, con lo cual se hace valer el principio del menor privilegio. También aprendió que, mediante la composición, una clase puede tener objetos de otras clases como miembros. Demostramos cómo usar las funciones `friend`.

Aprendió que el apuntador `this` se pasa como un argumento implícito a cada una de las funciones miembro no `static` de una clase, lo cual permite a las funciones acceder a los miembros de datos del objeto correcto, junto con otras funciones miembro no `static`. También vimos el uso explícito del apuntador `this` para acceder a los miembros de una clase y permitir las llamadas a funciones miembro en cascada. Motivamos la noción de los miembros de datos y las funciones miembro `static`, y demostramos cómo puede declararlos y usarlos en sus propias clases.

En el capítulo 10, continuaremos nuestro estudio sobre las clases y los objetos, al mostrarle cómo permitir que los operadores de C++ funcionen con *objetos de tipo clase*; a este proceso se le conoce como *sobrecarga de operadores*. Por ejemplo, veremos cómo sobrecargar el operador << de manera que se pueda utilizar para imprimir un arreglo completo, sin utilizar de manera explícita una instrucción de repetición.

Resumen

Sección 9.2 Caso de estudio con la clase `Tiempo`

- Las directivas del preprocesador `#ifndef` (que significa "si no está definido"; pág. 380) y `#endif` (pág. 380) se utilizan para evitar múltiples inclusiones de un encabezado. Si el código entre estas directivas no se ha incluido antes en una aplicación, `#define` (pág. 380) define un nombre que se puede utilizar para evitar futuras inclusiones, y el código se incluye en el archivo de código fuente.
- Antes de C++11, sólo los miembros de datos `static const int` podían inicializarse en donde se declaraban en el cuerpo de la clase. Por esta razón, lo recomendable es que estos miembros de datos se inicialicen mediante el constructor de la clase. A partir de C++11, ahora es posible usar un inicializador dentro de las clases, para inicializar cualquier miembro de datos en donde se declare en la definición de la clase.

- Las funciones de una clase pueden lanzar (pág. 381) excepciones (como `invalid_argument`; pág. 381) para indicar datos inválidos.
- El manipulador de flujo `setfill` (pág. 382) especifica el carácter de relleno (pág. 382) a mostrar cuando se imprime un entero en un campo que sea más ancho que el número de dígitos en el valor.
- Si una función miembro define una variable con el mismo nombre que una variable con alcance de clase (pág. 382), la variable con alcance de bloque oculta a la variable con alcance de clase en la función.
- De manera predeterminada, los caracteres de relleno aparecen antes de los dígitos en el número.
- El manipulador de flujo `setfill` es una opción "pegajosa", lo cual significa que una vez que se establece el carácter de relleno, se aplica para todos los campos subsiguientes que se impriman.
- Aun cuando una función miembro declarada en una definición de clase pueda definirse fuera de la definición de esa clase (y "enlazarse" a la clase a través del operador de resolución de ámbito), esa función miembro sigue dentro del alcance de esa clase.
- Si una función miembro se define en el cuerpo de una definición de clase, la función miembro se declara implícitamente en línea.
- Las clases pueden incluir objetos de otras clases como miembros, o pueden derivarse (pág. 385) de otras clases que proporcionen atributos y comportamientos que puedan usar las nuevas clases.

Sección 9.3 Alcance de las clases y acceso a los miembros de una clase

- Los miembros de datos de una clase y las funciones miembro pertenecen al alcance de esa clase.
- Las funciones que no son miembro se definen en alcance de espacio de nombres global.
- Dentro del alcance de una clase, los miembros de ésta son inmediatamente accesibles para todas las funciones miembro de esa clase, y se puede hacer referencia a ellos por su nombre.
- Fuera del alcance de una clase, los miembros de ésta se referencian a través de uno de los manejadores en un objeto: el nombre de un objeto, una referencia a un objeto o un apuntador a un objeto.
- Las variables que se declaran en una función miembro tienen alcance de bloque, y sólo esa función las conoce.
- Al operador punto (`.`) de selección de miembros se le antepone el nombre de un objeto o una referencia al objeto para acceder a los miembros `public` del objeto.
- Al operador flecha (`->`; pág. 386) de selección de miembros se le antepone un apuntador a un objeto para acceder a los miembros `public` de ese objeto.

Sección 9.4 Funciones de acceso y funciones utilitarias

- Las funciones de acceso (pág. 386) leen o muestran datos. También pueden usarse para evaluar la veracidad o falsedad de ciertas condiciones; a dichas funciones se les conoce comúnmente como funciones predicado.
- Una función utilitaria (pág. 386) es una función miembro `private` que soporta la operación de las funciones miembro `public` de la clase. Las funciones utilitarias no están diseñadas para que las utilicen los clientes de una clase.

Sección 9.5 Caso de estudio de la clase `Tiempo`: constructores con argumentos predeterminados

- Al igual que otras funciones, los constructores pueden especificar argumentos predeterminados.

Sección 9.6 Destructores

- El destructor de una clase (pág. 393) se llama de manera implícita cuando se destruye un objeto de la clase.
- El nombre del destructor para una clase es el carácter tilde (~) seguido del nombre de la clase.
- Un destructor no libera el almacenamiento de un objeto; realiza tareas de mantenimiento de terminación (pág. 393) antes de que el sistema reclame la memoria de un objeto, de manera que ésta se pueda reutilizar para contener nuevos objetos.
- Un destructor no recibe parámetros y no devuelve valores. Una clase sólo puede tener un destructor.

- Si no se proporciona un destructor de manera explícita, el compilador crea un destructor "vacío", de manera que cada clase tiene sólo un destructor.

Sección 9.7 Cuándo se hacen llamadas a los constructores y destructores

- El orden en el que se llaman los constructores y destructores depende del orden en el que la ejecución entra y sale de los alcances en los que se instancian los objetos.
- Por lo general, las llamadas al destructor se realizan en orden inverso a las llamadas correspondientes al constructor, pero las clases de almacenamiento de los objetos pueden alterar el orden en el que se llama a los destructores.

Sección 9.8 Caso de estudio con la clase `Tiempo`: una trampa sutil (devolver una referencia a un miembro de datos `private`)

- Una referencia a un objeto es un alias para el nombre del objeto y, por ende, se puede usar del lado izquierdo de una instrucción de asignación. En este contexto, la referencia se convierte en un *lvalue* perfectamente aceptable que puede recibir un valor.
- Si la función devuelve una referencia a datos `const`, entonces no puede usarse como *lvalue* modificable.

Sección 9.9 Asignación predeterminada a nivel de miembros

- El operador de asignación (=) se puede utilizar para asignar un objeto a otro del mismo tipo. De manera predeterminada, dicha asignación se realiza mediante la asignación a nivel de miembros (pág. 400).
- Los objetos se pueden pasar por valor o se pueden devolver de las funciones por valor. C++ crea un nuevo objeto y utiliza un constructor de copia (pág. 401) para copiar los valores del objeto original al nuevo objeto.
- Para cada clase, el compilador proporciona un constructor de copia predeterminado que copia cada miembro del objeto original en el miembro correspondiente del nuevo objeto.

Sección 9.10 Objetos `const` y funciones miembro `const`

- La palabra clave `const` se puede usar para especificar que un objeto no puede modificarse, y que cualquier intento por modificar el objeto debe producir un error de compilación.
- Los compiladores de C++ no permiten llamadas a funciones miembro no `const` en objetos `const`.
- El intento de una función miembro `const` por modificar un objeto de su clase es un error de compilación.
- Una función miembro se especifica como `const`, tanto en su prototipo como en su definición.
- Un objeto `const` se debe inicializar.
- Los constructores y destructores no se pueden declarar `const`.

Sección 9.11 Composición: objetos como miembros de clases

- Una clase puede tener objetos de otras clases como miembros; a este concepto se le conoce como composición.
- Los objetos miembro se construyen en el orden en el que se declaran en la definición de la clase y antes de construir los objetos de su clase circundante.
- Si no se proporciona un inicializador de miembros para un objeto miembro, se hará una llamada implícita al constructor predeterminado del objeto miembro (pág. 404).

Sección 9.12 Funciones `friend` y clases `friend`

- Una función `friend` (pág. 410) de una clase se define fuera del alcance de esa clase, pero aun así tiene el derecho de acceder a todos los miembros de la clase. Pueden declararse funciones independientes o clases completas como amigas de otras clases.
- Una declaración `friend` puede aparecer en cualquier lugar dentro de una clase.
- La relación de amistad no es simétrica ni transitiva.

Sección 9.13 Uso del apuntador `this`

- Todo objeto tiene acceso a su propia dirección a través del apuntador `this` (pág. 412).
- El apuntador `this` de un objeto no forma parte del objeto en sí; es decir, el tamaño de la memoria ocupada por el apuntador `this` no se refleja en el resultado de una operación `sizeof` en el objeto.
- El apuntador `this` se pasa como un argumento implícito para cada una de las funciones miembro no `static`.
- Los objetos usan el apuntador `this` de manera implícita (como se ha hecho hasta este momento), o de manera explícita para hacer referencia a sus miembros de datos y funciones miembro.
- El apuntador `this` permite llamadas en cascada a funciones miembro (pág. 414), en donde se invocan varias funciones en la misma instrucción.

Sección 9.14 Miembros de clase `static`

- Un miembro de datos `static` (pág. 418) representa información a "nivel de clase" (es decir, una propiedad de la clase compartida por todas las instancias, no una propiedad de un objeto específico de la clase).
- Los miembros de datos `static` tienen alcance de clase y se pueden declarar como `public`, `private` o `protected`.
- Los miembros `static` de una clase existen aún y cuando no existan objetos de esa clase.
- Para acceder a un miembro de clase `public static` cuando no existen objetos de la clase, simplemente hay que anteponer el nombre de clase y el operador de resolución de ámbito (`::`) al nombre del miembro de datos.
- La palabra clave `static` no puede aplicarse a la definición de un miembro que aparezca fuera de la definición de esa clase.
- Una función miembro debe declararse como `static` (pág. 419) si no accede a los miembros de datos no `static` o a las funciones miembro no `static` de la clase. A diferencia de las funciones miembro no `static`, una función miembro `static` no tiene un apuntador `this`, ya que los miembros de datos y las funciones miembro `static` existen de manera independiente de cualquier objeto de una clase.

Ejercicios de autoevaluación

9.1 Complete los siguientes enunciados:

a) Los miembros de una clase se utilizan mediante el operador _____ en conjunción con el nombre de un objeto (o referencia a un objeto) de la clase, o a través del operador _____ en conjunción con un apuntador a un objeto de la clase.
b) Los miembros de una clase que se especifican como _____ están accesibles sólo para las funciones miembro de la clase y sus funciones amigas.
c) Los miembros de una clase que se especifican como _____ están accesibles en cualquier parte en la que un objeto de la clase se encuentre dentro del alcance.
d) _____ se puede utilizar para asignar un objeto de una clase a otro objeto de la misma clase.
e) Una función no miembro se debe declarar como _____ de una clase para tener acceso a los miembros de datos `private` de esa clase.
f) Un objeto constante debe _____; no se puede modificar después de crearlo.
g) Un miembro de datos _____ representa la información a nivel de clase.
h) Las funciones miembro no `static` de un objeto tienen acceso a un "autoapuntador" al objeto, conocido como apuntador _____.
i) La palabra clave _____ especifica que un objeto o variable no puede modificarse.
j) Si no se proporciona un inicializador de miembros para un objeto miembro de una clase, se hace una llamada al _____ del objeto.
k) Una función miembro debe declararse `static` si no accede a los miembros de datos _____.
l) Los objetos miembro se construyen _____ del objeto de su clase circundante.

9.2 Busque el (los) error(es) en cada uno de los siguientes incisos, y explique cómo corregirlo(s).

a) Suponga que se declara el siguiente prototipo en la clase `Tiempo`:

```
void ~Tiempo( int );
```

b) Suponga que se declara el siguiente prototipo en la clase `Empleado`:

```
int Empleado( string, string );
```

c) La siguiente es una definición de la clase `Ejemplo`:

```
class Ejemplo
{
public:
   Ejemplo( int y = 10 )
      : datos( y )
   {
      // cuerpo vacío
   } // fin del constructor de Ejemplo

   int obtenerDatosIncrementados() const
   {
      return ++datos;
   } // fin de la función obtenerDatosIncrementados

   static int obtenerCuenta()
   {
      cout << "Los datos son " << data << endl;
      return cuenta;
   } // fin de la función obtenerCuenta
private:
   int datos;
   static int cuenta;
}; // fin de la clase Ejemplo
```

Respuestas a los ejercicios de autoevaluación

9.1 a) punto (.), flecha (->). b) `private`. c) `public`. d) La asignación predeterminada a nivel de miembros (realizada por el operador de asignación). e) `friend`. f) inicializarse. g) `static`. h) `this`. i) `const`. j) constructor predeterminado. k) no `static`. l) antes.

9.2 a) *Error:* los destructores no pueden devolver valores (o incluso especificar un tipo de valor de retorno) ni recibir argumentos.
Corrección: elimine el tipo de valor de retorno `void` y el parámetro `int` de la declaración.

b) *Error:* los constructores no pueden devolver valores.
Corrección: elimine el tipo de retorno `int` de la declaración.

c) *Error:* la definición de clase para `Ejemplo` tiene dos errores. El primero ocurre en la función `obtenerDatosIncrementados`. La función se declara `const`, pero modifica el objeto.
Corrección: para corregir el primer error, elimine la palabra clave `const` de la definición de `obtenerDatosIncrementados`. [*Nota:* también hubiera sido apropiado cambiar el nombre de la función miembro, ya que por lo general las funciones *obtener* son funciones miembro `const`].
Error: el segundo error ocurre en la función `obtenerCuenta`. Esta función se declara como `static`, por lo que no puede acceder a ningún miembro no `static` (es decir, datos) de la clase.
Corrección: para corregir el segundo error, elimine la línea de salida de la definición de `obtenerCuenta`.

Ejercicios

9.3 *(Operador de resolución de ámbito)* ¿Cuál es el propósito del operador de resolución de ámbito?

9.4 *(Mejora de la clase `Tiempo`)* Proporcione un constructor que sea capaz de usar el tiempo actual de las funciones `time` y `localtime` (declaradas en el encabezado `<ctime>` de la Biblioteca estándar de C++) para inicializar un objeto de la clase `Tiempo`.

9.5 *(Clase `Complejo`)* Cree una clase llamada `Complejo` para realizar operaciones aritméticas con números complejos. Escriba un programa para evaluar su clase. Los números complejos tienen la forma

```
parteReal + parteImaginaria * i
```

en donde *i* es

$$\sqrt{-1}$$

Use variables `double` para representar los datos `private` de la clase. Proporcione un constructor que permita a un objeto de la clase inicializarse al momento de ser declarado. El constructor debe contener valores predeterminados en caso de que no se proporcionen inicializadores. Proporcione funciones miembro `public` que realicen las siguientes tareas:

a) Sumar dos números `Complejo`: las partes reales se suman entre sí y las partes imaginarias se suman entre sí.
b) Restar dos números `Complejo`: la parte real del operando derecho se resta de la parte real del operando izquierdo, y la parte imaginaria del operando derecho se resta de la parte imaginaria del operando izquierdo.
c) Imprimir números `Complejo` de la forma `(a, b)`, en donde `a` es la parte real y `b` es la parte imaginaria.

9.6 *(Clase `Racional`)* Cree una clase llamada `Racional` para realizar operaciones aritméticas con fracciones. Escriba un programa para evaluar su clase. Use variables enteras para representar los datos `private` de la clase: el `numerador` y el `denominador`. Proporcione un constructor que permita a un objeto de esta clase inicializarse cuando se declare. El constructor debe contener valores predeterminados, en caso de que no se proporcionen inicializadores, y debe almacenar la fracción en forma reducida. Por ejemplo, la fracción

$$\frac{2}{4}$$

se almacenaría en el objeto como `1` en el `numerador` y `2` en el `denominador`. Proporcione funciones miembro `public` que realicen cada una de las siguientes tareas:

a) Sumar dos números `Racional`. El resultado debe almacenarse en forma reducida.
b) Restar dos números `Racional`. El resultado debe almacenarse en forma reducida.
c) Multiplicar dos números `Racional`. El resultado debe almacenarse en forma reducida.
d) Dividir dos números `Racional`. El resultado debe almacenarse en forma reducida.
e) Imprimir números `Racional` en la forma `a/b`, en donde `a` es el numerador y `b` es el denominador.
f) Imprimir números `Racional` en formato de punto flotante.

9.7 *(Mejora a la clase `Tiempo`)* Modifique la clase `Tiempo` de las figuras 9.4 y 9.5 para incluir una función miembro `tictac`, que incremente el tiempo almacenado en un objeto `Tiempo` por un segundo. Escriba un programa para probar la función miembro `tictac` en un ciclo que imprima la hora en formato estándar durante cada iteración del ciclo, para ilustrar que la función miembro funcione correctamente. Asegúrese de evaluar los siguientes casos:

a) Incrementar el minuto, de manera que cambie al siguiente minuto.
b) Incrementar la hora, de manera que cambie a la siguiente hora.
c) Incrementar el tiempo de manera que cambie al siguiente día (por ejemplo, de 11:59:59 PM a 12:00:00 AM).

9.8 *(Mejora a la clase `Fecha`)* Modifique la clase `Fecha` de las figuras 9.13 y 9.14 para realizar la comprobación de errores en los valores inicializadores para los miembros de datos `mes`, `dia` y `anio`. Además, proporcione una función llamada `siguienteDia` para incrementar el día en uno. Escriba un programa que evalúe la función `siguienteDia` en un ciclo que imprima la fecha durante cada iteración del ciclo, para mostrar que `siguienteDia` funciona correctamente. Asegúrese de evaluar los siguientes casos:

a) Incrementar la fecha de manera que cambie al siguiente mes.
b) Incrementar la fecha de manera que cambie al siguiente año.

9.9 *(Combinar la clase `Tiempo` y la clase `Fecha`)* Combine la clase `Tiempo` modificada del ejercicio 9.7 y la clase `Fecha` modificada del ejercicio 9.8 en una sola clase llamada `FechaYHora`. (En el capítulo 11 hablaremos sobre la herencia, que nos permitirá realizar esta tarea rápidamente sin tener que modificar las definiciones de las clases existentes). Modifique la función `tictac` para que llame a la función `siguienteDia` si el tiempo se incrementa y cambia al siguiente día. Modifique las funciones `imprimirEstandar` e `imprimirUniversal` para imprimir la fecha y hora. Escriba un programa para evaluar la nueva clase `FechaYHora`. En específico, pruebe a incrementar el tiempo para que cambie al siguiente día.

9.10 *(Devolver indicadores de error de las funciones establecer de la clase `Tiempo`)* Modifique las funciones *establecer* en la clase `Tiempo` de las figuras 9.4 y 9.5 para que devuelvan valores de error apropiados si hay un intento de *establecer* un miembro de datos de un objeto de la clase `Tiempo` en un valor inválido. Escriba un programa que evalúe su nueva versión de la clase `Tiempo`. Muestre mensajes de error cuando las funciones *establecer* devuelvan los valores de error.

9.11 *(Clase `Rectangulo`)* Cree una clase `Rectangulo` con los atributos `longitud` y `anchura`, cada una de las cuales tiene un valor predeterminado de 1. Proporcione funciones miembro que calculen el `perimetro` y el `area` del rectángulo. Además, proporcione funciones *establecer* y *obtener* para los atributos `longitud` y `anchura`. Las funciones *establecer* deben verificar que `longitud` y `anchura` sean números de punto flotante mayores que 0.0 y menores que 20.0.

9.12 *(Clase `Rectangulo` mejorada)* Cree una clase `Rectangulo` más sofisticada que la del ejercicio 9.11. Esta clase sólo almacena las coordenadas cartesianas de las cuatro esquinas del rectángulo. El constructor llama a una función *establecer* que acepta cuatro conjuntos de coordenadas y verifica que cada una de éstas se encuentre en el primer cuadrante, en donde ninguna coordenada *x* o *y* individual debe ser mayor que 20.0. La función *establecer* también verifica que las coordenadas suministradas especifiquen, de hecho, un rectángulo. Proporcione funciones miembro que calculen los valores de `longitud`, `anchura`, `perimetro` y `area`. La longitud es el valor mayor de las dos dimensiones. Incluya una función predicado llamada `cuadrado` que determine si el rectángulo es un `cuadrado`.

9.13 *(Clase `Rectangulo` mejorada)* Modifique la clase `Rectangulo` del ejercicio 9.12 para que incluya una función `dibujar` que muestre el rectángulo dentro de un cuadro de 25 por 25, que encierre la porción correspondiente al primer cuadrante en el que reside el rectángulo. Incluya una función `establecerCaracterRelleno` para especificar el carácter a partir del cual se dibujará el rectángulo. Incluya una función `establecerCaracterPerimetro` para especificar el carácter que se utilizará para dibujar el borde del rectángulo. Si se siente ambicioso, tal vez quiera incluir funciones para escalar el tamaño del rectángulo, girarlo y desplazarlo alrededor del interior de la porción designada del primer cuadrante.

9.14 *(Clase `EnteroEnorme`)* Cree una clase llamada `EnteroEnorme` que utilice un arreglo de 40 elementos de dígitos, para almacenar enteros de hasta 40 dígitos cada uno. Proporcione las funciones miembro `recibir`, `imprimir`, `sumar` y `restar`. Para comparar objetos `EnteroEnorme`, proporcione las funciones `esIgualA`, `noEsIgualA`, `esMayorQue`, `esMenorQue`, `esMayorOIgualQue` y `esMenorOIgualQue`; cada una de éstas es una función "predicado" que simplemente devuelve `true` si la relación se mantiene entre los dos objetos `EnteroEnorme` y devuelve `false` si la relación no se mantiene. Además, proporcione una función predicado `esCero`. Si se siente ambicioso, proporcione las funciones miembro `multiplicar`, `dividir` y `modulo`.

9.15 *(Clase `TresEnRaya`)* Cree una clase llamada `TresEnRaya` que le permita escribir un programa completo para jugar al "tres en raya". La clase debe contener como datos `private` un arreglo bidimensional de enteros, con un tamaño de 3 por 3. El constructor debe inicializar el tablero vacío con ceros. Permita dos jugadores humanos. Siempre que el primer jugador realice un movimiento, coloque un 1 en el cuadro especificado. Coloque un 2 siempre que el segundo jugador realice un movimiento. Cada movimiento debe hacerse en un cuadro vacío. Después de cada movimiento, determine si el juego se ha ganado o si hay un empate. Si desea hacer algo más, modifique su programa de manera que la computadora realice los movimientos para uno de los jugadores. Además, permita que el jugador especifique si desea el primer o segundo turno. Si se siente todavía más motivado, desarrolle un programa que reproduzca un juego de tres en raya tridimensional, en un tablero de 4 por 4 por 4. [*Advertencia*: ¡Éste es un proyecto extremadamente retador, que podría requerir de muchas semanas de esfuerzo!].

9.16 *(Amistad)* Explique la noción de amistad. Explique los aspectos negativos de la amistad, como se describe en el texto.

9.17 *(Sobrecarga de constructores)* ¿Puede una definición de la clase `Tiempo`, que incluye los *dos* constructores que se muestran a continuación:

```
Tiempo( int h = 0, int m = 0, int s = 0 );
Tiempo();
```

utilizarse para construir un objeto `Tiempo` de manera predeterminada? Si no es posible, explique por qué.

9.18 *(Constructores y destructores)* ¿Qué ocurre cuando se especifica un tipo de valor de retorno, incluso `void`, para un constructor o destructor?

9.19 *(Modificación de la clase `Fecha`)* Modifique la clase `Fecha` en la figura 9.17 para que tenga las siguientes herramientas:

a) Imprimir la fecha en varios formatos, como

```
DDD AAAA
MM/DD/AA
Junio 14, 1992
```

b) Usar constructores sobrecargados para crear objetos `Fecha` inicializados con fechas de los formatos en la parte (a).

c) Crear un constructor de `Fecha` que lea la fecha del sistema utilizando las funciones de la biblioteca estándar del encabezado `<ctime>`, y que establezca los miembros de `Fecha`. Consulte la documentación de referencia de su compilador, o el sitio `en.cpreference.com/w/cpp/chrono/c` para obtener información sobre las funciones en el encabezado `<ctime>`. Tal vez también quiera dar un vistazo a la nueva biblioteca `chrono` de C++11 en `en.cpreference.com/w/cpp/chrono`.

En el capítulo 10, podremos crear operadores para evaluar la igualdad de dos fechas, para comparar fechas y determinar si una fecha es anterior, o posterior, a otra.

9.20 *(Clase `CuentaAhorros`)* Cree una clase llamada `CuentaAhorros`. Use un miembro de datos `static` llamado `tasaInteresAnual` para almacenar la tasa de interés anual para cada uno de los ahorradores. Cada miembro de la clase debe contener un miembro de datos `private` llamado `saldoAhorros`, que indique el monto que tiene el ahorrador actualmente en depósito. Proporcione la función miembro `calcularInteresMensual` que calcule el interés mensual multiplicando el `saldoAhorros` por `tasaInteresAnual` dividido entre 12; este interés debe sumarse a `saldoAhorros`. Proporcione una función miembro `static` llamada `modificarTasaInteres` que establezca el miembro de datos `static tasaInteresAnual` a un nuevo valor. Escriba un programa controlador para probar la clase `CuentaAhorros`. Cree instancias de dos objetos distintos de la clase `CuentaAhorros` llamados `ahorrador1` y `ahorrador2`, con saldos de $2000.00 y $3000.00, respectivamente. Establezca la `tasaInteresAnual` al 3 por ciento. Después calcule el interés mensual e imprima los nuevos saldos para cada uno de los ahorradores. Después establezca la `tasaInteresAnual` al 4 por ciento, calcule el interés del siguiente mes e imprima los nuevos saldos para cada uno de los ahorradores.

9.21 *(Clase `ConjuntoEnteros`)* Cree la clase `ConjuntoEnteros` para la que cada objeto pueda contener enteros en el rango de 0 a 100. Represente el conjunto en forma interna como un `vector` de valores `bool`. El elemento `a[i]` del arreglo es `true` si el entero *i* se encuentra en el conjunto. El elemento `a[j]` es `false` si el entero *j* no se encuentra en el conjunto. El constructor predeterminado inicializa un conjunto con lo que se conoce como "conjunto vacío"; es decir, un conjunto en el que todos sus elementos contienen `false`.

a) Proporcione funciones miembro para las operaciones comunes de los conjuntos. Por ejemplo, proporcione una función miembro llamada `unionDeConjuntos` que cree un tercer conjunto que sea la unión teórica de dos conjuntos existentes (es decir, un elemento del resultado se establece en `true` si ese elemento es `true` en cualquiera, o en ambos de los conjuntos existentes, y un elemento del resultado se establece en `false` si ese elemento es `false` en cada uno de los conjuntos existentes).

b) Proporcione una función miembro llamada `interseccionDeConjuntos`, para crear un tercer conjunto que sea la intersección teórica de dos conjuntos existentes (es decir, un elemento del resultado se esta-

blece en `false` si ese elemento es `false` en uno o ambos de los conjuntos existentes, y un elemento del resultado se establece en `true` si ese elemento es `true` en cada uno de los conjuntos existentes).

c) Proporcione una función miembro llamada `insertarElemento`, que coloque un nuevo entero *k* en un conjunto, estableciendo `a[k]` en `true`. Proporcione una función miembro llamada `eliminarElemento` que elimine el entero *m*, estableciendo `a[m]` en `false`.

d) Proporcione una función miembro llamada `imprimirConjunto` que imprima un conjunto como una lista de números separados por espacios. Imprima sólo los elementos que estén presentes en el conjunto (es decir, que su posición en el `vector` tenga un valor de `true`). Imprima `---` para un conjunto vacío.

e) Proporcione una función miembro llamada `esIgualA` que determine si dos conjuntos son iguales.

f) Proporcione un constructor adicional que reciba un arreglo de enteros y el tamaño de ese arreglo, y que utilice el arreglo para inicializar un objeto conjunto.

Ahora escriba un programa controlador para probar su clase `ConjuntoEnteros`. Cree instancias de varios objetos `ConjuntoEnteros`. Pruebe que todas sus funciones miembro trabajen en forma apropiada.

9.22 ***(Clase `Tiempo` modificada)*** Sería perfectamente razonable que la clase `Tiempo` de las figuras 9.4 y 9.5 representaran el tiempo en forma interna como el número de segundos transcurridos desde medianoche, en vez de hacerlo con los tres valores enteros `hora`, `minuto` y `segundo`. Los clientes podrían usar los mismos métodos `public` y obtener los mismos resultados. Modifique la clase `Tiempo` de la figura 9.4 para implementar el tiempo como el número de segundos transcurridos desde medianoche y mostrar que no hay un cambio visible en la funcionalidad para los clientes de esa clase. [*Nota:* este ejercicio demuestra apropiadamente las virtudes del ocultamiento de información].

9.23 ***(Barajar y repartir cartas)*** Cree un programa para barajar y repartir un mazo de cartas. El programa debe consistir en las clases `Carta`, `MazoDeCartas` y un programa controlador. La clase `Carta` debe proporcionar:

a) Los miembros de datos `cara` y `palo` de tipo `int`.

b) Un constructor que recibe dos `int` que representan la cara y el palo, y los utiliza para inicializar los miembros de datos.

c) Dos arreglos `static` de objetos `string` que representan las caras y los palos.

d) Una función `aString` que devuelva la `Carta` como un objeto `string` en la forma "*cara* de *palo*". Puede usar el operador `+` para concatenar objetos `string`.

La clase `MazoDeCartas` debe contener:

a) Un arreglo de objetos `Carta` llamado `mazo` para guardar las cartas.

b) Una variable entera llamada `cartaActual` que represente la siguiente carta a repartir.

c) Un constructor predeterminado que inicialice los objetos `Carta` en el mazo.

d) Una función `barajar` para barajar los objetos `Carta` en el mazo. El algoritmo para barajar debe iterar a través del arreglo de objetos `Carta`. Para cada `Carta`, seleccione al azar otra `Carta` en el mazo e intercambie las dos cartas.

e) Una función `repartirCarta` que devuelva el siguiente objeto `Carta` del mazo.

f) Una función `masCartas` que devuelva un valor `bool` que indique si hay más objetos `Carta` por repartir.

El programa controlador debe crear un objeto `MazoDeCartas`, barajar y repartir las 52 cartas.

9.24 ***(Barajar y repartir cartas)*** Modifique el programa que desarrolló en el ejercicio 9.23, de modo que reparta una mano de póquer de cinco cartas. Luego escriba funciones para realizar cada una de las siguientes acciones:

a) Determinar si la mano contiene un par.

b) Determinar si la mano contiene dos pares.

c) Determinar si la mano contiene tres cartas del mismo tipo (por ejemplo, tres jotos).

d) Determinar si la mano contiene cuatro cartas del mismo tipo (por ejemplo, cuatro ases).

e) Determinar si la mano contiene una corrida (es decir, las cinco cartas del mismo palo).

f) Determinar si la mano contiene una escalera (es decir, cinco cartas de valor consecutivo de la misma cara).

9.25 ***(Proyecto: barajar y repartir cartas)*** Use las funciones del ejercicio 9.24 para escribir un programa que lidie con dos manos de póquer de cinco cartas, evalúe cada mano y determine cuál es la mejor.

9.26 ***(Proyecto: barajar y repartir cartas)*** Modifique el programa que desarrolló en el ejercicio 9.25 para poder simular el repartidor. La mano de cinco cartas del repartidor se reparte "boca abajo" para que el jugador no pueda verla. Después el programa debe evaluar la mano del repartidor y, con base en la calidad de ésta, debe sacar una, dos o tres cartas más para reemplazar el número correspondiente de cartas que no necesita en la mano original. Después, el programa debe reevaluar la mano del repartidor.

9.27 ***(Proyecto para barajar y repartir cartas)*** Modifique el programa que desarrolló en el ejercicio 9.26, de manera que pueda encargarse de la mano del repartidor en forma automática, pero debe permitir al jugador decidir cuáles cartas de su mano desea reemplazar. A continuación, el programa deberá evaluar ambas manos y determinar quién gana. Ahora utilice este nuevo programa para jugar 20 manos contra la computadora. ¿Quién gana más juegos, usted o la computadora? Haga que un amigo juegue 20 manos contra la computadora. ¿Quién gana más juegos? Con base en los resultados de estos juegos, realice las modificaciones apropiadas para refinar su programa para jugar póquer. Juegue 20 manos más. ¿Su programa modificado hace un mejor juego?

Hacer la diferencia

9.28 ***(Proyecto: clase de respuesta de emergencia)*** El servicio de respuesta de emergencia estadounidense, *9-1-1*, conecta a los que llaman a un Punto de respuesta del servicio público (PSAP) *local*. Por tradición, el PSAP pide al que llama cierta información de identificación, incluyendo su dirección, número telefónico y la naturaleza de la emergencia. Después despacha los respondedores de emergencia apropiados (como la policía, una ambulancia o el departamento de bomberos). El *9-1-1 mejorado (o E9-1-1)* usa computadoras y bases de datos para determinar el domicilio del que llama, dirige la llamada al PSAP más cercano y muestra tanto el número de teléfono como la dirección del que llama a la persona que toma la llamada. El *9-1-1 mejorado inalámbrico* proporciona, a los encargados de tomar las llamadas, la información de identificación para las llamadas inalámbricas. Se desplegó en dos fases; durante la primera fase, las empresas de telecomunicaciones tuvieron que proporcionar el número telefónico inalámbrico y la ubicación del sitio de la celda o la estación base que transmitía la llamada. En la segunda fase, las empresas de telecomunicaciones tuvieron que proveer la ubicación del que hacía la llamada (mediante el uso de tecnologías como GPS). Para aprender más sobre el 9-1-1, visite `www.fcc.gov/pshs/services/911-services/Welcome.html` y `people.howstuffworks.com/9-1-1.htm`.

Una parte importante de crear una clase es determinar sus atributos (variables de instancia). Para este ejercicio de diseño de clases, investigue los servicios 9-1-1 en Internet. Después, diseñe una clase llamada `Emergencia` que podría utilizarse en un sistema de respuesta de emergencia 9-1-1 orientado a objetos. Liste los atributos que podría usar un objeto de esta clase para representar la emergencia. Por ejemplo, la clase podría incluir información sobre quién reportó la emergencia (incluyendo su número telefónico), la ubicación de la emergencia, la hora del reporte, la naturaleza de la emergencia, el tipo de respuesta y el estado de la misma. Los atributos de la clase deben describir por completo la naturaleza del problema y lo que está ocurriendo para resolver ese problema.

Sobrecarga de operadores: la clase `string`

10

Hay dos hombres dentro del artista, el poeta y el artesano. Uno nace siendo poeta. El otro se convierte en artesano.

—Emile Zola

Algo bello es un gozo para siempre.

—John Keats

Objetivos

En este capítulo aprenderá a:

- Familiarizarse con la sobrecarga de operadores y cómo esto puede ayudarle a crear clases valiosas.
- Distinguir operadores unarios y binarios.
- Convertir objetos de una clase a otra clase.
- Usar los operadores sobrecargados y las características adicionales de la clase `string`.
- Crear las clases `NumeroTelefonico`, `Fecha` y `Arreglo` que proporcionen operadores sobrecargados.
- Realizar asignación dinámica de memoria con `new` y `delete`.
- Reconocer el uso de la palabra clave `explicit` para indicar que no se puede usar un constructor para conversiones implícitas.
- Experimentar un "momento de iluminación" cuando aprecie de verdad la elegancia y belleza del concepto de las clases.

Plan general

10.1 Introducción

En este capítulo mostraremos cómo permitir que los operadores de C++ trabajen con objetos; a este proceso se le conoce como **sobrecarga de operadores**. Un ejemplo de un operador sobrecargado integrado en C++ es <<, el cual se usa *como* operador de inserción de flujo *y como* operador de desplazamiento a la izquierda a nivel de bits (vea el capítulo 22, en inglés, en el sitio web). De manera similar, >> también está sobrecargado; se utiliza como operador de extracción de flujo y como operador de desplazamiento a la derecha a nivel de bits. Ambos operadores están sobrecargados en la Biblioteca estándar de C++. Las sobrecargas están integradas al mismo lenguaje C++ base. Por ejemplo, C++ sobrecarga los operadores de suma (+) y de resta (-) para que se desempeñen de manera distinta, dependiendo de su contexto en la aritmética de enteros, de punto flotante y de apuntadores con datos de tipos fundamentales.

C++ nos permite sobrecargar la *mayoría* de los operadores que se van a usar con objetos de clases; el compilador genera el código apropiado con base en los *tipos* de los operandos. Los trabajos que desempeñan los operadores sobrecargados también se pueden llevar a cabo mediante llamadas explícitas a funciones, pero la notación de los operadores es comúnmente más clara y natural.

Nuestros ejemplos comienzan por demostrar la clase `string` de la Biblioteca estándar de C++, que tiene muchos operadores sobrecargados. Esto le permitirá ver a los operadores sobrecargados en uso antes de que implemente sus propios operadores sobrecargados. A continuación crearemos una clase llamada `NumeroTelefonico` que nos permite usar los operadores sobrecargados << y >> para imprimir y recibir de manera conveniente números telefónicos de 10 dígitos, con todo y su formato correcto. Luego presentaremos una clase llamada `Fecha` que sobrecarga los operadores preincremento y postincremento (++) para sumar un día al valor de una `Fecha`. La clase también sobrecarga el operador += para permitir que un programa incremente una `Fecha` por el número de días especificados del lado derecho del operador.

Luego presentaremos un caso de estudio de culminación: una clase `Array` que utiliza operadores sobrecargados y otras herramientas para resolver varios problemas con arreglos basados en apuntador. Éste es uno de los casos de estudio más importantes del libro. Muchos de nuestros estudiantes nos han indicado que el caso de estudio de `Array` es su "momento de iluminación" para verdaderamente com-

prender los conceptos de clases y tecnología de objetos. Como parte de esta clase, vamos a sobrecargar los operadores de inserción de flujo, extracción de flujo, asignación, igualdad, relacionales y de subíndice. Una vez que domine esta clase `Array`, sin duda comprenderá la esencia de la tecnología de objetos: crear, usar y reutilizar clases valiosas.

El capítulo termina con una discusión de cómo realizar conversiones entre tipos (incluyendo tipos de clases), problemas con ciertas conversiones implícitas y cómo evitar estos problemas.

10.2 Uso de los operadores sobrecargados de la clase `string` de la Biblioteca estándar

La figura 10.1 demuestra muchos de los operadores sobrecargados de la clase `string` y varias otras funciones miembro útiles, incluyendo `empty`, `substr` y `at`. La función `empty` determina si un objeto `string` está vacío, la función `substr` devuelve un objeto `string` que representa una parte de un objeto `string` existente y la función `at` devuelve el carácter en el índice específico en un objeto `string` (después de verificar que el índice esté en el rango). En el capítulo 21 (en inglés en el sitio web) presentaremos la clase `string` con detalle.

```
1  // Fig. 10.1: fig10_01.cpp
2  // Programa de prueba de la clase string de la Biblioteca estándar.
3  #include <iostream>
4  #include <string>
5  using namespace std;
6
7  int main()
8  {
9     string s1( "feliz" );
10    string s2( " cumpleanios" );
11    string s3;
12
13    // prueba los operadores de igualdad y relacionales sobrecargados
14    cout << "s1 es \"" << s1 << "\"; s2 es \"" << s2
15       << "\"; s3 es \"" << s3 << '\"'
16       << "\n\nLos resultados de comparar s2 y s1:"
17       << "\ns2 == s1 produce " << ( s2 == s1 ? "true" : "false" )
18       << "\ns2 != s1 produce " << ( s2 != s1 ? "true" : "false" )
19       << "\ns2 >  s1 produce " << ( s2 > s1 ? "true" : "false" )
20       << "\ns2 <  s1 produce " << ( s2 < s1 ? "true" : "false" )
21       << "\ns2 >= s1 produce " << ( s2 >= s1 ? "true" : "false" )
22       << "\ns2 <= s1 produce " << ( s2 <= s1 ? "true" : "false" );
23
24    // prueba la función miembro empty de string
25    cout << "\n\nPrueba de s3.empty():" << endl;
26
27    if ( s3.empty() )
28    {
29       cout << "s3 esta vacia; se asigno s1 a s3;" << endl;
30       s3 = s1; // asigna s1 a s3
31       cout << "s3 es \"" << s3 << "\"";
32    } // fin de if
33
34    // prueba el operador de concatenación sobrecargado de string
35    cout << "\n\ns1 += s2 produce s1 = ";
```

Fig. 10.1 | Programa de prueba de la clase `string` de la Biblioteca estándar (parte 1 de 3).

```
36     s1 += s2; // prueba el operador de concatenación sobrecargado
37     cout << s1;
38
39     // prueba el operador de concatenación sobrecargado de string con una cadena
          estilo C
40     cout << "\n\ns1 += \" a ti\" produce" << endl;
41     s1 += " a ti";
42     cout << "s1 = " << s1 << "\n\n";
43
44     // prueba la función miembro substr de string
45     cout << "La subcadena de s1 que empieza en la ubicacion 0 para\n"
46        << "17 caracteres, s1.substr(0, 17), es:\n"
47        << s1.substr( 0, 17 ) << "\n\n";
48
49     // prueba la opción "hasta el final de la cadena" de substr
50     cout << "La subcadena de s1 qe empieza en\n"
51        << "la ubicacion 18, s1.substr(18), es:\n"
52        << s1.substr( 18 ) << endl;
53
54     // prueba el constructor de copia
55     string s4( s1 );
56     cout << "\ns4 = " << s4 << "\n\n";
57
58     // prueba el operador de asignación de copia sobrecargado(=)
          con la auto-asignación
59     cout << "asignando s4 a s4" << endl;
60     s4 = s4;
61     cout << "s4 = " << s4 << endl;
62
63     // prueba el uso del operador de subíndice sobrecargado para crear un lvalue
64     s1[ 0 ] = 'F';
65     s1[ 6 ] = 'C';
66     cout << "\ns1 despues de s1[0] = 'F' y s1[6] = 'C' es: "
67        << s1 << "\n\n";
68
69     // prueba el subíndice fuera de rango con la función miembro "at" de string
70     try
71     {
72        cout << "El intento de asignar 'd' a s1.at( 30 ) produce:" << endl;
73        s1.at( 30 ) = 'd'; // ERROR: subíndice fuera de rango
74     } // fin de try
75     catch ( out_of_range &ex )
76     {
77        cout << "Ocurrio una excepcion: " << ex.what() << endl;
78     } // fin de catch
79  } // fin de main
```

```
s1 es "feliz"; s2 es " cumpleanios"; s3 es ""

Los resultados de comparar s2 y s1:
s2 == s1 produce false
s2 != s1 produce true
s2 >  s1 produce false
s2 <  s1 produce true
```

Fig. 10.1 | Programa de prueba de la clase `string` de la Biblioteca estándar (parte 2 de 3).

```
s2 >= s1 produce false
s2 <= s1 produce true

Prueba de s3.empty():
s3 esta vacia; se asigno s1 a s3;
s3 es "feliz"

s1 += s2 produce s1 = feliz cumpleanios

s1 += " a ti" produce
s1 = feliz cumpleanios a ti

La subcadena de s1 que empieza en la ubicacion 0 para
17 caracteres, s1.substr(0, 17), es:
feliz cumpleanios

La subcadena de s1 que empieza en la
ubicacion 18, s1.substr(18), es:
a ti

s4 = feliz cumpleanios a ti

asignando s4 a s4
s4 = feliz cumpleanios a ti

s1 despues de s1[0] = 'F' y s1[6] = 'C' es: Feliz Cumpleaños a ti

El intento de asignar 'd' a s1.at( 30 ) produce:
Ocurrio uma excepcion: invalid string position
```

Fig. 10.1 | Programa de prueba de la clase `string` de la Biblioteca estándar (parte 3 de 3).

En las líneas 9 a 11 se crean tres objetos `string`: `s1` se inicializa con la literal `"feliz"`, `s2` se inicializa con la literal `" cumpleanios"` y `s3` utiliza el constructor de `string` predeterminado para crear un objeto `string` vacío. En las líneas 14 y 15 se imprimen estos tres objetos, usando `cout` y el operador <<, que los diseñadores de la clase `string` sobrecargaron para manejar objetos `string`. Después, en las líneas 16 a 22 se muestran los resultados de comparar `s2` con `s1`, usando los operadores de igualdad y relacionales sobrecargados de la clase `string`, que realizan comparaciones lexicográficas (como el orden en un diccionario) mediante el uso de los valores numéricos de los caracteres (vea el apéndice B, Conjunto de caracteres ASCII) en cada objeto `string`.

La clase `string` proporciona la función miembro **`empty`**, para determinar si un objeto `string` está vacío, lo cual demostraremos en la línea 27. La función miembro `empty` devuelve `true` si el objeto `string` está vacío; en caso contrario, devuelve `false`.

En la línea 30 se demuestra el operador de asignación de copia sobrecargado de la clase `string` mediante la asignación de `s1` a `s3`. En la línea 31 se imprime `s3` para demostrar que la asignación funcionó de manera correcta.

En la línea 36 se demuestra el operador += sobrecargado de la clase `string` para la *concatenación de cadenas*. En este caso, el contenido de `s2` se adjunta a `s1`. Después, en la línea 37 se imprime la cadena resultante que se almacena en `s1`. En la línea 41 se demuestra que una literal de cadena se puede adjuntar a un objeto `string` mediante el uso del operador +=. En la línea 42 se muestra el resultado.

La clase `string` proporciona la función miembro **`substr`** (líneas 47 y 52) para devolver una *parte* de una cadena como un objeto `string`. La llamada a `substr` en la línea 47 obtiene una subcadena de 17 caracteres (especificada por el segundo argumento) de `s1`, empezando en la posición 0 (especificada por el primer argumento). La llamada a `substr` en la línea 52 obtiene una subcadena que empieza desde la posición 15 de `s1`. Cuando el segundo argumento no se especifica, `substr` devuelve el *resto* del objeto `string` en el que se llamó.

En la línea 55 se crea el objeto `string s4` y se inicializa con una copia de `s1`. Esto produce una llamada al *constructor de copia* de la clase `string`. En la línea 60 se utiliza el operador de asignación de copia (=) sobrecargado de la clase `string` para demostrar que maneja la *auto-asignación* en forma apropiada; más adelante en el capítulo, al crear la clase `Array`, veremos que la auto-asignación puede ser peligrosa y también veremos cómo lidiar con estas cuestiones.

En las líneas 64 y 65 se utiliza el operador `[]` sobrecargado de la clase `string` para crear *lvalues* que permitan que nuevos caracteres reemplacen los caracteres existentes en `s1`. En la línea 67 se imprime el nuevo valor de `s1`. *El operador `[]` sobrecargado de la clase `string` no realiza la comprobación de límites.* Por lo tanto, *usted debe asegurar que las operaciones que utilicen el operador `[]` sobrecargado de la clase `string` no manipulen de manera accidental elementos fuera de los límites del objeto `string`.* La clase `string` *sí* proporciona la comprobación de límites en su función miembro **`at`**, la cual lanza una excepción si su argumento es un subíndice *inválido*. Si el subíndice es válido, la función `at` devuelve el carácter en la ubicación especificada como un *lvalue* modificable o un *lvalue* no modificable (es decir, una referencia `const`), dependiendo del contexto en el que aparece la llamada. En la línea 73 se demuestra una llamada a la función `at` con un subíndice inválido; en este caso se lanza una excepción `out_of_range`.

10.3 Fundamentos de la sobrecarga de operadores

Como vimos en la figura 10.1, los operadores proporcionan una notación concisa para manipular objetos `string`. Los programadores pueden usar operadores con sus propios tipos definidos por el usuario también. Aunque C++ *no* permite crear *nuevos* operadores, *sí* permite sobrecargar la mayoría de los operadores existentes para que, cuando éstos se utilicen con objetos, tengan un significado apropiado para esos objetos.

La sobrecarga de operadores *no* es automática: es necesario escribir funciones de sobrecarga de operadores para que realicen las operaciones deseadas. Para sobrecargar un operador, se escribe la definición de una función miembro no `static` o la definición de una función no miembro como se hace normalmente, excepto que el nombre empieza con la palabra clave `operator`, seguida del símbolo del operador que se va a sobrecargar. Por ejemplo, el nombre de función `operator+` se utilizaría para sobrecargar el operador de suma (+), para usarlo con objetos de una clase (o `enum`) específica. Cuando los operadores se sobrecargan como funciones miembro, deben ser no `static`, debido a que *se deben llamar en un objeto de la clase* y deben operar en ese objeto.

Para usar un operador en un objeto de una clase, hay que definir funciones de operadores sobrecargados para esa clase, con tres excepciones:

- El *operador de asignación (=)* se puede usar con la *mayoría* de las clases para realizar la *asignación a nivel de miembro* de los miembros de datos; cada miembro de datos se asigna del objeto "origen" (a la derecha) al objeto "destino" (a la izquierda) de la asignación. *La asignación a nivel de miembro es peligrosa para las clases con miembros apuntadores*, por lo que debemos sobrecargar de manera explícita el operador de asignación para dichas clases.
- El *operador dirección (&)* devuelve un apuntador al objeto; este operador también puede sobrecargarse.
- El *operador coma* evalúa la expresión a su izquierda, y después la expresión a su derecha, y devuelve el valor de esta última expresión. Este operador también se puede sobrecargar.

Operadores que no pueden sobrecargarse
La mayoría de los operadores de C++ se pueden sobrecargar. En la figura 10.2 se muestran los operadores que no se pueden sobrecargar.[1]

Operadores que no se pueden sobrecargar			
.	.* (pointer to member)	::	?:

Fig. 10.2 | Operadores que no se pueden sobrecargar.

Reglas y restricciones en cuanto a la sobrecarga de operadores
Como medida de preparación para sobrecargar operadores en sus propias clases, existen varias reglas y restricciones que debe tener en cuenta:

- *La precedencia de un operador no se puede cambiar mediante la sobrecarga*. Sin embargo, se pueden utilizar paréntesis para *forzar* el orden de evaluación de los operadores sobrecargados en una expresión.
- *La asociatividad de un operador no se puede cambiar mediante la sobrecarga*: si por lo general un operador asocia de izquierda a derecha, así lo harán todas sus versiones sobrecargadas.
- *No es posible modificar la "aridad" de un operador* (es decir, el número de operandos que recibe): los operadores unarios sobrecargados siguen siendo operadores unarios; los operadores binarios sobrecargados siguen siendo operadores binarios. Los operadores `&`, `*`, `+` y `-` tienen versiones unarias y binarias; cada una de estas versiones unarias y binarias se pueden sobrecargar por separado.
- *No es posible crear nuevos operadores; sólo los operadores existentes se pueden sobrecargar*.
- El significado de la forma en que trabaja un operador con valores de tipos fundamentales *no* se puede modificar mediante la sobrecarga de operadores. Por ejemplo, no podemos hacer que el operador `+` reste dos valores `int`. La sobrecarga de operadores funciona sólo con *objetos de los tipos definidos por el usuario, o con una mezcla de un objeto de un tipo definido por el usuario y un objeto de un tipo fundamental*.
- Los operadores relacionados, como `+` y `+=`, deben sobrecargarse por separado.
- Al sobrecargar los operadores `()`, `[]`, `->` o cualquiera de los operadores de asignación, la función de sobrecarga de operador *debe* declararse como miembro de la clase. Para todos los demás operadores sobrecargados, las funciones de sobrecarga de operadores pueden ser funciones miembro o no miembro.

Observación de Ingeniería de Software 10.1
Sobrecargue los operadores para tipos de clases, de modo que funcionen lo más parecido posible a la forma en que trabajan los operadores integrados con los tipos fundamentales.

10.4 Sobrecarga de operadores binarios

Un operador binario se puede sobrecargar como una función miembro no `static` *con un parámetro, o como una función no miembro con dos parámetros (uno de esos parámetros debe ser el objeto de una clase o una*

1 Aunque es posible sobrecargar los operadores dirección (`&`), coma (`,`), `&&` y `||`, lo recomendable es no hacerlo para evitar errores sutiles. Si desea más información sobre esto, consulte el lineamiento DCL10-CPP de CERT.

referencia al objeto de una clase). Por lo general, una función de operador no miembro se declara como amiga (`friend`) de una clase por cuestiones de rendimiento.

Operadores binarios sobrecargados como funciones miembro

Considere el uso del operador < para comparar dos objetos de una clase `String` que usted defina. Al sobrecargar el operador binario < como una función miembro no `static` de una clase `String`, si `y` y `z` son objetos de clases `String`, entonces `y < z` se considera como si se hubiera escrito `y.operator<( z )`, invocando a la función miembro `operator<` que se declara a continuación:

```
class String
{
public:
   bool operator<( const String & ) const;
   ...
}; // fin de la clase String
```

Las funciones de operador sobrecargadas para los operadores binarios pueden ser funciones miembro *sólo* cuando el operando *izquierdo* es un objeto de la clase de la que la función es miembro.

Operadores binarios sobrecargados como funciones no miembro

Como función no miembro, el operador binario < *debe* recibir *dos* argumentos (*uno* de los cuales *debe* ser un objeto (o una referencia a un objeto) de la clase con la que está asociado el operador sobrecargado. Si `y` y `z` son objetos de la clase `String` o referencias a objetos de esa clase, entonces `y < z` se considera como si se hubiera escrito la llamada `operator<(y, z)` en el programa, invocando a la función `operator<`, que se declara a continuación:

```
bool operator<( const String &, const String & );
```

10.5 Sobrecarga de los operadores binarios de inserción de flujo y extracción de flujo

Es posible recibir y enviar datos de tipos fundamentales mediante el uso del operador de extracción de flujo >> y del operador de inserción de flujo <<. Las bibliotecas de clases de C++ sobrecargan estos operadores binarios para cada tipo fundamental, incluyendo apuntadores y cadenas `char *`. También es posible sobrecargar estos operadores para realizar operaciones de entrada y salida para sus propios tipos. El programa de las figuras 10.3 a 10.5 sobrecarga estos operadores para recibir y enviar objetos `NumeroTelefonico` en el formato "`(000)000-0000`". El programa asume que los números telefónicos se introducen correctamente.

```
1  // Fig. 10.3: NumeroTelefonico.h
2  // Definición de la clase NumeroTelefonico
3  #ifndef NUMEROTELEFONICO_H
4  #define NUMEROTELEFONICO_H
5
6  #include <iostream>
7  #include <string>
8
9  class NumeroTelefonico
10 {
```

Fig. 10.3 | Clase `NumeroTelefonico` con los operadores de inserción de flujo y de extracción de flujo sobrecargados como funciones `friend` (parte 1 de 2).

```
11    friend std::ostream &operator<<( std::ostream &, const NumeroTelefonico & );
12    friend std::istream &operator>>( std::istream &, NumeroTelefonico & );
13    private:
14       std::string codigoArea; // código de área de 3 dígitos
15       std::string intercambio; // intercambio de 3 dígitos
16       std::string linea; // línea de 4 dígitos
17    }; // fin de la clase NumeroTelefonico
18
19    #endif
```

Fig. 10.3 | Clase `NumeroTelefonico` con los operadores de inserción de flujo y de extracción de flujo sobrecargados como funciones `friend` (parte 2 de 2).

```
1   // Fig. 10.4: NumeroTelefonico.cpp
2   // Operadores de inserción de flujo y de extracción de flujo sobrecargados
3   // para la clase NumeroTelefonico.
4   #include <iomanip>
5   #include "NumeroTelefonico.h"
6   using namespace std;
7
8   // operador de inserción de flujo sobrecargado; no puede ser
9   // una función miembro si deseamos invocarlo con
10  // cout << unNumeroTelefonico;
11  ostream &operator<<( ostream &salida, const NumeroTelefonico &numero )
12  {
13     salida << "(" << numero.codigoArea << ") "
14        << numero.intercambio << "-" << numero.linea;
15     return salida; // permite cout << a << b << c;
16  } // fin de la función operator<<
17
18  // operador de extracción de flujo sobrecargado; no puede ser
19  // una función miembro si deseamos invocarlo con
20  // cin >> unNumeroTelefonico;
21  istream &operator>>( istream &entrada, NumeroTelefonico &numero )
22  {
23     entrada.ignore(); // omite (
24     entrada >> setw( 3 ) >> numero.codigoArea; // recibe el código de área
25     entrada.ignore( 2 ); // omite ) y espacio
26     entrada >> setw( 3 ) >> nnumero.intercambio; // recibe intercambio
27     entrada.ignore(); // omite el guión corto (-)
28     entrada >> setw( 4 ) >> numero.linea; // recibe linea
29     return entrada; // permite cin >> a >> b >> c;
30  } // fin de la función operator>>
```

Fig. 10.4 | Operadores de inserción de flujo y de extracción de flujo sobrecargados para la clase `NumeroTelefonico`.

```
1   // Fig. 10.5: fig10_05.cpp
2   // Demostración de los operadores de inserción de flujo y de extracción
3   // de flujo sobrecargados de la clase NumeroTelefonico.
4   #include <iostream>
```

Fig. 10.5 | Operadores de inserción de flujo y de extracción de flujo sobrecargados (parte 1 de 2).

```
5  #include "NumeroTelefonico.h"
6  using namespace std;
7
8  int main()
9  {
10    NumeroTelefonico telefono; // crea el objeto telefono
11
12    cout << "Escriba el numero telefonico en la forma (123) 456-7890:" << endl;
13
14    // cin >> telefono invoca a operator>> generando de manera implícita
15    // la llamada a la función no miembro operator>>( cin, telefono )
16    cin >> telefono;
17
18    cout << "El numero telefonico introducido fue: ";
19
20    // cout << telefono invoca a operator<< generando de manera implícita
21    // la llamada a la función no miembro operator<<( cout, telefono )
22    cout << telefono << endl;
23 } // fin de main
```

```
Escriba el numero telefonico en la forma (123) 456-7890:
(800) 555-1212
El numero telefonico introducido fue: (800) 555-1212
```

Fig. 10.5 | Operadores de inserción de flujo y de extracción de flujo sobrecargados (parte 2 de 2).

Sobrecarga del operador de extracción de flujo (>>)

La función del operador de extracción de flujo `operator>>` (figura 10.4, líneas 21 a 30) recibe la referencia `istream` llamada `entrada` y la referencia `NumeroTelefonico` llamada `numero` como argumentos, y devuelve una referencia `istream`. La función de operador `operator>>` introduce números telefónicos de la forma

```
(800) 555-1212
```

en objetos de la clase `NumeroTelefonico`. Cuando el compilador ve la expresión

```
cin >> telefono
```

en la línea 16 de la figura 10.5, el compilador genera la *llamada a la función no miembro*

```
operator>>( cin, telefono );
```

Cuando se ejecute esta llamada, el parámetro de referencia `entrada` (figura 10.4, línea 21) se convierte en un alias para `cin` y el parámetro de referencia `numero` se convierte en un alias para `telefono`. La función de operador lee como objetos `string` las tres partes del número telefónico y las coloca en los miembros `codigoArea` (línea 24), `intercambio` (línea 26) y `linea` (línea 28) del objeto `NumeroTelefonico` al que hace referencia el parámetro `numero`. El manipulador de flujo `setw` limita el número de caracteres que se leen y se colocan en cada objeto `string`. *Cuando se utiliza con `cin` y objetos `string`, `setw` restringe el número de caracteres leídos al número de caracteres especificados por su argumento* (es decir, `setw(3)` permite que se lean tres caracteres). Los paréntesis, espacios y guiones cortos se omiten mediante una llamada a la función miembro `ignore` de `istream` (figura 10.4, líneas 23, 25 y 27), la cual descarta el número especificado de caracteres en el flujo de entrada (un carácter de manera predeterminada). La función `operator>>` devuelve la referencia `istream` llamada `entrada` (es decir, `cin`). Esto permite que las operaciones de entrada en los objetos `NumeroTelefonico` se realicen *en cascada* con las operaciones de entrada en otros objetos

`NumeroTelefonico`, o en objetos de otros tipos de datos. Por ejemplo, un programa puede recibir dos objetos `NumeroTelefonico` en una instrucción de la siguiente manera:

```
cin >> telefono1 >> telefono2;
```

Primero se ejecuta la expresión `cin >> telefono1`, mediante una llamada a la función no miembro

```
operator>>( cin, telefono1 );
```

Después, esta llamada devuelve una referencia a `cin` como el valor de `cin >> telefono1`, por lo que la porción restante de la expresión se interpreta simplemente como `cin >> telefono2`. Esto se ejecuta haciendo una *llamada a la función no miembro*

```
operator>>( cin, telefono2 );
```

Buena práctica de programación 10.1

Los operadores sobrecargados deben imitar la funcionalidad de sus contrapartes integrados; por ejemplo, el operador + debe realizar la suma, no la resta. Evite un uso excesivo o inconsistente de la sobrecarga de operadores, ya que esto puede hacer a un programa críptico y difícil de leer.

Sobrecarga del operador de inserción de flujo (<<)

La función del operador de inserción de flujo (figura 10.4, líneas 11 a 16) recibe una referencia `ostream (salida)` y una referencia `const NumeroTelefonico (numero)` como argumentos, y devuelve una referencia `ostream`. La función `operator<<` muestra objetos de tipo `NumeroTelefonico`. Cuando el compilador ve la expresión

```
cout << telefono
```

en la línea 22 de la figura 10.5, el compilador genera la *llamada a la función no miembro*

```
operator<<( cout, telefono );
```

La función `operator<<` muestra las partes del número telefónico como objetos `string`, ya que se almacenan como objetos `string`.

Los operadores sobrecargados como funciones `friend` no miembro

Las funciones `operator>>` y `operator<<` se declaran en `NumeroTelefonico` como funciones `friend` *no miembro* (figura 10.3, líneas 11 y 12). Son *funciones no miembro* debido a que el objeto de la clase `NumeroTelefonico` debe ser el operando *derecho* del operador. Si éstas fueran a ser *funciones miembro* de `NumeroTelefonico`, habría que usar las siguientes instrucciones incómodas para mostrar y recibir un `NumeroTelefonico`:

```
telefono << cout;
telefono >> cin;
```

Dichas instrucciones serían confusas para la mayoría de los programadores de C++, quienes están acostumbrados a que `cout` y `cin` aparezcan como los operandos *izquierdos* de `<<` y `>>`, respectivamente.

Las funciones de operadores sobrecargados para los operadores binarios pueden ser funciones miembro sólo cuando el operando *izquierdo* es un objeto de la clase en la que la función es miembro. *Los operadores de entrada y salida sobrecargados se declaran como `friend` si necesitan acceder a los miembros no `public` de las clases miembro directamente, o debido a que la clase tal vez no ofrezca funciones* obtener *apropiadas*. Además, la referencia `NumeroTelefonico` en la lista de parámetros de la función `operator<<` (figura 10.4, línea 11) es `const`, ya que el objeto `NumeroTelefonico` simplemente se enviará a la salida,

y la referencia `NumeroTelefonico` en la lista de parámetros de la función `operator>>` (línea 21) no es `const`, ya que el objeto `NumeroTelefonico` debe modificarse para almacenar el número telefónico de entrada en el objeto.

Observación de Ingeniería de Software 10.2

Las nuevas herramientas de entrada/salida para los tipos definidos por el usuario se agregan a C++ sin modificar las clases de la biblioteca de entrada/salida estándar. Éste es otro ejemplo de la extensibilidad de C++.

Por qué los operadores de inserción y extracción de flujos sobrecargados se sobrecargan como funciones no miembro

El operador de inserción de flujo (<<) se utiliza en una expresión en la que el operando izquierdo tiene el tipo `ostream &`, como en `cout << objetoClase`. Para usar el operador de esta forma en donde el operando *derecho* es un objeto de una clase definida por el usuario, debe sobrecargarse como *función no miembro*. Para ser una función miembro, el operador << tendría que ser miembro de la clase `ostream`. Esto *no* es posible para las clases definidas por el usuario, ya que *no tenemos permitido modificar las clases de la Biblioteca estándar de C++*. De manera similar, el operador de extracción de flujo sobrecargado (>>) se utiliza en una expresión en donde el operando *izquierdo* tiene el tipo `istream &`, como en `cin >> objetoClase`, y el operando *derecho* es un objeto de una clase definida por el usuario, por lo que también debe ser una *función no miembro*. Además, cada una de estas funciones de operador sobrecargado pueden requerir acceso a los miembros de datos `private` del objeto de la clase que se va a mostrar o a recibir, por lo que estas funciones de operador sobrecargado pueden hacerse funciones `friend` de la clase por cuestiones de rendimiento.

10.6 Sobrecarga de operadores unarios

Un operador unario para una clase se puede sobrecargar como función miembro no `static` *sin argumentos, o como función no miembro con un argumento que debe ser un objeto (o una referencia a un objeto) de la clase*. Las funciones miembro que implementan operadores sobrecargados deben ser no `static`, de manera que puedan acceder a los datos no `static` en cada objeto de la clase.

Los operadores unarios sobrecargados como funciones miembro

Considere sobrecargar el operador unario `!` para evaluar si un objeto de su propia clase `String` está vacío. Dicha función devolvería un resultado `bool`. Cuando un operador unario tal como `!` se sobrecarga como función miembro sin argumentos y el compilador ve la expresión `!s` (en donde `s` es un objeto de la clase `String`), éste genera la llamada a la función `s.operator!()`. El operando `s` es el objeto `String` para el que se está invocando la función miembro `operator!` de la clase `String`. La función se declara de la siguiente manera:

```
class String
{
public:
   bool operator!() const;
   ...
}; // fin de la clase String
```

Los operadores unarios sobrecargados como funciones no miembro

Un operador unario tal como `!` se puede sobrecargar como *función no miembro* con un parámetro. Si `s` es un objeto de la clase `String` (o una referencia a un objeto de la clase `String`), entonces `!s` se trata como si se hubiera escrito la llamada `operator!(s)`, invocando a la función *no miembro* `operator!` que se declara de la siguiente manera:

```
bool operator!( const String & );
```

10.7 Sobrecarga de los operadores unarios prefijo y postfijo ++ y --

Todas las versiones prefijo y postfijo de los operadores de incremento y decremento se pueden sobrecargar. A continuación veremos cómo el compilador diferencia entre la versión prefijo y la versión postfijo de un operador de incremento o decremento.

Para sobrecargar los operadores de incremento prefijo y postfijo, cada función de operador sobrecargado debe tener una firma distinta, de manera que el compilador pueda determinar cuál es la versión de ++ que se desea. Las versiones prefijo se sobrecargan de la misma forma que cualquier otro operador unario prefijo. Todo lo que se indica en esta sección para sobrecargar los operadores de incremento prefijo y postfijo se aplica a la sobrecarga de los operadores predecremento y postdecremento. En la siguiente sección examinaremos una clase `Fecha` con operadores de incremento prefijo y postfijo sobrecargados.

Sobrecarga del operador de incremento prefijo

Suponga que deseamos sumar 1 al día en el objeto `Fecha` llamado `d1`. Cuando el compilador ve la expresión de preincremento `++d1`, genera la *llamada a la función miembro*

```
d1.operator++()
```

El prototipo para esta función miembro de operador sería

```
Fecha &operator++();
```

Si el operador de incremento prefijo se implementa como una *función no miembro*, entonces cuando el compilador vea la expresión `++d1`, generará la llamada a la función

```
operator++( d1 )
```

El prototipo para esta función de operador no miembro se declararía de la siguiente manera:

```
Fecha &operator++( Fecha & );
```

Sobrecarga del operador de incremento postfijo

La sobrecarga del operador de incremento postfijo representa un reto, ya que el compilador debe tener la capacidad de diferenciar entre las firmas de las funciones de los operadores de incremento prefijo y postfijo. La *convención* que se ha adoptado es que, cuando el compilador ve la expresión de postincremento `d1++`; genera la *llamada a la función miembro*

```
d1.operator++( 0 )
```

El prototipo para esta función miembro de operador es

```
Fecha operator++( int )
```

El argumento `0` es estrictamente un *valor de muestra* que permite al compilador diferenciar entre las funciones de los operadores de incremento prefijo y postfijo. Se utiliza la misma sintaxis para diferenciar entre las funciones de los operadores de decremento prefijo y postfijo.

Si el operador de incremento prefijo se implementa como una *función no miembro*, entonces cuando el compilador ve la expresión `d1++`, genera la llamada a la función

```
operator++( d1, 0 )
```

El prototipo para esta función sería

```
Fecha operator++( Fecha &, int );
```

Una vez más, el compilador utiliza el argumento 0 para diferenciar entre los operadores de incremento prefijo y postfijo que se implementan como funciones no miembro. Observe que el *operador de incremento postfijo* devuelve objetos Fecha *por valor*, mientras que el operador de incremento prefijo devuelve objetos Fecha *por referencia*, ya que por lo general el operador de incremento postfijo devuelve un objeto temporal que contiene el valor original del objeto antes de que ocurra el incremento. C++ trata a dichos objetos como *rvalues*, los cuales *no se pueden utilizar del lado izquierdo de una asignación*. El operador de incremento prefijo devuelve el objeto real incrementado con su nuevo valor. Dicho objeto se *puede* utilizar como un *lvalue* en una expresión de continuación.

Tip de rendimiento 10.1

El objeto adicional que se crea mediante el operador de incremento (o decremento) postfijo *puede producir un problema de rendimiento; en especial cuando el operador se utiliza en un ciclo. Por esta razón, es preferible usar los operadores de incremento y decremento* prefijo *sobrecargados.*

10.8 Caso de estudio: una clase Fecha

El programa de las figuras 10.6 a 10.8 demuestra una clase Fecha, que usa operadores de incremento prefijo y postfijo sobrecargados para sumar 1 al día en un objeto Fecha, mientras que produce incrementos apropiados en el mes y el año, en caso de ser necesario. El encabezado de Fecha (figura 10.6) especifica que la interfaz public de Fecha incluye un operador de inserción de flujo sobrecargado (línea 11), un constructor predeterminado (línea 13), una función establecerFecha (línea 14), un operador de incremento prefijo sobrecargado (línea 15), un operador de incremento postfijo sobrecargado (línea 16), un operador de asignación de suma += sobrecargado (línea 17), una función para evaluar los años bisiestos (línea 18) y una función para determinar si un día es el último del mes (línea 19).

```
1  // Fig. 10.6: Fecha.h
2  // Definición de la clase Fecha con operadores de incremento sobrecargados.
3  #ifndef FECHA_H
4  #define FECHA_H
5
6  #include <array>
7  #include <iostream>
8
9  class Fecha
10 {
11    friend std::ostream &operator<<( std::ostreamm &, const Fecha & );
12 public:
13    Fecha( int m = 1, int d = 1, int a = 1900 ); // constructor predeterminado
14    void establecerFecha( int, int, int ); // establece mes, dia, año
15    Fecha &operator++(); // operador de incremento prefijo
16    Fecha operator++( int ); // operador de incremento postfijo
17    Fecha &operator+=( unsigned int ); // suma días, modifica el objeto
18    static bool anioBisiesto( int ); // ¿está la fecha en un año bisiesto?
19    bool finDeMes( int ) const; // ¿está la fecha en el fin del mes?
20 private:
21    unsigned int mes;
22    unsigned int dia;
23    unsigned int anio;
```

Fig. 10.6 | Definición de la clase Fecha con operadores de incremento sobrecargados (parte 1 de 2).

```
24
25    static const std::array< unsigned int, 13 > dias; // dias por mes
26    void ayudaIncremento(); // función utilitaria para incrementar la fecha
27 }; // fin de la clase Fecha
28
29 #endif
```

Fig. 10.6 | Definición de la clase `Fecha` con operadores de incremento sobrecargados (parte 2 de 2).

```
1  // Fig. 10.7: Fecha.cpp
2  // Definiciones de las funciones miembro y funciones friend de la clase Fecha.
3  #include <iostream>
4  #include <string>
5  #include "Fecha.h"
6  using namespace std;
7
8  // inicializa miembro estático; una copia a nivel de clase
9  const array< unsigned int, 13 > Fecha::dias =
10    { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
11
12 // constructor de Fecha
13 Fecha::Fecha( int mes, int dia, int anio )
14 {
15    establecerFecha( mes, dia, anio );
16 } // fin del constructor de Fecha
17
18 // establece mes, día y anio
19 void Fecha::establecerFecha( int mm, int dd, int aa )
20 {
21    if ( mm >= 1 && mm <= 12 )
22       mes = mm;
23    else
24       throw invalid_argument( "Mes debe estar entre 1 y 12" );
25
26    if ( aa >= 1900 && aa <= 2100 )
27       anio = aa;
28    else
29       throw invalid_argument( "Anio debe ser >= 1900 y <= 2100" );
30
31    // prueba si es año bisiesto
32    if ( ( mes == 2 && anioBisiesto( anio ) && dd >= 1 && dd <= 29 ) ||
33         ( dd >= 1 && dd <= dias[ mes ] ) )
34       dia = dd;
35    else
36       throw invalid_argument(
37          "Dia esta fuera de rango para el mes y anio actuales" );
38 } // fin de la función establecerFecha
39
40 // operador de incremento prefijo sobrecargado
41 Fecha &Fecha::operator++()
42 {
```

Fig. 10.7 | Definiciones de las funciones miembro y funciones `friend` de la clase `Fecha` (parte 1 de 3).

```
43    ayudaIncremento(); // incrementa la fecha
44    return *this; // devuelve referencia para crear un lvalue
45 } // fin de la función operator++
46
47 // operador de incremento postfijo sobrecargado; observe que el parámetro
48 // entero de muestra no tiene un nombre de parámetro
49 Fecha Fecha::operator++( int )
50 {
51    Fecha temp = *this; // contiene el estado actual del objeto
52    ayudaIncremento();
53
54    // devuelve objeto temporal almacenado y sin incrementar
55    return temp; // devuelve un valor; no devuelve una referencia
56 } // fin de la función operator++
57
58 // suma el número especificado de días a la fecha
59 Fecha &Fecha::operator+=( unsigned int diasAdicionales )
60 {
61    for ( int i = 0; i < diasAdicionales; ++i )
62       ayudaIncremento();
63
64    return *this; // permite la asignación en cascada
65 } // fin de la función operator+=
66
67 // si el año es bisiesto, devuelve true; en caso contrario, devuelve false
68 bool Fecha::anioBisiesto( int anioPrueba )
69 {
70    if ( anioPrueba % 400 == 0 ||
71       ( anioPrueba % 100 != 0 && anioPrueba % 4 == 0 ) )
72       return true; // un año bisiesto
73    else
74       return false; // no es un año bisiesto
75 } // fin de la función anioBisiesto
76
77 // determina si el día es el último del mes
78 bool Fecha::finDeMes( int diaPrueba ) const
79 {
80    if ( mes == 2 && anioBisiesto( anio ) )
81       return diaPrueba == 29; // último día de feb. en año bisiesto
82    else
83       return diaPrueba == dias[ mes ];
84 } // fin de la función finDeMes
85
86 // función para ayudar a incrementar la fecha
87 void Fecha::ayudaIncremento()
88 {
89    // dia no es fin de mes
90    if ( !finDeMes( dia ) )
91       ++dia; // incrementa dia
92    else
93       if ( mes < 12 ) // dia es fin de mes y mes < 12
94       {
95          ++mes; // incrementa mes
```

Fig. 10.7 | Definiciones de las funciones miembro y funciones `friend` de la clase `Fecha` (parte 2 de 3).

```
96          dia = 1; // primer dia del nuevo mes
97       } // fin de if
98       else // último dia de anio
99       {
100         ++anio; // incrementa anio
101         mes = 1; // primer mes del nuevo anio
102         dia = 1; // primer dia del nuevo mes
103      } // fin de else
104   } // fin de la función ayudaIncremento
105
106   // operador de salida sobrecargado
107   ostream &operator<<( ostream &salida, const Fecha &d )
108   {
109      static string nombreMes[ 13 ] = { "", "Enero", "Febrero",
110         "Marzo", "Abril", "Mayo", "Junio", "Julio", "Agosto",
111         "Septiembre", "Octubre", "Noviembre", "Diciembre" };
112      salida << nombreMes[ d.mes ] << ' ' << d.dia << ", " << d.anio;
113      return salida; // permite la asignación en cascada
114   } // fin de la función operator<<
```

Fig. 10.7 | Definiciones de las funciones miembro y funciones `friend` de la clase `Fecha` (parte 3 de 3).

```
1   // Fig. 10.8: fig10_08.cpp
2   // Programa de prueba de la clase Fecha.
3   #include <iostream>
4   #include Fecha.h" // definición de la clase Fecha
5   using namespace std;
6
7   int main()
8   {
9      Fecha d1( 12, 27, 2010 ); // Diciembre 27, 2010
10     Fecha d2; // recibe el valor predeterminado de Enero 1, 1900
11
12     cout << "d1 es " << d1 << "\nd2 es " << d2;
13     cout << "\n\nd1 += 7 es " << ( d1 += 7 );
14
15     d2.establecerFecha( 2, 28, 2008 );
16     cout << "\n\n  d2 es " << d2;
17     cout << "\n++d2 es " << ++d2 << " (anio bisiesto permite dia 29)";
18
19     Fecha d3( 7, 13, 2010 );
20
21     cout << "\n\nPrueba del operador de incremento prefijo:\n"
22        << "  d3 es " << d3 << endl;
23     cout << "++d3 es " << ++d3 << endl;
24     cout << "  d3 es "  << d3;
25
26     cout << "\n\nPrueba del operador de incremento postfijo:\n"
27        << "  d3 es " << d3 << endl;
28     cout << "d3++ es " << d3++ << endl;
29     cout << "  d3 es " << d3 << endl;
30  } // fin de main
```

Fig. 10.8 | Programa de prueba de la clase `Fecha` (parte 1 de 2).

```
d1 es Diciembre 27, 2010
d2 es Enero 1, 1900

d2 += 7 es Enero 3, 2011

  d2 es Febrero 28, 2008
++d2 es Febrero 29, 2008 (anio bisiesto permite dia 29)

Prueba del operador de incremento prefijo:
  d3 es Julio 13, 2010
++d3 es Julio 14, 2010
  d3 es Julio 14, 2010

Prueba del operador de incremento postfijo:
  d3 es Julio 14, 2010
d3++ es Julio 14, 20100
  d3 es Julio 15, 2010
```

Fig. 10.8 | Programa de prueba de la clase `Fecha` (parte 2 de 2).

La función `main` (figura 10.8) crea dos objetos `Fecha` (líneas 9 y 10): `d1` se inicializa con Diciembre 27, 2010 y `d2` se inicializa de manera predeterminada con Enero 1, 1900. El constructor de `Fecha` (definido en la figura 10.7, líneas 13 a 16) llama a `establecerFecha` (definida en la figura 10.7, líneas 19 a 38) para validar el mes, día y año especificados. Los valores inválidos para mes, día o año producen excepciones `invalid_argument`.

En la línea 12 de `main` (figura 10.8) se imprime cada uno de los objetos `Fecha`, usando el operador de inserción de flujo sobrecargado (definido en la figura 10.7, líneas 107 a 114). En la línea 13 de `main` se utiliza el operador sobrecargado `+=` (definido en la figura 10.7, líneas 59 a 65) para sumar siete días a `d1`. En la línea 15 de la figura 10.8 se utiliza la función `establecerFecha` para establecer `d2` a Febrero 28, 2008, que es un año bisiesto. Después, en la línea 17 se preincrementa `d2` para mostrar que la fecha se incrementa en forma apropiada a Febrero 29. A continuación, en la línea 19 se crea un objeto `Fecha` llamado `d3`, el cual se inicializa con la fecha Julio 13, 2010. Luego en la línea 23 se incrementa `d3` en 1 con el operador de incremento prefijo sobrecargado. En las líneas 21 a 24 se imprime `d3` antes y después de la operación de preincremento, para confirmar que haya funcionado en forma apropiada. Por último, en la línea 28 se incrementa `d3` con el operador de incremento postfijo sobrecargado. En las líneas 26 a 29 se imprime `d3` antes y después de la operación de postincremento, para confirmar que trabaje en forma apropiada.

Operador de incremento prefijo de la clase `Fecha`

La sobrecarga del operador de incremento prefijo es un proceso directo. El operador de incremento prefijo (definido en la figura 10.7, líneas 41 a 45) llama a la función utilitaria `ayudaIncremento` (definida en la figura 10.7, líneas 87 a 104) para incrementar la fecha. Esta función trata con "envolturas" o "acarreos" que ocurren cuando incrementamos el último día del mes. Estos acarreos requieren incrementar el mes. Si éste ya es 12, entonces el año también se debe incrementar y el mes debe establecerse en 1. La función `ayudaIncremento` usa la función `finDeMes` para determinar si se llegó al fin del mes e incrementar el día en forma apropiada.

El operador de incremento prefijo sobrecargado devuelve una referencia al objeto `Fecha` actual (es decir, el que se acaba de incrementar). Esto ocurre debido a que el objeto actual, `*this`, se devuelve como un objeto `Date &`. Esto permite que un objeto `Fecha` preincrementado se utilice como un *lvalue*, que es como trabaja el operador de incremento prefijo integrado para los tipos fundamentales.

Operador de incremento postfijo de la clase `Fecha`

La sobrecarga del operador de incremento postfijo (definido en la figura 10.7, líneas 49 a 56) es más complicada. Para emular el efecto del postincremento, debemos devolver una *copia sin incrementar* del objeto `Fecha`. Por ejemplo, si la variable `int x` tiene el valor `7`, la instrucción

```
cout << x++ << endl;
```

imprime el valor *original* de la variable `x`. Por lo tanto, nos gustaría que el operador de incremento postfijo operara de la misma forma con un objeto `Fecha`. Al entrar a `operator++`, guardamos el objeto actual (`*this`) en `temp` (línea 51). A continuación, llamamos a `ayudaIncremento` para incrementar el objeto `Fecha` actual. Después, en la línea 55 se devuelve la *copia sin incrementar* del objeto previamente almacenado en `temp`. Esta función *no puede* devolver una referencia al objeto `Fecha` local llamado `temp`, debido a que una variable local se destruye cuando la función en la que se declara termina su ejecución. Así, al declarar el tipo de valor de retorno para esta función como `Fecha &` se devolvería una referencia a un objeto que ya no existe.

Error común de programación 10.1

Devolver una referencia (o un apuntador) a una variable local es un error común para el cual la mayoría de los compiladores generarán una advertencia.

10.9 Administración dinámica de memoria

Es posible controlar la *asignación* y *desasignación* de memoria en un programa, para objetos y arreglos de cualquier tipo integrado o definido por el usuario. Esto se conoce como **administración dinámica de memoria** y se lleva a cabo mediante los operadores **`new`** y **`delete`**. Usaremos estas herramientas para implementar nuestra clase `Array` en la siguiente sección.

Es posible usar el operador `new` para **asignar** (reservar) en forma dinámica la cantidad exacta de memoria requerida para contener un objeto o arreglo integrado en tiempo de ejecución. El objeto o arreglo integrado se crea en el **almacenamiento libre** (también conocido como **heap** o **montón**): *una región de memoria asignada a cada programa para almacenar objetos asignados en forma dinámica*.[2] Una vez que se asigna la memoria en el almacenamiento libre, podemos obtener acceso a ésta mediante el apuntador que devuelve el operador `new`. Cuando ya no necesite la memoria, puede *devolverla* al almacenamiento libre mediante el uso del operador `delete` para **desasignar** (*liberar*) la memoria, que las operaciones posteriores con `new` pueden *reutilizar*.[3]

Obtener memoria dinámica mediante `new`

Considere la siguiente instrucción:

```
Tiempo *tiempoPtr = new Tiempo();
```

El operador `new` asigna el almacenamiento del tamaño apropiado para un objeto de tipo `Tiempo`, llama al constructor predeterminado para inicializar el objeto y devuelve un apuntador al tipo especificado a la derecha del operador `new` (es decir, un `Tiempo *`). Si `new` no puede encontrar suficiente espacio en memoria para el objeto, lanza una excepción para indicar que ocurrió un error.

---

2 El operador `new` podría fallar al tratar de obtener la memoria necesaria, en cuyo caso se produciría una excepción `bad_alloc`. En el capítulo 17 veremos cómo lidiar con las fallas al usar `new`.

3 Los operadores `new` y `delete` *pueden* sobrecargarse, pero esto se encuentra más allá del alcance del libro. Si sobrecarga `new`, entonces debe sobrecargar `delete` en el *mismo alcance* para evitar errores sutiles de administración de la memoria dinámica.

Liberación de la memoria dinámica mediante `delete`

Para destruir un objeto asignado en forma dinámica y liberar el espacio que éste ocupa, use el operador `delete` de la siguiente manera:

```
delete tiempoPtr;
```

Esta instrucción primero *llama al destructor para el objeto al que apunta* `tiempoPtr`*, y después desasigna la memoria asociada con el objeto, devolviendo la memoria al almacenamiento libre*.

Error común de programación 10.2

Si no se libera la memoria asignada en forma dinámica cuando ya no es necesaria, el sistema se puede quedar sin memoria antes de tiempo. A esto se le conoce algunas veces como "fuga de memoria".

Tip para prevenir errores 10.1

No elimine la memoria que no haya sido asignada por `new`*. Esto producirá un comportamiento indefinido.*

Tip para prevenir errores 10.2

Después de eliminar un bloque de memoria asignada en forma dinámica, asegúrese de no eliminar el mismo bloque de nuevo. Una forma de protegerse de esto es establecer de inmediato el apuntador a `nullptr`*. La acción de eliminar un* `nullptr` *no tiene efecto.*

Inicialización de la memoria dinámica

Podemos proporcionar un inicializador para una variable de tipo fundamental recién creada, como en

```
double *ptr = new double( 3.14159 );
```

la cual inicializa un valor `double` recién creado con `3.14159` y asigna el apuntador resultante a `ptr`. La misma sintaxis se puede utilizar para especificar una lista de argumentos separados por comas para el constructor de un objeto. Por ejemplo,

```
Tiempo *tiempoPtr = new Tiempo( 12, 45, 0 );
```

inicializa un objeto `Tiempo` recién creado con 12:45 PM y asigna el apuntador resultante a `tiempoPtr`.

Asignación dinámica de arreglos integrados mediante `new []`

También es posible usar el operador `new` para asignar arreglos integrados en forma dinámica. Por ejemplo, un arreglo entero de 10 elementos se puede asignar a `arregloCalificaciones` de la siguiente manera:

```
int *arregloCalificaciones = new int[ 10 ]();
```

esta instrucción declara el apuntador `int` llamado `arregloCalificaciones` y le asigna un apuntador al primer elemento de un arreglo de 10 elementos `int` asignado en forma dinámica. Los paréntesis que van después de `new int[10]` inicializan los elementos del arreglo; los tipos numéricos fundamentales se establecen en `0`, los tipos `bool` se establecen en `false`, los apuntadores en `nullptr` y los objetos de clases se inicializan mediante sus constructores predeterminados. El tamaño de un arreglo creado en tiempo de compilación se debe especificar mediante una expresión integral constante; sin embargo, el tamaño de un arreglo asignado en forma dinámica se puede especificar mediante *cualquier* expresión integral no negativa que se pueda evaluar en tiempo de ejecución.

C++11: uso de un inicializador de listas con un arreglo integrado asignado en forma dinámica

Antes de C++11, al asignar un arreglo de objetos en forma dinámica, *no se podían* pasar argumentos al constructor de cada objeto, sino que cada objeto se inicializaba mediante su constructor *predeterminado*.

En C++11 podemos usar un inicializador de listas para inicializar los elementos de un arreglo integrado asignado en forma dinámica, como en:

```
int *arregloCalificaciones = new int[ 10 ]{};
```

El conjunto vacío de llaves que se muestra aquí indica que debe usarse la *inicialización predeterminada* para cada elemento; para los tipos fundamentales, cada elemento se establece en 0. Las llaves también pueden contener una lista de inicializadores separados por comas para los elementos del arreglo.

Liberación de los arreglos integrados asignados en forma dinámica mediante `delete []`

Para desasignar la memoria a la que apunta `arregloCalificaciones`, use la siguiente instrucción:

```
delete [] arregloCalificaciones;
```

Si el apuntador apunta a un arreglo integrado de objetos, la instrucción primero llama al destructor para cada objeto en el arreglo, y después desasigna la memoria. Si la instrucción anterior *no* incluyera los corchetes (`[]`) y `arregloCalificaciones` apuntara a un arreglo integrado de objetos, el resultado sería *indefinido: algunos compiladores llaman al destructor sólo para el primer objeto en el arreglo. El uso de* `delete` *o* `delete []` *en un* `nullptr` *no tiene efecto.*

Error común de programación 10.3

El uso de `delete` *en vez de* `delete []` *para los arreglos de objetos puede provocar errores lógicos en tiempo de ejecución. Para asegurar que cada objeto en el arreglo reciba una llamada al destructor, siempre debe eliminar la memoria asignada como un arreglo con el operador* `delete []`*. De manera similar, siempre debe eliminar la memoria asignada como un elemento individual con el operador* `delete`*; el resultado de eliminar un solo objeto con el operador* `delete []` *es indefinido.*

C++11: administración de la memoria asignada en forma dinámica mediante `unique_ptr`

El nuevo apuntador `unique_ptr` de C++ es un "apuntador inteligente" para administrar la memoria asignada en forma dinámica. Cuando un `unique_ptr` queda fuera de alcance, su destructor devuelve *automáticamente* la memoria administrada al almacenamiento libre. En el capítulo 17, introduciremos el apuntador `unique_ptr` y le mostraremos cómo usarlo para administrar objetos asignados en forma dinámica o arreglos integrados asignados en forma dinámica.

10.10 Caso de estudio: la clase `Array`

En el capítulo 8 vimos los arreglos integrados. Los arreglos basados en apuntadores tienen varios problemas:

- Un programa puede "salirse" fácilmente de cualquier extremo de un arreglo integrado, ya que *C++ no comprueba si los subíndices están fuera del rango de un arreglo* (de todas formas se puede hacer esto explícitamente).
- Los arreglos integrados de tamaño n deben enumerar sus elementos así: 0, …, n – 1; *no* se permiten rangos de subíndices alternados.
- No es posible recibir ni enviar todo un arreglo integrado a la vez; se debe leer o escribir cada elemento de manera individual (a menos que el arreglo sea una cadena C con terminación nula).
- Dos arreglos integrados no pueden compararse significativamente con los operadores de igualdad o relacionales (debido a que los nombres de los arreglos son simplemente apuntadores a la dirección en la que empiezan los arreglos en memoria y dos arreglos siempre estarán en distintas ubicaciones de memoria).

- Cuando se pasa un arreglo integrado a una función de propósito general diseñada para manejar arreglos de cualquier tamaño, el *tamaño* del arreglo debe pasarse como argumento adicional.
- Un arreglo integrado no puede *asignarse* a otro con el (los) operador(es) de asignación.

El desarrollo de clases es una actividad interesante, creativa e intelectualmente desafiante; siempre con el objetivo de *crear clases valiosas*. Con C++ podemos implementar herramientas más robustas para los arreglos, mediante las clases y la sobrecarga de operadores, como se hizo con las plantillas de clases `array` y `vector` en la Biblioteca estándar de C++. En esta sección desarrollaremos nuestra propia clase de arreglos personalizada, que es preferible a usar arreglos integrados. Al referirnos a los "arreglos" en este caso de estudio, queremos indicar los arreglos integrados.

En este ejemplo, crearemos una poderosa clase `Array` que realiza comprobación de rangos para asegurar que los subíndices permanezcan dentro de los límites del objeto `Array`. La clase permite asignar un objeto `Array` a otro mediante el operador de asignación. Los objetos de la clase `Array` conocen su tamaño, por lo que éste no se necesita pasar por separado como argumento a las funciones que reciben parámetros `Array`. Pueden recibirse o enviarse objetos `Array` completos mediante los operadores de extracción de flujo e inserción de flujo, respectivamente. Pueden realizarse comparaciones entre objetos `Array` mediante los operadores de igualdad == y !=.

10.10.1 Uso de la clase `Array`

El programa de las figuras 10.9 a 10.11 muestra la clase `Array` y sus operadores sobrecargados. Primero recorremos `main` (figura 10.9) y los resultados del programa; después consideramos la definición de la clase (figura 10.10) y cada una de las definiciones de sus funciones miembro (figura 10.11).

```
1  // Fig. 10.9: fig10_09.cpp
2  // Programa de prueba de la clase Array.
3  #include <iostream>
4  #include <stdexcept>
5  #include "Array.h"
6  using namespace std;
7
8  int main()
9  {
10   Array enteros1( 7 ); // objeto Array de siete elementos
11   Array enteros2; // objeto Array de 10 elementos creado de manera predeterminada
12
13   // imprime el tamaño y contenido de enteros1
14   cout << "El tamanio del objeto Array enteros1 es "
15      << enteros1.obtenerTamanio()
16      << "\nEl objeto Array despues de la inicializacion es:\n" << enteros1;
17
18   // imprime el tamaño y el contenido de enteros2
19   cout << "\nEl tamanio del objeto Array enteros2 es "
20      << enteros2.getTamanio()
21      << "\nEl objeto Array despues de la inicializacion es:\n" << enteros2;
22
23   // recibe e imprime enteros1 y enteros2
24   cout << "\nIntroduzca 17 enteros:" << endl;
25   cin >> enteros1 >> enteros2;
```

Fig. 10.9 | Programa de prueba de la clase `Array` (parte 1 de 3).

```
26
27     cout << "\nDespues de la entrada, los objetos Array contienen:\n"
28        << "enteros1:\n" << enteros1
29        << "enteros2:\n" << enteros2;
30
31     // usa el operador de desigualdad (!=) sobrecargado
32     cout << "\nEvaluando: enteros1 != enteros2" << endl;
33
34     if ( enteros1 != enteros2 )
35        cout << "enteros1 y enteros2 no son iguales" << endl;
36
37     // crea el objeto Array enteros3, usando enteros1 como
38     // inicializador; imprime el tamaño y el contenido
39     Array enteros3( enteros1 ); // invoca el constructor de copia
40
41     cout << "\nEl tamanio del objeto Array enteros3 es "
42        << enteros3.obtenerTamanio()
43        << "\nObjeto Array despues de la inicializacion:\n" << enteros3;
44
45     // usa el operador de asignación (=) sobrecargado
46     cout << "\nAsignando enteros2 a enteros1:" << endl;
47     enteros1 = enteros2; // observe que el objeto Array de destino es más pequeño
48
49     cout << "enteros1:\n" << enteros1
50        << "enteros2:\n" << enteros2;
51
52     // usa el operador de igualdad (==) sobrecargado
53     cout << "\nEvaluando: enteros1 == enteros2" << endl;
54
55     if ( enteros1 == enteros2 )
56        cout << "enteros1 y enteros2 son iguales" << endl;
57
58     // usa el operador de subíndice sobrecargado para crear rvalue
59     cout << "\nenteros1[5] es " << enteros1[ 5 ];
60
61     // usa el operador de subíndice sobrecargado para crear lvalue
62     cout << "\n\nAsignando 1000 a enteros1[5]" << endl;
63     enteros1[ 5 ] = 1000;
64     cout << "enteros1:\n" << enteros1;
65
66     // trata de usar un subíndice fuera de rango
67     try
68     {
69        cout << "\nTrata de asignar 1000 a enteros1[15]" << endl;
70        enteros1[ 15 ] = 1000; // ERROR: subíndice fuera de rango
71     } // fin de try
72     catch ( out_of_range &ex )
73     {
74        cout << "Ocurrio una excepcion: " << ex.what() << endl;
75     } // fin de catch
76  } // fin de main
```

Fig. 10.9 | Programa de prueba de la clase `Array` (parte 2 de 3).

```
El tamanio del objeto Array enteros1 es 7
El objeto Array despues de la inicializacion es:
           0           0           0           0
           0           0           0

El tamanio del objeto Array enteros2 es 10
El objeto Array despues de la inicializacion es:
           0           0           0           0
           0           0           0           0
           0           0

Introduzca 17 enteros:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

Despues de la entrada, los objetos Array contienen:
enteros1:
           1           2           3           4
           5           6           7
enteros2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluando: enteros1 != enteros2
enteros1 y enteros2 no son iguales

El tamanio del objeto Array enteros3 es 7
Objeto Array despues de la inicializacion:
           1           2           3           4
           5           6           7

Asignando enteros2 a enteros1:
enteros1:
           8           9          10          11
          12          13          14          15
          16          17

enteros2:
           8           9          10          11
          12          13          14          15
          16          17

Evaluando: enteros1 == enteros2
enteros1 y enteros2 son iguales

enteros1[5] es 13

Asignando 1000 a enteros1[5]
enteros1:
           8           9          10          11
          12        1000          14          15
          16          17

Trata de asignar 1000 a enteros1[15]
Ocurrio una excepcion: Subindice fuera de rango
```

Fig. 10.9 | Programa de prueba de la clase `Array` (parte 3 de 3).

Cómo crear objetos `Array`*, imprimir su tamaño y mostrar su contenido*

El programa empieza por instanciar dos objetos de la clase `Array`: `enteros1` (figura 10.9, línea 10) con siete elementos, y `enteros2` (línea 11) con el tamaño predeterminado de `Array`: 10 elementos (especificado por el prototipo del constructor predeterminado de `Array` en la figura 10.10, línea 14). En las líneas 14 a 16 en la figura 10.9 se utiliza la función miembro `obtenerTamanio` para determinar el tamaño de `enteros1` e imprimir su contenido, usando el operador de inserción de flujo sobrecargado de `Array`. La salida del ejemplo confirma que los elementos del objeto `Array` se establecieron correctamente en cero mediante el constructor. A continuación, en las líneas 19 a 21 se imprime el tamaño del objeto `Array enteros2` y luego se imprime su contenido, usando el operador de inserción de flujo sobrecargado de `Array`.

Uso del operador de inserción de flujo sobrecargado para llenar un objeto `Array`

En la línea 24 se pide al usuario que introduzca 17 enteros. En la línea 25 se utiliza el operador de extracción de flujo sobrecargado de `Array` para leer los primeros siete valores y colocarlos en `enteros1`, y los 10 valores restantes en `enteros2`. En las líneas 27 a 29 se imprimen los dos arreglos con el operador de inserción de flujo sobrecargado de `Array` para confirmar que la entrada se haya realizado de manera correcta.

Uso del operador de desigualdad sobrecargado

En la línea 34 se prueba el operador de desigualdad sobrecargado, evaluando la condición

```
enteros1 != enteros2
```

La salida del programa muestra que los objetos `Array` no son iguales.

Cómo inicializar un nuevo objeto `Array` *con una copia del contenido de un objeto* `Array` *existente*

En la línea 39 se crea una instancia de un tercer objeto `Array` llamado `enteros3`, y se inicializa con una copia del objeto `Array enteros1`. Esto invoca al **constructor de copia** de la clase `Array` para copiar los elementos de `enteros1` a `enteros3`. En breve hablaremos sobre los detalles del constructor de copia. Este constructor de copia también se puede invocar si escribimos la línea 39 de la siguiente manera:

```
Array enteros3 = enteros1;
```

El signo de igual en la instrucción anterior *no* es el operador de asignación. Cuando aparece un signo de igual en la declaración de un objeto, se invoca a un constructor para ese objeto. Esta forma se puede utilizar para pasar sólo un argumento a un constructor; en específico, el valor del lado derecho del símbolo =.

En las líneas 41 a 43 se imprime el tamaño de `enteros3` y luego se imprime su contenido, usando el operador de inserción de flujo sobrecargado de `Array` para confirmar que los elementos de `enteros3` se hayan establecido correctamente mediante el constructor de copia.

Uso del operador de asignación sobrecargado

En la línea 47 se prueba el operador de asignación sobrecargado (=) mediante la asignación de `enteros2` a `enteros1`. En las líneas 49 y 50 se imprime el contenido de ambos objetos `Array` para confirmar que la asignación haya tenido éxito. En un principio el objeto `Array enteros1` contenía 7 enteros y se cambió su tamaño para que pudiera contener una copia de los 10 elementos en `enteros2`. Como veremos más adelante, el operador de asignación sobrecargado realiza esta operación de ajuste de tamaño de una forma que sea transparente para el código cliente.

Uso del operador de igualdad sobrecargado
En la línea 55 se utiliza el operador de igualdad sobrecargado (==) para confirmar que los objetos `enteros1` y `enteros2` sean sin duda *idénticos* después de la asignación en la línea 47.

Uso del operador de subíndice sobrecargado
En la línea 59 se utiliza el operador de subíndice sobrecargado para hacer referencia a `enteros1[5]`: un elemento de `enteros1` dentro del rango. Este nombre con subíndice se utiliza como *rvalue* para imprimir el valor asignado a `enteros1[5]`. En la línea 63 se utiliza `enteros1[5]` como un *lvalue* modificable del lado izquierdo de una instrucción de asignación para asignar un nuevo valor, `1000`, al elemento `5` de `enteros1`. Más adelante veremos que `operator[]` devuelve una referencia para usarla como el *lvalue* modificable, una vez que el operador confirma que `5` es un subíndice válido para `enteros1`.

En la línea 70 se trata de asignar el valor `1000` a `enteros1[15]`; un elemento *fuera de rango*. En este ejemplo, `operator[]` determina que el subíndice está fuera de rango y lanza una excepción `out_of_range`.

Es interesante observar que el *operador de subíndice de arreglo `[]` no está restringido para usarlo sólo con arreglos*; por ejemplo, también se puede utilizar para seleccionar elementos de otros tipos de *clases contenedoras*, como objetos `string` y diccionarios. Además, cuando se definen las funciones `operator[]` *los subíndices ya no tienen que ser enteros*; también se pueden usar caracteres, cadenas o incluso objetos de clases definidas por el usuario. En el capítulo 15 hablaremos sobre la clase `map` de la Biblioteca estándar que permite subíndices `string`.

10.10.2 Definición de la clase `Array`

Ahora que hemos visto cómo opera este programa, vamos a recorrer el encabezado de la clase (figura 10.10). A medida que hagamos referencia a cada función miembro en el encabezado, hablaremos sobre la implementación de esa función en la figura 10.11. En las líneas 34 y 35 de la figura 10.10 se representan los miembros de datos `private` de la clase `Array`. Cada objeto `Array` consiste en un miembro `tamanio` que indica el número de elementos en el objeto `Array` y un apuntador `int (ptr)` que apunta al arreglo de enteros basado en apuntador y asignado en forma dinámica que maneja el objeto `Array`.

```
1  // Fig. 10.10: Array.h
2  // Definición de la clase Array con operadores sobrecargados.
3  #ifndef ARRAY_H
4  #define ARRAY_H
5
6  #include <iostream>
7
8  class Array
9  {
10    friend std::ostream &operator<<( std::ostream &, const Array & );
11    friend std::istream &operator>>( std::istream &, Array & );
12
13 public:
14    explicit Array( int = 10 ); // constructor predeterminado
15    Array( const Array & ); // constructor de copia
16    ~Array(); // destructor
17    size_t obtenerTamanio() const; // devuelve el tamaño
18
19    const Array &operator=( const Array & ); // operador de asignación
20    bool operator==( const Array & ) const; // operador de igualdad
21
```

Fig. 10.10 | Definición de la clase `Array` con operadores sobrecargados (parte 1 de 2).

```
22     // operador de desigualdad; devuelve el opuesto del operador ==
23     bool operator!=( const Array &derecho ) const
24     {
25        return ! ( *this == derecho ); // invoca a Array::operator==
26     } // fin de la función operator!=
27
28     // el operador de subíndice para los objetos no const devuelve un lvalue modificable
29     int &operator[]( int );
30
31     // el operador de subíndice para los objetos const devuelve rvalue
32     int operator[]( int ) const;
33  private:
34     size_t tamanio; // arreglo tamanio basado en apuntador
35     int *ptr; // apuntador al primer elemento del arreglo basado en apuntador
36  }; // fin de la clase Array
37
38  #endif
```

Fig. 10.10 | Definición de la clase `Array` con operadores sobrecargados (parte 2 de 2).

```
1   // Fig. 10.11: Array.cpp
2   // Definiciones de las funciones miembro y friend de la clase Array.
3   #include <iostream>
4   #include <iomanip>
5   #include <stdexcept>
6
7   #include "Array.h" // Definición de la clase Array
8   using namespace std;
9
10  // constructor predeterminado para la clase Array (tamaño predeterminado 10)
11  Array::Array( int tamanioArreglo )
12     : tamanio( tamanioArreglo > 0 ? tamanioArreglo :
13          throw invalid_argument( "El tamanio del arreglo debe ser mayor que 0" ) ),
14       ptr( new int[ tamanio ] )
15  {
16     for ( size_t i = 0; i < tamanio; ++i )
17        ptr[ i ] = 0; // establece el elemento del arreglo basado en apuntador
18  } // fin del constructor predeterminado de Array
19
20  // constructor de copia para la clase Array;
21  // debe recibir una referencia a un objeto Array
22  Array::Array( const Array &arregloACopiar )
23     : tamanio( arregloACopiar.tamanio ),
24       ptr( new int[ tamanio ] )
25  {
26     for ( size_t i = 0; i < tamanio; ++i )
27        ptr[ i ] = arregloACopiar.ptr[ i ]; // lo copia en el objeto
28  } // fin del constructor de copia de Array
29
30  // destructor para la clase Array
31  Array::~Array()
32  {
```

Fig. 10.11 | Definiciones de las funciones miembro y funciones `friend` de la clase `Array` (parte 1 de 3).

```
33    delete [] ptr; // libera el espacio del arreglo basado en apuntador
34  } // fin del destructor
35
36  // devuelve el número de elementos del objeto Array
37  size_t Array::obtenerTamanio() const
38  {
39    return tamanio; // número de elementos en el objeto Array
40  } // fin de la función obtenerTamanio
41
42  // operador de asignación sobrecargado;
43  // devolución de const evita: ( a1 = a2 ) = a3
44  const Array &Array::operator=( const Array &derecho )
45  {
46    if ( &derecho != this ) // evita la auto-asignación
47    {
48      // para los objetos Array de distintos tamaños, desasigna el arreglo
49      // original del lado izquierdo, después asigna el nuevo arreglo del lado
           izquierdo
50      if ( tamanio != derecho.tamanio )
51      {
52        delete [] ptr; // libera espacio
53        tamanio = derecho.tamanio; // cambia el tamaño de este objeto
54        ptr = new int[ tamanio ]; // crea espacio para la copia del arreglo
55      } // fin del if interior
56
57      for ( size_t i = 0; i < tamanio; ++i )
58        ptr[ i ] = derecho.ptr[ i ]; // copia el arreglo en el objeto
59    } // fin del if exterior
60
61    return *this; // permite x = y = z, por ejemplo
62  } // fin de la función operator=
63
64  // determina si dos objetos Array son iguales y
65  // devuelve true, en caso contrario devuelve false
66  bool Array::operator==( const Array &derecho ) const
67  {
68    if ( tamanio != derecho.tamanio )
69      return false; // arreglos con distinto número de elementos
70
71    for ( size_t i = 0; i < tamanio; ++i )
72      if ( ptr[ i ] != derecho.ptr[ i ] )
73        return false; // el contenido de los objetos Array no es igual
74
75    return true; // los objetos Array son iguales
76  } // fin de la función operator==
77
78  // operador de subíndice sobrecargado para objetos Array no const;
79  // la devolución de una referencia crea un lvalue modificable
80  int &Array::operator[]( int subindice )
81  {
82    // comprueba error de subíndice fuera de rango
83    if ( subindice < 0 || subindice >= tamanio )
84      throw out_of_range( "Subindice fuera de rango" );
85
```

Fig. 10.11 | Definiciones de las funciones miembro y funciones `friend` de la clase `Array` (parte 2 de 3).

```
86    return ptr[ subindice ]; // devolución de referencia
87 } // fin de la función operator[]
88
89 // operador de subíndice sobrecargado para objetos Array const
90 // devolución de referencia const crea un rvalue
91 int Array::operator[]( int subindice ) const
92 {
93    // comprueba error de subíndice fuera de rango
94    if ( subindice < 0 || subindice >= tamanio )
95       throw out_of_range( "Subindice fuera de rango" );
96
97    return ptr[ subindice ]; // devuelve una copia de este elemento
98 } // fin de la función operator[]
99
100 // operador de entrada sobrecargado para la clase Array;
101 // recibe valores para el objeto Array completo
102 istream &operator>>( istream &entrada, Array &a )
103 {
104    for ( size_t i = 0; i < a.tamanio; ++i )
105       entrada >> a.ptr[ i ];
106
107    return entrada; // permite cin >> x >> y;
108 } // fin de la función
109
110 // operador de salida sobrecargado para la clase Array
111 ostream &operator<<( ostream &salida, const Array &a )
112 {
113    // imprime arreglo private basado en ptr
114    for ( size_t i = 0; i < a.tamanio; ++i )
115    {
116       salida << setw( 12 ) << a.ptr[ i ];
117
118       if ( ( i + 1 ) % 4 == 0 ) // 4 números por fila de salida
119          salida << endl;
120    } // fin de for
121
122    if ( a.tamanio % 4 != 0 ) // fin de la última línea de salida
123       salida << endl;
124
125    return salida; // permite cout << x << y;
126 } // fin de la función operator<<
```

Fig. 10.11 | Definiciones de las funciones miembro y funciones `friend` de la clase `Array` (parte 3 de 3).

Cómo sobrecargar los operadores de inserción de flujo y extracción de flujo como funciones `friend`

En las líneas 10 y 11 de la figura 10.10 se declaran el operador de inserción de flujo sobrecargado y el operador de extracción de flujo sobrecargado como funciones `friend` de la clase `Array`. Cuando el compilador ve una expresión como `cout << objetoArray`, invoca a la *función no miembro* `operator<<` con la llamada

```
operator<<( cout, objetoArray )
```

Cuando el compilador ve una expresión como `cin >> objetoArray`, invoca a la *función no miembro* `operator>>` con la llamada

```
operator>>( cin, objetoArray )
```

Observamos de nuevo que estas funciones de operador de inserción de flujo y operador de extracción de flujo *no pueden* ser miembros de la clase `Array`, ya que el objeto `Array` siempre se menciona del lado *derecho* del operador de inserción de flujo o del operador de extracción de flujo.

La función `operator<<` (definida en la figura 10.11, líneas 111 a 126) imprime el número de elementos indicados por `tamanio` a partir del arreglo de enteros al que apunta `ptr`. La función `operator>>` (definida en la figura 10.11, líneas 102 a 108) introduce los datos directamente en el arreglo al que apunta `ptr`. Cada una de estas funciones de operador devuelve una referencia apropiada para permitir instrucciones de salida o entrada *en cascada*, respectivamente. Estas funciones tienen acceso a los datos `private` de un objeto `Array`, debido a que estas funciones se declaran como funciones `friend` de la clase `Array`. Podríamos haber usado las funciones `obtenerTamanio` y `operator[]` de la clase `Array` en los cuerpos de `operator<<` y `operator>>`, en cuyo caso estas funciones de operador no tendrían que ser funciones `friend` de la clase `Array`.

Podría verse tentado a sustituir la instrucción `for` controlada por contador en las líneas 104 y 105, además de muchas de las otras instrucciones `for` en la implementación de la clase `Array`, con la instrucción `for` basada en rango de C++11. Por desgracia, esta instrucción `for` basada en rango *no* funciona con los arreglos integrados que se asignan en forma dinámica.

Constructor predeterminado de `Array`

En la línea 14 de la figura 10.10 se declara el *constructor predeterminado* para la clase y se especifica un tamaño predeterminado de 10 elementos. Cuando el compilador ve una declaración como la línea 11 de la figura 10.9, invoca al constructor predeterminado de `Array` para establecer su tamaño en 10 elementos. El constructor predeterminado (definido en la figura 10.11, líneas 11 a 18) valida y asigna el argumento al miembro de datos `tamanio`, utiliza `new` para obtener la memoria para la *representación interna basada en apuntador* de este arreglo y asigna el apuntador devuelto por `new` al miembro de datos `ptr`. Después, el constructor utiliza una instrucción `for` para establecer todos los elementos del arreglo en cero. Es posible tener una clase `Array` que no inicialice sus miembros si, por ejemplo, estos miembros se van a leer más adelante en cierto momento; pero esto se considera como mala práctica de programación. Los objetos `Array` y *los objetos en general, deben inicializarse de manera apropiada al momento de crearse*.

Constructor de copia de `Array`

En la línea 15 de la figura 10.10 se declara un *constructor de copia* (definido en la figura 10.11, líneas 22 a 28) que inicializa un objeto `Array`, para lo cual crea una copia de un objeto `Array` existente. *Dicha copia debe realizarse con cuidado, para evitar el problema de dejar ambos objetos* `Array` *apuntando a la misma memoria asignada en forma dinámica*. Esto es exactamente el problema que ocurriría con la *copia predeterminada a nivel de miembros*, si el compilador tiene permitido definir un constructor de copia predeterminado para esta clase. Los constructores de copia se invocan cada vez que se necesita una copia de un objeto, como cuando:

- se pasa un objeto por valor a una función;
- se devuelve un objeto por valor de una función, o
- se inicializa un objeto con una copia de otro objeto de la misma clase.

El constructor de copia se llama en una declaración cuando se instancia un objeto de la clase `Array` y se inicializa con otro objeto de la clase `Array`, como en la declaración en la línea 39 de la figura 10.9.

El constructor de copia para `Array` copia el `tamanio` del objeto `Array` inicializador en el miembro de datos `tamanio`, usa `new` para obtener la memoria para la representación interna basada en apuntador de este objeto `Array` y asigna el apuntador devuelto por `new` al miembro de datos `ptr`. Después, el constructor de copia utiliza una instrucción `for` para copiar todos los elementos del objeto `Array` inicializador en el nuevo objeto `Array`. Un objeto de una clase puede mirar los datos `private` de cualquier otro objeto de esa clase (usando un manejador que indique a cuál objeto acceder).

Observación de Ingeniería de Software 10.3

El argumento para un constructor de copia debe ser una referencia `const` *para permitir que se copie un objeto* `const`.

Error común de programación 10.4

Si el constructor de copia simplemente copió el apuntador del objeto de origen al apuntador del objeto de destino, entonces ambos objetos apuntarían a la misma *memoria asignada en forma dinámica. El primer destructor en ejecutarse eliminaría entonces la memoria asignada en forma dinámica, y el* `ptr` *del otro objeto apuntaría a una parte de memoria que ya no está asignada, una situación conocida como* ***apuntador suspendido****; probablemente esto produciría un grave error en tiempo de ejecución (como la terminación anticipada del programa) a la hora de utilizar el apuntador.*

Destructor de `Array`

En la línea 16 de la figura 10.10 se declara el destructor para la clase (definido en la figura 10.11, líneas 31 a 34). El destructor se invoca cuando un objeto de la clase `Array` queda fuera de alcance. El destructor usa `delete []` para liberar la memoria asignada en forma dinámica por `new` en el constructor.

Tip para prevenir errores 10.3

Si después de eliminar la memoria asignada en forma dinámica, el apuntador sigue existiendo *en memoria, establezca el valor del apuntador en* `nullptr` *para indicar que el apuntador ya no apunta a la memoria en el almacenamiento libre. Al establecer el apuntador a* `nullptr`*, el programa pierde el acceso a ese espacio en el almacenamiento libre, que podría reasignarse para un propósito distinto. Si no establece el apuntador a* `nullptr`*, su código podría* acceder de manera inadvertida a la memoria reasignada, provocando errores lógicos sutiles no repetibles. *No establecimos* `ptr` *a* `nullptr` *en la línea 33 de la figura 10.11 debido a que, después de que se ejecuta el destructor, el objeto* `Array` *ya no existe en la memoria.*

Función miembro `obtenerTamanio`

En la línea 17 de la figura 10.10 se declara la función `obtenerTamanio` (definida en la figura 10.11, líneas 37 a 40), que devuelve el número de elementos en el objeto `Array`.

Operador de asignación sobrecargado

En la línea 19 de la figura 10.10 se declara la función del operador de asignación sobrecargado para la clase. Cuando el compilador ve la expresión `enteros1 = enteros2` en la línea 47 de la figura 10.9, invoca a la función miembro `operator=` con la llamada

```
enteros1.operator=( enteros2 )
```

La implementación de la función miembro `operator=` (figura 10.11, líneas 44 a 62) prueba la **auto-asignación** (línea 46), en la que un objeto de la clase `Array` se asigna a sí mismo. Cuando `this` es igual a la dirección del operando `derecho`, se intenta una *auto-asignación*, por lo que se omite la asignación (es decir, el objeto ya es él mismo; en un momento veremos por qué la auto-asignación es peligrosa). Si no es una auto-asignación, entonces la función miembro determina si los tamaños de los dos arreglos son idénticos (línea 50); en ese caso, el arreglo original de enteros en el objeto `Array` del lado izquierdo *no* se reasigna. En caso contrario, `operator=` utiliza `delete[]` (línea 52) para liberar la memoria originalmente asignada al arreglo de destino, copia el `tamanio` del arreglo de origen al `tamanio` del arreglo de destino (línea 53), usa `new` para asignar memoria para el arreglo de destino y coloca el apuntador devuelto por `new` en el miembro `ptr` del arreglo. Después, la instrucción `for` en las líneas 57 y 58 copia los elementos del arreglo de origen al arreglo de destino. Sin importar que ésta sea o no

una auto-asignación, la función miembro devuelve el objeto actual (es decir, `*this` en la línea 61) como una referencia constante; esto permite asignaciones de objetos `Array` en cascada, como `x = y = z`, pero evita las asignaciones como `(x = y) = z` debido a que `z` no puede asignarse a la referencia `const Array` que devuelve `(x = y)`. Si ocurre la auto-asignación, y la función `operator=` no probó este caso, `operator=` copiaría de manera innecesaria los elementos del objeto `Array` a este mismo objeto.

Observación de Ingeniería de Software 10.4

Un constructor de copia, un destructor y un operador de asignación sobrecargado se proporcionan generalmente como un grupo para cualquier clase que utilice memoria asignada en forma dinámica. Con la adición de la semántica de movimiento en C++11, también deberían proporcionarse otras funciones, como veremos en el capítulo 24.

Error común de programación 10.5

Si no se proporcionan un operador de asignación sobrecargado y un constructor de copia para una clase cuando los objetos de esa clase contienen apuntadores a memoria asignada en forma dinámica, se puede producir un error lógico.

C++11: constructor de movimiento y operador de asignación de movimiento

C++11 agrega las nociones de un *constructor de movimiento* y de un *operador de asignación de movimiento*.

C++11: cómo eliminar funciones miembro no deseadas de su clase

Antes de C++11, era posible evitar que los objetos de clases se *copiaran* o *asignaran* si se declaraban como `private` tanto el constructor de copia de la clase, como el operador de asignación sobrecargado. A partir de C++11, sólo hay que *eliminar* estas funciones de la clase. Para hacer esto en `Array`, reemplace los prototipos en las líneas 15 y 19 de la figura 10.10 con lo siguiente:

```
Array( const Array & ) = delete;
const Array &operator=( const Array & ) = delete;
```

Aunque es posible eliminar *cualquier* función miembro, esta característica se utiliza en forma más común con las funciones miembro que el compilador puede *generar en forma automática*: el constructor predeterminado, el constructor de copia, el operador de asignación y, en C++11, el constructor de movimiento y el operador de asignación de movimiento.

Operadores de igualdad y desigualdad sobrecargados

En la línea 20 de la figura 10.10 se declara el operador de igualdad sobrecargado (==) para la clase. Cuando el compilador ve la expresión `enteros1==enteros2` en la línea 55 de la figura 10.9, invoca a la función miembro `operator==` con la llamada

```
enteros1.operator==( enteros2 )
```

La función miembro `operator==` (definida en la figura 10.11, líneas 66 a 76) devuelve inmediatamente `false`, si los miembros `tamanio` de los arreglos no son iguales. En caso contrario, `operator==` compara cada par de elementos. Si todos son iguales, la función devuelve `true`. El primer par de elementos que difieran provoca que la función devuelva `false` de inmediato.

En las líneas 23 a 26 de la figura 10.9 se define el operador de desigualdad sobrecargado (`!=`) para la clase. La función miembro `operator!=` utiliza la función `operator==` sobrecargada para determinar si un objeto `Array` es *igual* a otro, y después devuelve el *valor opuesto* de ese resultado. Al escribir `operator!=` de

esta forma, podemos *reutilizar* `operator==`, lo cual *reduce la cantidad de código que debemos escribir en la clase*. Observe además que la definición completa de la función para `operator!=` está en el archivo de encabezado de `Array`. Esto permite al compilador *poner en línea* la definición de `operator!=`.

Operadores de subíndice sobrecargados

En las líneas 29 y 32 de la figura 10.10 se declaran dos operadores de subíndice sobrecargados (definidos en la figura 10.11, en las líneas 80 a 87 y 91 a 98, respectivamente). Cuando el compilador ve la expresión `enteros1[5]` (figura 10.9, línea 59), invoca a la función miembro `operator[]` sobrecargada, para lo cual genera la llamada

```
enteros1.operator[]( 5 )
```

El compilador crea una llamada a la versión `const de operator[]` (figura 10.11, líneas 91 a 98) cuando se utiliza el operador de subíndice en un objeto `const Array`. Por ejemplo, si se pasa un objeto `Array` a una función que recibe este objeto como un `const Array &` llamado `z`, entonces la versión `const` de `operator[]` debe ejecutar una instrucción tal como

```
cout << z[ 3 ] << endl;
```

Recuerde que un programa sólo puede invocar a las funciones miembro `const` de un objeto `const`.

Cada definición de `operator[]` determina si el subíndice que recibe como argumento está *dentro del rango*. Si no es así, cada función lanza una excepción `out_of_range`. Si el subíndice está en el rango, la versión no `const` de `operator[]` devuelve el elemento apropiado del arreglo como una referencia, para que se pueda utilizar como un *lvalue* modificable (por ejemplo, del lado *izquierdo* de una instrucción de asignación). Si el subíndice está en el rango, la versión `const` de `operator[]` devuelve una copia del elemento apropiado del arreglo.

C++11: cómo administrar la memoria asignada en forma dinámica mediante `unique_ptr`

En este caso de estudio, el destructor de la clase `Array` utilizó `delete []` para devolver el arreglo integrado asignado en forma dinámica al almacenamiento libre. Si recuerda, C++11 le permite usar `unique_ptr` para asegurar que se elimine esta memoria asignada en forma dinámica cuando el objeto `Array` quede fuera de alcance. En el capítulo 17 (en el sitio web), introduciremos `unique_ptr` y le mostraremos cómo usarlo para administrar objetos asignados en forma dinámica o arreglos integrados asignados en forma dinámica.

C++11: cómo pasar un inicializador de lista a un constructor

En la figura 7.4 le mostramos cómo inicializar un objeto `array` con una lista de inicializadores separados por comas, encerrados entre llaves, como en:

```
array< int, 5 > n = { 32, 27, 64, 18, 95 };
```

En la sección 4.10 vimos que ahora C++11 nos permite inicializar *cualquier* objeto con un *inicializador de lista* y que la instrucción anterior también se puede escribir sin el signo `=`, como en el siguiente ejemplo:

```
array< int, 5 > n{ 32, 27, 64, 18, 95 };
```

C++11 también nos permite usar inicializadores de lista para declarar objetos de nuestras propias clases. Por ejemplo, ahora podemos proveer un constructor de `Array` que permita las siguientes declaraciones:

```
Array enteros = { 1, 2, 3, 4, 5 };
```

o

```
Array enteros{ 1, 2, 3, 4, 5 };
```

Cada una de las declaraciones anteriores crea un objeto `Array` con cinco elementos que contienen los enteros del 1 al 5.

Para apoyar la inicialización mediante listas, puede definir un constructor que reciba un *objeto* de la plantilla de clase `initializer_list`. Para la clase `Array`, hay que incluir el encabezado `<initializer_list>`. Después se define un constructor con la primera línea:

```
Array::Array( initializer_list< int > lista )
```

Puede determinar el número de elementos en el parámetro `lista` haciendo una llamada a su función miembro `tamanio`. Para obtener cada inicializador y copiarlo en el arreglo integrado asignado de forma dinámica del objeto `Array`, puede usar una instrucción `for` basada en rango como se muestra a continuación:

```
size_t i = 0;
for ( int elemento : lista )
  ptr[ i++ ] = elemento;
```

10.11 Comparación entre los operadores como funciones miembro y no miembro

Ya sea que una función de operador se implemente como *función miembro* o como *función no miembro*, el operador se sigue utilizando de la misma forma en las expresiones. Entonces, ¿cuál implementación es mejor?

Cuando una función de operador se implementa como *función miembro*, el operando de *más a la izquierda* (o el único) debe ser un objeto (o una referencia a un objeto) de la clase del operador. Si el operando izquierdo *debe* ser un objeto de una clase distinta o de un tipo fundamental, esta función de operador se *debe* implementar como una *función no miembro* (como hicimos en la sección 10.5, al sobrecargar << y >> como los operadores de inserción de flujo y de extracción de flujo, respectivamente). Una *función de operador no miembro* puede convertirse en amiga (`friend`) de una clase, si esa función debe acceder directamente a los miembros `private` o `protected` de esa clase.

Las funciones miembro de operador de una clase específica se llaman (de manera *implícita* por el compilador) sólo cuando el operando *izquierdo* de un operador binario es específicamente un objeto de esa clase, o cuando el único operando de un operador unario es un objeto de esa clase.

Operadores conmutativos

Otra razón por la que podríamos elegir una función no miembro para sobrecargar un operador sería para permitir que el operador fuera *conmutativo*. Por ejemplo, suponga que tenemos una *variable de tipo fundamental* llamada `numero`, de tipo `long int`, y un *objeto* llamado `enteroGrande1` de la clase `EnteroEnorme` (una clase en la que los enteros pueden ser arbitrariamente grandes, en vez de estar limitados por el tamaño de palabra del hardware subyacente del equipo; desarrollaremos la clase `EnteroEnorme` en los ejercicios del capítulo). El operador de suma (+) produce un objeto `EnteroEnorme` *temporal* como la suma de un `EnteroEnorme` y un `long int` (como en la expresión `enteroGrande1 + numero`), *o* como la suma de un `long int` y un `EnteroEnorme` (como en la expresión `numero + enteroGrande1`). Por ende, requerimos que el operador de suma sea *conmutativo* (exactamente como es con dos operandos de los tipos fundamentales). El problema es que el objeto de la clase *debe* aparecer del lado *izquierdo* del operador de suma, si ese operador se va a sobrecargar como *función miembro*. Por lo tanto, *también* sobrecargamos el operador como una *función no miembro* para permitir que el `EnteroEnorme` aparezca del lado *derecho* de la suma. La función `operator+`, que lidia con el `EnteroEnorme` a la *izquierda*, puede seguir siendo una *función miembro*. La *función no miembro* simplemente intercambia sus argumentos y llama a la *función miembro*.

10.12 Conversión entre tipos

La mayoría de los programas procesan información de muchos tipos. Algunas veces, todas las operaciones "permanecen dentro de un tipo". Por ejemplo, al sumar un `int` con un `int` se produce un `int`.

Sin embargo, con frecuencia es necesario convertir datos de un tipo a datos de otro tipo. Esto puede ocurrir en asignaciones, en cálculos, en el paso de valores a funciones y en la devolución de valores de las funciones. El compilador sabe cómo llevar a cabo ciertas conversiones entre los tipos fundamentales. Podemos usar *operadores de conversión de tipos* para *forzar* las conversiones entre los tipos fundamentales.

Pero ¿qué hay acerca de los tipos definidos por el usuario? El compilador no puede saber de antemano cómo convertir entre tipos definidos por el usuario, y entre tipos definidos por el usuario y tipos fundamentales, por lo que debemos especificar cómo hacer esto. Dichas conversiones se pueden llevar a cabo mediante **constructores de conversión**: constructores que se pueden llamar con un solo argumento (nos referiremos a éstos como *constructores de un solo argumento*). Dichos constructores pueden convertir objetos de otros tipos (incluyendo los tipos fundamentales) en objetos de una clase específica.

Operadores de conversión

Un **operador de conversión** (también llamado *operador cast*) se puede utilizar para convertir un objeto de una clase en otro tipo. Dicho operador de conversión debe ser una *función miembro no* `static`. El prototipo de función

```
MiClase::operator char *() const;
```

declara una función de operador de conversión de tipos sobrecargada para convertir un objeto de la clase `MiClase` en un objeto `char *` temporal. La función de operador se declara `const` dado que *no* modifica el objeto original. El tipo de valor de retorno de una **función de operador de conversión** sobrecargada es implícitamente el tipo al que se va a convertir el objeto. Si `s` es un objeto de la clase, cuando el compilador ve la expresión `static_cast<char *>(s)`, genera la llamada

```
s.operator char *()
```

para convertir el operando `s` en un `char *`.

Funciones de operador de conversión de tipos sobrecargado

Se pueden definir funciones de operador de conversión de tipos sobrecargado para convertir objetos de tipos definidos por el usuario en tipos fundamentales, o en objetos de otros tipos definidos por el usuario. Los prototipos

```
MiClase::operator int() const;
MiClase::operator OtraClase() const;
```

declaran *funciones de operador de conversión de tipos sobrecargadas*, que pueden convertir un objeto del tipo definido por el usuario `MiClase` en un entero, o en un objeto del tipo definido por el usuario `OtraClase`, respectivamente.

Llamadas implícitas a los operadores de conversión de tipos y a los constructores de conversión

Una de las características agradables de los operadores de conversión de tipos y los constructores de conversión es que, cuando es necesario, el compilador puede llamar a estas funciones de manera *implícita* para crear objetos *temporales*. Por ejemplo, si un objeto `s` de una clase `String` definida por el usuario aparece en un programa, en una ubicación en la que se espera un valor `char *` ordinario, como en

```
cout << s;
```

el compilador puede llamar a la función de operador de conversión de tipos sobrecargado `operator char *` para convertir el objeto en un `char *` y usar el `char *` resultante en la expresión. Al proporcionar este operador de conversión de tipos en nuestra clase `String`, el operador de inserción de flujo *no* tiene que sobrecargarse para imprimir un objeto `String` mediante `cout`.

Observación de Ingeniería de Software 10.5

Cuando se utiliza un constructor de conversión o un operador de conversión para realizar una conversión implícita, *C++ puede aplicar* sólo una *llamada al constructor implícito o a la función de operador (es decir, una sola conversión definida por el usuario) para tratar de relacionar las necesidades de otro operador sobrecargado. El compilador no satisfará las necesidades de un operador sobrecargado realizando una serie de conversiones implícitas definidas por el usuario.*

10.13 Constructores `explicit` y operadores de conversión

Recuerde que hemos estado declarando como `explicit` cada constructor que pueda llamarse con un argumento. Con la excepción de los constructores de copia, cualquier constructor que pueda llamarse con *un solo argumento* y *no* se declare como `explicit` podrá ser usado por el compilador para realizar una *conversión implícita*. El argumento del constructor se convierte en un objeto de la clase en la que éste se define. La conversión es automática y no hay que usar un operador de conversión de tipos. *En ciertas situaciones, las conversiones implícitas son indeseables o propensas a errores*. Por ejemplo, nuestra clase `Array` en la figura 10.10 define un constructor que recibe un solo argumento `int`. La intención de este constructor es crear un objeto `Array` que contenga el número de elementos especificados por el argumento `int`. Pero si este constructor no se declara como `explicit`, puede ser mal utilizado por el compilador para realizar una *conversión implícita*.

Error común de programación 10.6

Por desgracia, el compilador podría usar las conversiones implícitas en casos en los que no se espera, lo cual produce expresiones ambiguas que generan errores de compilación, o errores lógicos en tiempo de ejecución.

Uso accidental de un constructor con un solo argumento como un constructor de conversión

El programa (figura 10.12) utiliza la clase `Array` de las figuras 10.10 y 10.11 para demostrar una conversión implícita inapropiada. Para permitir esta conversión implícita, quitamos la palabra clave `explicit` de la línea 14 en `Array.h` (figura 10.10).

En la línea 11 de `main` (figura 10.12) se instancia el objeto `Array` llamado `enteros1` y se llama al *constructor con un solo argumento*, con el valor `int 7` para especificar el número de elementos en el objeto `Array`. En la figura 10.11 vimos que el constructor de `Array` que recibe un argumento `int` inicializa todos los elementos del arreglo con 0. En la línea 12 se hace una llamada a la función `imprimirArray` (definida en las líneas 17 a 21), la cual recibe como argumento un `const Array &` a un objeto `Array`. La función imprime el número de elementos en su argumento `Array` y el contenido del mismo. En este caso, el tamaño del objeto `Array` es `7`, por lo que se imprimen siete ceros.

En la línea 13 se hace una llamada a la función `imprimirArray` con el valor `int 3` como argumento. Sin embargo, este programa no contiene una función llamada `imprimirArray` que reciba un argumento `int`. Por lo tanto, el compilador determina si la clase `Array` proporciona un *constructor de conversión* que pueda convertir un `int` en un `Array`. Como el constructor de `Array` recibe un argumento `int`, el compilador asume que es un constructor de conversión y lo utiliza para convertir el argumento `3` en un objeto `Array` temporal que contiene tres elementos. Después, el compilador pasa el objeto `Array` temporal a la función `imprimirArray` para imprimir el contenido del objeto `Array`. Por ende, aun cuando no proporcionamos de manera *explícita* una función `imprimirArray` que reciba un argumento `int`, el compilador puede compilar la línea 13. Los resultados muestran el objeto `Array` de tres elementos que contiene ceros.

```
1  // Fig. 10.12: fig10_12.cpp
2  // Constructores con un solo argumento y conversiones implícitas.
3  #include <iostream>
4  #include "Array.h"
5  using namespace std;
6
7  void imprimirArray( const Array & ); // prototipo
8
9  int main()
10 {
11    Array enteros1( 7 ); // arreglo con 7 elementos
12    imprimirArray( enteros1 ); // imprime el objeto Array enteros1
13    imprimirArray( 3 ); // convierte 3 en un objeto Array e imprime su contenido
14 } // fin de main
15
16 // imprime el contenido de un objeto Array
17 void imprimirArray( const Array &arregloAImprimir )
18 {
19    cout << "El objeto Array recibido tiene " << arregloAImprimir.obtenerTamanio()
20       << " elementos. Su contenido es:\n" << arregloAImprimir << endl;
21 } // fin de imprimirArray
```

```
El objeto Array recibido tiene 7 elementos. Su contenido es:
            0           0           0           0
            0           0           0

El objeto Array recibido tiene 3 elementos. Su contenido es:
            0           0           0
```

Fig. 10.12 | Constructores con un solo argumento y conversiones implícitas.

Cómo evitar conversiones implícitas con constructores con un solo argumento

La razón por la que declaramos cada constructor de un solo argumento anteponiéndole la palabra clave `explicit` es para *suprimir las conversiones implícitas mediante los constructores de conversión cuando no deban permitirse dichas conversiones*. Un constructor que se declara `explicit` *no puede* usarse en una conversión *implícita*. En el ejemplo de la figura 10.13, usamos la versión original de `Array.h` de la figura 10.10, que incluía la palabra clave `explicit` en la declaración del *constructor con un solo argumento* en la línea 14:

```
explicit Array( int = 10 ); // constructor predeterminado
```

La figura 10.13 presenta una versión ligeramente modificada del programa de la figura 10.12. Cuando se compila este programa, el compilador produce un mensaje de error indicando que el valor de entero pasado a `imprimirArray` en la línea 13 *no se puede* convertir a un `const Arreglo &`. El mensaje de error emitido por el compilador (de Visual C++) se muestra en la ventana de salida. La línea 14 demuestra cómo se puede usar el constructor explícito para crear un `Array` temporal de 3 elementos y pasarlo a la función `imprimirArray`.

Tip para prevenir errores 10.4

Use siempre la palabra clave `explicit` en los constructores con un solo argumento, a menos que deban usarse como constructores de conversión.

```
1  // Fig. 10.13: fig10_13.cpp
2  // Demostración de un constructor implícito.
3  #include <iostream>
4  #include "Array.h"
5  using namespace std;
6
7  void imprimirArray( const Array & ); // prototipo
8
9  int main()
10 {
11    Array enteros1( 7 ); // arreglo con 7 elementos
12    imprimirArray( enteros1 ); // imprime el objeto Array enteros1
13    imprimirArray( 3 ); // convierte 3 en un objeto Array e imprime su contenido
14    imprimirArray( Array( 3 ) ); // llama al constructor explícito con un solo
      argumento
15 } // fin de main
16
17 // imprime el contenido del arreglo
18 void imprimirArray( const Array &arregloAImprimir )
19 {
20    cout << "El objeto Array recibido tiene " << arregloAImprimir.obtenerTamanio()
21       << " elementos. Su contenido es:\n" << arregloAImprimir << endl;
22 } // fin de imprimirArray
```

```
c:\libros\2012\cpphtp9\ejemplos\cap10\fig10_13\fig10_13.cpp(13): error C2664:
'imprimirArray' : no se puede convertir el parámetro 1 de 'int' a 'const Array &'
        Razón: no se puede realizar la conversión de 'int' a 'const Array'
        El constructor de class 'Array' está declarado como 'explicit'
```

Fig. 10.13 | Demostración de un constructor `explicit`.

C++11: operadores de conversión **`explicit`**

A partir de C++11, de la misma forma que se declaran los constructores con un solo argumento como `explicit`, es posible declarar los operadores de conversión `explicit` para evitar que el compilador los use para realizar conversiones implícitas. Por ejemplo, el prototipo:

```
explicit MiClases::operator char *() const;
```

declara al operador de conversión `char *` de `MiClase` como `explicit`.

10.14 Sobrecarga del operador () de llamada a función

Es poderoso sobrecargar el **operador () de llamada a función**, ya que las funciones pueden recibir un número *arbitrario* de parámetros separados por comas. Por ejemplo, en una clase `String` *personalizada* podríamos cargar este operador para seleccionar una subcadena a partir de un objeto `String`; los dos parámetros enteros del operador podrían especificar la *ubicación inicial* y la *longitud de la subcadena a seleccionar*. La función `operator()` podría comprobar errores tales como una *ubicación inicial fuera de rango* o una *longitud negativa de subíndice*.

El operador sobrecargado de llamada a función debe ser una función miembro *no* `static` y podría definirse con la primera línea:

```
String String::operator()( size_t index, indice, size_t longitud ) const
```

En este caso debería ser una función miembro `const`, ya que el hecho de obtener una subcadena *no* debería modificar el objeto `String` original.

Suponga que `cadena1` es un objeto `String` que contiene la cadena `"AEIOU"`. Cuando el compilador encuentra la expresión `cadena1(2, 3)`, genera la siguiente llamada a la función miembro:

```
cadena1.operator()( 2, 3 )
```

la cual devuelve un objeto `String` que contiene `"IOU"`.

Otro posible uso del operador de llamada a función es para permitir una notación alternativa de subíndice de `Array`. En vez de usar la notación de doble corchete de C++, como en `tableroAjedrez[fila][columna]`, tal vez prefiera sobrecargar el operador de llamada a función para permitir la notación `tableroAjedrez(fila, columna)`, en donde `tableroAjedrez` es un objeto de una clase `Array` bidimensional modificada. El ejercicio 10.7 le pedirá que construya esta clase. El uso principal del operador de llamada a función es para definir objetos de función, que veremos en el capítulo 16 (en el sitio web).

10.15 Conclusión

En este capítulo, aprendió a sobrecargar operadores para trabajar con objetos de clases. Mostramos la clase `string` estándar de C++, que hace un uso extenso de los operadores sobrecargados para crear una clase robusta y reutilizable que pueda reemplazar las cadenas estilo C. Después hablamos acerca de varias restricciones que impone el estándar de C++ sobre los operadores sobrecargados. Más tarde presentamos una clase llamada `NumeroTelefonico`, en la que se sobrecargaron los operadores << y >> para imprimir y recibir números telefónicos de manera conveniente. Además vimos una clase llamada `Fecha`, la cual sobrecargaba los operadores de incremento prefijo y postfijo (++), y le mostramos una sintaxis especial que se requiere para diferenciar entre las versiones prefijo y postfijo del operador de incremento (++).

Posteriormente introdujimos el concepto de la asignación dinámica de memoria. Aprendió que puede crear y destruir objetos en forma dinámica mediante los operadores `new` y `delete`, respectivamente. Después presentamos un caso de estudio de culminación de la clase `Array`, en el que se utilizaron operadores sobrecargados y otras herramientas para resolver varios problemas con los arreglos basados en apuntador. Este caso de estudio le ayudó verdaderamente a comprender la tecnología de las clases y los objetos: creación, uso y reutilización de clases valiosas. Como parte de esta clase, vio los operadores sobrecargados de inserción de flujo, extracción de flujo, asignación, igualdad y subíndice.

Aprendió las razones de implementar operadores sobrecargados como funciones miembro o como funciones no miembro. El capítulo concluyó con discusiones sobre cómo realizar conversiones entre tipos (incluyendo tipos de clases), problemas con ciertas conversiones implícitas definidas mediante constructores con un solo argumento y cómo evitar esos problemas mediante el uso de constructores `explicit`.

En el siguiente capítulo continuaremos con nuestra discusión sobre las clases, en donde presentaremos una forma de reutilización de software conocida como herencia. Ahí veremos que cuando las clases comparten atributos y comportamientos comunes, es posible definir esos atributos y comportamientos en una clase "base" común, y "heredar" esas capacidades en nuevas definiciones de clases, lo cual nos permitirá crear las nuevas clases con una mínima cantidad de código.

Resumen

Sección 10.1 Introducción

- C++ nos permite sobrecargar la mayoría de los operadores, para que sean sensibles al contexto en el que se utilizan; el compilador genera el código apropiado con base en los tipos de los operandos.

- Un ejemplo de un operador sobrecargado integrado en C++ es <<, el cual se usa como operador de inserción de flujo y como operador de desplazamiento a la izquierda a nivel de bits. De manera similar, >> también está sobrecargado; se utiliza como operador de extracción de flujo y como operador de desplazamiento a la derecha a nivel de bits. Ambos operadores están sobrecargados en la Biblioteca estándar de C++.
- C++ sobrecarga los operadores + y – de modo que tengan un desempeño diferente, dependiendo de su contexto en la aritmética de enteros, de punto flotante y de apuntadores.
- Los trabajos realizados por los operadores sobrecargados también se pueden llevar a cabo mediante llamadas a funciones, pero la notación de operador es a menudo más natural.

Sección 10.2 Uso de los operadores sobrecargados de la clase `string` de la Biblioteca estándar

- La clase `string` estándar se define en el encabezado `<string>` y pertenece al espacio de nombres `std`.
- La clase `string` proporciona muchos operadores sobrecargados, incluyendo los operadores de igualdad, relacionales, de asignación, de asignación de suma (para la concatenación) y de subíndice.
- La clase `string` proporciona la función miembro `emtpy` (pág. 437), que devuelve `true` si el objeto `string` está vacío; en caso contrario devuelve `false`.
- La función miembro `substr` de la clase `string` estándar (pág. 437) obtiene una subcadena de una longitud especificada por el segundo argumento, empezando en la posición especificada por el primer argumento. Cuando no se especifica el segundo argumento, `substr` devuelve el resto del objeto `string` en el que se llamó.
- El operador `[]` sobrecargado de la clase `string` no realiza comprobación de límites. Por lo tanto, el programador se debe asegurar que las operaciones que utilizan el operador `[]` sobrecargado de la clase `string` estándar no manipule accidentalmente los elementos fuera de los límites del objeto `string`.
- La clase `string` estándar proporciona la comprobación de límites en su función miembro `at` (pág. 438), la cual "lanza una excepción" si su argumento es un subíndice inválido. De manera predeterminada, esto hace que el programa termine. Si el subíndice es válido, la función `at` devuelve una referencia o una referencia `const` al carácter en la ubicación especificada, dependiendo del contexto.

Sección 10.3 Fundamentos de la sobrecarga de operadores

- Para sobrecargar un operador, se escribe la definición de una función miembro no `static` o la definición de una función no miembro en donde el nombre de la función es la palabra clave `operator`, seguida del símbolo del operador que se va a sobrecargar.
- Cuando los operadores se sobrecargan como funciones miembro, deben ser no `static` debido a que se deben llamar en un objeto de la clase y deben operar en ese objeto.
- Para usar un operador en objetos de clases, es necesario definir una función de operador sobrecargado, con tres excepciones: el operador de asignación (`=`), el operador de dirección (`&`) y el operador de coma (`,`).
- No es posible cambiar la precedencia y asociatividad de un operador mediante la sobrecarga.
- No es posible cambiar la "aridad" de un operador (el número de operandos que recibe).
- No es posible crear nuevos operadores; sólo se pueden sobrecargar los que ya existen.
- No podemos cambiar el significado de la forma en que un objeto trabaja sobre objetos de tipos fundamentales.
- Sobrecargar un operador de asignación y un operador de suma para una clase no implica que el operador `+=` también esté sobrecargado. Debemos sobrecargar el operador `+=` de manera explícita para esa clase.
- Los operadores sobrecargados `()`, `[]`, `->` y de asignación deben declararse como miembros de la clase. Para el resto de los operadores, las funciones de sobrecarga de operadores pueden ser miembros de clases o funciones no miembro.

Sección 10.4 Sobrecarga de operadores binarios

- Un operador binario se puede sobrecargar como una función miembro no `static` con un argumento, o como una función no miembro con dos argumentos (uno de esos argumentos debe ser el objeto de una clase o una referencia al objeto de una clase).

Sección 10.5 Sobrecarga de los operadores binarios de inserción de flujo y extracción de flujo

- El operador de inserción de flujo sobrecargado (<<) se utiliza en una expresión en la que el operando izquierdo tiene el tipo `ostream &`. Por esta razón, se debe sobrecargar como una función no miembro. De manera similar, el operador de extracción de flujo sobrecargado (>>) debe ser una función no miembro.
- Otra razón por la que se debe elegir una función global para sobrecargar un operador es para permitir que éste sea conmutativo.
- Cuando se utiliza con `cin`, `setw` restringe el número de caracteres leídos al número de caracteres especificados por su argumento.
- La función miembro `ignore` de `istream` descarta el número especificado de caracteres en el flujo de entrada (un carácter de manera predeterminada).
- Los operadores de entrada y salida sobrecargados se declaran como funciones `friend` si necesitan acceder a los miembros no `public` de la clase directamente, por cuestiones de rendimiento.

Sección 10.6 Sobrecarga de operadores unarios

- Un operador unario para una clase se puede sobrecargar como función miembro no `static` sin argumentos, o como función no miembro con un argumento; ese argumento debe ser un objeto de la clase, o una referencia a un objeto de la misma.
- Las funciones miembro que implementan operadores sobrecargados deben ser no `static`, de manera que puedan acceder a los datos no `static` en cada objeto de la clase.

Sección 10.7 Sobrecarga de los operadores unarios prefijo y postfijo ++ y --

- Todos los operadores de incremento y decremento prefijo y postfijo se pueden sobrecargar.
- Para sobrecargar los operadores de preincremento y postincremento, cada función de operador sobrecargado debe tener una firma distinta. Las versiones prefijo se sobrecargan de la misma forma que cualquier otro operador unario. La firma única del operador de incremento postfijo se obtiene al proveer un segundo argumento, que debe ser de tipo `int`. Este argumento no se proporciona en el código cliente. El compilador lo utiliza de manera implícita para diferenciar entre las versiones prefijo y postfijo del operador de incremento. Se utiliza la misma sintaxis para diferenciar entre las funciones de operador de decremento prefijo y postfijo.

Sección 10.9 Administración dinámica de memoria

- La administración dinámica de memoria (pág. 451) nos permite controlar la asignación y desasignación de memoria en un programa, para cualquier tipo integrado o definido por el usuario.
- El almacenamiento libre (conocido también como heap o montón; pág. 451) es una región de memoria asignada a cada programa para almacenar objetos asignados en forma dinámica, en tiempo de ejecución.
- El operador `new` (pág. 451) asigna almacenamiento del tamaño apropiado para un objeto, ejecuta su constructor y devuelve un apuntador del tipo correcto. Si `new` no puede encontrar espacio en memoria para el objeto, indica que ocurrió un error "lanzando" una "excepción". Esto por lo general hace que el programa termine de inmediato, a menos que se maneje la excepción.
- Para destruir un objeto asignado en forma dinámica y liberar su espacio, use el operador `delete` (pág. 451).
- Es posible asignar un arreglo integrado de objetos en forma dinámica mediante `new`, como en:

```
int *ptr = new int[ 100 ]();
```

que asigna un arreglo integrado de 100 enteros, inicializa cada uno con `0` mediante la inicialización por valor y asigna la ubicación inicial del arreglo integrado a `ptr`. El arreglo integrado anterior se elimina (pág. 453) con la siguiente instrucción:

```
delete [] ptr;
```

Sección 10.10 Caso de estudio: la clase `Array`

- Para inicializar un nuevo objeto de una clase, un constructor de copia realiza una copia de los miembros de un objeto existente de esa clase. Por lo general, las clases que contienen memoria asignada en forma dinámica proporcionan un constructor de copia, un destructor y un operador de asignación sobrecargado.
- La implementación de la función miembro `operator=` debe evaluar la auto-asignación (pág. 463), en la cual un objeto se asigna a sí mismo.
- El compilador llama a la versión `const` de `operator[]` cuando el operador de subíndice se utiliza en un objeto `const`, y llama a la versión no `const` del operador cuando se utiliza en un objeto no `const`.
- El operador de subíndice (`[]`) puede usarse para seleccionar elementos de otros tipos de contenedores. Además, con la sobrecarga los valores de los subíndices no necesitan ser enteros.

Sección 10.11 Comparación entre los operadores como funciones miembro y no miembro

- Las funciones de operadores pueden ser funciones miembro o funciones no miembro; por lo general, las funciones no miembro se hacen `friend` por cuestiones de rendimiento. Las funciones miembro utilizan el apuntador `this` de manera implícita para obtener uno de los argumentos de los objetos de su clase (el operando izquierdo para operadores binarios). Los argumentos para ambos operandos de un operador binario deben listarse de manera explícita en la llamada a una función no miembro.
- Cuando una función de operador se implementa como función miembro, el operando de más a la izquierda (o el único operando) debe ser un objeto (o una referencia a un objeto) de la clase del operador.
- Si el operando izquierdo debe ser un objeto de una clase distinta o de un tipo fundamental, esta función de operador se debe implementar como una función no miembro.
- Una función de operador no miembro puede convertirse en `friend` de una clase, si esa función debe acceder directamente a los miembros `private` o `protected` de esa clase.

Sección 10.12 Conversión entre tipos

- El compilador no puede saber de antemano cómo convertir entre tipos definidos por el usuario, y entre tipos definidos por el usuario y tipos fundamentales, por lo que debemos especificar cómo hacer esto. Dichas conversiones se pueden llevar a cabo mediante constructores de conversión (pág. 467): constructores con un solo argumento que convierten objetos de otros tipos (incluyendo los tipos fundamentales) en objetos de una clase específica.
- Un constructor que puede llamarse con un solo argumento puede usarse como constructor de conversión.
- Un operador de conversión (pág. 467) debe ser una función miembro no `static`. Las funciones de operador de conversión sobrecargado (pág. 467) pueden definirse para convertir objetos de tipos definidos por el usuario en tipos fundamentales o en objetos de otros tipos definidos por el usuario.
- Una función de operador de conversión sobrecargado no especifica un tipo de valor de retorno; éste viene siendo el tipo al que se va a convertir el objeto.
- Si es necesario, el compilador puede llamar a los operadores de conversión de tipos y a los constructores de conversión de manera implícita.

Sección 10.13 Constructores `explicit` *y operadores de conversión*

- Un constructor que se declara como `explicit` (pág. 469) no se puede utilizar en una conversión implícita.

Sección 10.14 Sobrecarga del operador `()` *de llamada a función*

- Es poderoso sobrecargar el operador `()` (pág. 470) de llamada a función, ya que las funciones pueden tener un número arbitrario de parámetros.

Ejercicios de autoevaluación

10.1 Complete los siguientes enunciados:

a) Suponga que `a` y `b` son variables enteras y que formamos la suma `a + b`. Ahora suponga que `c` y `d` son variables de punto flotante y que formamos la suma `c + d`. Los dos operadores `+` de aquí se están utilizando claramente para distintos fines. Éste es un ejemplo de _____.

b) La palabra clave _____ introduce la definición de una función de operador sobrecargado.
c) Para usar operadores en objetos de clases, éstos deben sobrecargarse con la excepción de los operadores _____, _____ y _____.
d) La _____, _____ y _____ de un operador no se puede modificar al sobrecargar este operador.
e) Los operadores que no se pueden sobrecargar son _____, _____, _____ y _____.
f) El operador _____ reclama la memoria asignada previamente por `new`.
g) El operador _____ asigna memoria en forma dinámica para un objeto de un tipo especificado y devuelve un(a) _____ a ese tipo.

10.2 Explique los múltiples significados de los operadores << y >>.

10.3 ¿En qué contexto se podría utilizar el nombre `operator/`?

10.4 (Verdadero/falso) Sólo los operadores existentes se pueden sobrecargar.

10.5 ¿Cómo se compara la precedencia de un operador sobrecargado con la precedencia del operador original?

Respuestas a los ejercicios de autoevaluación

10.1 a) sobrecarga de operadores. b) `operator`. c) asignación (=), dirección (&), coma (,). d) precedencia, asociatividad, "aridad". e) `.`, `?:`, `.*` y `::`. f) `delete`. g) `new`, apuntador.

10.2 El operador >> es tanto el operador de desplazamiento a la derecha como el operador de extracción de flujo, dependiendo de su contexto. El operador << es tanto el operador de desplazamiento a la izquierda como el operador de inserción de flujo, dependiendo de su contexto.

10.3 Para la sobrecarga de operadores: sería el nombre de una función que proporcionara una versión sobrecargada del operador / para una clase específica.

10.4 Verdadero.

10.5 La precedencia es idéntica.

Ejercicios

10.6 ***(Operadores de asignación y desasignación de memoria)*** Compare y contraste los operadores de asignación y desasignación dinámica de memoria `new`, `new []`, `delete` y `delete []`.

10.7 ***(Sobrecarga del operador de paréntesis)*** Un buen ejemplo de cómo sobrecargar el operador `()` de llamada a función es permitir otra forma de subíndices dobles de arreglos comunes en ciertos lenguajes de programación. En vez de decir

```
tableroAjedrez[ fila ][ columna ]
```

para un arreglo de objetos, sobrecargue el operador de llamadas a funciones para permitir la siguiente forma alterna:

```
tableroAjedrez( fila, columna )
```

Cree una clase llamada `ArregloSubindiceDoble` que tenga características similares a la clase `Array` de las figuras 10.10 y 10.11. Al momento de la construcción, la clase debe poder crear un arreglo `ArregloSubindiceDoble` de cualquier número de filas y columnas. La clase debe proporcionar el `operator()` para realizar operaciones con doble subíndice. Por ejemplo, en un objeto `ArregloSubindiceDoble` de 3 por 5 llamado `tableroAjedrez`, el usuario podría escribir `tableroAjedrez( 1, 3 )` para acceder al elemento en la fila `1` y la columna `3`. Recuerde que `operator()` puede recibir *cualquier* número de argumentos. La representación subyacente del `ArregloSubindiceDoble` podría ser un arreglo de enteros con un solo subíndice, con un número de elementos equivalente a *filas * columnas*. La función `operator()` debe realizar la aritmética de apuntadores apropiada para acceder a cada elemento del arreglo subyacente. Debe haber *dos* versiones de `operator()`: una que devuelva `int &` (de manera que un elemento de un objeto `ArregloSubindiceDoble` pueda usarse como *lvalue*) y otra que devuelva `int`. La clase debe también

proporcionar los siguientes operadores: ==, !=, =, << (para imprimir el ArregloSubindiceDoble en formato de fila y columna) y >> (para recibir todo el contenido completo del ArregloSubindiceDoble).

10.8 ***(Clase Complejo)*** Considere la clase Complejo que se muestra en las figuras 10.14 a 10.16. Esta clase permite operaciones con *números complejos*. Estos son números de la forma parteReal + parteImaginaria * *i*, en donde *i* tiene el valor

$$\sqrt{-1}$$

a) Modifique la clase para permitir la entrada y salida de números complejos a través de los operadores sobrecargados >> y <<, respectivamente (debe eliminar la función imprimir de la clase).
b) Sobrecargue el operador de multiplicación para permitir la multiplicación de dos números complejos, como en álgebra.
c) Sobrecargue los operadores == y != para permitir comparaciones de números complejos.

Después de realizar este ejercicio, tal vez quiera leer sobre la clase complex de la Biblioteca estándar (del encabezado <complex>).

```
1  // Fig. 10.14: Complejo.h
2  // Definición de la clase Complejo.
3  #ifndef COMPLEJO_H
4  #define COMPLEJO_H
5
6  class Complejo
7  {
8  public:
9     explicit Complejo( double = 0.0, double = 0.0 ); // constructor
10    Complejo operator+( const Complejo & ) const; // suma
11    Complejo operator-( const Complejo & ) const; // resta
12    void imprimir() const; // salida
13 private:
14    double real; // parte real
15    double imaginaria; // parte imaginaria
16 }; // fin de la clase Complejo
17
18 #endif
```

Fig. 10.14 | Definición de la clase Complejo.

```
1  // Fig. 10.15: Complejo.cpp
2  // Definiciones de las funciones miembro de la clase Complejo.
3  #include <iostream>
4  #include "Complejo.h" // definición de la clase Complejo
5  using namespace std;
6
7  // Constructor
8  Complejo::Complejo( double parteReal, double parteImaginaria )
9     : real( parteReal ),
10     imaginaria( parteImaginaria )
11 {
12    // cuerpo vacío
13 } // fin del constructor de Complejo
14
```

Fig. 10.15 | Definiciones de las funciones miembro de la clase Complejo (parte 1 de 2).

```
15  // operador de suma
16  Complejo Complejo::operator+( const Complejo &operando2 ) const
17  {
18     return Complejo( real + operando2.real,
19        imaginaria + operando2.imaginaria );
20  } // fin de la función operator+
21
22  // operador de resta
23  Complejo Complejo::operator-( const Complejo &operando2 ) const
24  {
25     return Complejo( real - operando2.real,
26        imaginaria - operando2.imaginaria );
27  } // fin de la función operator-
28
29  // muestra un objeto Complejo en la forma: (a, b)
30  void Complejo::imprimir() const
31  {
32     cout << '(' << real << ", " << imaginaria << ')';
33  } // fin de la función imprimir
```

Fig. 10.15 | Definiciones de las funciones miembro de la clase `Complejo` (parte 2 de 2).

```
34  // Fig. 10.16: fig10_16.cpp
35  // Programa de prueba de la clase Complejo.
36  #include <iostream>
37  #include "Complejo.h"
38  using namespace std;
39
40  int main()
41  {
42     Complejo x;
43     Complejo y( 4.3, 8.2 );
44     Complejo z( 3.3, 1.1 );
45
46     cout << "x: ";
47     x.imprimir();
48     cout << "\ny: ";
49     y.imprimir();
50     cout << "\nz: ";
51     z.imprimir();
52
53     x = y + z;
54     cout << "\n\nx = y + z:" << endl;
55     x.imprimir();
56     cout << " = ";
57     y.imprimir();
58     cout << " + ";
59     z.imprimir();
60
61     x = y - z;
62     cout << "\n\nx = y - z:" << endl;
63     x.imprimir();
64     cout << " = ";
65     y.imprimir();
66     cout << " - ";
67     z.imprimir();
68     cout << endl;
69  } // fin de main
```

Fig. 10.16 | Programa de prueba de la clase `Complejo` (parte 1 de 2).

```
x: (0, 0)
y: (4.3, 8.2)
z: (3.3, 1.1)

x = y + z:
(7.6, 9.3) = (4.3, 8.2) + (3.3, 1.1)

x = y - z:
(1, 7.1) = (4.3, 8.2) - (3.3, 1.1)
```

Fig. 10.16 | Programa de prueba de la clase `Complejo` (parte 2 de 2).

10.9 ***(Clase EnteroEnorme)*** Una máquina con enteros de 32 bits puede representar enteros en el rango aproximado de –2 mil millones a +2 mil millones. Esta restricción de tamaño fijo raras veces presenta problemas, pero hay aplicaciones en las que sería conveniente poder usar un rango de enteros mucho más amplio. Esto es para lo que se creó C++, a saber, para crear nuevos y poderosos tipos de datos. Considere la clase `EnteroEnorme` de las figuras 10.17 a 10.19. Estudie la clase con cuidado y después responda a lo siguiente:

a) Describa la forma en que opera con precisión.
b) ¿Qué restricciones tiene la clase?
c) Sobrecargue el operador de multiplicación `*`.
d) Sobrecargue el operador de división `/`.
e) Sobrecargue todos los operadores relacionales y de igualdad.

[*Nota:* no mostramos un operador de asignación o constructor de copia para la clase `EnteroEnorme`, debido a que el operador de asignación y el constructor de copia que proporciona el compilador son capaces de copiar el miembro de datos `array` completo de manera apropiada].

```
1  // Fig. 10.17: EnteroEnorme.h
2  // Definición de la clase EnteroEnorme.
3  #ifndef ENTEROENORME_H
4  #define ENTEROENORME_H
5
6  #include <array>
7  #include <iostream>
8  #include <string>
9
10 class EnteroEnorme
11 {
12    friend std::ostream &operator<<( std::ostream &, const EnteroEnorme & );
13 public:
14    static const int digitos = 30; // digitos máximos en un EnteroEnorme
15
16    EnteroEnorme( long = 0 ); // constructor de conversión/predeterminado
17    EnteroEnorme( const std::string & ); // constructor de conversión
18
19    // operador de suma; EnteroEnorme + EnteroEnorme
20    EnteroEnorme operator+( const EnteroEnorme & ) const;
21
22    // operador de suma; EnteroEnorme + int
23    EnteroEnorme operator+( int ) const;
24
25    // operador de suma;
26    // EnteroEnorme + string que representa un valor entero mayor
27    EnteroEnorme operator+( const std::string & ) const;
```

Fig. 10.17 | Definición de la clase `EnteroEnorme` (parte 1 de 2).

```
28   private:
29      std::array< short, digitos > entero;
30   }; // fin de la clase EnteroEnorme
31
32   #endif
```

Fig. 10.17 | Definición de la clase `EnteroEnorme` (parte 2 de 2).

```
1    // Fig. 10.18: EnteroEnorme.cpp
2    // Definiciones de las funciones miembro y funciones friend de EnteroEnorme.
3    #include <cctype> // prototipo de la función isdigit
4    #include "EnteroEnorme.h" // definición de la clase EnteroEnorme
5    using namespace std;
6
7    // constructor predeterminado; constructor de conversión que convierte
8    // un entero long en un objeto EnteroEnorme
9    EnteroEnorme::EnteroEnorme( long valor )
10   {
11      // inicializa el arreglo con cero
12      for ( short &elemento : entero )
13         elemento = 0;
14
15      // coloca los dígitos del argumento en el arreglo
16      for ( size_t j = digitos - 1; valor != 0 && j >= 0; j-- )
17      {
18         entero[ j ] = valor % 10;
19         valor /= 10;
20      } // fin de for
21   } // fin del constructor predeterminado/de conversión de EnteroEnorme
22
23   // constructor de conversión que convierte una cadena de caracteres
24   // que representa a un entero grande en un objeto EnteroEnorme
25   EnteroEnorme::EnteroEnorme( const string &numero )
26   {
27      // inicializa el arreglo con cero
28      for ( short &elemento : entero )
29         elemento = 0;
30
31      // coloca los dígitos del argumento en el arreglo
32      size_t longitud = numero.size();
33
34      for ( size_t j = digitos - longitud, k = 0; j < digitos; ++j, ++k )
35         if ( isdigit( numero[ k ] ) ) // asegura que carácter sea dígito
36            entero[ j ] = numero[ k ] - '0';
37   } // fin del constructor de conversión de EnteroEnorme
38
39   // operador de suma; EnteroEnorme + EnteroEnorme
40   EnteroEnorme EnteroEnorme::operator+( const EnteroEnorme &op2 ) const
41   {
42      EnteroEnorme temp; // resultado temporal
43      int acarreo = 0;
44
45      for ( int i = digitos - 1; i >= 0; i-- )
46      {
47         temp.entero[ i ] = entero[ i ] + op2.entero[ i ] + acarreo;
48
```

Fig. 10.18 | Definiciones de las funciones miembro y funciones `friend` de la clase `EnteroEnorme` (parte 1 de 2).

```
49        // determina si se acarrea un 1
50        if ( temp.entero[ i ] > 9 )
51        {
52           temp.entero[ i ] %= 10;  // lo reduce 0-9
53           acarreo = 1;
54        } // fin de if
55        else // no hay acarreo
56           acarreo = 0;
57     } // fin de for
58
59     return temp; // devuelve una copia del objeto temporal
60  } // fin de la función operator+
61
62  // operador de suma; EnteroEnorme + int
63  EnteroEnorme EnteroEnorme::operator+( int op2 ) const
64  {
65     // convierte op2 en un objeto EnteroEnorme, después invoca
66     // a operator+ para dos objetos EnteroEnorme
67     return *this + EnteroEnormet( op2 );
68  } // fin de la función operator+
69
70  // operador de suma;
71  // EnteroEnorme + cadena que representa un valor entero grande
72  EnteroEnorme EnteroEnorme::operator+( const string &op2 ) const
73  {
74     // convierte op2 en un objeto EnteroEnorme, después invoca
75     // a operator+ para dos objetos EnteroEnorme
76     return *this + EnteroEnorme( op2 );
77  } // fin de operator+
78
79  // operador de salida sobrecargado
80  ostream& operator<<( ostream &salida, const EnteroEnorme &num )
81  {
82     int i;
83
84     for ( i = 0; ( i < EnteroEnorme::digitos ) && ( 0 == num.entero[ i ] ); ++i )
85        ; // omite ceros a la izquierda
86
87     if ( i == EnormeEntero::digitos )
88        salida << 0;
89     else
90        for ( ; i < EnteroEnorme::digitos; ++i )
91           salida << num.entero[ i ];
92
93     return salida;
94  } // fin de la función operator<<
```

Fig. 10.18 | Definiciones de las funciones miembro y funciones `friend` de la clase `EnteroEnorme` (parte 2 de 2).

```
1  // Fig. 10.19: fig10_19.cpp
2  // Programa de prueba de EnteroEnorme.
3  #include <iostream>
4  #include "EnteroEnorme.h"
5  using namespace std;
6
7  int main()
8  {
```

Fig. 10.19 | Programa de prueba de `EnteroEnorme` (parte 1 de 2).

```
9      EnteroEnorme n1( 7654321 );
10     EnteroEnorme n2( 7891234 );
11     EnteroEnorme n3( "99999999999999999999999999999" );
12     EnteroEnorme n4( "1" );
13     EnteroEnorme n5;
14
15     cout << "n1 es " << n1 << "\nn2 es " << n2
16        << "\nn3 es " << n3 << "\nn4 es " << n4
17        << "\nn5 es " << n5 << "\n\n";
18
19     n5 = n1 + n2;
20     cout << n1 << " + " << n2 << " = " << n5 << "\n\n";
21
22     cout << n3 << " + " << n4 << "\n= " << ( n3 + n4 ) << "\n\n";
23
24     n5 = n1 + 9;
25     cout << n1 << " + " << 9 << " = " << n5 << "\n\n";
26
27     n5 = n2 + "10000";
28     cout << n2 << " + " << "10000" << " = " << n5 << endl;
29  } // fin de main
```

```
n1 is 7654321
n2 is 7891234
n3 is 99999999999999999999999999999
n4 is 1
n5 is 0

7654321 + 7891234 = 15545555

99999999999999999999999999999 + 1
= 100000000000000000000000000000

7654321 + 9 = 7654330

7891234 + 10000 = 7901234
```

Fig. 10.19 | Programa de prueba de `EnteroEnorme` (parte 2 de 2).

10.10 *(Clase `NumeroRacional`)* Cree una clase llamada `NumeroRacional` (fracciones) con las siguientes capacidades:

a) Cree un constructor que evite un denominador 0 en una fracción, que reduzca o simplifique fracciones que no estén en forma reducida y que evite los denominadores negativos.
b) Sobrecargue los operadores de suma, resta, multiplicación y división para esta clase.
c) Sobrecargue los operadores relacionales y de igualdad para esta clase.

10.11 *(Clase `Polinomio`)* Desarrolle la clase `Polinomio`. La representación interna de un `Polinomio` es un arreglo de términos. Cada término contiene un coeficiente y un exponente; por ejemplo el termino

$$2x^4$$

tiene el coeficiente 2 y el exponente 4. Desarrolle una clase completa que contenga las funciones apropiadas del constructor y destructor, así como funciones *establecer* y *obtener*. La clase también debe proporcionar las siguientes herramientas de operadores sobrecargados:

a) Sobrecargue el operador de suma (+) para sumar dos objetos `Polinomio`.
b) Sobrecargue el operador de resta (-) para restar dos objetos `Polinomio`.
c) Sobrecargue el operador de asignación para asignar un `Polinomio` a otro.
d) Sobrecargue el operador de multiplicación (*) para multiplicar dos objetos `Polinomio`.
e) Sobrecargue el operador de asignación de suma (+=), el operador de asignación de resta (-=) y el operador de asignación de multiplicación (*=).

11

Programación orientada a objetos: herencia

No digas que conoces
a otro por completo,
sino hasta que hayas dividido
una herencia con él.

—Johann Kasper Lavater

Este método es definir como
el número de una clase,
la clase de todas las clases
similares a la clase dada.

—Bertrand Russell

Preserva la autoridad base
de los libros de otros.

—William Shakespeare

Objetivos

En este capítulo aprenderá a:

- Definir qué es la herencia y cómo promueve la reutilización de código.
- Familiarizarse con los conceptos de clases bases y clases derivadas, y las relaciones entre éstas.
- Familiarizarse con el especificador de acceso a miembros `protected`.
- Utilizar constructores y destructores en las jerarquías de herencias.
- Reconocer el orden en el que se llama a los constructores y destructores en las jerarquías de herencia de clases.
- Diferenciar entre herencia `public`, `protected` y `private`.
- Usar la herencia para personalizar el software existente.

Plan general

11.1 Introducción

En este capítulo continuaremos nuestra discusión acerca de la programación orientada a objetos (POO), mediante la introducción de la **herencia**: una forma de reutilización de software, en la cual para crear una nueva clase se absorben las herramientas de una clase existente y luego se *personalizan* o mejoran. La reutilización de software ahorra tiempo durante el desarrollo de programas al sacar provecho del software comprobado de alta calidad.

Al crear una clase, en lugar de escribir miembros de datos y funciones miembro completamente nuevos, especificamos que la nueva clase **herede** los miembros de una ya existente. La clase existente se llama **clase base**, y la clase nueva es la **clase derivada**. Otros lenguajes de programación, como Java y C#, se refieren a la clase base como la **superclase** y a la clase derivada como la **subclase**. Una clase derivada representa a un grupo *más especializado* de objetos.

C++ ofrece herencia `public`, `protected` y `private`. En este capítulo nos concentraremos en la herencia `public` y explicaremos brevemente los otros dos tipos. *Con la herencia `public`, todo objeto de una clase derivada es también un objeto de la clase base de esa clase derivada*. Sin embargo, los objetos de la clase base *no* son objetos de sus clases derivadas. Por ejemplo, si tenemos `Vehiculo` como clase base y `Auto` como clase derivada, entonces todos los `Autos` son `Vehiculos`, pero no todos los `Vehiculos` son `Autos`; por ejemplo, un `Vehiculo` podría ser también un `Camion` o un `Bote`.

Hay una diferencia entre la **relación *es-un*** y la relación *tiene-un*. La relación *es-un* representa la herencia. En una relación del tipo *es un*, un objeto de una clase derivada también puede tratarse como un objeto de su clase base; por ejemplo, un `Auto` *es-un* `Vehiculo`, por lo que cualquier atributo y comportamiento de un `Vehiculo` es también atributo y comportamiento de un `Auto`. En contraste, la relación *tiene-un* representa la *composición*, que vimos en el capítulo 9. En una relación del tipo *tiene-un*, un objeto *contiene* uno o más objetos de otras clases como miembros. Por ejemplo, un `Auto` incluye muchos componentes: *tiene-un* volante, *tiene-un* pedal de freno, *tiene-una* transmisión, etcétera.

11.2 Clases base y clases derivadas

En la figura 11.1 se listan varios ejemplos simples de las clases base y las clases derivadas. Las clases base tienden a ser *más generales* y las clases derivadas a ser *más específicas.*

Clase base	Clases derivadas
`Estudiante`	`EstudianteGraduado, EstudianteNoGraduado`
`Figura`	`Circulo, Triangulo, Rectangulo, Esfera, Cubo`
`Prestamo`	`PrestamoAuto, PrestamoMejoraHogar, PrestamoHipotecario`
`Empleado`	`Docente, Administrativo`
`CuentaBanco`	`CuentaCheques, CuentaAhorros`

Fig. 11.1 | Ejemplos de herencia.

Como todo objeto de una clase derivada *es-un* objeto de su clase base, y una clase base puede tener *muchas* clases derivadas, el conjunto de objetos representados por una clase base es por lo general *mayor* que el conjunto de objetos representado por cualquiera de sus clases derivadas. Por ejemplo, la clase base `Vehiculo` representa a todos los vehículos, incluyendo autos, camiones, barcos, aviones, bicicletas, etcétera. En contraste, la clase derivada `Auto` representa un subconjunto *más pequeño y específico* de todos los vehículos.

Las relaciones de herencia forman **jerarquías de clases**. Una clase base existe en una relación jerárquica con sus clases derivadas. Aunque las clases pueden existir de manera independiente, una vez que se emplean en relaciones de herencia, se afilian con otras clases. Una clase se convierte ya sea en una clase base (suministrando miembros a las otras clases), en una clase derivada (heredando sus miembros de otras clases), o en *ambas*.

La jerarquía de clases `MiembroDeLaComunidad`

Vamos a desarrollar una jerarquía de herencia simple con cinco niveles (representada por el diagrama de clases de UML de la figura 11.2). Una comunidad universitaria tiene miles de miembros de la comunidad (objetos `MiembroDeLaComunidad`).

Estos `MiembroDeLaComunidad` consisten en `Empleados`, `Estudiantes` y `Exalumnos` (cada uno de la clase `alumno`). Los `Empleados` pueden ser `Docentes` o `Administrativos`. Los miembros `Docentes` pueden

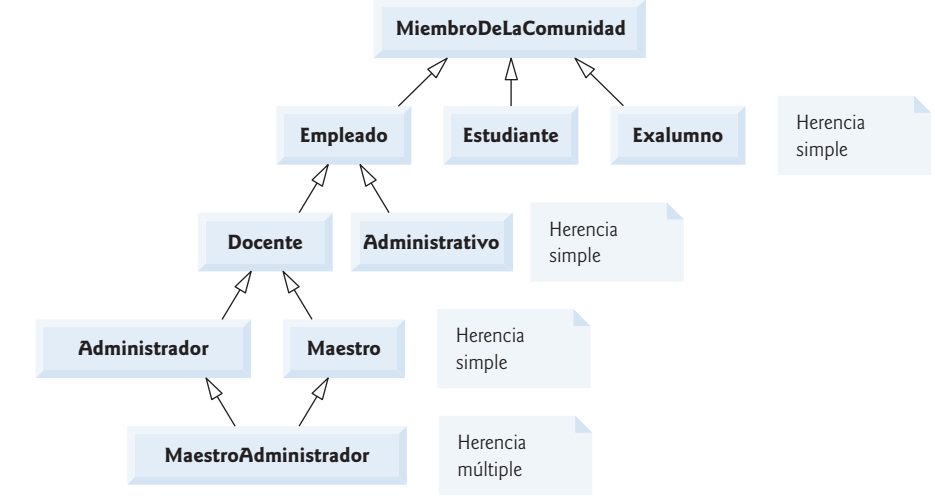


Fig. 11.2 | Jerarquía de herencia para cada `MiembroDeLaComunidad` de una universidad.

ser `Administradores` o `Maestros`. Sin embargo, algunos `Administradores` también son `Maestros`. Utilizamos la *herencia múltiple* para formar la clase `MaestroAdministrador`. Con la **herencia `simple`**, una clase se deriva de *una* clase base. Con la **herencia múltiple**, una clase derivada hereda al mismo tiempo de *dos o más* clases base (que tal vez no estén relacionadas). (Consulte el capítulo 23, Other Topics, en el sitio web).

Cada flecha en la jerarquía (figura 11.2) representa una relación *es-un*. Por ejemplo, al seguir las flechas en esta jerarquía de clases podemos decir que "un `Empleado` *es-un* `MiembroDeLaComunidad`" y que "un `Maestro` *es-un* miembro `Docente`". `MiembroDeLaComunidad` es la **clase base directa** de `Empleado`, `Estudiante` y `ExAlumno`. Además, `MiembroDeLaComunidad` es una **clase base indirecta** de todas las demás clases en el diagrama. Una clase base indirecta se hereda de dos o más niveles de la parte superior de la jerarquía de clases.

Empezando desde la parte inferior del diagrama, usted puede seguir las flechas y aplicar la relación *es-un* hasta la clase base superior. Por ejemplo, un `MaestroAdministrador` *es-un* `Administrador`, *es-un* miembro `Docente`, *es-un* `Empleado` y *es-un* `MiembroDeLaComunidad`.

Jerarquía de la clase `Figura`

Ahora considere la jerarquía de herencia para `Figura` en la figura 11.3. Esta jerarquía comienza con la clase base `Figura`. Las clases `FiguraBidimensional` y `FiguraTridimensional` se derivan de la clase base `Figura`: un objeto `Figura` *es-una* `FiguraBidimensional` o *es-una* `FiguraTridimensional`. El tercer nivel de esta jerarquía contiene algunos tipos *más específicos* de objetos `FiguraBidimensional` y `FiguraTridimensional`. Al igual que en la figura 11.2, podemos seguir las flechas desde la parte inferior del diagrama de clases hasta la clase base superior en esta jerarquía de clases, para identificar varias relaciones del tipo *es-un*. Por ejemplo, un `Triangulo` *es-una* `FiguraBidimensional` y *es-una* `Figura`, mientras que una `Esfera` *es-una* `FiguraTridimensional` y *es-una* `Figura`.

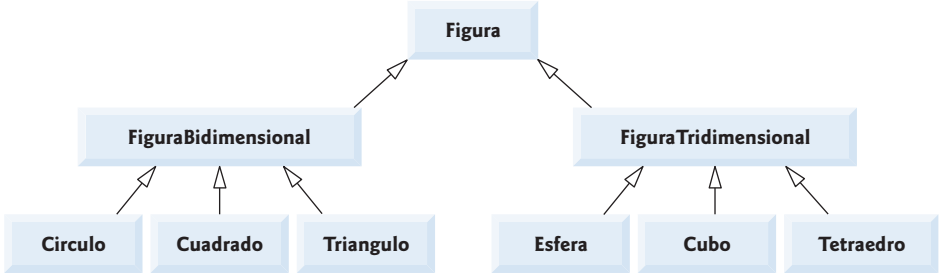


Fig. 11.3 | Jerarquía de herencia para objetos `Figura`.

Para especificar que la clase `FiguraBidimensional` (figura 11.3) se deriva de (o hereda de) la clase `Figura`, la definición de la clase `FiguraBidimensional` podría empezar de la siguiente manera:

```
class FiguraBidimensional : public Figura
```

Éste es un ejemplo de **herencia `public`**, la forma de uso más común. También hablaremos sobre la **herencia `private`** y la **herencia `protected`** (sección 11.5). Con todas las formas de herencia, los miembros `private` de una clase base *no* pueden utilizarse directamente desde las clases derivadas de esa clase, pero estos miembros `private` de la clase base de todas formas se heredan (es decir, se consideran parte de las clases derivadas). Con la herencia `public`, todos los demás miembros de la clase base retienen su acceso original a los miembros cuando se convierten en miembros de la clase derivada (por ejemplo, los miembros `public` de la clase base se convierten en miembros `public` de la clase derivada y, como veremos

pronto, los miembros `protected` de la clase base se convierten en miembros `protected` de la clase derivada). A través de estos miembros heredados de la clase base, la clase derivada puede manipular los miembros `private` de la clase base (si estas funciones miembro heredadas proporcionan dicha funcionalidad en la clase base). Observe que las funciones `friend` no se heredan.

La herencia *no* es apropiada para todas las relaciones de las clases. En el capítulo 9 hablamos sobre la relación *tiene-un*, en la cual las clases tienen miembros que son objetos de otras clases. Dichas relaciones crean clases mediante la *composición* de clases existentes. Por ejemplo, dadas las clases `Empleado`, `FechaNacimiento` y `NumeroTelefonico`, es impropio decir que un `Empleado` *es-una* `FechaNacimiento` o que un `Empleado` *es-un* `NumeroTelefonico`. Sin embargo, es apropiado decir que un `Empleado` *tiene-una* `FechaNacimiento` y que un `Empleado` *tiene-un* `NumeroTelefonico`.

Es posible tratar a los objetos de la clase base y a los objetos de las clases derivadas de manera similar; sus características comunes se expresan en los miembros de la clase base. Los objetos de todas las clases derivadas a partir de una clase base común se pueden tratar como objetos de esa clase base (es decir, dichos objetos tienen una relación *es-un* con la clase base). En el capítulo 12, consideraremos muchos ejemplos que aprovechan esta relación.

11.3 Relación entre las clases base y las clases derivadas

En esta sección usaremos una jerarquía de herencia que contiene tipos de empleados en la aplicación de nómina de una compañía, para hablar sobre la relación entre una clase base y una clase derivada. A los empleados por comisión (que se representan como objetos de una clase base) se les paga un porcentaje de sus ventas, mientras que los empleados por comisión con salario base (que se representan como objetos de una clase derivada) reciben un salario base, más un porcentaje de sus ventas. Dividiremos nuestra discusión sobre la relación entre los empleados por comisión y los empleados por comisión con salario base en una serie cuidadosamente pautada de cinco ejemplos.

10.3.1 Creación y uso de una clase `EmpleadoPorComision`

Analizaremos la definición de la clase `EmpleadoPorComision` (figuras 11.4 a 11.5). El encabezado de `EmpleadoPorComision` (figura 11.4) especifica los servicios `public` de la clase `EmpleadoPorComision`, que incluyen un constructor (líneas 11 y 12) y las funciones miembro `ingresos` (línea 29) e `imprimir` (línea 30). En las líneas 14 a 27 se declaran funciones `public` *establecer* y *obtener* que manipulan los miembros de datos de la clase (declarados en las líneas 32 a 36) `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision`. Por ejemplo, las funciones miembro `establecerVentasBrutas` (definida en las líneas 57 a 63 de la figura 11.5) y `establecerTarifaComision` (definida en las líneas 72 a 78 de la figura 11.5) validan sus argumentos antes de asignar los valores a los miembros de datos `ventasBrutas` y `tarifaComision`, respectivamente.

```
1  // Fig. 11.4: EmpleadoPorComision.h
2  // La definición de la clase EmpleadoPorComision representa a un empleado por
      comisión.
3  #ifndef COMISION_H
4  #define COMISION_H
5
6  #include <string> // la clase string estándar de C++
7
```

Fig. 11.4 | Archivo de encabezado de la clase `EmpleadoPorComision` (parte 1 de 2).

```
8  class EmpleadoPorComision
9  {
10 public:
11    EmpleadoPorComision( const std::string &, const std::string &,
12       const std::string &, double = 0.0, double = 0.0 );
13
14    void establecerPrimerNombre( const std::string & );
      // establece el primer nombre
15    std::string obtenerPrimerNombre() const; // devuelve el primer nombre
16
17    void establecerApellidoPaterno( const std::string & );
      // establece el apellido paterno
18    std::string obtenerApellidoPaterno() const; // devuelve el apellido paterno
19
20    void establecerNumeroSeguroSocial( const std::string & ); // establece el NSS
21    std::string obtenerNumeroSeguroSocial() const; // devuelve el NSS
22
23    void establecerVentasBrutas( double ); // establece el monto de ventas brutas
24    double obtenerVentasBrutas() const; // devuelve el monto de ventas brutas
25
26    void establecerTarifaComision( double );
      // establece la tarifa de comisión (porcentaje)
27    double obtenerTarifaComision() const; // devuelve la tarifa de comisión
28
29    double ingresos() const; // calcula los ingresos
30    void imprimir() const; // imprime el objeto EmpleadoPorComision
31 private:
32    std::string primerNombre;
33    std::string apellidoPaterno;
34    std::string numeroSeguroSocial;
35    double ventasBrutas; // ventas brutas por semana
36    double tarifaComision; // porcentaje de comisión
37 }; // fin de la clase EmpleadoPorComision
38
39 #endif
```

Fig. 11.4 | Archivo de encabezado de la clase `EmpleadoPorComision` (parte 2 de 2).

```
1  // Fig. 11.5: EmpleadoPorComision.cpp
2  // Definiciones de las funciones miembro de la clase EmpleadoPorComision.
3  #include <iostream>
4  #include <stdexcept>
5  #include "EmpleadoPorComision.h" // definición de la clase EmpleadoPorComision
6  using namespace std;
7
8  // constructor
9  EmpleadoPorComision::EmpleadoPorComision(
10    const string &nombre, const string &apellido, const string &nss,
11    double ventas, double tarifa )
12 {
13    primerNombre = nombre; // debe validar
14    apellidoPaterno = apellido; // debe validar
15    numeroSeguroSocial = nss; // debe validar
16    establecerVentasBrutas( ventas ); // valida y almacena las ventas brutas
```

Fig. 11.5 | Archivo de implementación para la clase `EmpleadoPorComision` que representa a un empleado que recibe un porcentaje de las ventas brutas (parte 1 de 3).

```
17    establecerTarifaComision( tarifa ); // valida y almacena la tarifa de comisión
18 } // fin del constructor de EmpleadoPorComision
19
20 // establece el primer nombre
21 void EmpleadoPorComision::establecerPrimerNombre( const string &nombre )
22 {
23    primerNombre = nombre; // debe validar
24 } // fin de la función establecerPrimerNombre
25
26 // devuelve el primer nombre
27 EmpleadoPorComision::obtenerPrimerNombre() const
28 {
29    return primerNombre;
30 } // fin de la función obtenerPrimerNombre
31
32 // establece el apellido paterno
33 void EmpleadoPorComision::establecerApellidoPaterno( const string &apellido )
34 {
35    apellidoPaterno = apellido; // debe validar
36 } // fin de la función establecerApellidoPaterno
37
38 // devuelve el apellido paterno
39 string EmpleadoPorComision::obtenerApellidoPaterno() const
40 {
41    return apellidoPaterno;
42 } // fin de la función obtenerApellidoPaterno
43
44 // establece el número de seguro social
45 void EmpleadoPorComision::establecerNumeroSeguroSocial( const string &nss )
46 {
47    numeroSeguroSocial = nss; // debe validar
48 } // fin de la función establecerNumeroSeguroSocial
49
50 // return social security number
51 string EmpleadoPorComision::obtenerNumeroSeguroSocial() const
52 {
53    return numeroSeguroSocial;
54 } // fin de la función obtenerNumeroSeguroSocial
55
56 // establece el monto de ventas brutas
57 void EmpleadoPorComision::establecerVentasBrutas( double ventas )
58 {
59    if ( ventas >= 0.0 )
60       ventasBrutas = ventas;
61    else
62       throw invalid_argument( "Las ventas brutas deben ser >= 0.0" );
63 } // fin de la función establecerVentasBrutas
64
65 // devuelve el monto de ventas brutas
66 double EmpleadoPorComision::obtenerVentasBrutas() const
67 {
```

Fig. 11.5 | Archivo de implementación para la clase `EmpleadoPorComision` que representa a un empleado que recibe un porcentaje de las ventas brutas (parte 2 de 3).

```
68    return ventasBrutas;
69 } // fin de la función obtenerVentasBrutas
70
71 // establece la tarifa de comisión
72 void EmpleadoPorComision::establecerTarifaComision( double tarifa )
73 {
74    if ( tarifa > 0.0 && tarifa < 1.0 )
75       tarifaComision = tarifa;
76    else
77       throw invalid_argument( "La tarifa de comision debe ser > 0.0 y < 1.0" );
78 } // fin de la función establecerTarifaComision
79
80 // devuelve la tarifa de comisión
81 double EmpleadoPorComision::obtenerTarifaComision() const
82 {
83    return tarifaComision;
84 } // fin de la función obtenerTarifaComision
85
86 // calcula los ingresos
87 double EmpleadoPorComision::ingresos() const
88 {
89    return tarifaComision * ventasBrutas;
90 } // fin de la función ingresos
91
92 // imprime el objeto EmpleadoPorComision
93 void EmpleadoPorComision::imprimir() const
94 {
95    cout << "empleado por comision: " << primerNombre << ' ' << apellidoPaterno
96       << "\nnumero de seguro social: " << numeroSeguroSocial
97       << "\nventas brutas: " << ventasBrutas
98       << "\ntarifa de comision: " << tarifaComision;
99 } // fin de la función imprimir
```

Fig. 11.5 | Archivo de implementación para la clase `EmpleadoPorComision` que representa a un empleado que recibe un porcentaje de las ventas brutas (parte 3 de 3).

Constructor de `EmpleadoPorComision`

La definición del constructor de `EmpleadoPorComision` *no utiliza a propósito la sintaxis de inicialización de miembros* en los primeros ejemplos de esta sección, para que podamos demostrar cómo afectan los especificadores `private` y `protected` al acceso a los miembros en las clases derivadas. Como se muestra en la figura 11.5, en las líneas 13 a 15 asignamos valores a los miembros de datos `primerNombre`, `apellidoPaterno` y `numeroSeguroSocial` en el cuerpo del constructor. Más adelante en esta sección, volveremos a usar listas de inicialización de miembros en los constructores.

No validamos los valores de los argumentos `nombre`, `apellido` y `nss` del constructor antes de asignarlos a los correspondientes miembros de datos. Sin duda, hubiéramos podido validar el nombre y el apellido; tal vez asegurándonos que sean de una longitud razonable. De manera similar, se podría validar un número de seguro social para asegurar que contenga nueve dígitos "o el número correspondiente según el país), con o sin guiones cortos (por ejemplo, `123-45-6789` o `123456789`).

Las funciones miembro `ingresos` *e* `imprimir` *de* `EmpleadoPorComision`

La función miembro `ingresos` (líneas 87 a 90) calcula los ingresos de un `EmpleadoPorComision`. En la línea 89 se multiplica la `tarifaComision` por las `ventasBrutas` y se devuelve el resultado. La

función miembro `imprimir` (líneas 93 a 99) muestra los valores de los miembros de datos de un objeto `EmpleadoPorComision`.

Prueba de la clase `EmpleadoPorComision`

La figura 11.6 prueba la clase `EmpleadoPorComision`. En las líneas 11 y 12 se instancia el objeto `empleado` de la clase `EmpleadoPorComision` y se invoca su constructor para inicializar el objeto con `"Sue"` como primer nombre, `"Jones"` como apellido paterno, `"222-22-2222"` como número de seguro social, `10000` como el monto de ventas brutas y `.06` como la tarifa de comisión. En las líneas 19 a 24 se utilizan las funciones *obtener* de `empleado` para mostrar los valores de sus miembros de datos. En las líneas 26 y 27 se invocan las funciones miembro del objeto `establecerVentasBrutas` y `establecerTarifaComision` para modificar los valores de los datos miembro `ventasBrutas` y `tarifaComision`, respectivamente. Después, en la línea 31 se llama a la función miembro `imprimir` de `empleado` para imprimir la información actualizada del `EmpleadoPorComision`. Por último, en la línea 34 se muestran los ingresos del `EmpleadoPorComision`, calculados por la función miembro `ingresos` del objeto mediante el uso de los valores actualizados de los datos miembro `ventasBrutas` y `tarifaComision`.

```
1  // Fig. 11.6: fig11_06.cpp
2  // Prueba de la clase EmpleadoPorComision.
3  #include <iostream>
4  #include <iomanip>
5  #include "EmpleadoPorComision.h" // definición de la clase EmpleadoPorComision
6  using namespace std;
7
8  int main()
9  {
10    // instancia un objeto EmpleadoPorComision
11    EmpleadoPorComision empleado(
12       "Sue", "Jones", "222-22-2222", 10000, .06 );
13
14    // establece el formato de salida de punto flotante
15    cout << fixed << setprecision( 2 );
16
17    // obtiene los datos del empleado por comisión
18    cout << "Informacion del empleado obtenida por las funciones obtener: \n"
19       << "\nEl primer nombre es " << empleado.obtenerPrimerNombre()
20       << "\nEl apellido paterno es " << empleado.obtenerApellidoPaterno()
21       << "\nEl numero de seguro social es "
22       << empleado.obtenerNumeroSeguroSocial()
23       << "\nLas ventas brutas son " << empleado.obtenerVentasBrutas()
24       << "\nLa tarifa de comision es " << empleado.obtenerTarifaComision()
         << endl;
25
26    empleado.establecerVentasBrutas( 8000 ); // establece las ventas brutas
27    empleado.establecerTarifaComision( .1 ); // establece la tarifa de comisión
28
29    cout << "\nInformacion actualizada del empleado, mostrada por la funcion
                imprimir: \n"
30       << endl;
31    empleado.imprimir(); // muestra la nueva informacion del empleado
32
33    // muestra los ingresos del empleado
34    cout << "\n\nIngresos del empleado: $" << empleado.ingresos() << endl;
35 } // fin de main
```

Fig. 11.6 | Programa de prueba de la clase `EmpleadoPorComision` (parte 1 de 2).

```
Informacion del empleado obtenida por las funciones obtener:

El primer nombre es Sue
El apellido paterno es Jones
El numero de seguro social es 222-22-2222
Las ventas brutas son 10000.00
La tarifa de comision es 0.06

Informacion actualizada del empleado, mostrada por la funcion imprimir:

empleado por comision: Sue Jones
numero de seguro social: 222-22-2222
ventas brutas: 8000.00
tarifa de comision: 0.10

Ingresos del empleado: $800.00
```

Fig. 11.6 | Programa de prueba de la clase `EmpleadoPorComision` (parte 2 de 2).

10.3.2 Creación de una clase `EmpleadoBaseMasComision` sin usar la herencia

Ahora veremos la segunda parte de nuestra introducción a la herencia, para ello crearemos y probaremos una clase (completamente nueva e independiente) llamada `EmpleadoBaseMasComision` (figuras 11.7 a 11.8), la cual contiene un primer nombre, apellido paterno, número de seguro social, monto de ventas brutas, tarifa de comisión *y* salario base.

```
1  // Fig. 11.7: EmpleadoBaseMasComision.h
2  // Definición de la clase EmpleadoBaseMasComision que representa
3  // a un empleado que recibe un salario base además de la comisión.
4  #ifndef BASEMAS_H
5  #define BASEMAS_H
6
7  #include <string> // clase string estándar de C++
8
9  class EmpleadoBaseMasComision
10 {
11 public:
12    EmpleadoBaseMasComision( const std::string &, const std::string &,
13       const std::string &, double = 0.0, double = 0.0, double = 0.0 );
14
15    void obtenerPrimerNombre( const std::string & ); // establece el primer nombre
16    std::obtenerPrimerNombre() const; // devuelve el primer nombre
17
18    void obtenerApellidoPaterno( const std::string & );
      // establece el apellido paterno
19    std::obtenerApellidoPaterno() const; // devuelve el apellido paterno
20
21    void establecerNumeroSeguroSocial( const std::string & ); // establece el NSS
22    std::obtenerNumeroSeguroSocial() const; // devuelve el NSS
23
24    void establecerVentasBrutas( double ); // establece el monto de ventas brutas
25    double obtenerVentasBrutas() const; // devuelve el monto de ventas brutas
```

Fig. 11.7 | Archivo de encabezado de la clase `EmpleadoBaseMasComision` (parte 1 de 2).

```
26
27    void establecerTarifaComision( double ); // establece la tarifa de comisión
28    double obtenerTarifaComision() const; // devuelve la tarifa de comisión
29
30    void establecerSalarioBase( double ); // establece el salario base
31    double obtenerSalarioBase() const; // devuelve el salario base
32
33    double ingresos() const; // calcula los ingresos
34    void imprimir() const; // imprime el objeto EmpleadoBaseMasComision
35 private:
36    std::string primerNombre;
37    std::string apellidoPaterno;
38    std::string numeroSeguroSocial;
39    double ventasBrutas; // ventas brutas por semana
40    double tarifaComision; // porcentaje de comisión
41    double salarioBase; // salario base
42 }; // fin de la clase EmpleadoBaseMasComision
43
44 #endif
```

Fig. 11.7 | Archivo de encabezado de la clase `EmpleadoBaseMasComision` (parte 2 de 2).

```
1  // Fig. 11.8: EmpleadoBaseMasComision.cpp
2  // Definiciones de las funciones miembro de la clase EmpleadoBaseMasComision.
3  #include <iostream>
4  #include <stdexcept>
5  #include "EmpleadoBaseMasComision.h"
6  using namespace std;
7
8  // constructor
9  EmpleadoBaseMasComision::EmpleadoBaseMasComision(
10    const string &nombre, const string &apellido, const string &nss,
11    double ventas, double tarifa, double salario )
12 {
13    primerNombre = nombre; // debe validar
14    apellidoPaterno = apellido; // debe validar
15    numeroSeguroSocial = nss; // debe validar
16    establecerVentasBrutas( ventas ); // valida y almacena las ventas brutas
17    establecerTarifaComision( tarifa ); // valida y almacena la tarifa de comisión
18    establecerSalarioBase( salario ); // valida y almacena el salario base
19 } // fin del constructor de EmpleadoBaseMasComision
20
21 // establece el primer nombre
22 void EmpleadoBaseMasComision::obtenerPrimerNombre( const string &nombre )
23 {
24    primerNombre = nombre; // debe validar
25 } // fin de la función obtenerPrimerNombre
26
27 // devuelve el primer nombre
28 string EmpleadoBaseMasComision::obtenerPrimerNombre() const
29 {
```

Fig. 11.8 | La clase `EmpleadoBaseMasComision` representa a un empleado que recibe un salario base, además de una comisión (parte 1 de 3).

```
30    return primerNombre;
31  } // fin de la función obtenerPrimerNombre
32
33  // establece el apellido paterno
34  void EmpleadoBaseMasComision::obtenerApellidoPaterno( const string &apellido )
35  {
36    apellidoPaterno = apellido; // debe validar
37  } // fin de la función obtenerApellidoPaterno
38
39  // devuelve el apellido paterno
40  string EmpleadoBaseMasComision::obtenerApellidoPaterno() const
41  {
42    return pellidoPaterno;
43  } // fin de la función obtenerApellidoPaterno
44
45  // establece el número de seguro social
46  void EmpleadoBaseMasComision::establecerNumeroSeguroSocial(
47    const string &nss )
48  {
49    numeroSeguroSocial = nss; // debe validar
50  } // fin de la función setNumeroSeguroSocial
51
52  // devuelve el número de seguro social
53  string EmpleadoBaseMasComision::obtenerNumeroSeguroSocial() const
54  {
55    return numeroSeguroSocial;
56  } // fin de la función obtenerNumeroSeguroSocial
57
58  // establece el monto de ventas brutas
59  void EmpleadoBaseMasComision::establecerVentasBrutas( double ventas )
60  {
61    if ( ventas >= 0.0 )
62      ventasBrutas = ventas;
63    else
64      throw invalid_argument( "Las ventas brutas deben ser >= 0.0" );
65  } // fin de la función establecerVentasBrutas
66
67  // devuelve el monto de ventas brutas
68  double EmpleadoBaseMasComision::obtenerVentasBrutas() const
69  {
70    return ventasBrutas;
71  } // fin de la función obtenerVentasBrutas
72
73  // establece la tarifa de comisión
74  void EmpleadoBaseMasComision::establecerTarifaComision( double tarifa )
75  {
76    if ( tarifa > 0.0 && tarifa < 1.0 )
77      tarifaComision = tarifa;
78    else
79      throw invalid_argument( "La tarifa de comision debe ser > 0.0 y < 1.0" );
80  } // fin de la función setTarifaComision
81
```

Fig. 11.8 | La clase `EmpleadoBaseMasComision` representa a un empleado que recibe un salario base, además de una comisión (parte 2 de 3).

```
82 // devuelve la tarifa de comisión
83 double EmpleadoBaseMasComision::obtenerTarifaComision() const
84 {
85    return tarifaComision;
86 } // fin de la función obtenerTarifaComision
87
88 // establece el salario base
89 void EmpleadoBaseMasComision::establecerSalarioBase( double salario )
90 {
91    if ( salario >= 0.0 )
92       salarioBase = salario;
93    else
94       throw invalid_argument( "El salario debe ser >= 0.0" );
95 } // fin de la función establecerSalarioBase
96
97 // devuelve el salario base
98 double EmpleadoBaseMasComision::obtenerSalarioBase() const
99 {
100    return salarioBase;
101 } // fin de la función obtenerSalarioBase
102
103 // calcula los ingresos
104 double EmpleadoBaseMasComision::ingresos() const
105 {
106    return salarioBase + ( tarifaComision * ventasBrutas );
107 } // fin de la función ingresos
108
109 // imprime el objeto EmpleadoBaseMasComision
110 void EmpleadoBaseMasComision::imprimir() const
111 {
112    cout << "empleado por comision con salario base: " << primerNombre << ' '
113       << apellidoPaterno << "\nnumero de seguro social: " << numeroSeguroSocial
114       << "\nventas brutas: " << ventasBrutas
115       << "\ntarifa de comision: " << tarifaComision
116       << "\nsalario base: " << salarioBase
117 } // fin de la función imprimir
```

Fig. 11.8 | La clase `EmpleadoBaseMasComision` representa a un empleado que recibe un salario base, además de una comisión (parte 3 de 3).

Definición de la clase `EmpleadoBaseMasComision`

El encabezado de `EmpleadoBaseMasComision` (figura 11.7) especifica los servicios `public` de la clase `EmpleadoBaseMasComision`, que incluyen el constructor de `EmpleadoBaseMasComision` (líneas 12 y 13) y las funciones miembro `ingresos` (línea 33) e `imprimir` (línea 34). En las líneas 15 a 31 se declaran funciones `public` *obtener* y *establecer* para los miembros de datos `private` de la clase (que se declaran en las líneas 36 a 41): `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas`, `tarifaComision` *y* `salarioBase`. Estas variables y las funciones miembro encapsulan todas las características necesarias de un empleado por comisión con salario base. Observe la similitud entre esta clase y la clase `EmpleadoPorComision` (figuras 11.4 y 11.5); en este ejemplo *no* explotaremos todavía esa similitud.

La función miembro `ingresos` de la clase `EmpleadoBaseMasComision` (definida en las líneas 104 a 107 de la figura 11.8) calcula los ingresos de un empleado por comisión con salario base. En la línea 106 se devuelve el resultado de sumar el salario base del empleado al producto de la tarifa de comisión y las ventas brutas del empleado.

Prueba de la clase `EmpleadoBaseMasComision`

La figura 11.9 prueba la clase `EmpleadoBaseMasComision`. En las líneas 11 y 12 se instancia el objeto `empleado` de la clase `EmpleadoBaseMasComision`, y se pasan los datos `"Bob"`, `"Lewis"`, `"333-33-3333"`, `5000`, `.04` y `300` al constructor como primer nombre, apellido paterno, número de seguro social, ventas brutas, tarifa de comisión *y* salario base, respectivamente. En las líneas 19 a 25 se utilizan las funciones *obtener* de `EmpleadoBaseMasComision` para obtener los valores de los miembros de datos del objeto para la salida. En la línea 27 se invoca la función miembro `establecerSalarioBase` del objeto para modificar el salario base. La función miembro `establecerSalarioBase` (figura 11.8, líneas 89 a 95) asegura que al miembro de datos `salarioBase` no se le asigne un valor negativo, ya que el salario base de un empleado no puede ser negativo. En la línea 31 de la figura 11.9 se invoca la función miembro `imprimir` del objeto para imprimir la información actualizada del `EmpleadoBaseMasComision`, y en la línea 34 se llama a la función miembro `ingresos` para mostrar los ingresos de `EmpleadoBaseMasComision`.

```
1  // Fig. 11.9: fig11_09.cpp
2  // Programa de prueba de la clase EmpleadoBaseMasComision.
3  #include <iostream>
4  #include <iomanip>
5  #include "EmpleadoBaseMasComision.h"
6  using namespace std;
7
8  int main()
9  {
10    // instancia un objeto EmpleadoBaseMasComision
11    EmpleadoBaseMasComision
12       empleado( "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
13
14    // establece el formato de salida de punto flotante
15    cout << fixed << setprecision( 2 );
16
17    // obtiene los datos del empleado por comisión
18    cout << "Informacion del empleado obtenida por las funciones obtener: \n"
19       << "\nEl primer nombre es " << empleado.obtenerPrimerNombre()
20       << "\nEl apellido paterno es " << empleado.obtenerApellidoPaterno()
21       << "\nEl numero de seguro social es "
22       << empleado.obtenerNumeroSeguroSocial()
23       << "\nLas ventas brutas son " << empleado.obtenerVentasBrutas()
24       << "\nLa tarifa de comision es " << empleado.obtenerTarifaComision()
25       << "\nEl salario base es " << empleado.obtenerSalarioBase() << endl;
26
27    empleado.establecerSalarioBase( 1000 ); // establece el salario base
28
29    cout << "\nInformacion actualizada del empleado, impresa por la funcion
             imprimir: \n"
30       << endl;
31    empleado.imprimir(); // muestra la nueva informacion del empleado
32
33    // muestra los ingresos del empleado
34    cout << "\n\nIngresos del empleado: $" << empleado.ingresos() << endl;
35 } // fin de main
```

Fig. 11.9 | Programa de prueba de la clase `EmpleadoBaseMasComision` (parte 1 de 2).

```
Informacion del empleado obtenida por las funciones obtener:

El primer nombre es Bob
El apellido paterno es Lewis
El numero de seguro social es 333-33-3333
Las ventas brutas son 5000.00
La tarifa de comision es 0.04
El salario base es 300.00

Informacion actualizada del empleado, impresa por la funcion imprimir:

empleado por comision con salario base: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa de comision: 0.04
salario base: 1000.00

Ingresos del empleado: $1200.00
```

Fig. 11.9 | Programa de prueba de la clase `EmpleadoBaseMasComision` (parte 2 de 2).

Exploración de las similitudes entre la clase `EmpleadoBaseMasComision` *y la clase* `EmpleadoPorComision`

La mayoría del código para la clase `EmpleadoBaseMasComision` (figuras 11.7 y 11.8) es *similar (si no es que idéntico)* al código de la clase `EmpleadoPorComision` (figuras 11.4 y 11.5). Por ejemplo, en la clase `EmpleadoBasePorComision`, los miembros de datos `private` `primerNombre` y `apellidoPaterno`, y las funciones miembro `obtenerPrimerNombre`, `obtenerPrimerNombre`, `obtenerApellidoPaterno` y `establecerApellidoPaterno` son idénticos a los de la clase `EmpleadoPorComision`. Las clases `EmpleadoPorComision` y `EmpleadoBaseMasComision` también contienen los miembros de datos `private` `numeroSeguroSocial`, `tarifaComision` y `ventasBrutas`, así como funciones *obtener* y *establecer* para manipular esos miembros. Además, el constructor de `EmpleadoBaseMasComision` es *casi* idéntico al de la clase `EmpleadoPorComision`, excepto que el constructor de `EmpleadoBaseMasComision` *también* establece el `salarioBase`. Las demás adiciones a la clase `EmpleadoBaseMasComision` son el miembro de datos `private` `salarioBase`, *y* las funciones miembro `establecerSalario` y `obtenerSalarioBase`. La función miembro `imprimir` de la clase `EmpleadoBaseMasComision` es *casi idéntica* a la de la clase `EmpleadoPorComision`, excepto que la función `imprimir` de `EmpleadoBaseMasComision` *también* imprime el valor del miembro de datos `salarioBase`.

Literalmente *copiamos* el código de la clase `EmpleadoPorComision` y lo *pegamos* en la clase `EmpleadoBaseMasComision`, después modificamos la clase `EmpleadoBaseMasComision` para incluir un salario base y las funciones miembro que lo manipulan. Este método *de copiar y pegar* es propensa a errores y consume mucho tiempo.

Observación de Ingeniería de Software 11.1

Copiar y pegar código *de una clase a otra puede esparcir muchas* copias *físicas* del mismo código *y puede esparcir* errores *a través de un sistema, creando una pesadilla para el mantenimiento de código. Para evitar duplicar código (y posiblemente los errores), use la* herencia *en vez del método de "copiar y pegar" en situaciones en las que una clase debe "absorber" los miembros de datos y las funciones miembro de otra clase.*

Observación de Ingeniería de Software 11.2

Con la herencia, los miembros de datos y las funciones miembro comunes de todas las clases en la jerarquía se declaran en una clase base. Cuando se requieren cambios para esas características comunes, es necesario realizar los cambios sólo en la clase base; así, las clases derivadas pueden heredar los cambios. Sin la herencia, los cambios tendrían que realizarse en todos *los archivos de código fuente que contengan una copia del código en cuestión.*

10.3.3 Creación de una jerarquía de herencia EmpleadoPorComision-EmpleadoBaseMasComision

Ahora crearemos y probaremos una nueva clase `EmpleadoBaseMasComision` (figuras 11.10 y 11.11) que se *deriva de* la clase `EmpleadoPorComision` (figuras 11.4 y 11.5). En este ejemplo, un objeto `EmpleadoBaseMasComision` *es-un* `EmpleadoPorComision` (debido a que la herencia transfiere las capacidades de la clase `EmpleadoPorComision`), pero la clase `EmpleadoBaseMasComision` *también* tiene el miembro de datos `salarioBase` (figura 11.10, línea 22). El *signo de dos puntos (:)* en la línea 10 de la definición de clase indica la herencia. La palabra clave `public` indica el *tipo de herencia*. Como clase derivada (que se forma con herencia `public`), `EmpleadoBaseMasComision` hereda *todos* los miembros de la clase `EmpleadoPorComision`, *excepto* el constructor; cada clase proporciona sus *propios* constructores específicos (los destructores tampoco se heredan). Por ende, los servicios `public` de `EmpleadoBaseMasComision` incluyen a su constructor (líneas 13 y 14) y las funciones miembro `public` heredadas de la clase `EmpleadoPorComision`; *aunque no podemos ver estas funciones miembro heredadas en el* código fuente de `EmpleadoBaseMasComision`, forman sin duda parte de esta clase derivada. Los servicios `public` de la clase derivada también incluyen las funciones miembro `establecerSalarioBase`, `obtenerSalarioBase`, `ingresos` e `imprimir` (líneas 16 a 20).

```
1  // Fig. 11.10: EmpleadoBaseMasComision.h
2  // Clase EmpleadoBaseMasComision derivada de la clase
3  // EmpleadoPorComision.
4  #ifndef BASEMAS_H
5  #define BASEMAS_H
6
7  #include <string> // clase string estándar de C++
8  #include "EmpleadoPorComision.h" // declaración de la clase EmpleadoPorComision
9
10 class EmpleadoBaseMasComision : public EmpleadoPorComision
11 {
12 public:
13    EmpleadoBaseMasComision( const std::string &, const std::string &,
14       const std::string &, double = 0.0, double = 0.0, double = 0.0 );
15
16    void establecerSalarioBase( double ); // establece el salario base
17    double obtenerSalarioBase() const; // devuelve el salario base
18
19    double ingresos() const; // calcula los ingresos
20    void imprimir() const; // imprime el objeto EmpleadoBaseMasComision
```

Fig. 11.10 | Definición de la clase `EmpleadoBaseMasComision` que indica la relación de herencia con la clase `EmpleadoPorComision` (parte 1 de 2).

```
21   private:
22      double salarioBase; // salario base
23   }; // fin de la clase EmpleadoBaseMasComision
24
25   #endif
```

Fig. 11.10 | Definición de la clase `EmpleadoBaseMasComision` que indica la relación de herencia con la clase `EmpleadoPorComision` (parte 2 de 2).

```
1   // Fig. 11.11: EmpleadoBaseMasComision.cpp
2   // Definiciones de las funciones miembro de la clase EmpleadoBaseMasComision.
3   #include <iostream>
4   #include <stdexcept>
5   #include "EmpleadoBaseMasComision.h"
6   using namespace std;
7
8   // constructor
9   EmpleadoBaseMasComision::EmpleadoBaseMasComision(
10     const string &nombre, const string &apellido, const string &nss,
11     double ventas, double tarifa, double salario )
12     // llama explícitamente al constructor de la clase base
13     : EmpleadoPorComision( nombre, apellido, nss, ventas, tarifa )
14  {
15     establecerSalarioBase( salario ); // valida y almacena el salario base
16  } // fin del constructor de EmpleadoBaseMasComision
17
18  // establece el salario base
19  void EmpleadoBaseMasComision::establecerSalarioBase( double salario )
20  {
21     if ( salario >= 0.0 )
22        salarioBase = salario;
23     else
24        throw invalid_argument( "El salario debe ser >= 0.0" );
25  } // fin de la función setSalarioBase
26
27  // devuelve el salario base
28  double EmpleadoBaseMasComision::obtenerSalarioBase() const
29  {
30     return salarioBase;
31  } // fin de la función obtenerSalarioBase
32
33  // calcula los ingresos
34  double EmpleadoBaseMasComision::ingresos() const
35  {
36     // la clase derivada no puede acceder a los datos privados de la clase base
37     return salarioBase + ( tarifaComision * ventasBrutas );
38  } // fin de la función ingresos
39
```

Fig. 11.11 | Archivo de implementación de `EmpleadoBaseMasComision`: la clase derivada no puede acceder a los datos `private` de la clase base (parte 1 de 2).

```
40  // imprime el objeto EmpleadoBaseMasComision
41  void EmpleadoBaseMasComision::imprimir() const
42  {
43     // la clase derivada no puede acceder a los datos privados de la clase base
44     cout << "empleado por comision con salario base: " << primerNombre << ' '
45        << apellidoPaterno << "\nnumero de seguro social: " << numeroSeguroSocial
46        << "\nventas brutas: " << ventasBrutas
47        << "\ntarifa de comision: " << tarifaComision
48        << "\nsalario base: " << salarioBase;
49  } // fin de la función imprimir
```

Errores de compilación del compilador LLVM en Xcode 4.5

```
EmpleadoBaseMasComision.cpp:37:26:
   'tarifaComision' is a private member of 'EmpleadoPorComision'
EmpleadoBaseMasComision.cpp:37:43:
   'ventasBrutas' is a private member of 'EmpleadoPorComision'
EmpleadoBaseMasComision.cpp:44:53:
   'primerNombre' is a private member of 'EmpleadoPorComision'
EmpleadoBaseMasComision.cpp:45:10:
   'apellidoPaterno' is a private member of 'EmpleadoPorComision'
EmpleadoBaseMasComision.cpp:45:54:
   'numeroSeguroSocial' is a private member of 'EmpleadoPorComision'
EmpleadoBaseMasComision.cpp:46:31:
   'ventasBrutas' is a private member of 'EmpleadoPorComision'
EmpleadoBaseMasComision.cpp:47:35:
   'tarifaComision' is a private member of 'EmpleadoPorComision'
```

Fig. 11.11 | Archivo de implementación de `EmpleadoBaseMasComision`: la clase derivada no puede acceder a los datos `private` de la clase base (parte 2 de 2).

La figura 11.11 muestra las implementaciones de las funciones miembro de `EmpleadoBaseMasComision`. El constructor (líneas 9 a 16) introduce la **sintaxis de inicialización de clase base** (línea 13), la cual utiliza un inicializador de miembros para pasar argumentos al constructor de la clase base (`EmpleadoPorComision`). C++ requiere que el constructor de una clase derivada llame al constructor de su clase base para inicializar los miembros de datos de la clase base que se heredan en la clase derivada. En la línea 13 se realiza esta tarea, invocando *explícitamente* al constructor de `EmpleadoPorComision` por su nombre, y pasando los parámetros del constructor `nombre`, `apellido`, `nss`, `ventas` y `tarifa` como argumentos para inicializar los miembros de datos de la clase base `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision`, respectivamente. Si el constructor de `EmpleadoBaseMasComision` *no* invocara al constructor de la clase `EmpleadoPorComision` de manera *explícita*, C++ trataría de invocar de manera implícita al constructor predeterminado de la clase `EmpleadoPorComision`; pero esta clase *no* tiene un constructor de este tipo, por lo que el compilador generaría un *error*. En el capítulo 3 vimos que el compilador proporciona un constructor predeterminado sin parámetros en cualquier clase que *no* incluya de manera explícita un constructor. Sin embargo, `EmpleadoPorComision` *sí incluye* de manera explícita un constructor, por lo que *no* se proporciona un constructor predeterminado.

Error común de programación 11.1

Si el constructor de una clase derivada llama a uno de los constructores de su clase base, los argumentos que se pasan al constructor de la clase base deben ser consistentes con el número y tipos de los parámetros especificados en uno de los constructores de la clase base; de lo contrario se producirá un error de compilación.

Tip de rendimiento 11.1

En el constructor de una clase derivada, al inicializar los objetos miembro e invocar a los constructores de la clase base de manera explícita en la lista de inicializadores de miembros, se evita duplicar la inicialización en la que se hace la llamada a un constructor predeterminado, y después los miembros de datos se modifican de nuevo en el cuerpo del constructor de la clase derivada.

Errores de compilación al acceder a los miembros `private` de la clase base

El compilador genera errores para la línea 37 de la figura 11.11 debido a que los miembros de datos `tarifaComision` y `ventasBrutas` de la clase base `EmpleadoPorComision` son `private`; las funciones miembro de la clase derivada `EmpleadoBaseMasComision` *no* pueden acceder a los datos `private` de la clase base `EmpleadoPorComision`. El compilador genera errores adicionales en las líneas 44 a 47 de la función miembro `imprimir` de `EmpleadoBaseMasComision` por la misma razón. Como podemos ver, C++ hace cumplir rígidamente las restricciones en cuanto al acceso a los miembros de datos `private`, de tal forma que *hasta una clase derivada (que está íntimamente relacionada con su clase base) no puede acceder a los datos `private` de la clase base*.

Evitar los errores en `EmpleadoBaseMasComision`

Incluimos a propósito el código erróneo en la figura 11.11 para enfatizar que las funciones miembro de una clase derivada *no pueden* acceder a los datos `private` de su clase base. Los errores en `EmpleadoBaseMasComision` se podían haber evitado mediante el uso de las funciones miembro *obtener* heredadas de la clase `EmpleadoPorComision`. Por ejemplo, en la línea 37 se podía haber invocado a `obtenerTarifaComision` y a `establecerVentasBrutas` para acceder a los datos miembro `private` `tarifaComision` y `ventasBrutas` de `EmpleadoPorComision`, respectivamente. De manera similar, en las líneas 44 a 47 se pudieron haber utilizado funciones miembro *obtener* apropiadas para obtener los valores de los miembros de datos de la clase base. En el siguiente ejemplo le mostraremos cómo el uso de datos `protected` *también* nos permite evitar los errores que encontramos en este ejemplo.

Cómo incluir el encabezado de la clase base en el encabezado de la clase derivada mediante `#include`

Observe que incluimos el encabezado `#include` de la clase base en el encabezado de la clase derivada (línea 8 de la figura 11.10). Esto es necesario por tres razones. En primer lugar, para que la clase derivada utilice el nombre de la clase base en la línea 10, debemos indicar al compilador que la clase base existe; la definición de clase en `EmpleadoPorComision.h` hace exactamente eso.

La segunda razón es que el compilador utiliza una definición de clase para determinar el *tamaño* de un objeto de esa clase (como vimos en la sección 3.6). Un programa cliente que crea un objeto de una clase debe incluir (mediante `#include`) la definición de clase para permitir que el compilador reserve la cantidad apropiada de memoria para el objeto. Al usar herencia, el tamaño de un objeto de una clase derivada depende de los miembros de datos declarados explícitamente en su definición de clase, *y* de los miembros de datos *heredados* de sus clases base directa e indirecta. Al incluir la definición de la clase base en la línea 8, permitimos que el compilador determine los requerimientos de memoria para los miembros de datos de la clase base que se conviertan en parte del objeto de una clase derivada, y por ende contribuyen al tamaño total del objeto de la clase derivada.

La última razón de incluir la línea 8 es para permitir al compilador determinar si la clase derivada utiliza los miembros heredados de la clase base en forma apropiada. Por ejemplo, en el programa de las figuras 11.10 y 11.11, el compilador utiliza el encabezado de la clase base para determinar que los miembros de datos que está usando la clase derivada son `private` en la clase base. Como éstos son *inaccesibles* para la clase derivada, el compilador genera errores. El compilador también utiliza los *prototipos de las funciones* de la clase base para *validar* las llamadas a funciones realizadas por la clase derivada a las funciones heredadas de la clase base.

El proceso de enlace en una jerarquía de herencia

En la sección 3.7, hablamos sobre el proceso de enlace para crear una aplicación `LibroCalificaciones` ejecutable. En ese ejemplo, vimos que el código objeto del cliente se enlazó con el código objeto para la clase `LibroCalificaciones`, así como con el código objeto para cualquier clase de la Biblioteca estándar de C++ utilizada en el código cliente o en la clase `LibroCalificaciones`.

El proceso de enlace es similar para un programa que utiliza clases en una jerarquía de herencia. El proceso requiere el código objeto para todas las clases utilizadas en el programa, y el código objeto para las clases base directas e indirectas de cualquier clase derivada utilizada por el programa. Suponga que un cliente desea crear una aplicación que utilice la clase `EmpleadoBaseMasComision`, la cual es una clase derivada de `EmpleadoPorComision` (en la sección 11.3.4 veremos un ejemplo de esto). Al compilar la aplicación cliente, el código objeto del cliente debe enlazarse con el código objeto para las clases `EmpleadoBaseMasComision` y `EmpleadoPorComision`, ya que la clase `EmpleadoBaseMasComision` hereda las funciones miembro de su clase base `EmpleadoPorComision`. El código también se enlaza con el código objeto para cualquier clase de la Biblioteca estándar de C++ que se utilice en las clases `EmpleadoPorComision`, `EmpleadoBaseMasComision` o en el código cliente. Esto proporciona al programa acceso a las implementaciones de toda la funcionalidad que el programa puede utilizar.

10.3.4 La jerarquía de herencia `EmpleadoPorComision-EmpleadoBaseMasComision` mediante el uso de datos `protected`

En el capítulo 3 presentamos los especificadores de acceso `public` y `private`. Los miembros `public` de una clase base pueden utilizarse dentro de su cuerpo y en cualquier parte en donde el programa tenga un manejador (es decir, un nombre, referencia o apuntador) a un objeto de esa clase, o a uno de sus clases derivadas. Los miembros `private` de una clase base pueden utilizarse sólo dentro de su cuerpo y en las funciones `friend` de esa clase base. En esta sección presentaremos el especificador de acceso **`protected`**.

El uso del acceso `protected` ofrece un nivel intermedio de protección entre los accesos `public` y `private`. Para permitir que la clase `EmpleadoBaseMasComision` *acceda directamente* a los miembros de datos `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` de `EmpleadoPorComision`, podemos declarar esos miembros como `protected` en la clase base. Los miembros `protected` de una clase base *pueden* ser utilizados dentro del cuerpo de esa clase base, por los miembros y funciones `friend` de la clase base, y por los miembros y funciones `friend` de cualquier clase derivada de esa clase base.

Definición de la clase base `EmpleadoPorComision` con datos `protected`

La clase `EmpleadoPorComision` (figura 11.12) declara ahora los miembros de datos `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` como `protected` (líneas 31 a 36) en vez de `private`. Las implementaciones de las funciones miembro son idénticas a las de la figura 11.5, por lo que aquí no mostraremos el código de `EmpleadoPorComision.cpp`.

```
1  // Fig. 11.12: EmpleadoPorComision.h
2  // Definición de la clase EmpleadoPorComision con datos protected.
3  #ifndef COMISION_H
4  #define COMISION_H
5
```

Fig. 11.12 | Definición de la clase `EmpleadoPorComision` que declara datos `protected`, para permitir que las clases derivadas accedan a éstos (parte 1 de 2).

```
6  #include <string> // clase string estándar de C++
7
8  class EmpleadoPorComision
9  {
10 public:
11    EmpleadoPorComision( const std::string &, const std::string &,
12       const std::string &, double = 0.0, double = 0.0 );
13
14    void obtenerPrimerNombre( const std::string & ); // establece el primer nombre
15    std::string obtenerPrimerNombre() const; // devuelve el primer nombre
16
17    void obtenerApellidoPaterno( const std::string & );
      // establece el apellido paterno
18    std::string obtenerApellidoPaterno() const; // devuelve el apellido paterno
19
20    void establecerNumeroSeguroSocial( const std::string & ); // establece el NSS
21    std::obtenerNumeroSeguroSocial() const; // devuelve el NSS
22
23    void establecerVentasBrutas( double ); // establece el monto de ventas brutas
24    double obtenerVentasBrutas() const; // devuelve el monto de ventas brutas
25
26    void establecerTarifaComision( double ); // establece la tarifa de comisión
27    double obtenerTarifaComision() const; // devuelve la tarifa de comisión
28
29    double ingresos() const; // calcula los ingresos
30    void imprimir() const; // imprime el objeto EmpleadoPorComision
31 protected:
32    std::string primerNombre;
33    std::string apellidoPaterno;
34    std::string numeroSeguroSocial;
35    double ventasBrutas; // ventas brutas por semana
36    double tarifaComision; // porcentaje de comisión
37 }; // fin de la clase EmpleadoPorComision
38
39 #endif
```

Fig. 11.12 | Definición de la clase `EmpleadoPorComision` que declara datos `protected`, para permitir que las clases derivadas accedan a éstos (parte 2 de 2).

La clase `EmpleadoBaseMasComision`

La definición de la clase `EmpleadoBaseMasComision` de las figuras 11.10 y 11.11 permanece *sin modificaciones*, por lo que *no* la mostraremos aquí de nuevo. Ahora que `EmpleadoBaseMasComision` hereda de la clase `EmpleadoPorComision` actualizada (figura 11.12), los objetos de la clase `EmpleadoBaseMasComision` *pueden* acceder a los miembros de datos heredados que se declaran `protected` en la clase `EmpleadoPorComision` (es decir, los miembros de datos `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision`). Como resultado, el compilador *no* genera errores al compilar las definiciones de las funciones miembro `ingresos` e `imprimir` de `EmpleadoPorComision` en la figura 11.11 (líneas 34 a 38 y 41 a 49, respectivamente). Esto muestra los privilegios especiales que se otorgan a una clase derivada para acceder a los miembros de datos `protected` de su clase base. Los objetos de una clase derivada también pueden acceder a los miembros `protected` en *cualquiera* de las clases base *indirectas* de la clase derivada.

La clase `EmpleadoBaseMasComision` *no* hereda el constructor de la clase `EmpleadoPorComision`. Sin embargo, el constructor de la clase `EmpleadoBaseMasComision` (figura 11.11, líneas 9 a 16) llama al constructor de la clase `EmpleadoPorComision` de manera explícita, mediante la sintaxis de inicializador

de miembros (línea 13). Recuerde que el constructor de `EmpleadoBaseMasComision` debe llamar en forma *explícita* al constructor de la clase `EmpleadoPorComision`, ya que `EmpleadoPorComision` *no* contiene un constructor predeterminado que pueda invocarse en forma implícita.

Prueba de la clase `EmpleadoBaseMasComision` modificada

Para probar la jerarquía de clases actualizada, reutilizamos el programa de prueba de la figura 11.9. Como se muestra en la figura 11.13, la salida es idéntica a la de la figura 11.9. Creamos la primera clase `EmpleadoBaseMasComision` *sin usar la herencia*, y creamos esta versión de `EmpleadoBaseMasComision` *usando la herencia*; sin embargo, ambas clases proporcionan la *misma* funcionalidad. El código para la clase `EmpleadoBaseMasComision` (es decir, los archivos de encabezado y de implementación), que es de 74 líneas, es considerablemente *más corto* que el código para la versión no heredada de la clase, que es de 161 líneas, debido a que la versión heredada absorbe parte de su funcionalidad de `EmpleadoPorComision`, mientras que la versión no heredada no absorbe ninguna funcionalidad. Además, ahora sólo hay *una* copia de la funcionalidad de `EmpleadoPorComision` declarada y definida en la clase `EmpleadoPorComision`. Esto facilita el mantenimiento, la modificación y depuración del código fuente, ya que el código fuente relacionado a un `EmpleadoPorComision` sólo existe en los archivos `EmpleadoPorComision.h` y `EmpleadoPorComision.cpp`.

```
Informacion del empleado obtenida por las funciones obtener:

El primer nombre es Bob
El apellido paterno es Lewis
El numero de seguro social es 333-33-3333
Las ventas brutas son 5000.00
La tarifa de comision es 0.04
El salario base es 300.00

Informacion actualizada del empleado, impresa por la funcion imprimir:

empleado por comision con salario base: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa de comision: 0.04
salario base: 1000.00

Ingresos del empleado: $1200.00
```

Fig. 11.13 | Los datos `protected` de la clase base se pueden utilizar desde la clase derivada.

Observaciones acerca de los datos `protected`

En este ejemplo, declaramos los miembros de datos de la clase base como `protected`, de manera que las clases derivadas puedan modificar los datos de manera directa. Al heredar los miembros de datos `protected` se incrementa ligeramente el rendimiento, ya que podemos acceder directamente a los miembros sin incurrir en la sobrecarga de las llamadas a funciones *establecer* u *obtener*.

Observación de Ingeniería de Software 11.3

En la mayoría de los casos es mejor usar miembros de datos `private` para fomentar la ingeniería de software apropiada, y dejar las cuestiones de optimización de código al compilador. El código del programador será más fácil de mantener, modificar y depurar.

El uso de miembros de datos `protected` crea dos problemas graves. En primer lugar, el objeto de la clase derivada *no* tiene que usar una función miembro para establecer el valor del miembro de datos `protected` de la clase base. Puede asignarse fácilmente un valor *inválido* al miembro de datos `protected`, con lo cual se deja el objeto en un estado *inconsistente*; por ejemplo, con el miembro de datos `ventasBrutas` de `EmpleadoPorComision` declarado como `protected`, el objeto de una clase derivada puede asignar un valor negativo a `ventasBrutas`. El segundo problema con el uso de miembros de datos `protected` es que es más probable que las funciones miembro de la clase derivada se escriban de manera que *dependan en la implementación de la clase base*. Las clases derivadas sólo deben depender de los servicios de la clase base (es decir, las funciones miembro no `private`) y *no* de su implementación. Con los miembros de datos `protected` en la clase base, si se modifica la implementación de la clase base, tal vez haya que modificar *todas* las clases derivadas de esa clase base. Por ejemplo, si por alguna razón tuviéramos que modificar los nombres de los miembros de datos `primerNombre` y `apellidoPaterno` a `nombre` y `apellido`, entonces tendríamos que hacerlo para todas las ocurrencias en las que una clase derivada haga referencia a estos miembros de datos de la clase base directamente. En tal caso, se dice que el software es **frágil** o **quebradizo**, ya que una pequeña modificación en la clase base puede "quebrantar" la implementación de la clase derivada. El programador debe tener la capacidad de modificar la implementación de la clase base y a la vez debe poder seguir proporcionando los *mismos* servicios a las clases derivadas. Desde luego, si cambian los servicios de la clase base, debemos reimplementar nuestras clases derivadas; el buen diseño orientado a objetos trata de evitar esto.

Observación de Ingeniería de Software 11.4

Es apropiado usar el especificador de acceso `protected` cuando una clase base debe proporcionar un servicio (es decir, una función miembro) sólo a sus clases derivadas y funciones `friend`.

Observación de Ingeniería de Software 11.5

Declarar los datos miembro de la clase base como `private` (en vez de declararlos `protected`) permite a los programadores modificar la implementación de la clase base, sin tener que modificar las implementaciones de las clases derivadas.

10.3.5 La jerarquía de herencia `EmpleadoPorComision-EmpleadoBaseMasComision` mediante el uso de datos `private`

Ahora vamos a reexaminar nuestra jerarquía una vez más, pero esta vez incluiremos las *mejores prácticas de ingeniería de software*. La clase `EmpleadoPorComision` ahora declara los miembros de datos `primerNombre`, `apellidoPaterno`, `numeroSeguroSocial`, `ventasBrutas` y `tarifaComision` como `private`, como se muestra anteriormente en las líneas 31 a 36 de la figura 11.4.

Modificaciones en las definiciones de las funciones miembro de `EmpleadoPorComision`

En la implementación del constructor de `EmpleadoPorComision` (figura 11.14, líneas 9 a 16), usamos inicializadores de miembros (línea 12) para establecer los valores de los miembros `primerNombre`, `apellidoPaterno` y `numeroSeguroSocial`. Mostraremos cómo la clase derivada `EmpleadoBaseMasComision` (figura 11.15) puede invocar a las funciones miembro no `private` de la clase base (`establecerPrimerNombre`, `obtenerPrimerNombre`, `establecerApellidoPaterno`, `obtenerApellidoPaterno`, `establecerNumeroSeguroSocial` y `obtenerNumeroSeguroSocial`) para manipular estos miembros de datos.

En el cuerpo del constructor y en los cuerpos de las funciones miembro `ingresos` (figura 11.14, líneas 85 a 88) e `imprimir` (líneas 91 a 98), llamamos a las funciones miembro *establecer* y *obtener* de la clase para acceder a los miembros de datos `private` de ésta. Si decidimos modificar los nombres de los

miembros de datos, las definiciones de `ingresos` e `imprimir` *no* necesitarán modificarse; sólo habrá que cambiar las definiciones de las funciones *obtener* y *establecer* que manipulan directamente a los miembros de datos. *Estos cambios ocurren únicamente dentro de la clase base; no es necesario modificar la clase derivada*. Localizar los efectos de las modificaciones de esta forma es una buena práctica de ingeniería de software.

```
1  // Fig. 11.14: EmpleadoPorComision.cpp
2  // Definiciones de las funciones miembro de la clase EmpleadoPorComision.
3  #include <iostream>
4  #include <stdexcept>
5  #include "EmpleadoPorComision.h" // definición de la clase EmpleadoPorComision
6  using namespace std;
7
8  // constructor
9  EmpleadoPorComision::EmpleadoPorComision(
10    const string &nombre, const string &apellido, const string &nss,
11    double ventas, double tarifa )
12    : primerNombre( nombre ), apellidoPaterno( apellido ), numeroSeguroSocial( nss )
13 {
14    establecerVentasBrutas( ventas ); // valida y almacena las ventas brutas
15    establecerTarifaComision( tarifa ); // valida y almacena la tarifa de comisión
16 } // fin del constructor de EmpleadoPorComision
17
18 // establece el primer nombre
19 void EmpleadoPorComision::obtenerPrimerNombre( const string &nombre )
20 {
21    primerNombre = nombre; // debe validar
22 } // fin de la función obtenerPrimerNombre
23
24 // devuelve el primer nombre
25 string EmpleadoPorComision::obtenerPrimerNombre() const
26 {
27    return primerNombre;
28 } // fin de la función obtenerPrimerNombre
29
30 // establece el apellido paterno
31 void EmpleadoPorComision::obtenerApellidoPaterno( const string &apellido )
32 {
33    apellidoPaterno = apellido; // debe validar
34 } // fin de la función obtenerApellidoPaterno
35
36 // devuelve el apellido paterno
37 string EmpleadoPorComision::obtenerApellidoPaterno() const
38 {
39    return apellidoPaterno;
40 } // fin de la función obtenerApellidoPaterno
41
42 // establece el número de seguro social
43 void EmpleadoPorComision::establecerNumeroSeguroSocial( const string &nss )
44 {
45    numeroSeguroSocial = nss; // debe validar
46 } // fin de la función establecerNumeroSeguroSocial
47
```

Fig. 11.14 | Archivo de implementación de la clase `EmpleadoPorComision`: la clase `EmpleadoPorComision` utiliza funciones miembro para manipular sus datos `private` (parte 1 de 2).

```
48  // devuelve el número de seguro social
49  string EmpleadoPorComision::obtenerNumeroSeguroSocial() const
50  {
51     return numeroSeguroSocial;
52  } // fin de la función obtenerNumeroSeguroSocial
53
54  // establece el monto de ventas brutas
55  void EmpleadoPorComision::establecerVentasBrutas( double ventas )
56  {
57     if ( ventas >= 0.0 )
58        ventasBrutas = ventas;
59     else
60        throw invalid_argument( "Las ventas brutas deben ser >= 0.0" );
61  } // fin de la función establecerVentasBrutas
62
63  // devuelve el monto de ventas brutas
64  double EmpleadoPorComision::obtenerVentasBrutas() const
65  {
66     return ventasBrutas;
67  } // fin de la función obtenerVentasBrutas
68
69  // establece la tarifa de comisión
70  void EmpleadoPorComision::establecerTarifaComision( double tarifa )
71  {
72     if ( tarifa > 0.0 && tarifa < 1.0 )
73        tarifaComision = tarifa;
74     else
75        throw invalid_argument( "La tarifa de comision debe ser > 0.0 y < 1.0" );
76  } // fin de la función establecerTarifaComision
77
78  // devuelve la tarifa de comisión
79  double EmpleadoPorComision::obtenerTarifaComision() const
80  {
81     return tarifaComision;
82  } // fin de la función obtenerTarifaComision
83
84  // calcula los ingresos
85  double EmpleadoPorComision::ingresos() const
86  {
87     return obtenerTarifaComision() * obtenerVentasBrutas();
88  } // fin de la función ingresos
89
90  // imprime el objeto EmpleadoPorComision
91  void EmpleadoPorComision::imprimir() const
92  {
93     cout << "empleado por comision: "
94        << obtenerPrimerNombre() << ' ' << obtenerApellidoPaterno()
95        << "\nnumero de seguro social: " << obtenerNumeroSeguroSocial()
96        << "\nventas brutas: " << obtenerVentasBrutas()
97        << "\ntarifa de comision: " << obtenerTarifaComision();
98  } // fin de la función imprimir
```

Fig. 11.14 | Archivo de implementación de la clase `EmpleadoPorComision`: la clase `EmpleadoPorComision` utiliza funciones miembro para manipular sus datos `private` (parte 2 de 2).

Tip de rendimiento 11.2

Usar una función miembro para acceder al valor de un miembro de datos puede ser un poco más lento que acceder a los datos directamente. Sin embargo, los compiladores optimizadores de hoy en día están diseñados cuidadosamente para realizar muchas optimizaciones de manera implícita (como poner en línea las llamadas a funciones establecer *y* obtener*). Los programadores deben escribir código que se adhiera a los principios de la ingeniería de software apropiada, y dejar las cuestiones de optimización al compilador. Una buena regla es, "no dudar del compilador".*

Modificaciones en las definiciones de las funciones miembro de la clase `EmpleadoBaseMasComision`

La clase `EmpleadoBaseMasComision` hereda las funciones miembro `public` de `EmpleadoPorComision` y puede acceder a los miembros `private` de la clase base a través de las funciones miembro heredadas. El encabezado de la clase permanece igual que en la figura 11.10. La clase tiene varias modificaciones en las implementaciones de sus funciones miembro (figura 11.15) que la diferencian de la versión anterior de la clase (figuras 11.10 y 11.11). Las funciones miembro `ingresos` (figura 11.15, líneas 34 a 37) e `imprimir` (líneas 40 a 48) invocan a la función miembro `obtenerSalarioBase` para obtener el valor del salario base, en vez de acceder directamente a `salarioBase`. Esto aísla a `ingresos` e `imprimir` de las potenciales modificaciones a la implementación del miembro de datos `salarioBase`. Por ejemplo, si decidimos renombrar el miembro de datos `salarioBase` o modificar su tipo, sólo las funciones miembro `establecerSalarioBase` y `obtenerSalarioBase` tendrán que modificarse.

```
1  // Fig. 11.15: EmpleadoBaseMasComision.cpp
2  // Definiciones de las funciones miembro de la clase EmpleadoBaseMasComision.
3  #include <iostream>
4  #include <stdexcept>
5  #include "EmpleadoBaseMasComision.h"
6  using namespace std;
7
8  // constructor
9  EmpleadoBaseMasComision::EmpleadoBaseMasComision(
10    const string &nombre, const string &apellido, const string &nss,
11    double ventas, double tarifa, double salario )
12    // llama explícitamente al constructor de la clase base
13    : EmpleadoPorComision( nombre, apellido, nss, ventas, tarifa )
14 {
15    establecerSalarioBase( salario ); // valida y almacena el salario base
16 } // fin del constructor de EmpleadoBaseMasComision
17
18 // establece el salario base
19 void EmpleadoBaseMasComision::establecerSalarioBase( double salario )
20 {
21    if ( salario >= 0.0 )
22       salarioBase = salario;
23    else
24       throw invalid_argument( "El salario debe ser >= 0.0" );
25 } // fin de la función establecerSalarioBase
26
27 // devuelve el salario base
28 double EmpleadoBaseMasComision::obtenerSalarioBase() const
29 {
```

Fig. 11.15 | Clase `EmpleadoBaseMasComision` que hereda de la clase `EmpleadoPorComision` pero no puede acceder directamente a los datos `private` de la clase (parte 1 de 2).

```
30    return salarioBase;
31 } // fin de la función obtenerSalarioBase
32
33 // calcula los ingresos
34 double EmpleadoBaseMasComision::ingresos() const
35 {
36    return obtenerSalarioBase() + EmpleadoPorComision::ingresos();
37 } // fin de la función ingresos
38
39 // imprime el objeto EmpleadoBaseMasComision
40 void EmpleadoBaseMasComision::imprimir() const
41 {
42    cout << "con salario base ";
43
44    // invoca a la función imprimir de EmpleadoPorComision
45    EmpleadoPorComision::imprimir();
46
47    cout << "\nsalario base: " << obtenerSalarioBase();
48 } // fin de la función imprimir
```

Fig. 11.15 | Clase `EmpleadoBaseMasComision` que hereda de la clase `EmpleadoPorComision` pero no puede acceder directamente a los datos `private` de la clase (parte 2 de 2).

La función miembro `ingresos` *de* `EmpleadoBaseMasComision`

La función `ingresos` de la clase `EmpleadoBaseMasComision` (figura 11.15, líneas 34 a 37) redefine la función miembro `ingresos` de la clase `EmpleadoPorComision` (figura 11.14, líneas 85 a 88) para calcular los ingresos de un empleado por comisión con salario base. La versión de `ingresos` de la clase `EmpleadoBaseMasComision` obtiene la porción de los ingresos del empleado únicamente con base en la comisión, llamando a la función `ingresos` de la clase base `EmpleadoPorComision` con la expresión `EmpleadoPorComision::ingresos()` (figura 11.15, línea 36). Después, la función `ingresos` de `EmpleadoBaseMasComision` suma el salario base a este valor para calcular los ingresos totales del empleado. Observe la sintaxis utilizada para invocar a una función miembro de la clase base redefinida desde una clase derivada: se coloca el nombre de la clase base y el operador de resolución de ámbito (`::`) antes del nombre de la función miembro de la clase base. Esta invocación a la función miembro es una buena práctica de ingeniería de software: en el capítulo 9 vimos que, si la función miembro de un objeto realiza las acciones que necesita otro objeto, debemos llamar a esa función miembro en vez de duplicar su cuerpo de código. Al hacer que la función `ingresos` de `EmpleadoBaseMasComision` invoque a la función `ingresos` de `EmpleadoPorComision` para calcular parte de los ingresos de un objeto `EmpleadoBaseMasComision`, evitamos duplicar el código y reducimos los problemas de mantenimiento de código.

Error común de programación 11.2

Cuando se redefine una función miembro de la clase base en una clase derivada, a menudo la versión de la clase derivada llama a la versión de la clase base para realizar trabajo adicional. Si no se utiliza el nombre de la clase base antes del operador `::` al hacer referencia a la función miembro de la clase base, se produce una recursividad infinita, ya que entonces la función miembro de la clase derivada se llamaría a sí misma.

La función miembro `imprimir` *de* `EmpleadoBaseMasComision`

De manera similar, la función `imprimir` de `EmpleadoBaseMasComision` (figura 11.15, líneas 40 a 48) redefine la función `imprimir` de `EmpleadoPorComision` (figura 11.14, líneas 91 a 98) para imprimir la información apropiada para un empleado por comisión con salario base. La nueva versión muestra

parte de la información de un objeto `EmpleadoBaseMasComision` (es decir, la cadena `"empleado por comision"` y los valores de los miembros de datos `private` de la clase `EmpleadoPorComision`) llamando a la función miembro `imprimir` de `EmpleadoPorComision` con el nombre calificado `EmpleadoPorComision::imprimir()` (figura 11.15, línea 45). Después, la función `imprimir` de `EmpleadoBaseMasComision` imprime el resto de la información de un objeto `EmpleadoBaseMasComision` (es decir, el valor del salario base de la clase `EmpleadoBaseMasComision`).

Prueba de la jerarquía de clases modificada

Una vez más, este ejemplo utiliza el programa de prueba de `EmpleadoBaseMasComision` de la figura 11.9 y produce el mismo resultado. Aunque cada clase "empleado por comisión con salario base" se comporta en forma idéntica, la versión en este ejemplo es la mejor diseñada. *Al usar la herencia y llamar a las funciones miembro que ocultan los datos y aseguran la consistencia, hemos construido en forma eficiente y eficaz una clase bien diseñada.*

Resumen de los ejemplos `EmpleadoPorComision-EmpleadoBaseMasComision`

En esta sección vimos un conjunto evolutivo de ejemplos que se diseñó cuidadosamente para enseñar las herramientas clave para la buena ingeniería de software mediante la herencia. El lector aprendió a crear una clase derivada mediante el uso de la herencia, a utilizar miembros `protected` de la clase base para permitir que una clase derivada acceda a los miembros de datos heredados de la clase base, y cómo redefinir las funciones de la clase base para proporcionar versiones que sean más apropiadas para los objetos de la clase derivada. Además, aprendió a aplicar las técnicas de ingeniería de software del capítulo 9 y de este capítulo, para crear clases que sean fáciles de mantener, modificar y depurar.

11.4 Constructores y destructores en clases derivadas

Como explicamos en la sección anterior, al instanciar un objeto de una clase derivada se inicia una *cadena* de llamadas a constructores, en donde el constructor de la clase derivada, antes de realizar sus propias tareas, invoca al constructor de su clase base directa, ya sea de manera explícita (a través de un inicializador de miembros de la clase base) o implícita (llamando al constructor predeterminado de la clase base). De manera similar, si la clase base se deriva de otra clase, se requiere el constructor de la clase base para invocar al constructor de la siguiente clase hacia arriba de la jerarquía, y así en lo sucesivo. El último constructor llamado en esta cadena es el constructor de la clase en la base de la jerarquía, cuyo cuerpo en realidad termina *primero* de ejecutarse. El cuerpo del constructor de la clase más derivada original termina de ejecutarse al último. El constructor de cada clase base inicializa los miembros de datos de la clase base que hereda el objeto de la clase derivada. En la jerarquía `EmpleadoPorComision/EmpleadoBaseMasComision` que hemos estado estudiando, cuando un programa crea un objeto de la clase `EmpleadoBaseMasComision`, se hace una llamada al constructor de `EmpleadoPorComision`. Como la clase `EmpleadoPorComision` se encuentra en la base de la jerarquía, se ejecuta su constructor y se inicializan los datos miembro `private` de `EmpleadoPorComision` que forman parte del objeto `EmpleadoBaseMasComision`. Cuando el constructor de `EmpleadoPorComision` completa su ejecución, devuelve el control al constructor de `EmpleadoBaseMasComision`, el cual inicializa al objeto `salarioBase` de `EmpleadoBaseMasComision`.

Observación de Ingeniería de Software 11.6

Cuando un programa crea un objeto de una clase derivada, el constructor de la clase derivada llama de inmediato al constructor de la clase base, se ejecuta el cuerpo del constructor de la clase base y después se ejecutan los inicializadores de miembros *de la clase derivada; al último, se ejecuta el cuerpo del constructor de la clase derivada. Este proceso avanza en cascada hacia arriba por la jerarquía, si contiene más de dos niveles.*

Cuando se destruye un objeto de una clase derivada, el programa llama al destructor de ese objeto. Esto empieza una cadena (o cascada) de llamadas a destructores en las que el destructor de la clase derivada y los destructores de las clases base directas e indirectas y los miembros de las clases se ejecutan en orden *inverso* al orden en el que se ejecutaron los constructores. Cuando se hace una llamada al destructor de un objeto de una clase derivada, el destructor realiza su tarea y después invoca al destructor de la siguiente clase base hacia arriba por la jerarquía. Este proceso se repite hasta que se hace una llamada al destructor de la clase base final en la parte superior de la jerarquía. Después, el objeto se elimina de la memoria.

Observación de Ingeniería de Software 11.7

Suponga que creamos un objeto de una clase derivada, en donde tanto la clase base como la clase derivada contienen (a través de la composición) objetos de otras clases. Cuando se crea un objeto de esa clase derivada, primero se ejecutan los constructores para los objetos miembro de la clase base, después se ejecuta el cuerpo del constructor de la clase base, luego se ejecutan los objetos miembro de la clase derivada, y después se ejecuta el cuerpo del constructor de la clase derivada. Los destructores para los objetos de una clase derivada se llaman en orden inverso al orden en el que se llamaron sus correspondientes constructores.

Las clases derivadas *no* heredan los constructores, destructores ni operadores de asignación sobrecargados (capítulo 10) de la clase base. Sin embargo, los constructores, destructores y operadores de asignación sobrecargados de la clase derivada pueden llamar a todas las versiones de la clase base.

C++11: heredar los constructores de la clase base

Algunas veces, los constructores de la clase derivada simplemente imitan a los constructores de la clase base. Una característica de conveniencia para C++11 que se solicitaba con frecuencia era la habilidad de *heredar* los constructores de una clase base. Ahora podemos hacer esto si incluimos de manera *explícita* una declaración `using` de la forma

```
using ClaseBase::ClaseBase;
```

en cualquier parte de la definición de la clase derivada. En la anterior declaración, *ClaseBase* es el nombre de la clase base. Con unas cuantas excepciones (que se indican a continuación), por cada constructor en la clase base, el compilador genera un constructor de clase derivada que llama al correspondiente constructor de la clase base. Los constructores generados realizan sólo la *inicialización predeterminada* para los miembros de datos adicionales de la clase derivada. Cuando se heredan constructores:

- De manera predeterminada, cada constructor heredado tiene el *mismo* nivel de acceso (`public`, `protected` o `private`) que el constructor de su clase base correspondiente.
- Los constructores predeterminado, de copia y de movimiento *no* se heredan.
- Si se *elimina* el constructor en la clase base al colocar `= delete` en su prototipo, el constructor correspondiente en la clase derivada *también* se elimina.
- Si la clase derivada no define constructores de manera *explícita*, el compilador genera un constructor predeterminado en la clase derivada; *incluso* aunque herede otros constructores de su clase base.
- Si un constructor que usted defina de manera *explícita* en una clase derivada tiene la *misma* lista de parámetros que un constructor de la clase base, entonces *no* se hereda el constructor de la clase base.
- Los argumentos predeterminados del constructor de una clase base *no* se heredan. En vez de ello, el compilador genera *constructores sobrecargados* en la clase derivada. Por ejemplo, si la clase base declara el constructor

```
ClaseBase( int = 0, double = 0.0 );
```

el compilador genera los siguientes *dos* constructores de clase derivada *sin* argumentos predeterminados:

```
ClaseDerivada( int );
ClaseDerivada( int, double );
```

Cada uno de éstos llama al constructor *ClaseBase* que especifica los argumentos predeterminados.

11.5 Herencia public, protected y private

Al derivar una clase de una clase base, ésta se puede heredar a través de la herencia `public`, `protected` o `private`. Por lo general, en este libro usamos herencia `public`. El uso de la herencia `protected` es poco común. En el capítulo 19 (en inglés en el sitio web) se demuestra el uso de la herencia `private` como una alternativa para la composición. La figura 11.16 sintetiza, para cada tipo de herencia, la accesibilidad de los miembros de la clase base en una clase derivada. La primera columna contiene los especificadores de acceso de la clase base.

Especificador de acceso a miembros de la clase base	Tipo de herencia		
	Herencia `public`	Herencia `protected`	Herencia `private`
`public`	`public` en la clase derivada. Puede ser utilizado directamente por las funciones miembro, las funciones `friend` y las funciones no miembro.	`protected` en la clase derivada. Puede ser utilizado directamente por las funciones miembro y las funciones `friend`.	`private` en la clase derivada. Puede ser utilizado directamente por las funciones miembro y las funciones `friend`.
`protected`	`protected` en la clase derivada. Puede ser utilizado directamente por las funciones miembro y las funciones `friend`.	`protected` en la clase derivada. Puede ser utilizado directamente por las funciones miembro y las funciones `friend`.	`private` en la clase derivada. Puede ser utilizado directamente por las funciones miembro y las funciones `friend`.
`private`	Oculto en la clase derivada. Puede ser utilizado por las funciones miembro y las funciones `friend` a través de las funciones miembro `public` o `protected` de la clase base.	Oculto en la clase derivada. Puede ser utilizado por las funciones miembro y las funciones `friend` a través de funciones miembro `public` o `protected` de la clase base.	Oculto en la clase derivada. Puede ser utilizado por funciones miembro y funciones `friend` a través de funciones miembro `public` o `protected` de la clase base.

Fig. 11.16 | Resumen de la accesibilidad de los miembros de una clase base en una clase derivada.

Al derivar una clase con herencia `public`, los miembros `public` de la clase base se convierten en miembros `public` de la clase derivada, y los miembros `protected` de la clase base se convierten en miembros `protected` de la clase derivada. Los miembros `private` de la clase base *nunca* pueden utilizarse directamente desde una clase derivada, pero se puede acceder a ellos a través de llamadas a los miembros `public` y `protected` de la clase base.

Al derivar de una clase con herencia `protected`, los miembros `public` y `protected` de la clase base se convierten en miembros `protected` de la clase derivada. Al derivar de una clase con herencia `private`, los miembros `public` y `protected` de la clase base se convierten en miembros `private` (por ejemplo, las funciones se convierten en funciones utilitarias) de la clase derivada. Las relaciones de herencia `private` y `protected` no son relaciones del tipo *es-un*.

11.6 Ingeniería de software mediante la herencia

Algunas veces es difícil para los estudiantes apreciar el alcance de los problemas a los que se enfrentan los diseñadores que trabajan en proyectos de software de gran escala en la industria. La gente experimentada con dichos proyectos dice que la reutilización efectiva de software mejora el proceso de desarrollo de software. La programación orientada a objetos facilita la reutilización de software, con lo cual se reducen los tiempos de desarrollo y se mejora la calidad del software.

Cuando usamos la herencia para crear una nueva clase a partir de una existente, la nueva clase hereda los miembros de datos y las funciones miembro de la clase existente, como se describe en la figura 11.16. Podemos personalizar la nueva clase para que cumpla con nuestras necesidades al redefinir los miembros de la clase base e incluir miembros adicionales. El programador de la clase derivada hace esto en C++ *sin* acceder al código fuente de la clase base (la clase derivada debe ser capaz de *ligarse* con el código objeto de la clase base). Esta poderosa herramienta es atractiva para los desarrolladores de software. Ellos pueden desarrollar clases propietarias para venta o licencia, y poner esas clases a disposición de los usuarios en formato de código objeto. Después, los usuarios pueden derivar con rapidez nuevas clases a partir de estas bibliotecas de clases y sin acceder al código fuente propietario. Los desarrolladores de software necesitan suministrar los encabezados junto con el código objeto.

La disponibilidad de bibliotecas de clases extensas y útiles produce los máximos beneficios de la reutilización de software a través de la herencia. Las Bibliotecas estándar de C++ tienden a ser de propósito general y tienen su alcance limitado. Hay un compromiso a nivel mundial en relación con el desarrollo de las bibliotecas de clases para una amplia variedad de áreas de aplicaciones.

Observación de Ingeniería de Software 11.8

En la etapa de diseño de un sistema orientado a objetos, a menudo el diseñador determina que ciertas clases están estrechamente relacionadas. El diseñador debe "factorizar" los atributos y comportamientos comunes, colocarlos en una clase base, y después usar la herencia para formar clases derivadas.

Observación de Ingeniería de Software 11.9

La creación de una clase derivada no afecta al código fuente de su clase base. La herencia preserva la integridad de una clase base.

11.7 Repaso

En este capítulo se introdujo la herencia: la habilidad de crear una clase al absorber los miembros de datos y las funciones miembro de una clase existente, y completarlos con nuevas capacidades. A través de una serie de ejemplos mediante el uso de una jerarquía de herencia, el lector aprendió las nociones de las clases base y las clases derivadas, y utilizó la herencia `public` para crear una clase derivada que hereda miembros de una clase base. El capítulo introdujo el especificador de acceso `protected`; las funciones miembro de la clase derivada pueden acceder a los miembros `protected` de la clase base. El lector aprendió a acceder a los miembros redefinidos de la clase base, calificando sus nombres con el nombre de la clase base y el operador de resolución de ámbito (`::`). También vio el orden en el que se llaman los

constructores y destructores para los objetos de clases que forman parte de una jerarquía de herencia. Por último, explicamos los tres tipos de herencia (`public`, `protected` y `private`) y la accesibilidad de los miembros de la clase base en una clase derivada, al usar cada uno de los tipos de herencia.

En el capítulo 12, Programación orientada a objetos: polimorfismo, continuaremos con nuestra discusión sobre la herencia al introducir el polimorfismo: un concepto orientado a objetos que nos permite escribir programas que manejen, de una forma más general, los objetos de una amplia variedad de clases relacionadas por la herencia. Después de estudiar el capítulo 12, el lector estará familiarizado con las clases, objetos, encapsulación, herencia y polimorfismo: los conceptos esenciales de la programación orientada a objetos.

Resumen

Sección 11.1 Introducción

- La reutilización de software reduce el tiempo y el costo del desarrollo de programas.
- La herencia (pág. 483) es una forma de reutilización de software en la que el programador crea una clase que absorbe las herramientas de una clase existente, y luego las personaliza o mejora. La clase existente se llama clase base (pág. 483), y la nueva clase se conoce como clase derivada (pág. 483).
- Todo objeto de una clase derivada es también un objeto de la clase base de esa clase. Sin embargo, un objeto de la clase base no es un objeto de las clases derivadas de esa clase.
- La relación *es-un* (pág. 483) representa la herencia. En una relación *es-un*, un objeto de una clase derivada también puede tratarse como un objeto de su clase base.
- La relación *tiene-un* (pág. 483) representa la composición: un objeto contiene uno o más objetos de otras clases como miembros, pero no divulga su comportamiento directamente en su interfaz.

Sección 11.2 Clases base y clases derivadas

- Una clase base directa (pág. 485) es aquella de la que una clase derivada hereda de forma explícita. Una clase base indirecta (pág. 485) se hereda de dos o más niveles hacia arriba en la jerarquía de clases (pág. 484).
- Con la herencia simple (pág. 485), una clase se deriva de una clase base. Con la herencia múltiple (pág. 485), una clase hereda de varias clases base (posiblemente no relacionadas).
- Una clase derivada representa a un grupo más especializado de objetos.
- Las relaciones de herencia forman jerarquías de clases.
- Es posible tratar a los objetos de la clase base y a los objetos de la clase derivada de manera similar; las características comunes compartidas entre los tipos de los objetos se expresan en los miembros de datos y en las funciones miembro de la clase base.

Sección 11.4 Constructores y destructores en clases derivadas

- Cuando se instancia un objeto de una clase derivada, el constructor de la clase base se llama de inmediato para inicializar los miembros de datos de la clase base en el objeto de la clase derivada, y después el constructor de la clase derivada inicializa los miembros de datos adicionales de la clase derivada.
- Cuando se destruye un objeto de una clase derivada, los destructores se llaman en el orden inverso al de los constructores; primero se llama el destructor de la clase derivada, y después se llama el destructor de la clase base.
- Los miembros `public` de una clase base pueden utilizarse en cualquier parte en donde el programa tenga un manejador a un objeto de esa clase base, o a un objeto de una de las clases derivadas de la clase base; o, cuando se utilice el operador de resolución de ámbito, en cualquier parte en donde el nombre de la clase esté dentro del alcance.
- Los miembros `private` de una clase base pueden utilizarse sólo dentro de la clase base o desde sus funciones `friend`.
- Los miembros `protected` de una clase base pueden ser utilizados por los miembros y funciones `friend` de esa clase base, y por los miembros y funciones `friend` de las clases que se deriven de esa clase base.

- En C++11, una clase derivada puede heredar constructores de su clase base al incluir en cualquier parte de la definición de la clase derivada una declaración `using` de la forma

  `using` *ClaseBase*`::`*ClaseBase*;

Sección 11.5 Herencia `public`, `protected` *y* `private`

- Al declarar los datos miembro `private`, proporcionando a la vez funciones miembro no `private` para manipular y realizar la comprobación de validez en estos datos, se hace cumplir la buena ingeniería de software.
- Al derivar una clase, la clase base se puede declarar como `public`, `protected` o `private`.
- Al derivar una clase con herencia `public` (pág. 485), los miembros `public` de la clase base se convierten en miembros `public` de la clase derivada, y los miembros `protected` de la clase base se convierten en miembros `protected` de la clase derivada.
- Al derivar una clase con herencia `protected` (pág. 485), los miembros `public` y `protected` de la clase base se convierten en miembros `protected` de la clase derivada.
- Al derivar una clase con herencia `private` (pág. 485) los miembros `public` y `protected` de la clase base se convierten en miembros `private` de la clase derivada.

Ejercicios de autoevaluación

11.1 Complete los siguientes enunciados:

a) _____ es una forma de reutilización de software, en la que nuevas clases absorben los datos y comportamientos de las clases existentes, y completan estas clases con nuevas capacidades.
b) Los miembros _____ de una clase base pueden utilizarse sólo en la definición de la clase base, o en las definiciones de la clase derivada y en las funciones `friend` de la clase base y de sus clases derivadas.
c) En una relación _____, un objeto de una clase derivada se puede tratar también como un objeto de su clase base.
d) En una relación _____, el objeto de una clase tiene uno o más objetos de otras clases como miembros.
e) En la herencia simple, una clase existe en una relación _____ con sus clases derivadas.
f) Los miembros _____ de una clase base se pueden utilizar dentro de esa clase base, y en cualquier parte en donde el programa tenga un manejador a un objeto de esa clase, o a un objeto de una de sus clases derivadas.
g) Los miembros de acceso `protected` de una clase base tienen un nivel de protección entre los de acceso `public` y _____.
h) C++ cuenta con _____, la cual permite a una clase derivada heredar de muchas clases base, incluso aunque las clases base no estén relacionadas.
i) Cuando se instancia un objeto de una clase derivada, el _____ de la clase base se llama de manera implícita o explícita para realizar la inicialización necesaria de los miembros de datos de la clase base en el objeto de la clase derivada.
j) Al derivar una clase con herencia `public`, los miembros `public` de la clase base se convierten en miembros _____ de la clase derivada, y los miembros `protected` de la clase base se convierten en miembros _____ de la clase derivada.
k) Al derivar una clase de una clase base con herencia `protected`, los miembros `public` de la clase base se convierten en miembros _____ de la clase derivada, y los miembros `protected` de la clase base se convierten en miembros _____ de la clase derivada.

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

a) Los constructores de la clase base no son heredados por las clases derivadas.
b) Una relación *tiene-un* se implementa mediante la herencia.
c) Una clase `Auto` tiene una relación *es-un* con las clases `VolanteDireccion` y `Frenos`.
d) La herencia fomenta la reutilización de software comprobado, de alta calidad.
e) Cuando se destruye un objeto de una clase derivada, los destructores se llaman en el orden inverso al de los constructores.

Respuestas a los ejercicios de autoevaluación

11.1 a) Herencia. b) `protected`. c) *es-un* o de herencia (por herencia `public`). d) *tiene-un*, o composición, o agregación. e) jerárquica. f) `public`. g) `private`. h) herencia múltiple. i) constructor. j) `public`, `protected`. k) `protected`, `protected`.

11.2 a) Verdadero. b) Falso. Una relación *tiene-un* se implementa mediante la composición. Una relación *es-un* se implementa mediante la herencia. c) Falso. Éste es un ejemplo de una relación *tiene-un*. La clase `Auto` tiene una relación *es-un* con la clase `Vehiculo`. d) Verdadero. e) Verdadero.

Ejercicios

11.3 ***(Composiciones como alternativa para la herencia)*** Muchos programas escritos con herencia podrían escribirse mediante la composición, y viceversa. Vuelva a escribir la clase `EmpleadoBaseMasComision` de la jerarquía `EmpleadoPorComision-EmpleadoBaseMasComision` para usar la composición en vez de la herencia. Una vez que haga esto, valore los méritos relativos de los dos métodos para diseñar las clases `EmpleadoPorComision` y `EmpleadoBaseMasComision`, así como también para los programas orientados a objetos en general. ¿Cuál método es más natural? ¿Por qué?

11.4 ***(Ventaja de la herencia)*** Describa las formas en las que la herencia promueve la reutilización de software, ahorra tiempo durante el desarrollo de los programas y ayuda a prevenir errores.

11.5 ***(Comparación entre clases base protegidas y privadas)*** Algunos programadores prefieren no utilizar el acceso `protected`, ya que creen que quebranta el encapsulamiento de la clase base. Hable sobre los méritos relativos de utilizar el acceso `protected` en comparación con el acceso `private` en las clases base.

11.6 ***(Jerarquía de herencia de estudiantes)*** Dibuje una jerarquía de herencia para los estudiantes en una universidad, de manera similar a la jerarquía que se muestra en la figura 11.2. Use a `Estudiante` como la superclase de la jerarquía, y después incluya las clases `EstudianteNoGraduado` y `EstudianteGraduado`, que se deriven de `Estudiante`. Siga extendiendo la jerarquía con el mayor número de niveles que sea posible. Por ejemplo, `EstudiantePrimerAnio`, `EstudianteSegundoAnio`, `EstudianteTercerAnio` y `EstudianteCuartoAnio` podrían derivarse de `EstudianteNoGraduado`, y `EstudianteDoctorado` y `EstudianteMaestria` podrían derivarse de `EstudianteGraduado`. Después de dibujar la jerarquía, hable sobre las relaciones que existen entre las clases. [*Nota*: no necesita escribir código para este ejercicio].

11.7 ***(Jerarquía de figuras más completa)*** El mundo de las figuras es más extenso que las figuras incluidas en la jerarquía de herencia de la figura 11.3. Anote todas las figuras en las que pueda pensar (tanto bidimensionales como tridimensionales) e intégrelas en una jerarquía `Figura` más completa, con todos los niveles que sea posible. Su jerarquía debe tener la clase base `Figura`, de la que se deriven las clases `FiguraBidimensional` y `FiguraTridimensional`. [*Nota:* no necesita escribir código para este ejercicio]. Utilizaremos esta jerarquía en los ejercicios del capítulo 12 para procesar un conjunto de figuras distintas como objetos de la clase base `Figura`. (Esta técnica, conocida como polimorfismo, es el tema del capítulo 12).

11.8 ***(Jerarquía de herencia de cuadriláteros)*** Dibuje una jerarquía de herencia para las clases `Cuadrilatero`, `Trapezoide`, `Paralelogramo`, `Rectangulo` y `Cuadrado`. Use `Cuadrilatero` como la clase base de la jerarquía. Agregue todos los niveles que sea posible a la jerarquía.

11.9 (***Jerarquía de herencia `Paquete`***) Los servicios de entrega de paquetes como FedEx®, DHL® y UPS® ofrecen una variedad de opciones de envío distintas, cada una con los costos específicos asociados. Cree una jerarquía de herencia para representar varios tipos de paquetes. Use `Paquete` como la clase base de la jerarquía; después incluya las clases `PaqueteDosDias` y `PaqueteNocturno` que se deriven de `Paquete`. La clase base `Paquete` debe incluir miembros de datos que representen el nombre, dirección, ciudad, estado y código postal para el emisor y el destinatario del paquete, además de los datos miembro que almacenan el peso (en kilogramos) y el costo por kilogramo para enviar el paquete. El constructor de `Paquete` debe inicializar estos miembros de datos. Asegúrese

que el peso y costo por kilogramo contengan valores positivos. `Paquete` debe proporcionar una función miembro `public` llamada `calcularCosto` que devuelva un valor `double` para indicar el costo asociado con el envío del paquete. La función `calcularCosto` de `Paquete` debe determinar el costo al multiplicar el peso por el costo por kilogramo. La clase derivada `PaqueteDosDias` debe heredar la funcionalidad de la clase base `Paquete`, pero también debe incluir un miembro de datos que represente una cuota fija que cobre la compañía de envío por el servicio de entrega de dos días. El constructor de `PaqueteDosDias` debe recibir un valor para inicializar este miembro de datos. `PaqueteDosDias` debe redefinir la función miembro `calcularCosto`, de manera que calcule el costo sumando la cuota fija al costo basado en el peso, calculado por la función `calcularCosto` de la clase base `Paquete`. La clase `PaqueteNocturno` debe heredar directamente de la clase `Paquete` y debe contener un miembro de datos adicional que represente una cuota adicional por cada kilogramo que se cobre por el servicio de entrega nocturna. `PaqueteNocturno` debe redefinir la función miembro `calcularCosto`, de manera que sume la cuota adicional por kilogramo al costo estándar por kilogramo, antes de calcular el costo de envío. Escriba un programa de prueba para crear objetos de cada tipo de `Paquete` y evaluar la función miembro `calcularCosto`.

11.10 ***(Jerarquía de herencia `Cuenta`)*** Cree una jerarquía de herencia que podría usar un banco para representar las cuentas bancarias de los clientes. Todos los clientes en este banco pueden depositar (es decir, abonar) dinero en sus cuentas, y retirar (es decir, cargar) dinero de ellas. También existen tipos más específicos de cuentas. Por ejemplo, las cuentas de ahorro obtienen intereses sobre el dinero que contienen. Por otro lado, las cuentas de cheques cobran una cuota por transacción (es decir, abono o cargo).

Cree una jerarquía de herencia que contenga la clase base `Cuenta`, junto con las clases derivadas `CuentaAhorros` y `CuentaCheques` que hereden de la clase `Cuenta`. La clase base `Cuenta` debe incluir un miembro de datos de tipo `double` para representar el saldo de la cuenta. La clase debe proporcionar un constructor que reciba un saldo inicial y lo utilice para inicializar el miembro de datos. El constructor debe validar el saldo inicial, para asegurar que sea mayor o igual a `0.0`. De no ser así, el saldo debe establecerse en `0.0` y el constructor debe mostrar un mensaje de error, indicando que el saldo inicial es inválido. La clase debe proporcionar tres funciones miembro. La función miembro `abonar` debe sumar un monto al saldo actual. La función miembro `cargar` debe retirar dinero de la `Cuenta` y asegurar que el monto a cargar no exceda el saldo de la `Cuenta`. Si lo hace, el saldo debe permanecer sin cambio y la función debe imprimir el mensaje `"El monto a cargar excedio el saldo de la cuenta"`. La función miembro `obtenerSaldo` debe devolver el saldo actual.

La clase derivada `CuentaAhorros` debe heredar la funcionalidad de una `Cuenta`, pero también debe incluir un miembro de datos de tipo `double` que indique la tasa de interés (porcentaje) asignada a la `Cuenta`. El constructor de `CuentaAhorros` debe recibir el saldo inicial, así como un valor inicial para la tasa de interés de `CuentaAhorros`. `CuentaAhorros` debe proporcionar una función miembro `public` llamada `calcularInteres`, que devuelva un valor `double` que indique el monto de interés obtenido por una cuenta. La función miembro `calcularInteres` debe determinar este monto, multiplicando la tasa de interés por el saldo de la cuenta. [*Nota:* `CuentaAhorros` debe heredar las funciones miembro `abonar` y `cargar` como están, sin redefinirlas].

La clase derivada `CuentaCheques` debe heredar de la clase base `Cuenta` e incluir un miembro de datos adicional de tipo `double`, que represente la cuota que se cobra por transacción. El constructor de `CuentaCheques` debe recibir el saldo inicial, así como un parámetro que indique el monto de la cuota. La clase `CuentaCheques` debe redefinir las funciones miembro `abonar` y `cargar` de manera que resten la cuota del saldo de la cuenta, cada vez que se realice una de esas transacciones con éxito. Las versiones de `CuentaCheques` de estas funciones deben invocar la versión de la clase base `Cuenta` para realizar las actualizaciones en el saldo de una cuenta. La función `cargar` de `CuentaCheques` debe cobrar una cuota sólo si realmente se retiró dinero (es decir, que el monto a cargar no exceda el saldo de la cuenta). [*Sugerencia:* defina la función `cargar` de `Cuenta` de manera que devuelva un valor `bool` que indique si se retiró dinero. Después use el valor de retorno para determinar si se debe cobrar una cuota].

Después de definir las clases en esta jerarquía, escriba un programa para crear objetos de cada clase y evaluar sus funciones miembro. Agregue interés al objeto `CuentaAhorros`, primero invocando a su función `calcularInteres` y después pasando el monto de interés devuelto a la función `abonar` del objeto.

Programación orientada a objetos: polimorfismo

12

El silencio, a menudo de pura inocencia,
persuade cuando el habla falla.
—William Shakespeare

Las proposiciones generales no deciden casos concretos.
—Oliver Wendell Holmes

Un filósofo de imponente estatura no piensa en un vacío. Incluso sus ideas más abstractas son, en cierta medida, condicionadas por lo que se conoce o no en el tiempo en que vive.
—Alfred North Whitehead

Objetivos

En este capítulo aprenderá a:

- Reconocer que el polimorfismo hace la programación más conveniente y los sistemas más extensibles.
- Distinguir entre clases abstractas y concretas, y a crear clases abstractas.
- Usar la información de tipos en tiempo de ejecución (RTTI).
- Distinguir cómo implementa C++ las funciones `virtual` y la vinculación dinámica.
- Reconocer cómo es que los destructores `virtual` aseguran que se ejecuten todos los destructores apropiados en un objeto.

Plan general

12.1 Introducción

Ahora continuaremos nuestro estudio de la POO al explicar y demostrar el **polimorfismo** con las jerarquías de herencia. El polimorfismo nos permite "programar en *general*" en vez de "programar de manera *específica*". En especial, el polimorfismo nos permite escribir programas que procesen objetos de clases que formen parte de la *misma* jerarquía de clases, como si todos fueran objetos de la clase base de la jerarquía. Como veremos en breve, el polimorfismo trabaja con los *manejadores de apuntadores* de clase base y *manejadores de referencias* de clase base, pero *no* con los manejadores de nombres.

Implementación para extensibilidad

Con el polimorfismo podemos diseñar e implementar sistemas que puedan *extenderse* con facilidad; pueden agregarse nuevas clases con sólo modificar un poco (o nada) las porciones generales del programa, siempre y cuando las nuevas clases sean parte de la jerarquía de herencia que el programa procesa en forma genérica. Las únicas partes de un programa que deben alterarse para dar cabida a las nuevas clases son las que requieren un conocimiento directo de las nuevas clases que el programador va a agregar a la jerarquía. Por ejemplo, si creamos la clase `Tortuga` que hereda de la clase `Animal` (que podría responder a un mensaje `mover` caminando una pulgada), necesitamos escribir sólo la clase `Tortuga` y la parte de la simulación que crea una instancia de un objeto `Tortuga`. Las porciones de la simulación que procesan a cada `Animal` en forma genérica pueden permanecer iguales.

Discusión opcional sobre el polimorfismo "detrás de las cámaras"

Una característica clave de este capítulo es su discusión detallada (opcional) acerca del polimorfismo, las funciones `virtual` y la vinculación dinámica "detrás de las cámaras", la cual utiliza un diagrama detallado para explicar cómo se puede implementar el polimorfismo en C++.

12.2 Introducción al polimorfismo: videojuego polimórfico

Suponga que vamos a diseñar un videojuego que manipule objetos de muchos tipos *distintos*, incluyendo objetos de las clases `Marciano`, `Venusino`, `Plutoniano`, `NaveEspacial` y `RayoLaser`. Imagine que cada clase hereda de la clase base común llamada `ObjetoEspacial`, la cual contiene la función miembro `dibujar`. Cada clase derivada implementa a esta función de una manera apropiada para esa clase. Un programa administrador de la pantalla mantiene un contenedor (por ejemplo, un `vector`) que contiene *apuntadores* `ObjetoEspacial` a objetos de las distintas clases. Para actualizar la pantalla, el administrador de pantalla envía el *mismo* mensaje a cada objeto; a saber, `dibujar`. Cada tipo de objeto responde de manera única. Por ejemplo, un objeto `Marciano` podría dibujarse a sí mismo en color rojo, con el número apropiado de antenas, un objeto `NaveEspacial` podría dibujarse a sí mismo como un platillo volador de color plateado y un objeto `RayoLaser` podría dibujarse a sí mismo como un rayo color rojo brillante a lo largo de la pantalla. El *mismo* mensaje (en este caso, `dibujar`) que se envía a una *variedad* de objetos tiene *muchas formas* de resultados; de aquí que se use el término polimorfismo.

Un administrador de pantalla polimórfico facilita el proceso de agregar nuevas clases a un sistema, con el mínimo de modificaciones a su código. Suponga que deseamos agregar objetos de la clase `Mercuriano` a nuestro videojuego. Para ello, debemos crear una clase `Mercuriano` que herede de `ObjetoEspacial`, pero proporcione su propia definición de la función miembro `dibujar`. Después, cuando aparezcan *apuntadores* a objetos de la clase `Mercuriano` en el contenedor, no será necesario modificar el código para el administrador de pantalla. Éste invocará a la función miembro `dibujar` en *cada* objeto en el contenedor, *sin importar* el tipo del objeto, por lo que los nuevos objetos `Mercuriano` simplemente "se integran en forma automática". Así, sin modificar el sistema (más que para crear e incluir las mismas clases), los programadores pueden utilizar el polimorfismo para acomodar clases adicionales, incluyendo las que *no se hayan considerado* a la hora de crear el sistema.

Observación de Ingeniería de Software 12.1

El polimorfismo nos permite tratar con las generalidades *y dejar que el entorno en tiempo de ejecución se encargue de los detalles específicos. Los programadores pueden ordenar a una variedad de objetos que se comporten en formas apropiadas para ellos,* sin necesidad de conocer sus tipos*, siempre y cuando éstos pertenezcan a la misma jerarquía de herencia y se utilicen a través de un apuntador o una referencia común de la clase base.*

Observación de Ingeniería de Software 12.2

El polimorfismo promueve la extensibilidad*: el software escrito para invocar el comportamiento polimórfico se escribe* de manera independiente *de los tipos de los objetos a los cuales se envían los mensajes. Así, se pueden incorporar en un sistema nuevos tipos de objetos que pueden responder a los mensajes* existentes*, sin necesidad de modificar el sistema base. Sólo el código cliente que crea instancias de los nuevos objetos debe modificarse para dar cabida a los nuevos tipos.*

12.3 Relaciones entre los objetos en una jerarquía de herencia

En la sección 11.3 se creó una jerarquía de clases de empleados, en la cual la clase `EmpleadoBaseMasComision` heredó de la clase `EmpleadoPorComision`. Los ejemplos del capítulo 11 manipularon objetos

`EmpleadoPorComision` y objetos `EmpleadoBaseMasComision` mediante el uso de sus nombres para invocar a sus funciones miembro. Ahora examinaremos las relaciones entre las clases en una jerarquía con más detalle. Las siguientes secciones presentan una serie de ejemplos que demuestran cómo se pueden orientar *apuntadores* de la clase base y de la clase derivada a objetos de la clase base y de la clase derivada, y cómo se pueden utilizar esos apuntadores para invocar a funciones miembro que manipulen a esos objetos.

- En la sección 12.3.1 asignaremos la dirección de un objeto de la clase derivada a un apuntador de la clase base, y después mostraremos que al invocar una función a través del apuntador de la clase base se invoca a la *funcionalidad de la clase base* en el objeto de la clase derivada; es decir, el *tipo del manejador determina cuál función se llama*.
- En la sección 12.3.2 asignaremos la dirección de un objeto de la clase base a un apuntador de la clase derivada, lo cual produce un error de compilación. Hablaremos sobre el mensaje de error e investigaremos por qué el compilador *no* permite dicha asignación.
- En la sección 12.3.3 asignaremos la dirección de un objeto de la clase derivada a un apuntador de la clase base, y después examinaremos cómo puede usarse el apuntador de la clase base para invocar sólo la funcionalidad de la clase base; *al tratar de invocar funciones miembro de la clase derivada a través del apuntador de la clase base, se producen errores de compilación*.
- Finalmente, en la sección 12.3.4 demostraremos cómo obtener un comportamiento polimórfico de los apuntadores de la clase base dirigidos a objetos de la clase derivada. Introduciremos las funciones `virtual` y el polimorfismo, al declarar una función de la clase base como `virtual`. Después asignaremos la dirección de un objeto de la clase derivada al apuntador de la clase base y utilizaremos ese apuntador para invocar la funcionalidad de la clase derivada; *precisamente la capacidad que necesitamos para lograr el comportamiento polimórfico*.

Un concepto clave en estos ejemplos es demostrar que con la herencia `public` *un objeto de una clase derivada puede tratarse como un objeto de su clase base*. Esto permite varias manipulaciones interesantes. Por ejemplo, un programa puede crear un arreglo de apuntadores de la clase base que apunten a objetos de muchos tipos de clases derivadas. A pesar del hecho de que los objetos de las clases derivadas son de *distintos tipos*, el compilador lo permite debido a que cada objeto de una clase derivada *es-un* objeto de su clase base. Sin embargo, *no podemos tratar a un objeto de la clase base como un objeto de una de sus clases derivadas*. Por ejemplo, un `EmpleadoPorComision` no es un `EmpleadoBaseMasComision` en la jerarquía definida en el capítulo 11; un `EmpleadoPorComision` *no* tiene un miembro de datos `salarioBase` y *no* tiene las funciones miembro `establecerSalarioBase` y `obtenerSalarioBase`. La relación *es-un* se aplica sólo de una *clase derivada* a sus *clases base directa e indirectas*.

12.3.1 Invocación de funciones de la clase base desde objetos de una clase derivada

El ejemplo en la figura 12.1 reutiliza las versiones finales de las clases `EmpleadoPorComision` y `EmpleadoBaseMasComision` de la sección 11.3.5. Este ejemplo demuestra tres formas de orientar los apuntadores de la clase base y los apuntadores de la clase derivada a objetos de la clase base y objetos de la clase derivada. Las primeras dos son simples y directas: orientamos un apuntador de la clase base a un objeto de la clase base e invocamos la funcionalidad de la clase base, y orientamos un apuntador de la clase derivada a un objeto de la clase derivada e invocamos la funcionalidad de la clase derivada. Después, demostramos la relación entre las clases derivadas y las clases base (es decir, la relación *es-un* de herencia) al orientar un apuntador de la clase base a un objeto de la clase derivada y mostrar que la funcionalidad de la clase base está evidentemente disponible en el objeto de la clase derivada.

```
1  // Fig. 12.1: fig12_01.cpp
2  // Cómo orientar los apuntadores de la clase base y la clase derivada a los
3  // objetos de la clase base y la clase derivada, respectivamente.
4  #include <iostream>
5  #include <iomanip>
6  #include "EmpleadoPorComision.h"
7  #include "EmpleadoBaseMasComision.h"
8  using namespace std;
9
10 int main()
11 {
12    // crea el objeto de la clase base
13    EmpleadoPorComision empleadoPorComision(
14       "Sue", "Jones", "222-22-2222", 10000, .06 );
15
16    // crea un apuntador de la clase base
17    EmpleadoPorComision *empleadoPorComisionPtr = nullptr;
18
19    // crea un objeto de la clase derivada
20    EmpleadoBaseMasComision empleadoBaseMasComision(
21       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
22
23    // crea un apuntador de la clase derivada
24    EmpleadoBaseMasComision *empleadoBaseMasComisionPtr = nullptr;
25
26    // establece el formato de salida de punto flotante
27    cout << fixed << setprecision( 2 );
28
29    // imprime los objetos empleadoPorComision y empleadoBaseMasComision
30    cout << "Impresion de los objetos de clase base y clase derivada:\n\n";
31    empleadoPorComision.imprimir();
      // invoca a la función imprimir de la clase base
32    cout << "\n\n";
33    empleadoBaseMasComision.imprimir();
      // invoca a la función imprimir de la clase derivada
34
35    // orienta el apuntador de la clase base al objeto de la clase base e imprime
36    empleadoPorComisionPtr = &empleadoPorComision; // perfectamente natural
37    cout << "\n\n\nAl llamar a imprimir con el apuntador de clase base al "
38       << "\nobjeto de clase base se invoca la funcion imprimir de la clase base:
            \n\n";
39    empleadoPorComisionPtr->imprimir();
      // invoca a la función imprimir de la clase base
40
41    // orienta el apuntador de clase derivada al objeto de clase derivada e imprime
42    empleadoBaseMasComisionPtr = &empleadoBaseMasComision; // natural
43    cout << "\n\n\nAl llamar a imprimir con el apuntador de clase derivada al "
44       << "\nobjeto de clase derivada se invoca a la funcion imprimir "
45       << "de la clase derivada:\n\n";
46    empleadoBaseMasComisionPtr->imprimir();
      // invoca a la función imprimir de la clase derivada
47
48    // orienta el apuntador de clase base al objeto de clase derivada e imprime
49    empleadoPorComisionPtr = &empleadoBaseMasComision;
50    cout << "\n\n\nAl llamar a imprimir con el apuntador de clase base al "
51       << "objeto de clase derivada\nse invoca a la funcion imprimir de la "
```

Fig. 12.1 | Asignación de direcciones de objetos de la clase base y la clase derivada a apuntadores de la clase base y la clase derivada (parte 1 de 2).

```
52        << "clase base en ese objeto de la clase derivada:\n\n";
53     empleadoPorComisionPtr->imprimir();
       // invoca a la función imprimir de la clase base
54     cout << endl;
55  } // fin de main
```

```
Impresion de los objetos de clase base y clase derivada:

empleado por comision: Sue Jones
numero de seguro social: 222-22-2222
ventas brutas: 10000.00
tarifa de comision: 0.06

con salario base empleado por comision: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa de comision: 0.04
salario base: 300.00


Al llamar a imprimir con el apuntador de clase base al
objeto de clase base se invoca la funcion imprimir de la clase base:

empleado por comision: Sue Jones
numero de seguro social: 222-22-2222
ventas brutas: 10000.00
tarifa de comision: 0.06


Al llamar a imprimir con el apuntador de clase derivada al
objeto de clase derivada se invoca a la funcion imprimir de la clase derivada:

con salario base empleado por comision: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa de comision: 0.04
salario base: 300.00


Al llamar a imprimir con el apuntador de clase base al objeto de clase derivada
se invoca a la funcion imprimir de la clase base en ese objeto de la clase derivada:

empleado por comision: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa de comision: 0.04
```

Observe que no se muestra el salario base

Fig. 12.1 | Asignación de direcciones de objetos de la clase base y la clase derivada a apuntadores de la clase base y la clase derivada (parte 2 de 2).

Recuerde que cada objeto `EmpleadoBaseMasComision` *es un* `EmpleadoPorComision` que *también* tiene un salario base. La función miembro `ingresos` de la clase `EmpleadoBaseMasComision` (líneas 34 a 37 de la figura 11.15) redefine a la función miembro `ingresos` de la clase `EmpleadoPorComision` (líneas 85 a 88 de la figura 11.14) para incluir el salario base del objeto. La función miembro `imprimir` de la clase `EmpleadoBaseMasComision` (líneas 40 a 48 de la figura 11.15) redefine la versión de la clase `EmpleadoPorComision` (líneas 91 a 98 de la figura 11.14) para mostrar la misma información *más* el salario base del empleado.

Creación de objetos y visualización de su contenido
En la figura 12.1, en las líneas 13 y 14 se crea un objeto `EmpleadoPorComision` y en la línea 17 se crea un apuntador a un objeto `EmpleadoPorComision`; en las líneas 20 y 21 se crea un objeto `EmpleadoBaseMasComision` y en la línea 24 se crea un apuntador a un objeto `EmpleadoBaseMasComision`. En las líneas 31 y 33 se utiliza el nombre de cada objeto para invocar a su función miembro `imprimir`.

Orientar un apuntador de la clase base a un objeto de la clase base
En la línea 36 se asigna la dirección del objeto `empleadoPorComision` de la clase base al apuntador `empleadoPorComisionPtr` de la clase base, que la línea 39 utiliza para invocar a la función miembro `imprimir` en ese objeto `EmpleadoPorComision`. Esto invoca a la versión de `imprimir` definida en la clase base `EmpleadoPorComision`.

Orientar un apuntador de la clase derivada a un objeto de la clase derivada
De manera similar, en la línea 42 se asigna la dirección del objeto `empleadoBaseMasComision` de la clase derivada al apuntador `empleadoBaseMasComisionPtr` de la clase derivada, que la línea 46 utiliza para invocar a la función miembro `imprimir` en ese objeto `EmpleadoBaseMasComision`. Esto invoca a la versión de `imprimir` definida en la clase derivada `EmpleadoBaseMasComision`.

Orientar un apuntador de la clase base a un objeto de la clase derivada
Después, en la línea 49 se asigna la dirección del objeto `empleadoBaseMasComision` de la clase derivada al apuntador `empleadoPorComisionPtr` de la clase base, que en la línea 53 se utiliza para invocar a la función miembro `imprimir`. Se permite esta "contradicción", ya que un objeto de una clase derivada *es un* objeto de su clase base. A pesar del hecho de que el apuntador de la clase base `EmpleadoPorComision` apunta a un objeto de la *clase derivada* `EmpleadoBaseMasComision`, se invoca a la función miembro `imprimir` de la *clase base* `EmpleadoPorComision` (en vez de la función `imprimir` de `EmpleadoBaseMasComision`). Los resultados de cada invocación a cada una de las funciones miembro `imprimir` en este programa revelan que *la funcionalidad invocada depende del tipo del manejador (o referencia) que se utiliza para invocar la función, no del tipo del objeto para el que se hace la llamada a la función miembro.* En la sección 12.3.4, cuando presentemos las funciones `virtual`, demostraremos que es posible invocar la funcionalidad del tipo del objeto, *en vez de* invocar la funcionalidad del tipo del manejador. Veremos que esto es crucial para implementar el comportamiento polimórfico: el tema clave de este capítulo.

12.3.2 Cómo orientar los apuntadores de una clase derivada a objetos de la clase base

En la sección 12.3.1, asignamos la dirección de un objeto de la clase derivada a un apuntador de la clase base y explicamos que el compilador de C++ permite esta asignación, debido a que un objeto de una clase derivada *es un* objeto de la clase base. Tomamos la metodología opuesta en la figura 12.2, al orientar un apuntador de la clase derivada a un objeto de la clase base. [*Nota:* este programa reutiliza las versiones finales de las clases `EmpleadoPorComision` y `EmpleadoBaseMasComision` de la sección 11.3.5]. Las líneas 8 y 9 de la figura 12.2 crean un objeto `EmpleadoPorComision`, y en la línea 10 se crea un apuntador `EmpleadoBaseMasComision`. En la línea 14 se trata de asignar la dirección del objeto `empleadoPorComision` de la clase base al apuntador `empleadoBaseMasComisionPtr` de la clase derivada, pero el compilador genera un error. El compilador evita esta asignación, ya que un `EmpleadoPorComision` *no es* un `EmpleadoBaseMasComision`.

Considere las consecuencias si el compilador permitiera esta asignación. A través de un apuntador `EmpleadoBaseMasComision` podemos invocar *cualquier* función miembro de `EmpleadoBaseMasComision`, incluyendo `establecerSalarioBase`, para el objeto al que apunta el apuntador (es decir, el objeto `empleadoPorComision` de la clase base). Sin embargo, el objeto `EmpleadoPorComision` *no* proporciona una función miembro `establecerSalarioBase`, *ni* proporciona un miembro de datos `salarioBase`

para establecer su valor. Esto podría generar problemas, debido a que la función miembro `establecerSalarioBase` supondría que hay un miembro de datos `salarioBase` que se debe establecer en su "ubicación usual" en un objeto `EmpleadoBaseMasComision`. Esta memoria no pertenece al objeto `EmpleadoPorComision`, por lo que la función miembro `establecerSalarioBase` podría sobrescribir otros datos importantes en la memoria, posiblemente datos que pertenezcan a un objeto distinto.

```
1  // Fig. 12.2: fig12_02.cpp
2  // Cómo orientar un apuntador de clase derivada a un objeto de clase base.
3  #include "EmpleadoPorComision.h"
4  #include "EmpleadoBaseMasComision.h"
5
6  int main()
7  {
8     EmpleadoPorComision empleadoPorComision(
9        "Sue", "Jones", "222-22-2222", 10000, .06 );
10    EmpleadoBaseMasComision *empleadoBaseMasComisionPtr = nullptr;
11
12    // orienta el apuntador de la clase derivada al objeto de la clase base
13    // Error: un EmpleadoPorComision no es un EmpleadoBaseMasComision
14    empleadoBaseMasComisionPtr = &empleadoPorComision;
15 } // fin de main
```

Mensaje de error del compilador Microsoft Visual C++:

```
C:\cpphtp8_ejemplos\cap12\Fig12_02\fig12_02.cpp(14): error C2440: '=' :
   cannot convert from 'EmpleadoPorComision *' to 'EmpleadoBaseMasComision *'
         Cast from base to derived requires dynamic_cast or static_cast
```

Fig. 12.2 | Orientación de un apuntador de la clase derivada a un objeto de la clase base.

12.3.3 Llamadas a funciones miembro de una clase derivada a través de apuntadores de la clase base

Desde un apuntador de la clase base, el compilador nos permite invocar *sólo* a las funciones miembro de la clase base. Por ende, si un apuntador de la clase base se orienta a un objeto de la clase derivada, y se trata de acceder a una *función miembro que sólo pertenezca a la clase derivada*, se producirá un error de compilación.

En la figura 12.3 se muestran las consecuencias de tratar de invocar a una función miembro de la clase derivada desde un apuntador de la clase base. [*Nota:* estamos reutilizando de nuevo las versiones de las clases `EmpleadoPorComision` y `EmpleadoBaseMasComision` de la sección 11.3.5]. En la línea 11 se crea `empleadoPorComisionPtr` (un apuntador a un objeto `EmpleadoPorComision`) y en las líneas 12 y 13 se crea un objeto `EmpleadoBaseMasComision`. En la línea 16 se orienta el apuntador `empleadoPorComisionPtr` de la clase base al objeto de la clase derivada llamado `empleadoBaseMasComision`. En la sección 12.3.1 vimos que esto está permitido, ya que un `EmpleadoBaseMasComision` *es un* `EmpleadoPorComision` (en el sentido en el que un objeto `EmpleadoBaseMasComision` contiene toda la funcionalidad de un objeto `EmpleadoPorComision`). En las líneas 20 a 24 se invocan las funciones miembro de la clase base `obtenerPrimerNombre`, `obtenerApellidoPaterno`, `obtenerNumeroSeguroSocial`, `obtenerVentasBrutas` y `obtenerTarifaComision` desde el apuntador de la clase base. Todas estas llamadas son legítimas, ya que `EmpleadoBaseMasComision` *hereda* estas funciones miembro de `EmpleadoPorComision`. Sabemos que `empleadoPorComisionPtr` está orientado a un objeto `EmpleadoBaseMasComision`, por lo que en las líneas 28 y 29 tratamos de invocar a las funciones miembro de `EmpleadoBaseMasComision` llamadas `obtenerSalarioBase` y `establecerSalarioBase`. El compilador genera errores en ambas llamadas, debido a que *no* se hacen a las funciones miembro de la clase base `EmpleadoPorComision`. El manejador se

puede utilizar para invocar *sólo* a las funciones que son miembros del tipo de clase asociado de ese manejador. (En este caso, desde un `EmpleadoPorComision *` sólo podemos invocar a las funciones miembro de `EmpleadoPorComision` llamadas `establecerPrimerNombre`, `obtenerPrimerNombre`, `establecerApellidoPaterno`, `obtenerApellidoPaterno`, `establecerNumeroSeguroSocial`, `obtenerNumeroSeguroSocial`, `establecerVentasBrutas`, `obtenerVentasBrutas`, `establecerTarifaComision`, `obtenerTarifaComision`, `ingresos` e `imprimir`).

```
1  // Fig. 12.3: fig12_03.cpp
2  // Intento de invocar a las funciones miembro que sólo son de
3  // la clase derivada a través de un apuntador de la clase base.
4  #include <string>
5  #include "EmpleadoPorComision.h"
6  #include "EmpleadoBaseMasComision.h"
7  using namespace std;
8
9  int main()
10 {
11    EmpleadoPorComision *empleadoPorComisionPtr = nullptr;
      // apuntador de clase base
12    EmpleadoBaseMasComision empleadoBaseMasComision(
13       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 ); // clase derivada
14
15    // orienta el apuntador de la clase base al objeto de la clase derivada
         (permitido)
16    empleadoPorComisionPtr = &empleadoBaseMasComision;
17
18    // invoca a las funciones miembro de la clase base en el objeto de la
19    // clase derivada a través de un apuntador de la clase base (permitido)
20    string primerNombre = empleadoPorComisionPtr->obtenerPrimerNombre();
21    string apellidoPaterno = empleadoPorComisionPtr->obtenerApellidoPaterno();
22    string nss = empleadoPorComisionPtr->obtenerNumeroSeguroSocial();
23    double ventasBrutas = empleadoPorComisionPtr->obtenerVentasBrutas();
24    double tarifaComision = empleadoPorComisionPtr->obtenerTarifaComision();
25
26    // intento de invocar a las funciones miembro que sólo son de la clase derivada
27    // en un objeto de la clase derivada a través de un apuntador de la
         clase base (no permitido)
28    double salarioBase = empleadoPorComisionPtr->obtenerSalarioBase();
29    empleadoPorComisionPtr->establecerSalarioBase( 500 );
30 } // fin de main
```

Mensajes de error del compilador GNU C++:

```
fig12_03.cpp:28:47: error: 'class EmpleadoPorComision' has no member named
  'obtenerSalarioBase'
fig12_03.cpp:29:27: error: 'class EmpleadoPorComision' has no member named
  'establecerSalarioBase'
```

Fig. 12.3 | Intento de invocar a las funciones que sólo son de la clase derivada, a través de un apuntador de la clase base.

Conversión descendente

El compilador permite el acceso a los miembros que sólo son de la clase derivada desde un apuntador de la clase base que esté orientado a un objeto de la clase derivada, *si* convertimos de manera explícita el apuntador de la clase base a un apuntador de la clase derivada; a esta técnica se le conoce como **conversión descendente**. Como sabe, es posible orientar un apuntador de la clase base a un objeto de la clase derivada.

Sin embargo, como demostramos en la figura 12.3, un apuntador de la clase base se puede usar para invocar *sólo* las funciones declaradas en la clase base. La conversión descendente permite una operación específica de la clase derivada en un objeto de la clase derivada al que apunta un apuntador de la clase base. Después de una conversión descendente, el programa *puede* invocar las funciones de la clase derivada que no están en la clase base. La conversión descendente es una operación potencialmente peligrosa. En la sección 12.8 le mostraremos cómo usar la conversión descendente *en forma segura*.

Observación de Ingeniería de Software 12.3

Si la dirección de un objeto de la clase derivada se ha asignado a un apuntador de una de sus clases base directas o indirectas, es aceptable convertir ese apuntador de la clase base de vuelta a un apuntador del tipo de la clase derivada. De hecho, esto debe *hacerse para llamar a las funciones miembro de la clase derivada que no aparezcan en la clase base.*

12.3.4 Funciones y destructores virtuales

En la sección 12.3.1, orientamos un apuntador de la clase base `EmpleadoPorComision` a un objeto de la clase derivada `EmpleadoBaseMasComision`, y después invocamos a la función miembro `imprimir` a través de ese apuntador. Recuerde que el *tipo del manejador* determina cuál funcionalidad de la clase se va a invocar. En este caso, el apuntador `EmpleadoPorComision` invocó a la función miembro `imprimir` de `EmpleadoPorComision` en el objeto `EmpleadoBaseMasComision`, aun cuando el apuntador estaba orientado a un objeto `EmpleadoBaseMasComision` que tiene su propia función `imprimir` personalizada.

Observación de Ingeniería de Software 12.4

Con las funciones `virtual`*, el tipo del objeto y* no *el tipo del manejador utilizado para invocar a la función miembro, es el que determina cuál versión de una función* `virtual` *se debe invocar.*

Por qué son útiles las funciones `virtual`

Primero vamos a considerar por qué son útiles las funciones `virtual`. Suponga que un conjunto de clases de figuras como `Circulo`, `Triangulo`, `Rectangulo` y `Cuadrado` se derivan de la clase base `Figura`. Cada una de estas clases podría estar dotada con la habilidad de *dibujarse a sí misma* a través de una función miembro llamada `dibujar`, pero la función para cada figura es bastante distinta. En un programa que dibuja un conjunto de figuras, sería útil poder tratar a todas las figuras en forma genérica como objetos de la clase base `Figura`. Después, para dibujar cualquier figura podríamos simplemente usar un apuntador de la clase base `Figura` para invocar a la función `dibujar`, y dejar que el programa determine en forma *dinámica* (es decir, en tiempo de ejecución) de cuál clase derivada se va a utilizar la función `dibujar`, con base en el tipo del objeto al que apunta el apuntador de la clase base `Figura` en cualquier momento dado. Éste es un *comportamiento polimórfico*.

Declaración de funciones `virtual`

Para permitir este tipo de comportamiento, declaramos a `dibujar` en la clase base como una **función `virtual`**, y **sobrescribimos** a `dibujar` en *cada* una de las clases derivadas para dibujar la figura apropiada. Desde una perspectiva de implementación, *sobrescribir* una función no es algo distinto a *redefinirla* (que es la metodología que hemos estado usando hasta ahora). Una función sobrescrita en una clase derivada tiene *la misma firma y el mismo tipo de valor de retorno* (es decir, *prototipo*) que la función que sobrescribe en su clase base. Si no declaramos la función de la clase base como `virtual`, podemos *redefinir* esa función. En contraste, si declaramos la función de la clase base como `virtual`, podemos *sobrescribir* esa función para permitir el *comportamiento polimórfico*. Para declarar una función `virtual`, anteponemos al prototipo de la función la palabra clave `virtual` en la clase base. Por ejemplo,

```
virtual void dibujar() const;
```

aparecería en la clase base `Figura`. El prototipo anterior declara que la función `dibujar` es una función `virtual` que no recibe argumentos y no devuelve nada. Esta función se declara `const` debido a que, por lo general, una función `dibujar` no realizaría modificaciones al objeto `Figura` en el cual se invoca; las funciones virtuales *no* tienen que ser funciones `const`.

Observación de Ingeniería de Software 12.5

Una vez que una función se declara `virtual`, permanece `virtual` en todos los niveles hacia abajo de la jerarquía de herencia desde ese punto, aun si esa función no se declara explícitamente como `virtual` cuando una clase derivada la sobrescribe.

Buena práctica de programación 12.1

Aun cuando ciertas funciones son implícitamente `virtual` debido a la declaración que se hace en un nivel superior en la jerarquía de clases, debe declarar explícitamente estas funciones como `virtual` en cada nivel de la jerarquía de herencia para promover la claridad del programa.

Observación de Ingeniería de Software 12.6

Cuando una clase derivada opta por no *sobrescribir una función `virtual` de su clase base, la clase derivada simplemente hereda la implementación de la función `virtual` de su clase base.*

Invocación de una función `virtual` a través de un apuntador o referencia de la clase base

Si un programa invoca a una función `virtual` a través de un apuntador de la clase base a un objeto de la clase derivada (por ejemplo, `figuraPtr->dibujar()`) o una referencia de la clase base a un objeto de la clase derivada (por ejemplo, `figuraRef.dibujar()`), el programa elegirá la función `dibujar` correcta de la clase derivada *en forma dinámica* (es decir, en tiempo de ejecución) *con base en el tipo del objeto, no en el tipo del apuntador o referencia*. Al proceso de elegir la función apropiada a llamar en tiempo de ejecución (en vez de hacerlo en tiempo de compilación) se le conoce como **vinculación dinámica** o **vinculación tardía**.

Invocación de una función `virtual` a través del nombre de un objeto

Cuando una función `virtual` se llama al referenciar un objeto específico por su *nombre* y usando el operador punto de selección de miembros (por ejemplo, `objetoCuadrado.dibujar()`), la invocación de la función se *resuelve en tiempo de compilación* (a esto se le conoce como **vinculación estática**) y la función `virtual` que se llama es la que se define para (o se hereda por) la clase de ese objeto específico; esto *no* es comportamiento polimórfico. Por ende, la vinculación dinámica con las funciones `virtual` sólo ocurre a partir de apuntadores (y, como pronto veremos, de referencias).

Funciones `virtual` en la jerarquía `EmpleadoPorComision`

Ahora veamos cómo las funciones `virtual` pueden permitir el comportamiento polimórfico en nuestra jerarquía de empleados. Las figuras 12.4 y 12.5 son los encabezados para las clases `EmpleadoPorComision` y `EmpleadoBaseMasComision`, respectivamente. Modificamos estos archivos para declarar las funciones miembro `ingresos` e `imprimir` de cada clase como `virtual` (líneas 29 y 30 de la figura 12.4, y líneas 19 y 20 de la figura 12.5). Como las funciones `ingresos` e `imprimir` son `virtual` en la clase `EmpleadoPorComision`, las funciones `ingresos` e `imprimir` de la clase `EmpleadoBaseMasComision` *sobrescriben* a las de la clase `EmpleadoPorComision`. Además, las funciones `ingresos` e `imprimir` de la clase `EmpleadoBaseMasComision` se declaran como **`override`**.

Tip para prevenir errores 12.1

Aplique la palabra clave `override` *de C++11 a* toda *función sobrescrita en una clase derivada. Esto obliga al compilador a comprobar si la clase base tiene una función miembro con el mismo nombre y lista de parámetros (es decir, la misma firma). De no ser así, el compilador genera un error.*

Ahora, si orientamos un apuntador de la clase base `EmpleadoPorComision` a un objeto de la clase derivada `EmpleadoBaseMasComision`, y el programa utiliza ese apuntador para llamar a la función `ingresos` o `imprimir`, se invocará la función correspondiente del objeto `EmpleadoBaseMasComision`. *No* hubo modificaciones a las implementaciones de las funciones miembro de las clases `EmpleadoPorComision` y `EmpleadoBaseMasComision`, por lo que reutilizamos las versiones de las figuras 11.14 y 11.15.

```
1  // Fig. 12.4: EmpleadoPorComision.h
2  // Encabezado de la clase EmpleadoPorComision que declara a ingresos e imprimir
      como virtual.
3  #ifndef COMISION_H
4  #define COMISION_H
5
6  #include <string> // clase string estándar de C++
7
8  class EmpleadoPorComision
9  {
10 public:
11    EmpleadoPorComision( const std::string &, const std::string &,
12       const std::string &, double = 0.0, double = 0.0 );
13
14    void establecerPrimerNombre( const std::string & );
      // establece el primer nombre
15    std::obtenerPrimerNombre() const; // devuelve el primer nombre
16
17    void establecerApellidoPaterno( const std::string & );
      // establece el apellido paterno
18    std::obtenerApellidoPaterno() const; // devuelve el apellido paterno
19
20    void establecerNumeroSeguroSocial( const std::string & ); // establece el NSS
21    std::obtenerNumeroSeguroSocial() const; // devuelve el NSS
22
23    void establecerVentasBrutas( double ); // establece el monto de ventas brutas
24    double obtenerVentasBrutas() const; // devuelve el monto de ventas brutas
25
26    void establecerTarifaComision( double ); // establece la tarifa de comisión
27    double obtenerTarifaComision() const; // devuelve la tarifa de comisión
28
29    virtual double ingresos() const; // calcula los ingresos
30    virtual void imprimir() const; // imprime el objeto
31 private:
32    std::string primerNombre;
33    std::string apellidoPaterno;
34    std::string numeroSeguroSocial;
35    double ventasBrutas; // ventas brutas por semana
36    double tarifaComision; // porcentaje de comisión
37 }; // fin de la clase EmpleadoPorComision
38
39 #endif
```

Fig. 12.4 | Encabezado de la clase `EmpleadoPorComision`, que declara a las funciones `ingresos` e `imprimir` como `virtual`.

```
1  // Fig. 12.5: EmpleadoBaseMasComision.h
2  // Clase EmpleadoBaseMasComision derivada de la clase
3  // EmpleadoPorComision.
4  #ifndef BASEMAS_H
5  #define BASEMAS_H
6
7  #include <string> // clase string estándar de C++
8  #include "EmpleadoPorComision.h" // declaración de la clase EmpleadoPorComision
9
10 class EmpleadoBaseMasComision : public EmpleadoPorComision
11 {
12 public:
13    EmpleadoBaseMasComision( const std::string &, const std::string &,
14       const std::string &, double = 0.0, double = 0.0, double = 0.0 );
15
16    void establecerSalarioBase( double ); // establece el salario base
17    double obtenerSalarioBase() const; // devuelve el salario base
18
19    virtual double ingresos() const override; // calcula los ingresos
20    virtual void imprimir() const override; // imprime el objeto
21 private:
22    double salarioBase; // salario base
23 }; // fin de la clase EmpleadoBaseMasComision
24
25 #endif
```

Fig. 12.5 | Encabezado de la clase `EmpleadoBaseMasComision` que declara a las funciones `ingresos` e `imprimir` como `virtual`.

Modificamos la figura 12.1 para crear el programa de la figura 12.6. En las líneas 40 a 51 de la figura 12.6 se demuestra otra vez que un apuntador `EmpleadoPorComision` orientado a un objeto `EmpleadoPorComision` puede utilizarse para invocar la funcionalidad de `EmpleadoPorComision`, y que un apuntador `EmpleadoBaseMasComision` orientado a un objeto `EmpleadoBaseMasComision` puede utilizarse para invocar la funcionalidad de `EmpleadoBaseMasComision`. En la línea 54 se orienta el apuntador `empleadoPorComisionPtr` de la clase base al objeto `empleadoBaseMasComision` de la clase derivada. Observe que cuando en la línea 61 se invoca a la función miembro `imprimir` desde el apuntador de la clase base, se invoca a la función miembro `imprimir` de la clase derivada `EmpleadoBaseMasComision`, por lo que en la línea 61 se imprime un texto distinto al de la línea 53 en la figura 12.1 (cuando la función miembro `imprimir` *no* se declaró `virtual`). Podemos ver que, al declarar una función miembro `virtual`, el programa determina en forma dinámica cuál función debe invocar *con base en el tipo de objeto al que apunta el manejador, en vez de basarse en el tipo del manejador*. Observe de nuevo que cuando `empleadoPorComisionPtr` apunta a un objeto `EmpleadoPorComision`, se invoca a la función `imprimir` de la clase `EmpleadoPorComision` (figura 12.6, línea 40) y cuando `empleadoPorComisionPtr` apunta a un objeto `EmpleadoBaseMasComision`, se invoca a la función `imprimir` de `EmpleadoBaseMasComision` (línea 61). Por ende, el mismo mensaje (`imprimir`, en este caso) que se envía (desde un apuntador de la clase base) a una variedad de objetos relacionados por la herencia a esa clase base, toma muchas formas; éste es el comportamiento polimórfico.

```
1  // Fig. 12.6: fig12_06.cpp
2  // Introducción al polimorfismo, las funciones virtuales y la vinculación
      postergada.
3  #include <iostream>
4  #include <iomanip>
5  #include "EmpleadoPorComision.h"
6  #include "EmpleadoBaseMasComision.h"
7  using namespace std;
8
9  int main()
10 {
11    // crea un objeto de la clase base
12    EmpleadoPorComision empleadoPorComision(
13       "Sue", "Jones", "222-22-2222", 10000, .06 );
14
15    // crea un apuntador de la clase base
16    EmpleadoPorComision *empleadoPorComisionPtr = nullptr;
17
18    // crea un objeto de la clase derivada
19    EmpleadoBaseMasComision empleadoBaseMasComision(
20       "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
21
22    // crea un apuntador de la clase derivada
23    EmpleadoBaseMasComision *empleadoBaseMasComisionPtr = nullptr;
24
25    // establece el formato de salida de punto flotante
26    cout << fixed << setprecision( 2 );
27
28    // imprime los objetos usando la vinculación estática
29    cout << "Invocando a la funcion imprimir en objetos de la clase base "
30       << "\ny la clase derivada con vinculacion estatica\n\n";
31    empleadoPorComision.imprimir(); // vinculación estática
32    cout << "\n\n";
33    empleadoBaseMasComision.imprimir(); // vinculación estática
34
35    // imprime los objetos usando vinculación dinámica
36    cout << "\n\n\nInvocando a la funcion imprimir en objetos de la clase base "
37       << "y la \nclase derivada con vinculacion dinamica";
38
39    // orienta el apuntador de la clase base al objeto de la clase base e imprime
40    empleadoPorComisionPtr = &empleadoPorComision;
41    cout << "\n\nAl llamar a la funcion virtual imprimir con un apuntador"
42       << "\nde la clase base a un objeto de la clase base se invoca a la "
43       << "funcion imprimir de la clase base:\n\n";
44    empleadoPorComisionPtr->imprimir();
      // invoca a la función imprimir de la clase base
45
46    // orienta un apuntador de la clase derivada al objeto de la
         clase derivada e imprime
47    empleadoBaseMasComisionPtr = &empleadoBaseMasComision;
48    cout << "\n\nAl llamar a la funcion virtual imprimir con un apuntador "
49       << "de la clase derivada\na un objeto de la clase derivada se invoca a "
50       << "la funcion imprimir de la clase derivada:\n\n";
51    empleadoBaseMasComisionPtr->imprimir();
      // invoca a la función imprimir de la clase derivada
```

Fig. 12.6 | Demostración del polimorfismo al invocar una función `virtual` de la clase derivada a través de un apuntador de la clase base a un objeto de la clase derivada (parte 1 de 2).

```
52
53    // orienta un apuntador de la clase base a un objeto de la clase derivada e
         imprime
54    empleadoPorComisionPtr = &empleadoBaseMasComision;
55    cout << "\n\nAl llamar a la funcion virtual imprimir con un apuntador de la
            clase base"
56       << "\na un objeto de la clase derivada se invoca a la funcion "
57       << "imprimir de la clase derivada:\n\n";
58
59    // polimorfismo; invoca a la función imprimir de EmpleadoBaseMasComision;
60    // apuntador de la clase base a un objeto de la clase derivada
61    empleadoPorComisionPtr->imprimir();
62    cout << endl;
63 } // fin de main
```

```
Invocando a la funcion imprimir en objetos de la clase base
y la clase derivada con vinculacion estatica

empleado por comision: Sue Jones
numero de seguro social: 222-22-2222
ventas brutas: 10000.00
tarifa de comision: 0.06

con salario base empleado por comision: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa de comision: 0.04
salario base: 300.00


Invocando a la funcion imprimir en objetos de la clase base y la
clase derivada con vinculacion dinamica

Al llamar a la funcion virtual imprimir con un apuntador
de la clase base a un objeto de la clase base se invoca a la funcion
imprimir de la clase base:

empleado por comision: Sue Jones
numero de seguro social: 222-22-2222
ventas brutas: 10000.00
tarifa de comision: 0.06

Al llamar a la funcion virtual imprimir con un apuntador de la clase derivada
a un objeto de la clase derivada se invoca a la funcion imprimir de la clase
derivada:

con salario base empleado por comision: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa de comision: 0.04
salario base: 300.00

Al llamar a la funcion virtual imprimir con un apuntador de la clase base
a un objeto de la clase derivada se invoca a la funcion imprimir de la clase
derivada:

con salario base empleado por comision: Bob Lewis
numero de seguro social: 333-33-3333
ventas brutas: 5000.00
tarifa de comision: 0.04
salario base: 300.00 ———— Observe que no se muestra el salario base
```

Fig. 12.6 | Demostración del polimorfismo al invocar una función `virtual` de la clase derivada a través de un apuntador de la clase base a un objeto de la clase derivada (parte 2 de 2).

Destructores virtual

Puede ocurrir un problema al utilizar polimorfismo para procesar los objetos de una jerarquía de clases que se asignan en forma dinámica. Hasta ahora hemos visto los destructores que no se declaran con la palabra clave `virtual`. Si se destruye un objeto de la clase derivada con un destructor no `virtual` mediante la aplicación del operador `delete` a un *apuntador de la clase base* al objeto, el estándar de C++ especifica que el comportamiento es *indefinido*.

La solución simple para este problema es crear un **destructor `virtual`** en la clase base. Si el destructor de una clase se declara `virtual`, los destructores de todas las clases derivadas serán *también* `virtual` y *sobrescribirán* al destructor de la clase base. Por ejemplo, en la definición de la clase `EmpleadoPorComision`, podemos definir el destructor `virtual` de la siguiente manera:

```
virtual ~EmpleadoPorComisione() { }
```

Ahora, si un objeto en la jerarquía se destruye explícitamente al aplicar el operador `delete` a un *apuntador de la clase base*, se hace una llamada al destructor para la *clase apropiada* con base en el objeto al que apunta el apuntador de la clase base. Recuerde, cuando se destruye un objeto de la clase derivada, la parte del objeto de la clase derivada correspondiente a la clase base también se destruye, por lo que es importante que se ejecuten *ambos* destructores de la clase derivada y la clase base. El destructor de la clase base se ejecuta de manera automática, después del destructor de la clase derivada. De aquí en adelante, incluiremos un destructor virtual en *cada* clase que contenga funciones `virtual`.

Tip para prevenir errores 12.2

Si una clase tiene funciones `virtual`, siempre debemos *proporcionar un destructor `virtual` aunque no se requiera uno para la clase. Esto asegura que se invoque a un destructor personalizado de la clase derivada (si hay uno) cuando se elimine un objeto de la clase derivada mediante un apuntador de la clase base.*

Error común de programación 12.1

Los constructores no pueden ser `virtual`. Declarar un constructor como `virtual` es un error de compilación.

C++11: Funciones miembro y clases `final`

Antes de C++11, una clase derivada podía sobrescribir *cualquiera* de las funciones `virtual` de su clase base. En C++11, una función `virtual` de la clase base que se declara **`final`** en su prototipo, como en

```
virtual unaFunción( parámetros ) final;
```

no puede sobrescribirse en ninguna clase derivada; esto garantiza que la definición de la función miembro `final` de la clase base sea utilizada por todos los objetos de la clase base y por todos los objetos de las clases derivadas directas *e* indirectas de la clase base. De manera similar, antes de C++11 era posible usar *cualquier* clase existente como clase base en una jerarquía. A partir de C++11, podemos declarar una clase como `final` para evitar que se utilice como una clase base, como en el siguiente ejemplo:

```
class MiClase final // esta clase no puede ser una clase base
{
   // cuerpo de la clase
};
```

Si intentamos sobrescribir una función miembro `final` o heredar de una clase base `final`, se produce un error de compilación.

12.4 Tipos de campos e instrucciones switch

Una manera de determinar el tipo de un objeto es utilizar una instrucción `switch` para comprobar el valor de un campo en el objeto. Esto nos permite diferenciar entre los tipos de los objetos, y después invocar una acción apropiada para un objeto específico. Por ejemplo, en una jerarquía de figuras en las que cada objeto figura tiene un atributo `tipoFigura`, una instrucción `switch` podría comprobar el `tipoFigura` del objeto para determinar cuál función `imprimir` debe llamar.

El uso de la lógica de `switch` expone a los programas a una variedad de problemas potenciales. Por ejemplo, el programador podría olvidar incluir una prueba de tipos cuando sea obligatorio hacerla, o podría olvidar evaluar todos los casos posibles en una instrucción `switch`. Al modificar un sistema basado en `switch` agregando nuevos tipos, el programador podría olvidar insertar los nuevos casos en *todas* las instrucciones `switch` relevantes. Cada adición o eliminación de una clase requiere la modificación de todas las instrucciones `switch` en el sistema; rastrear estas instrucciones puede ser un proceso que consuma mucho tiempo y esté propenso a errores.

Observación de Ingeniería de Software 12.7

La programación polimórfica puede eliminar la necesidad de la lógica de `switch`. Al utilizar el mecanismo polimórfico para realizar la lógica equivalente, los programadores pueden evitar el tipo de errores que se asocian generalmente con la lógica de `switch`.

Observación de Ingeniería de Software 12.8

Una consecuencia interesante de utilizar el polimorfismo es que los programas toman una apariencia simplificada. Contienen menos lógica de bifurcación y un código secuencial más simple.

12.5 Clases abstractas y funciones virtual puras

Cuando pensamos en una clase como un tipo, suponemos que los programas crearán objetos de ese tipo. Sin embargo, hay casos en los que es útil definir las *clases de las cuales nunca se tendrá la intención de instanciar objetos*. Dichas clases se conocen como **clases abstractas**. Como estas clases se utilizan comúnmente como clases base en las jerarquías de herencia, nos referimos a ellas como **clases base abstractas**. Estas clases no se pueden utilizar para instanciar objetos ya que, como veremos pronto, las clases abstractas están *incompletas*; las clases derivadas deben definir las "piezas faltantes" antes de poder instanciar objetos de estas clases. En la sección 12.6 crearemos programas con clases abstractas.

Una clase abstracta es una clase base a partir de la cual otras clases puedan heredar. Las clases que se pueden utilizar para instanciar objetos se conocen como **clases concretas**. Dichas clases definen o heredan implementaciones para *todas* las funciones miembro que declaran. Podríamos tener una clase *abstracta* llamada `FiguraBidimensional` y derivar de ella clases *concretas* tales como `Cuadrado`, `Circulo` y `Triangulo`. Podríamos tener también una clase base *abstracta* llamada `FiguraTridimensional` y derivar de ella clases *concretas* tales como `Cubo`, `Esfera` y `Cilindro`. Las clases base abstractas son *demasiado genéricas* como para definir objetos reales; necesitamos ser *más específicos* antes de poder pensar en instanciar objetos. Por ejemplo, si alguien nos dice que "dibujemos la figura bidimensional", ¿qué figura dibujaríamos? Las clases concretas proporcionan los detalles *específicos* que hacen que sea razonable instanciar objetos.

Una jerarquía de herencia *no* necesita contener clases abstractas, pero muchos sistemas orientados a objetos tienen jerarquías de clases encabezadas por clases base abstractas. En ciertos casos, las clases abstractas constituyen unos cuantos niveles superiores de la jerarquía. Un buen ejemplo de ello es la jerarquía de figuras en la figura 11.3, la cual empieza con la clase base abstracta `Figura`. En el siguiente

nivel de la jerarquía tenemos dos clases base abstractas más; a saber, `FiguraBidimensional` y `FiguraTridimensional`. El siguiente nivel de la jerarquía define clases *concretas* para las figuras bidimensionales (a saber, `Circulo`, `Cuadrado` y `Triangulo`) y para las figuras tridimensionales (a saber, `Esfera`, `Cubo` y `Tetraedro`).

Funciones virtuales puras

Para hacer una clase abstracta, se declara una o más de sus funciones `virtual` como "puras". Una **función `virtual` pura** se especifica colocando "`= 0`" en su declaración, como en

```
virtual void dibujar() const = 0; // función virtual pura
```

El "`= 0`" se conoce como un **especificador puro**. Por lo general, las funciones `virtual` puras *no* proporcionan implementaciones, aunque pueden hacerlo. Cada clase derivada *concreta debe sobrescribir a todas* las funciones `virtual` puras de la clase base con implementaciones concretas de esas funciones; de lo contrario, la clase derivada también es abstracta. La diferencia entre una función `virtual` y una función `virtual` pura es que una función `virtual` *tiene* una implementación y proporciona a la clase derivada la *opción* de sobrescribir la función; en contraste, una función `virtual` pura *no* proporciona una implementación y *requiere* que la clase derivada sobrescriba la función para que esa clase derivada sea concreta; en caso contrario, la clase derivada permanece *abstracta*.

Las funciones `virtual` puras se utilizan cuando *no* tiene sentido para la clase base tener una implementación de una función, pero es conveniente que todas las clases derivadas concretas implementen la función. Regresando a nuestro ejemplo anterior de los objetos espaciales, no tiene sentido para la clase base `ObjetoEspacial` tener una implementación para la función `dibujar` (ya que no hay forma de dibujar un objeto espacial genérico sin tener más información acerca del tipo de objeto espacial que se va a dibujar). Un ejemplo de una función que se definiría como `virtual` (y no como `virtual` pura) sería una que devuelve un nombre para el objeto. Podemos nombrar a un `ObjetoEspacial` genérico (por ejemplo, como `"objeto espacial"`), por lo que se puede proporcionar una implementación predeterminada para esta función, y la función no necesita ser `virtual` *pura*. Sin embargo, la función se sigue declarando como `virtual`, ya que se espera que las clases derivadas sobrescriban esta función para proporcionar nombres *más específicos* para los objetos de la clase derivada.

Observación de Ingeniería de Software 12.9

Una clase abstracta define una interfaz pública común para las diversas clases en una jerarquía de clases. Una clase abstracta contiene una o más funciones `virtual` puras que las clases derivadas concretas deben *sobrescribir.*

Error común de programación 12.2

Si no se sobrescribe una función `virtual` pura en una clase derivada, esa clase se hace abstracta. Tratar de instanciar un objeto de una clase abstracta produce un error de compilación.

Observación de Ingeniería de Software 12.10

Una clase abstracta tiene por lo menos una función `virtual` pura. Una clase abstracta también puede tener miembros de datos y funciones concretas (incluyendo los constructores y destructores), que están sujetos a las reglas normales de la herencia por las clases derivadas.

Aunque *no podemos* instanciar objetos de una clase base abstracta, *sí podemos* usar la clase base abstracta para declarar *apuntadores* y *referencias* que puedan referirse a objetos de cualquier clase *concreta*

derivada de la clase abstracta. Por lo general, los programas utilizan dichos apuntadores y referencias para manipular los objetos de la clase derivada mediante el polimorfismo.

Controladores de dispositivos y polimorfismo

El polimorfismo es particularmente efectivo para implementar *sistemas de software en niveles*. Por ejemplo, en los sistemas operativos cada tipo de dispositivo físico podría operar en forma muy distinta de los otros. Aun así, los comandos para *leer* o *escribir* datos desde y hacia los dispositivos podrían tener cierta uniformidad. El mensaje *escribir* que se envía a un objeto *controlador de dispositivo* necesita interpretarse de manera específica en el contexto de ese controlador de dispositivo, y en la forma en que ese controlador de dispositivo manipula dispositivos de un tipo específico. Sin embargo, la llamada *escribir* en sí en realidad no es distinta de la *escritura* a cualquier otro dispositivo en el sistema; se coloca cierto número de bytes de memoria en ese dispositivo. Un sistema operativo orientado a objetos podría utilizar una clase base abstracta para proporcionar una interfaz apropiada para todos los controladores de dispositivos. Después, a través de la herencia de esa clase base abstracta, se forman clases derivadas que operen todas de manera similar. Las herramientas (es decir, la funciones `public`) ofrecidas por los controladores de dispositivos se proporcionan como funciones `virtual` puras en la clase base abstracta. Las implementaciones de esas funciones `virtual` puras se proporcionan en las clases derivadas que corresponden a los tipos específicos de controladores de dispositivos. Esta arquitectura también permite *agregar* nuevos dispositivos a un sistema con facilidad. El usuario simplemente puede conectar el dispositivo e instalar el controlador de su nuevo dispositivo. El sistema operativo "habla" con este nuevo dispositivo a través de su controlador de dispositivos, que tiene las mismas funciones miembro `public` que todos los demás controladores de dispositivos; aquellas definidas en la clase base abstracta del controlador de dispositivos.

12.6 Caso de estudio: sistema de nómina mediante el uso de polimorfismo

En esta sección volvemos a examinar la jerarquía `EmpleadoPorComision-EmpleadoBaseMasComision` que exploramos a lo largo de la sección 11.3. En este ejemplo utilizamos una clase abstracta y polimorfismo para realizar cálculos de nómina con base en el tipo de empleado. Creamos una jerarquía de empleados mejorada para resolver el siguiente problema:

> *Una empresa paga semanalmente a sus empleados. Los empleados son de tres tipos: los* empleados asalariados *reciben un salario semanal fijo, sin importar el número de horas trabajadas; los* empleados por comisión *reciben un porcentaje de sus ventas y los* empleados por comisión con salario base *reciben un salario base, más un porcentaje de sus ventas. Para el periodo de pago actual, la empresa ha decidido recompensar a los empleados con salario base más comisión, agregando un 10 por ciento a su salario base. La empresa desea implementar un programa en C++ que realice sus cálculos de nómina por medio del polimorfismo.*

Utilizamos la clase abstracta `Empleado` para representar el concepto general de un empleado. Las clases que se derivan directamente de `Empleado` son: `EmpleadoAsalariado` y `EmpleadoPorComision`. La clase `EmpleadoBaseMasComision` (que se deriva de `EmpleadoPorComision`) representa el último tipo de empleado. El diagrama de clases de UML de la figura 12.7 muestra la jerarquía de herencia para nuestra aplicación de nómina de empleados polimórfica. El nombre de la clase abstracta `Empleado` está en *cursiva*, según la convención del UML.

La clase base abstracta `Empleado` declara la "interfaz" para la jerarquía; es decir, el conjunto de funciones miembro que un programa puede invocar en todos los objetos `Empleado`. Cada empleado, sin importar la forma en que se calculen sus ingresos, tiene un primer nombre, un apellido paterno y un número de seguro social, por lo que aparecen los datos miembro `private primerNombre`, `apellidoPaterno` y `numeroSeguroSocial` en la clase base abstracta `Empleado`.

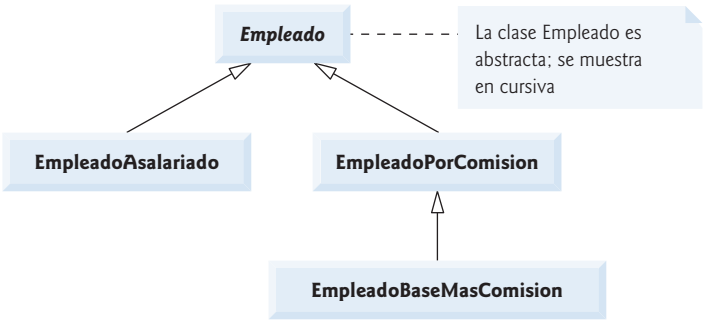


Fig. 12.7 | Diagrama de clases de UML de la jerarquía `Empleado`.

Observación de Ingeniería de Software 12.11

Una clase derivada puede heredar la interfaz y/o implementación de una clase base. Las jerarquías diseñadas para la ***herencia de implementación*** *tienden a tener su funcionalidad en un nivel alto en la jerarquía; cada nueva clase derivada hereda una o más funciones miembro que se definieron en una clase base, y la clase derivada utiliza las definiciones de la clase base. Las jerarquías diseñadas para la* ***herencia de interfaz*** *tienden a tener su funcionalidad en un nivel bajo en la jerarquía; una clase base especifica una o más funciones que deben definirse para cada clase en la jerarquía (es decir, tienen el mismo prototipo), pero las clases derivadas individuales proporcionan sus propias implementaciones de la(s) función(es).*

En las siguientes secciones se implementa la jerarquía de la clase `Empleado`. Las primeras cinco secciones implementan cada una de las clases abstractas o concretas. La última sección implementa un programa de prueba que crea objetos de todas estas clases y procesa los objetos mediante el polimorfismo.

12.6.1 Creación de la clase base abstracta Empleado

La clase `Empleado` (figuras 12.9 y 12.10, que veremos con detalle en breve) proporciona las funciones `ingresos` e `imprimir`, además de varias funciones *establecer* y *obtener* que manipulan los miembros de datos `Empleado`. Una función `ingresos` se aplica evidentemente en forma genérica a todos los empleados, pero cada cálculo de los ingresos depende de la clase del empleado. Por lo tanto, declaramos `ingresos` como `virtual` pura en la clase base `Empleado` debido a que *una implementación predeterminada no tiene sentido* para esa función; no hay suficiente información para determinar qué cantidad debe devolver `ingresos`. Cada clase derivada *sobrescribe* a `ingresos` con una implementación apropiada. Para calcular los ingresos de un empleado, el programa asigna la dirección de un objeto empleado a un *apuntador* de la clase base `Empleado`, y después invoca a la función `ingresos` en ese objeto. Mantenemos un `vector` de apuntadores `Empleado`, cada uno de los cuales apunta a un objeto `Empleado`. *Desde luego, no puede haber objetos* `Empleado` *debido a que es una clase base abstracta; sin embargo, debido a la herencia todos los objetos de todas las clases derivadas de* `Empleado` *pueden considerarse como objetos* `Empleado`. El programa itera a través del `vector` y llama a la función `ingresos` para cada objeto `Empleado`. C++ procesa estas llamadas a funciones en forma *polimórfica*. Al incluir a `ingresos` como una función `virtual` pura en `Empleado`, se *obliga* a todas las clases derivadas directas de `Empleado` que deseen ser una clase *concreta* a sobrescribir `ingresos`.

La función `imprimir` en la clase `Empleado` muestra el primer nombre, apellido paterno y número de seguro social del empleado. Como veremos, cada clase derivada de `Empleado` sobrescribe a la función

`imprimir` para mostrar el tipo del empleado (por ejemplo, `"empleado asalariado:"`), seguido del resto de la información del empleado. La función `imprimir` en las clases derivadas podría llamar también a `ingresos`, aun cuando ésta sea una función `virtual` pura en la clase base `Empleado`.

El diagrama de la figura 12.8 muestra a cada una de las cuatro clases en la jerarquía hacia abajo en el lado izquierdo, y a las funciones `ingresos` e `imprimir` a lo largo de la parte superior. Para cada clase, el diagrama muestra los resultados deseados de cada función. El texto en cursiva representa en dónde se utilizan los valores de un objeto específico en las funciones `ingresos` e `imprimir`. La clase `Empleado` especifica "`= 0`" para la función `ingresos`, para indicar que es una función `virtual` pura y, por ende, *no* tiene implementación. Cada clase derivada sobrescribe a esta función para proporcionar una implementación apropiada. *No* listamos las funciones *establecer* y *obtener* de la clase base `Empleado` debido a que *no* se sobrescriben en ninguna de las clases derivadas; cada una de estas funciones es heredada y utilizada "así como está" por cada una de las clases derivadas.

	`ingresos`	`imprimir`
`Empleado`	`= 0`	*primerNombre apellidoPaterno* `numero de seguro social:` *NSS*
`Empleado-` `Asalariado`	*salarioSemanal*	`empleado asalariado:` *primerNombre apellidoPaterno* `numero de seguro social:` *NSS* `salario semanal:` *salarioSemanal*
`Empleado-` `PorComision`	*tarifaComision * ventasBrutas*	`empleado por comision:` *primerNombre apellidoPaterno* `numero de seguro social:` *NSS* `ventas brutas:` *ventasBrutas*`;` `tarifa de comision:` *tarifaComision*

`Empleado-`
`Base-`
`MasComision`
	(*tarifaComision * ventasBrutas*) + *salarioBase*	`empleado por comision con salario base:`
		primerNombre apellidoPaterno
		`numero de seguro social:` *NSS*
		`ventas brutas:` *ventasBrutas*`;`
		`tarifa de comision:` *tarifaComision*`;`
		`salario base:` *salarioBase*

Fig. 12.8 | Interfaz polimórfica para las clases de la jerarquía `Empleado`.

Encabezado de la clase `Empleado`

Vamos a considerar el encabezado de la clase `Empleado` (figura 12.9). Las funciones miembro `public` incluyen un constructor que recibe el primer nombre, apellido paterno y número de seguro social como argumentos (línea 11 y 12); un destructor virtual (línea 13); funciones *establecer* que establecen el primer nombre, apellido paterno y número de seguro social (líneas 15, 18 y 21, respectivamente); funciones *obtener* que devuelven el primer nombre, apellido paterno y número de seguro social (líneas 16, 19 y 22, respectivamente); la función `virtual` pura `ingresos` (línea 25) y la función `virtual` `imprimir` (línea 26).

```
1  // Fig. 12.9: Empleado.h
2  // Clase base abstracta Empleado.
3  #ifndef EMPLEADO_H
```

Fig. 12.9 | La clase base abstracta `Empleado` (parte 1 de 2).

```
4  #define EMPLEADO_H
5
6  #include <string> // clase string estándar de C++
7
8  class Empleado
9  {
10 public:
11    Empleado( const std::string &, const std::string &,
12       const std::string & );
13    virtual ~Empleado() { } // destructor virtual
14
15    void establecerPrimerNombre( const std::string & ); // establece el primer nombre
16    std::string obtenerPrimerNombre() const; // devuelve el primer nombre
17
18    void establecerApellidoPaterno( const std::string & );
      // establece el apellido paterno
19    std::string obtenerApellidoPaterno() const; // devuelve el apellido paterno
20
21    void establecerNumeroSeguroSocial( const std::string & ); // establece el NSS
22    std::string obtenerNumeroSeguroSocial() const; // devuelve el NSS
23
24    // la función virtual pura hace de Empleado una clase base abstracta
25    virtual double ingresos() const = 0; // virtual pura
26    virtual void imprimir() const; // virtual
27 private:
28    std::string primerNombre;
29    std::string apellidoPaterno;
30    std::string numeroSeguroSocial;
31 }; // fin de la clase Empleado
32
33 #endif // EMPLEADO_H
```

Fig. 12.9 | La clase base abstracta `Empleado` (parte 2 de 2).

Recuerde que declaramos a `ingresos` como una función `virtual` pura, debido a que primero debemos conocer el tipo de `Empleado` *específico* para determinar los cálculos apropiados de `ingresos`. Al declarar esta función como `virtual` pura se indica que cada clase derivada concreta *debe* proporcionar una implementación para `ingresos`, y que un programa puede usar apuntadores de la clase base `Empleado` para invocar a la función `ingresos` de manera *polimórfica* para *cualquier* tipo de `Empleado`.

Definiciones de las funciones miembro de la clase `Empleado`

La figura 12.10 contiene las implementaciones de las funciones miembro para la clase `Empleado`. No se proporciona una implementación para la función `virtual ingresos`. El constructor de `Empleado` (líneas 9 a 14) no valida el número de seguro social. Por lo general, se debe proporcionar dicha validación.

```
1  // Fig. 12.10: Empleado.cpp
2  // Definiciones de las funciones miembro de la clase base abstracta Empleado.
3  // Nota: no se proporcionan definiciones para las funciones virtuales puras.
4  #include <iostream>
5  #include "Empleado.h" // definición de la clase Empleado
6  using namespace std;
```

Fig. 12.10 | Archivo de implementación de la clase `Empleado` (parte 1 de 2).

```
 7
 8  // constructor
 9  Empleado::Empleado( const string &nombre, const string &apellido,
10     const string &nss )
11     : primerNombre( nombre ), apellidoPaterno( apellido ),
          numeroSeguroSocial( nss )
12  {
13     // cuerpo vacío
14  } // fin del constructor de Empleado
15
16  // establece el primer nombre
17  void Empleado::establecerPrimerNombre( const string &nombre )
18  {
19     primerNombre = nombre;
20  } // fin de la función establecerPrimerNombre
21
22  // devuelve el primer nombre
23  string Empleado::obtenerPrimerNombre() const
24  {
25     return primerNombre;
26  } // fin de la función establecerPrimerNombre
27
28  // establece el apellido paterno
29  void Empleado::establecerApellidoPaterno( const string &apellido )
30  {
31     apellidoPaterno = apellido;
32  } // fin de la función establecerApellidoPaterno
33
34  // devuelve el apellido paterno
35  string Empleado::obtenerApellidoPaterno() const
36  {
37     return apellidoPaterno;
38  } // fin de la función obtenerApellidoPaterno
39
40  // establece el número de seguro social
41  void Empleado::establecerNumeroSeguroSocial( const string &nss )
42  {
43     numeroSeguroSocial = nss; // debe validar
44  } // fin de la función establecerNumeroSeguroSocial
45
46  // devuelve el número de seguro social
47  string Empleado::obtenerNumeroSeguroSocial() const
48  {
49     return numeroSeguroSocial;
50  } // fin de la función obtenerNumeroSeguroSocial
51
52  // imprime la información del Empleado (virtual, pero no virtual pura)
53  void Empleado::imprimir() const
54  {
55     cout << obtenerPrimerNombre() << ' ' << obtenerApellidoPaterno()
56        << "\nnumero de seguro social: " << obtenerNumeroSeguroSocial();
57  } // fin de la función imprimir
```

Fig. 12.10 | Archivo de implementación de la clase `Empleado` (parte 2 de 2).

La función `virtual` `imprimir` (líneas 53 a 57) proporciona una *implementación* que se *sobrescribirá* en *cada una* de las clases derivadas. Sin embargo, cada una de estas funciones utilizará la versión de `imprimir` de la clase abstracta para imprimir información *común para todas las clases* en la jerarquía de `Empleado`.

12.6.2 Creación de la clase derivada concreta `EmpleadoAsalariado`

La clase `EmpleadoAsalariado` (figuras 12.11 y 12.12) se deriva de la clase `Empleado` (línea 9 de la figura 12.11). Las funciones miembro `public` incluyen a un constructor que recibe un primer nombre, un apellido paterno, un número de seguro social y un salario semanal como argumentos (líneas 12 y 13); un destructor `virtual` (línea 14); una función *establecer* para asignar un nuevo valor no negativo al miembro de datos `salarioSemanal` (línea 16); una función *obtener* para devolver el valor de `salarioSemanal` (línea 17); una función `virtual` llamada `ingresos` que calcula los ingresos de un `EmpleadoAsalariado` (línea 20) y una función `virtual` llamada `imprimir` (línea 21) que imprime el tipo del empleado, a saber, `"empleado asalariado: "` seguido de la información específica del empleado producida por la función `imprimir` de la clase base `Empleado` y la función `obtenerSalarioSemanal` de `EmpleadoAsalariado`.

```
1  // Fig. 12.11: EmpleadoAsalariado.h
2  // Clase EmpleadoAsalariado derivada de Empleado.
3  #ifndef ASALARIADO_H
4  #define ASALARIADO_H
5
6  #include <string> // clase string estándar de C++
7  #include "Empleado.h" // definición de la clase Empleado
8
9  class EmpleadoAsalariado : public Empleado
10 {
11 public:
12    EmpleadoAsalariado( const std::string &, const std::string &,
13       const std::string &, double = 0.0 );
14    virtual ~EmpleadoAsalariado() { } // destructor virtual
15
16    void establecerSalarioSemanal( double ); // establece el salario semanal
17    double obtenerSalarioSemanal() const; // devuelve el salario semanal
18
19    // la palabra clave virtual indica el intento de sobrescribir
20    virtual double ingresos() const override; // calcula los ingresos
21    virtual void imprimir() const override; // imprime el objeto
22 private:
23    double salarioSemanal; // salario por semana
24 }; // fin de la clase EmpleadoAsalariado
25
26 #endif // ASALARIADO_H
```

Fig. 12.11 | Encabezado de la clase `EmpleadoAsalariado`.

Definiciones de las funciones miembro de la clase `EmpleadoAsalariado`

La figura 12.12 contiene las definiciones de las funciones miembro para `EmpleadoAsalariado`. El constructor de la clase pasa el primer nombre, apellido paterno y número de seguro social al constructor de `Empleado` (línea 11) para inicializar los miembros de datos `private` que se heredan de la clase base, pero que no se pueden utilizar directamente en la clase derivada. La función `ingresos` (líneas 33 a 36) sobrescribe la función `virtual` pura `ingresos` en `Empleado` para proporcionar una implementación

concreta que devuelva el salario semanal del `EmpleadoAsalariado`. Si no definiéramos `ingresos`, la clase `EmpleadoAsalariado` sería una clase *abstracta* y cualquier intento de instanciar un objeto de la clase produciría un error de compilación. En el encabezado de la clase `EmpleadoAsalariado` declaramos las funciones miembro `ingresos` e `imprimir` como `virtual` (líneas 20 y 21 de la figura 12.11); en realidad, colocar la palabra clave `virtual` antes de estas funciones miembro es *redundante*. Las definimos como `virtual` en la clase base `Empleado`, por lo que siguen siendo funciones `virtual` a través de la jerarquía de clases. Declarar dichas funciones explícitamente como `virtual` en cada nivel de la jerarquía puede promover la claridad del programa. Si no declaramos `ingresos` como `virtual` pura, indicamos nuestra intención de proporcionar una implementación en esta clase concreta.

```
1  // Fig. 12.12: EmpleadoAsalariado.cpp
2  // Definiciones de las funciones miembro de la clase EmpleadoAsalariado.
3  #include <iostream>
4  #include <stdexcept>
5  #include "EmpleadoAsalariado.h" // definición de la clase EmpleadoAsalariado
6  using namespace std;
7
8  // constructor
9  EmpleadoAsalariado::EmpleadoAsalariado( const string &nombre,
10    const string &apellido, const string &nss, double salario )
11    : Empleado( nombre, apellido, nss )
12 {
13    establecerSalarioSemanal( salario );
14 } // fin del constructor de EmpleadoAsalariado
15
16 // establece el salario
17 void EmpleadoAsalariado::establecerSalarioSemanal( double salario )
18 {
19    if ( salario >= 0.0 )
20       salarioSemanal = salario;
21    else
22       throw invalid_argument( "El salario semanal debe ser >= 0.0" );
23 } // fin de la función establecerSalarioSemanal
24
25 // devuelve el salario
26 double EmpleadoAsalariado::obtenerSalarioSemanal() const
27 {
28    return salarioSemanal;
29 } // fin de la función obtenerSalarioSemanal
30
31 // calcula los ingresos;
32 // sobrescribe a la función virtual pura ingresos en Empleado
33 double EmpleadoAsalariado::ingresos() const
34 {
35    return obtenerSalarioSemanal();
36 } // fin de la función ingresos
37
38 // imprime la información del EmpleadoAsalariado
39 void EmpleadoAsalariado::imprimir() const
40 {
```

Fig. 12.12 | Archivo de implementación de la clase `EmpleadoAsalariado` (parte 1 de 2).

```
41     cout << "empleado asalariado: ";
42     Empleado::imprimir(); // reutiliza la función imprimir de la clase base abstracta
43     cout << "\nsalario semanal: " << obtenerSalarioSemanal();
44  } // fin de la función imprimir
```

Fig. 12.12 | Archivo de implementación de la clase `EmpleadoAsalariado` (parte 2 de 2).

La función `imprimir` de la clase `EmpleadoAsalariado` (líneas 39 a 44 de la figura 12.12) sobrescribe a la función `imprimir` de `Empleado`. Si la clase `EmpleadoAsalariado` no sobrescribiera a `imprimir`, `EmpleadoAsalariado` heredaría la versión de `imprimir` correspondiente a `Empleado`. En ese caso, la función `imprimir` de `EmpleadoAsalariado` simplemente devolvería el nombre completo del empleado y el número de seguro social, lo cual no representa en forma adecuada a un `EmpleadoAsalariado`. Para imprimir la información completa de un `EmpleadoAsalariado`, la función `imprimir` de la clase derivada imprime el texto `"empleado asalariado: "` seguido de la información específica de la clase base `Empleado` (es decir, primer nombre, apellido paterno y número de seguro social) que se imprime al *invocar la función `imprimir` de la clase base*, usando el operador de resolución de ámbito (línea 42); éste es un buen ejemplo de reutilización de código. Sin el operador de resolución de ámbito, la llamada a `imprimir` provocaría *recursividad infinita*. Los resultados producidos por la función `imprimir` de `EmpleadoAsalariado` también contienen el salario semanal del empleado que se obtiene al invocar la función `obtenerSalarioSemanal` de la clase.

12.6.3 Creación de la clase derivada concreta `EmpleadoPorComision`

La clase `EmpleadoPorComision` (figuras 12.13 y 12.14) se deriva de la clase `Empleado` (figura 12.13, línea 9). Las implementaciones de las funciones miembro (figura 12.14) incluyen a un constructor (líneas 9 a 15) que recibe un primer nombre, un apellido paterno, un número de seguro social, un monto de ventas y una tarifa de comisión; funciones *establecer* (líneas 18 a 24 y 33 a 39) para asignar nuevos valores a los miembros de datos `tarifaComision` y `ventasBrutas`, respectivamente; funciones *obtener* (líneas 27 a 30 y 42 a 45) que obtienen sus valores; la función `ingresos` (líneas 48 a 51) para calcular los ingresos de un `EmpleadoPorComision` y la función `imprimir` (líneas 54 a 60), que imprime el tipo del empleado, a saber, `"empleado por comision: "`, y la información específica del empleado. El constructor pasa el primer nombre, apellido paterno y número de seguro social al constructor de `Empleado` (línea 11) para inicializar los miembros de datos `private` de `Empleado`. La función `imprimir` llama a la función `imprimir` de la clase base (línea 57) para mostrar la información específica de `Empleado`.

```
1  // Fig. 12.13: EmpleadoPorComision.h
2  // Clase EmpleadoPorComision derivada de Empleado.
3  #ifndef COMISION_H
4  #define COMISION_H
5
6  #include <string> // clase string estándar de C++
7  #include "Empleado.h" // definición de la clase Empleado
8
9  class EmpleadoPorComision : public Empleado
10 {
11 public:
12    EmpleadoPorComision( const std::string &, const std::string &,
13       const std::string &, double = 0.0, double = 0.0 );
```

Fig. 12.13 | Encabezado de la clase `EmpleadoPorComision` (parte 1 de 2).

```
14    virtual ~EmpleadoPorComision() { } // destructor virtual
15
16    void establecerTarifaComision( double ); // establece la tarifa de comisión
17    double obtenerTarifaComision() const; // devuelve la tarifa de comisión
18
19    void establecerVentasBrutas( double ); // establece el monto de ventas brutas
20    double obtenerVentasBrutas() const; // devuelve el monto de ventas brutas
21
22    // la palabra clave virtual indica la intención de sobrescribir
23    virtual double ingresos() const override; // calcula los ingresos
24    virtual void imprimir() const override; // imprime el objeto
25 private:
26    double ventasBrutas; // ventas brutas semanales
27    double tarifaComision; // porcentaje de comisión
28 }; // fin de la clase EmpleadoPorComision
29
30 #endif // COMISION_H
```

Fig. 12.13 | Encabezado de la clase `EmpleadoPorComision` (parte 2 de 2).

```
1  // Fig. 12.14: EmpleadoPorComision.cpp
2  // Definiciones de las funciones miembro de la clase EmpleadoPorComision.
3  #include <iostream>
4  #include <stdexcept>
5  #include "EmpleadoPorComision.h" // definición de la clase EmpleadoPorComision
6  using namespace std;
7
8  // constructor
9  EmpleadoPorComision::EmpleadoPorComision( const string &nombre,
10    const string &apellido, const string &nss, double ventas, double tarifa )
11    : Empleado( nombre, apellido, nss )
12 {
13    establecerVentasBrutas( ventas );
14    establecerTarifaComision( tarifa );
15 } // fin del constructor de EmpleadoPorComision
16
17 // establece el monto de ventas brutas
18 void EmpleadoPorComision::establecerVentasBrutas( double ventas )
19 {
20    if ( ventas >= 0.0 )
21       ventasBrutas = ventas;
22    else
23       throw invalid_argument( "Las ventas brutas deben ser >= 0.0" );
24 } // fin de la función establecerVentasBrutas
25
26 // devuelve el monto de ventas brutas
27 double EmpleadoPorComision::obtenerVentasBrutas() const
28 {
29    return ventasBrutas;
30 } // fin de la función obtenerVentasBrutas
31
```

Fig. 12.14 | Archivo de implementación de la clase `EmpleadoPorComision` (parte 1 de 2).

```
32 // establece la tarifa de comisión
33 void EmpleadoPorComision::establecerTarifaComision( double tarifa )
34 {
35    if ( tarifa > 0.0 && tarifa < 1.0 )
36       tarifaComision = tarifa;
37    else
38       throw invalid_argument( "La tarifa de comision debe ser > 0.0 y < 1.0" );
39 } // fin de la función establecerTarifaComision
40
41 // devuelve la tarifa de comisión
42 double EmpleadoPorComision::obtenerTarifaComision() const
43 {
44    return tarifaComision;
45 } // fin de la función obtenerTarifaComision
46
47 // calcula los ingresos; sobrescribe la función virtual pura ingresos en Empleado
48 double EmpleadoPorComision::ingresos() const
49 {
50    return obtenerTarifaComision() * obtenerVentasBrutas();
51 } // fin de la función ingresos
52
53 // imprime la información del EmpleadoPorComision
54 void EmpleadoPorComision::imprimir() const
55 {
56    cout << "empleado por comision: ";
57    Empleado::imprimir(); // reutilización de código
58    cout << "\nventas brutas: " << obtenerVentasBrutas()
59       << "; tarifa de comision: " << obtenerTarifaComision();
60 } // fin de la función imprimir
```

Fig. 12.14 | Archivo de implementación de la clase `EmpleadoPorComision` (parte 2 de 2).

12.6.4 Creación de la clase derivada concreta indirecta `EmpleadoBaseMasComision`

La clase `EmpleadoBaseMasComision` (figuras 12.15 y 12.16) hereda directamente de la clase `EmpleadoPorComision` (línea 9 de la figura 12.15), y por lo tanto es una clase derivada *indirecta* de la clase `Empleado`. Las implementaciones de las funciones miembro de la clase `EmpleadoBaseMasComision` incluyen a un constructor (líneas 9 a 15 de la figura 12.16) que recibe como argumentos un primer nombre, un apellido paterno, un número de seguro social, un monto de ventas, una tarifa de comisión *y* un salario base. Después pasa el primer nombre, apellido paterno, número de seguro social, monto de ventas y tarifa de comisión al constructor de `EmpleadoPorComision` (línea 12) para inicializar los miembros heredados. `EmpleadoBaseMasComision` también contiene una función *establecer* (líneas 18 a 24) para asignar un nuevo valor al miembro de datos `salarioBase` y una función *obtener* (líneas 27 a 30) para devolver el valor de `salarioBase`. La función `ingresos` (líneas 34 a 37) calcula los ingresos de un `EmpleadoBaseMasComision`. En la línea 36 de la función `ingresos` se llama a la función `ingresos` de la clase base `EmpleadoPorComision` para calcular la porción basada en comisiones de los ingresos del empleado. Éste es otro buen ejemplo de reutilización de código. La función `imprimir` de `EmpleadoBaseMasComision` (líneas 40 a 45) imprime el texto `"con salario base"`, seguido de los resultados de la función `imprimir` de la clase base `EmpleadoPorComision` (otro ejemplo de reutilización de código), y después el salario base. La salida resultante empieza con el texto `"con salario base empleado por comision: "` seguido del resto de la información de `EmpleadoBaseMasComision`. Recuerde que la función `imprimir` de `EmpleadoPorComision` muestra el primer nombre,

apellido paterno y número de seguro social del empleado al invocar la función `imprimir` de su clase base (es decir, `Empleado`); otro ejemplo más de reutilización de código. La función `imprimir` de `EmpleadoBaseMasComision` inicia una cadena de funciones que abarca *los tres niveles* de la jerarquía de `Empleado`.

```
1  // Fig. 12.15: EmpleadoBaseMasComision.h
2  // Clase EmpleadoBaseMasComision derivada de EmpleadoPorComision.
3  #ifndef BASEMAS_H
4  #define BASEMAS_H
5
6  #include <string> // clase string estándar de C++
7  #include "EmpleadoPorComision.h" // definición de la clase EmpleadoPorComision
8
9  class EmpleadoBaseMasComision : public EmpleadoPorComision
10 {
11 public:
12    EmpleadoBaseMasComision( const std::string &, const std::string &,
13       const std::string &, double = 0.0, double = 0.0, double = 0.0 );
14    virtual ~EmpleadoPorComision() { } // destructor virtual
15
16    void establecerSalarioBase( double ); // establece el salario base
17    double obtenerSalarioBase() const; // devuelve el salario base
18
19    // la palabra clave virtual indica el intento de sobrescribir
20    virtual double ingresos() const override; // calcula los ingresos
21    virtual void imprimir() const override; // imprime el objeto
22 private:
23    double salarioBase; // salario base por semana
24 }; // fin de la clase EmpleadoBaseMasComision
25
26 #endif // BASEMAS_H
```

Fig. 12.15 | Encabezado de `EmpleadoBaseMasComision`.

```
1  // Fig. 12.16: EmpleadoBaseMasComision.cpp
2  // Definiciones de las funciones miembro de EmpleadoBaseMasComision.
3  #include <iostream>
4  #include <stdexcept>
5  #include "EmpleadoBaseMasComision.h"
6  using namespace std;
7
8  // constructor
9  EmpleadoBaseMasComision::EmpleadoBaseMasComision(
10    const string &nombre, const string &apellido, const string &nss,
11    double ventas, double tarifa, double salario )
12    : EmpleadoPorComision( nombre, apellido, nss, ventas, tarifa )
13 {
14    establecerSalarioBase( salario ); // valida y almacena el salario base
15 } // fin del constructor de EmpleadoBaseMasComision
16
```

Fig. 12.16 | Archivo de implementación de la clase `EmpleadoBaseMasComision` (parte 1 de 2).

```
17 // establece el salario base
18 void EmpleadoBaseMasComision::establecerSalarioBase( double salario )
19 {
20    if ( salario >= 0.0 )
21       salarioBase = salario;
22    else
23       throw invalid_argument( "El salario debe ser >= 0.0" );
24 } // fin de la función establecerSalarioBase
25
26 // devuelve el salario base
27 double EmpleadoBaseMasComision::obtenerSalarioBase() const
28 {
29    return salarioBase;
30 } // fin de la función obtenerSalarioBase
31
32 // calcula los ingresos;
33 // sobrescribe la función virtual ingresos en EmpleadoPorComision
34 double EmpleadoBaseMasComision::ingresos() const
35 {
36    return obtenerSalarioBase() + EmpleadoPorComision::ingresos();
37 } // fin de la función ingresos
38
39 // imprime la información del EmpleadoBaseMasComision
40 void EmpleadoBaseMasComision::imprimir() const
41 {
42    cout << "con salario base ";
43    EmpleadoPorComision::imprimir(); // reutilización de código
44    cout << "; salario base: " << obtenerSalarioBase();
45 } // fin de la función imprimir
```

Fig. 12.16 | Archivo de implementación de la clase `EmpleadoBaseMasComision` (parte 2 de 2).

12.6.5 Demostración del procesamiento polimórfico

Para evaluar nuestra jerarquía de `Empleado`, el programa de la figura 12.17 crea un objeto de cada una de las tres clases concretas `EmpleadoAsalariado`, `EmpleadoPorComision` y `EmpleadoBaseMasComision`. El programa manipula estos objetos, primero con la *vinculación estática* y después en forma *polimórfica*, usando un `vector` de apuntadores `Empleado`. En las líneas 22 a 27 se crean objetos de cada una de las tres clases derivadas concretas de `Empleado`. En las líneas 32 a 38 se imprime la información y los ingresos de cada `Empleado`. La invocación a cada función miembro en las líneas 32 a 37 es un ejemplo de *vinculación estática*; *en tiempo de compilación*, puesto que estamos usando *manejadores de nombres* (y no *apuntadores* o *referencias* que podrían establecerse en *tiempo de ejecución*), el *compilador* puede identificar el tipo de cada objeto para determinar cuáles funciones de `imprimir` e `ingresos` se van a llamar.

```
1 // Fig. 12.17: fig12_17.cpp
2 // Procesamiento de objetos de clases derivadas de Empleado en forma
3 // individual y polimórfica, mediante el uso de la vinculación dinámica.
4 #include <iostream>
5 #include <iomanip>
6 #include <vector>
```

Fig. 12.17 | Programa controlador de la jerarquía de clases de `Empleado` (parte 1 de 4).

```
7  #include "Empleado.h"
8  #include "EmpleadoAsalariado.h"
9  #include "EmpleadoPorComision.h"
10 #include "EmpleadoBaseMasComision.h"
11 using namespace std;
12
13 void virtualViaApuntador( const Empleado * const ); // prototipo
14 void virtualViaReferencia( const Empleado & ); // prototipo
15
16 int main()
17 {
18    // establece el formato de salida de punto flotante
19    cout << fixed << setprecision( 2 );
20
21    // crea objetos de las clases derivadas
22    EmpleadoAsalariado empleadoAsalariado(
23       "John", "Smith", "111-11-1111", 800 );
24    EmpleadoPorComision empleadoPorComision(
25       "Sue", "Jones", "333-33-3333", 10000, .06 );
26    EmpleadoBaseMasComision empleadoBaseMasComision(
27       "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
28
29    cout << "Empleados procesados en forma individual, usando vinculacion
             estatica:\n\n";
30
31    // imprime la información de cada empleado y sus ingresos, usando vinculación
         estática
32    empleadoAsalariado.imprimir();
33    cout << "\nobtuvo $" << empleadoAsalariado.ingresos() << "\n\n";
34    empleadoPorComision.imprimir();
35    cout << "\nobtuvo $" << empleadoPorComision.ingresos() << "\n\n";
36    empleadoBaseMasComision.imprimir();
37    cout << "\nobtuvo $" << empleadoBaseMasComision.ingresos()
38       << "\n\n";
39
40    // crea un vector de tres apuntadores de la clase base
41    vector< Empleado * > empleados( 3 );
42
43    // inicializa el vector con apuntadores a objetos Empleado
44    empleados[ 0 ] = &empleadoAsalariado;
45    empleados[ 1 ] = &empleadoPorComision;
46    empleados[ 2 ] = &empleadoBaseMasComision;
47
48    cout << "Empleados procesados en forma polimorfica mediante vinculacion
             dinamica:\n\n";
49
50    // llama a virtualViaApuntador para imprimir la información de cada Empleado
51    // y a ingresos mediante el uso de la vinculación dinámica
52    cout << "Llamadas a funciones virtuales realizadas desde apuntadores de la
             clase base:\n\n";
53
54    for ( const Empleado *empleadoPtr : empleados )
55       virtualViaApuntador( empleadoPtr );
56
57    // llama a virtualViaReferencia para imprimir la información de cada Empleado
58    // y a ingresos mediante el uso de vinculación dinámica
59    cout << "Llamadas a funciones virtuales realizadas desde referencias de la
             clase base:\n\n";
```

Fig. 12.17 | Programa controlador de la jerarquía de clases de `Empleado` (parte 2 de 4).

```
60
61    for ( const Empleado *empleadoPtr : empleados )
62       virtualViaReferencia( *empleadoPtr ); // observe la desreferencia
63 } // fin de main
64
65 // llama a las funciones virtuales imprimir e ingresos de Empleado desde un
66 // apuntador de la clase base mediante la vinculación dinámica
67 void virtualViaApuntador( const Empleado * const claseBasePtr )
68 {
69    claseBasePtr->imprimir();
70    cout << "\nobtuvo $" << claseBasePtr->ingresos() << "\n\n";
71 } // fin de la función virtualViaApuntador
72
73 // llama a las funciones virtuales imprimir e ingresos de Empleado desde una
74 // referencia de la clase base mediante la vinculación dinámica
75 void virtualViaReferencia( const Empleado &claseBaseRef )
76 {
77    claseBaseRef.imprimir();
78    cout << "\nobtuvo $" << claseBaseRef.ingresos() << "\n\n";
79 } // fin de la función virtualViaReferencia
```

```
Empleados procesados en forma individual, usando vinculacion estatica:

empleado asalariado: John Smith
numero de seguro social: 111-11-1111
salario semanal: 800.00
obtuvo $800.00

empleado por comision: Sue Jones
numero de seguro social: 333-33-3333
ventas brutas: 10000.00; tarifa de comision: 0.06
obtuvo $600.00

con salario base empleado por comision: Bob Lewis
numero de seguro social: 444-44-4444
ventas brutas: 5000.00; tarifa de comision: 0.04; salario base: 300.00
obtuvo $500.00

Empleados procesados en forma polimorfica mediante vinculacion dinamica:

Llamadas a funciones virtuales realizadas desde apuntadores de la clase base:

empleado asalariado: John Smith
numero de seguro social: 111-11-1111
salario semanal: 800.00
obtuvo $800.000

empleado por comision: Sue Jones
numero de seguro social: 333-33-3333
ventas brutas: 10000.00; tarifa de comision: 0.06
obtuvo $600.00

con salario base empleado por comision: Bob Lewis
numero de seguro social: 444-44-4444
ventas brutas: 5000.00; tarifa de comision: 0.04; salario base: 300.00
obtuvo $500.00
```

Fig. 12.17 | Programa controlador de la jerarquía de clases de `Empleado` (parte 3 de 4).

```
Llamadas a funciones virtuales realizadas desde referencias de la clase base:

empleado asalariado: John Smith
numero de seguro social: 111-11-1111
salario semanal: 800.00
obtuvo $800.00

empleado por comision: Sue Jones
numero de seguro social: 333-33-3333
ventas brutas: 10000.00; tarifa de comision: 0.06
obtuvo $600.00

con salario base empleado por comision: Bob Lewis
numero de seguro social: 444-44-4444
ventas brutas: 5000.00; tarifa de comision: 0.04; salario base: 300.00
obtuvo $500.00
```

Fig. 12.17 | Programa controlador de la jerarquía de clases de `Empleado` (parte 4 de 4).

En la línea 41 se crea el vector `empleados`, que contiene tres apuntadores `Empleado`. En la línea 44 se orienta `empleados[0]` al objeto `empleadoAsalariado`. En la línea 45 se orienta `empleados[1]` al objeto `empleadoPorComision`. En la línea 46 se orienta `empleados[2]` al objeto `empleadoBaseMasComision`. El compilador permite estas asignaciones, ya que un `EmpleadoAsalariado` *es un* `Empleado`, un `EmpleadoPorComision` *es-un* `Empleado` y un `EmpleadoBaseMasComision` *es-un* `Empleado`. Por lo tanto, podemos asignar las direcciones de los objetos `EmpleadoAsalariado`, `EmpleadoPorComision` y `EmpleadoBaseMasComision` a los apuntadores de la clase base `Empleado` (aun cuando `Empleado` sea una clase *abstracta*).

El ciclo en las líneas 54 y 55 recorre el `vector empleados` e invoca a la función `virtualViaApuntador` (líneas 67 a 71) para cada elemento en `empleados`. La función `virtualViaApuntador` recibe en el parámetro `claseBasePtr` la dirección almacenada en un elemento de `empleados`. Cada llamada a `virtualViaApuntador` utiliza a `claseBasePtr` para invocar a las funciones `virtual imprimir` (línea 69) e `ingresos` (línea 70). La función `virtualViaApuntador` *no* contiene *ninguna* información de los tipos `EmpleadoAsalariado`, `EmpleadoPorComision` o `EmpleadoBaseMasComision`. La función *sólo* sabe acerca del tipo de su clase base `Empleado`**.** Por lo tanto, el compilador *no puede saber* cuáles funciones de la clase concreta debe llamar a través de `claseBasePtr`. Aún en tiempo de ejecución, cada invocación a una función virtual llama *correctamente* a la función en el objeto al que apunta `claseBasePtr` en ese momento. La salida ilustra que *sin duda se invocan las funciones apropiadas para cada clase*, y que se muestra la información apropiada de cada objeto. Por ejemplo, el salario semanal se muestra para el `EmpleadoAsalariado`, y las ventas brutas se muestran para `EmpleadoPorComision` y `EmpleadoBaseMasComision`. Además, al obtener los ingresos de cada `Empleado` en forma polimórfica en la línea 70 se producen los mismos resultados que obtener los ingresos de esos empleados mediante la *vinculación estática* en las líneas 33, 35 y 37. Todas las llamadas a las funciones `virtual imprimir` e `ingresos` se resuelven en *tiempo de ejecución* con la *vinculación dinámica*.

Por último, el ciclo en las líneas 61 y 62 recorre a `empleados` e invoca la función `virtualViaReferencia` (líneas 75 a 79) para cada elemento en el `vector`. La función `virtualViaReferencia` recibe en su parámetro `claseBaseRef` (de tipo `const Empleado &`) una *referencia* al objeto, que se obtiene al *desreferenciar el apuntador* almacenado en cada elemento de `empleados` (línea 62). Cada llamada a `virtualViaReferencia` invoca a las funciones `virtual imprimir` (línea 77) e `ingresos` (línea 78) a través de la referencia `claseBaseRef` para demostrar que *el procesamiento polimórfico ocurre también con las referencias de la clase base*. Cada invocación a una función `virtual` llama a la función en el objeto

al que `claseBaseRef` hace referencia en tiempo de ejecución. Éste es otro ejemplo de *vinculación dinámica*. Los resultados producidos usando referencias de la clase base son idénticos a los resultados que se producen usando apuntadores de la clase base.

12.7 (Opcional) Polimorfismo, funciones virtuales y vinculación dinámica "detrás de las cámaras"

C++ facilita la programación del polimorfismo. Evidentemente, es posible programar para el polimorfismo en los lenguajes no orientados a objetos como C, pero para ello se requieren manipulaciones de apuntadores complejas y potencialmente peligrosas. Esta sección habla acerca de cómo C++ puede implementar el polimorfismo, las funciones `virtual` y la vinculación dinámica de manera interna. Esto proporcionará al lector una sólida comprensión de la forma en que realmente funcionan estas herramientas. Lo que es más importante, le ayudará a apreciar la *sobrecarga* del polimorfismo; en términos de *consumo de memoria* y *tiempo de procesador* adicionales. Esto le ayudará a determinar cuándo usar el polimorfismo y cuándo evitarlo. Las clases de la Biblioteca estándar de C++ como `array` y `vector` se implementan *sin* polimorfismo ni funciones `virtual` para evitar la sobrecarga asociada en tiempo de ejecución y lograr un rendimiento óptimo.

Primero explicaremos las estructuras de datos que el compilador genera en *tiempo de compilación* para dar soporte al polimorfismo en tiempo de ejecución. Veremos que el polimorfismo se lleva a cabo a través de tres niveles de apuntadores (es decir, *triple indirección*). Después mostraremos cómo un programa en ejecución utiliza estas estructuras de datos para ejecutar funciones `virtual` y lograr la *vinculación dinámica* asociada con el polimorfismo. Nuestra discusión explica una *posible* implementación; esto no es un requerimiento del lenguaje.

Cuando C++ compila una clase que tiene una o más funciones `virtual`, genera una **tabla de funciones virtuales** (***vtable***) para esa clase. La *vtable* contiene apuntadores a las funciones `virtual` de la clase. Así como el nombre de un arreglo integrado contiene la dirección en memoria del primer elemento del arreglo, un **apuntador a una función** contiene la dirección inicial en memoria del código que realiza la tarea de la función. Un programa en ejecución utiliza la *vtable* para seleccionar la implementación de la función apropiada cada vez que se llama a una función `virtual` de esa clase. La columna de más a la izquierda de la figura 12.18 ilustra las *vtables* para las clases `Empleado`, `EmpleadoAsalariado`, `EmpleadoPorComision` y `EmpleadoBaseMasComision`.

Vtable *de la clase* `Empleado`

En la *vtable* para la clase `Empleado`, el apuntador a la primera función se establece en `0` (es decir, `nullptr`), ya que la función `ingresos` es una función `virtual` *pura*, y por lo tanto *carece de una implementación*. El apuntador a la segunda función apunta a la función `imprimir`, la cual muestra el nombre completo y número de seguro social del empleado. [*Nota:* hemos abreviado la salida de cada función `imprimir` en esta figura para conservar espacio]. Cualquier clase que tenga uno o más apuntadores nulos en su *vtable* es una clase abstracta. Las clases sin apuntadores nulos en su *vtable* (como `EmpleadoAsalariado`, `EmpleadoPorComision` y `EmpleadoBaseMasComision`) son clases *concretas*.

Vtable *de la clase* `EmpleadoAsalariado`

La clase `EmpleadoAsalariado` sobrescribe a la función `ingresos` para regresar el salario semanal del empleado, de manera que el apuntador a función apunta a la función `ingresos` de la clase `EmpleadoAsalariado`. Esta clase también sobrescribe a `imprimir`, por lo que el apuntador a la función correspondiente apunta a la función miembro de `EmpleadoAsalariado` que imprime el texto `"empleado asalariado: "` seguido del nombre del empleado, su número de seguro social y su salario semanal.

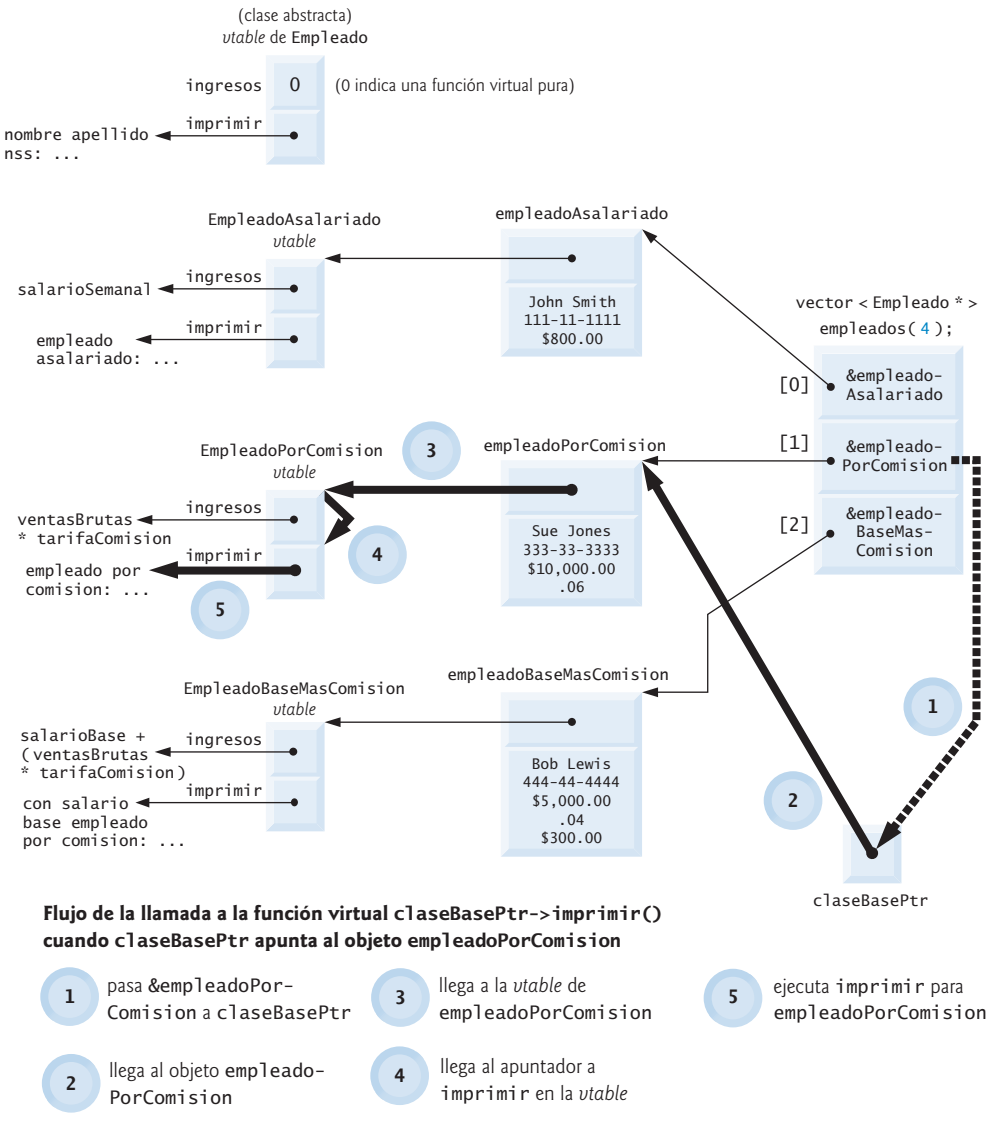


Fig. 12.18 | Cómo funcionan las llamadas a funciones `virtual`.

Vtable *de la clase* `EmpleadoPorComision`

El apuntador a la función `ingresos` en la *vtable* para la clase `EmpleadoPorComision` apunta a la función `ingresos` de `EmpleadoAsalariado` que devuelve las ventas brutas del empleado, multiplicadas por la tarifa de comisión. El apuntador a la función `imprimir` apunta a la versión de la función correspondiente a `EmpleadoPorComision`, la cual imprime el tipo de empleado, el nombre, número de seguro social, tarifa de comisión y ventas brutas. Ambas funciones sobrescriben a las funciones en la clase `Empleado`.

Vtable *de la clase* `EmpleadoBaseMasComision`

El apuntador a la función `ingresos` en la *vtable* para la clase `EmpleadoBaseMasComision` apunta a la función `ingresos` de `EmpleadoBaseMasComision`, la cual devuelve el salario base del empleado más las ventas brutas, multiplicadas por la tarifa de comisión. El apuntador a la función `imprimir` apunta a la versión de la función correspondiente a `EmpleadoBaseMasComision`, la cual imprime el salario base del empleado más el tipo, nombre, número de seguro social, tarifa de comisión y ventas brutas. Ambas funciones sobrescriben a las funciones en la clase `EmpleadoPorComision`.

Heredar funciones `virtual` *concretas*

En nuestro caso de estudio de `Empleado`, cada clase *concreta* proporciona su propia implementación para las funciones `virtual ingresos` e `imprimir`. El lector aprendió que cada clase que hereda *directamente* de la clase base abstracta `Empleado` *debe implementar a* `ingresos` para poder ser una clase *concreta*, ya que `ingresos` es una función `virtual` pura. Sin embargo, estas clases no necesitan implementar la función `imprimir` para considerarse concretas; `imprimir` *no es* una función `virtual` *pura* y las clases derivadas pueden heredar la implementación de `imprimir` correspondiente a la clase `Empleado`. Lo que es más, la clase `EmpleadoBaseMasComision` *no* necesita implementar ni la función `imprimir` ni la función `ingresos`; ambas implementaciones de las funciones se pueden heredar de la clase concreta `EmpleadoPorComision`. Si una clase en nuestra jerarquía fuera a heredar las implementaciones de las funciones de esta forma, los apuntadores de la *vtable* para estas funciones simplemente apuntarían a la implementación de la función que se vaya a heredar. Por ejemplo, si `EmpleadoBaseMasComision` no sobrescribiera a `ingresos`, el apuntador a la función `ingresos` en la *vtable* para la clase `EmpleadoBaseMasComision` apuntaría a la misma función `ingresos` que a la que apunta la *vtable* para la clase `EmpleadoPorComision`.

Tres niveles de apuntadores para implementar el polimorfismo

El polimorfismo se lleva a cabo a través de una elegante estructura de datos que implica *tres niveles de apuntadores*. Hemos hablado sobre un nivel: los apuntadores a las funciones en la *vtable*. Estos apuntan a las funciones actuales que se ejecutan cuando se invoca a una función `virtual`.

Ahora consideraremos el segundo nivel de apuntadores. *Cada vez que se instancia un objeto de una clase con una o más funciones* `virtual`*, el compilador adjunta al objeto un apuntador a la* vtable *para esa clase*. Por lo general, este apuntador está en la parte frontal del objeto, pero no se requiere implementarlo de esa forma. En la figura 12.18, estos apuntadores se asocian con los objetos creados en la figura 12.17 (un objeto para cada uno de los tipos `EmpleadoAsalariado`, `EmpleadoPorComision` y `EmpleadoBaseMasComision`). El diagrama muestra los valores de cada uno de los miembros de datos del objeto. Por ejemplo, el objeto `empleadoAsalariado` contiene un apuntador a la *vtable* de `EmpleadoAsalariado`; el objeto también contiene los valores `John Smith`, `111-11-1111` y `$800.00`.

El tercer nivel de apuntadores simplemente contiene los manejadores para los objetos que reciben las llamadas a las funciones `virtual`. Los manejadores en este nivel también pueden ser *referencias*. La figura 12.18 describe el `vector empleados` que contiene *apuntadores* `Empleado`.

Ahora veamos cómo se ejecuta una llamada a una función `virtual` ordinaria. Considere la llamada `claseBasePtr->imprimir()` en la función `virtualViaApuntador` (línea 69 de la figura 12.17). Suponga que `claseBasePtr` contiene `empleados[ 1 ]` (es decir, la dirección del objeto `empleadoPorComision` en `empleados`). Cuando el compilador compila esta instrucción, determina que la llamada sin duda se está realizando a través de un *apuntador de la clase base*, y que `imprimir` es una función `virtual`.

El compilador determina que `imprimir` es la *segunda* entrada en cada una de las *vtables*. Para localizar esta entrada, el compilador observa que necesitará omitir la primera entrada. Por ende, el compilador compila un **desplazamiento** en la tabla de apuntadores de código objeto de lenguaje máquina para encontrar el código que ejecutará la llamada a la función `virtual`. El tamaño en bytes del desplazamiento depende del número de bytes que se utilicen para representar a un apuntador de función en una

plataforma específica. Por ejemplo, en una plataforma de 32 bits, un apuntador se almacena comúnmente en cuatro bytes, mientras que en una plataforma de 64 bits un apuntador se almacena por lo general en ocho bytes. Para esta discusión vamos a suponer que son cuatro bytes.

El compilador genera código que realiza las siguientes operaciones [*Nota:* los números en la lista corresponden a los números con círculos en la figura 12.18]:

1. Seleccionar la *i*-ésima entrada de `empleados` (en este caso, la dirección del objeto `empleadoPorComision`), y pasarla como un argumento a la función `virtualViaApuntador`. Esto establece el parámetro `claseBasePtr` para que apunte a `empleadoPorComision`.
2. *Desreferenciar* ese apuntador para llegar al objeto `empleadoPorComision`; que, como recordará, empieza con un apuntador a la *vtable* de `EmpleadoPorComision`.
3. *Desreferenciar* el apuntador de la *vtable* de `empleadoPorComision` para llegar a la *vtable* de `EmpleadoPorComision`.
4. Omitir el desplazamiento de cuatro bytes para seleccionar el apuntador a la función `imprimir`.
5. *Desreferenciar* el apuntador a la función `imprimir` para formar el "nombre" de la función actual que se va a ejecutar, y usar el operador de llamada a función `()` para ejecutar la función `imprimir` apropiada, que en este caso imprime el tipo, nombre, número de seguro social, ventas brutas y tarifa de comisión.

Las estructuras de datos de la figura 12.18 pueden parecer complejas, pero esta complejidad es administrada por el compilador y se *oculta* al programador, lo cual hace de la programación polimórfica un proceso simple. Las operaciones de desreferenciamiento de apuntadores y los accesos a memoria que ocurren en cada llamada a una función `virtual` requieren cierto tiempo de ejecución adicional. Las *vtables* y los apuntadores a una *vtable* que se agregan a los objetos requieren cierta memoria adicional.

Tip de rendimiento 12.1

El polimorfismo, según su implementación común con funciones `virtual` y vinculación dinámica en C++, es eficiente. Los programadores pueden utilizar estas herramientas con un impacto nominal en el rendimiento.

Tip de rendimiento 12.2

Las funciones virtuales y la vinculación dinámica permiten la programación polimórfica como una alternativa a la programación con la lógica de `switch`. Por lo común, los compiladores optimizadores generan código polimórfico que se ejecuta con la misma eficiencia que la lógica basada en `switch`, codificada a mano. La sobrecarga del polimorfismo es aceptable para la mayoría de las aplicaciones. Pero en ciertas situaciones (aplicaciones de tiempo real con requerimientos estrictos en cuanto al rendimiento, por ejemplo), la sobrecarga del polimorfismo puede ser demasiado alta.

12.8 Caso de estudio: sistema de nómina mediante el uso de polimorfismo e información de tipos en tiempo de ejecución con conversión descendente, `dynamic_cast`, `typeid` y `type_info`

En el enunciado del problema anterior al principio de la sección 12.6 vimos que, para el periodo de pago actual, nuestra empresa ficticia ha decidido recompensar a cada `EmpleadoBaseMasComision`, agregando un 10 por ciento a sus salarios base. Al procesar objetos `Empleado` mediante el polimorfismo en la sección 12.6.5, no tuvimos que preocuparnos por los "detalles específicos". Ahora, sin embargo, para ajustar los salarios base de cada `EmpleadoBaseMasComision` tenemos que determinar

el *tipo específico* de cada objeto `Empleado` *en tiempo de ejecución*, y después actuar en forma apropiada. Esta sección demuestra las poderosas herramientas de la **información de tipos en tiempo de ejecución** (**RTTI**) y la **conversión dinámica de tipos**, las cuales permiten a un programa determinar el tipo de un objeto en tiempo de ejecución, y actuar sobre ese objeto en forma acorde.[1]

En la figura 12.19 se utiliza la jerarquía de `Empleado` desarrollada en la sección 12.6, y se incrementa en un 10 por ciento el salario base de cada `EmpleadoBaseMasComision`. En la línea 21 se declara el `vector` de tres elementos llamado `empleados`, que almacena apuntadores a objetos `Empleado`. En las líneas 24 a 29 se llena el `vector` con las *direcciones de los objetos asignados en forma dinámica* de las clases `EmpleadoAsalariado` (figuras 12.11 y 12.12), `EmpleadoPorComision` (figuras 12.13 y 12.14) y `EmpleadoBaseMasComision` (figuras 12.15 y 12.16). La instrucción `for` en las líneas 32 a 52 itera a través del `vector empleados` y muestra la información de cada `Empleado`, para lo cual invoca a la función miembro `imprimir` (línea 34). Recuerde que como `imprimir` se declara `virtual` en la *clase base* `Empleado`, el sistema invoca a la función `imprimir` del objeto de la *clase derivada* apropiada.

```
1  // Fig. 12.19: fig12_19.cpp
2  // Demostración de la conversión descendente y la información de tipos en tiempo
      de ejecución.
3  // NOTA: Tal vez necesite habilitar la RTTI en su compilador
4  // para poder compilar esta aplicación.
5  #include <iostream>
6  #include <iomanip>
7  #include <vector>
8  #include <typeinfo>
9  #include "Empleado.h"
10 #include "EmpleadoAsalariado.h"
11 #include "EmpleadoPorComision.h"
12 #include "EmpleadoBaseMasComision.h"
13 using namespace std;
14
15 int main()
16 {
17    // establece el formato de salida de punto flotante
18    cout << fixed << setprecision( 2 );
19
20    // crea un vector de tres apuntadores de la clase base
21    vector < Empleado * > empleados( 3 );
22
23    // inicializa el vector con varios tipos de objetos Empleado
24    empleados[ 0 ] = new EmpleadoAsalariado(
25       "John", "Smith", "111-11-1111", 800 );
26    empleados[ 1 ] = new EmpleadoPorComision(
27       "Sue", "Jones", "333-33-3333", 10000, .06 );
28    empleados[ 2 ] = new EmpleadoBaseMasComision(
29       "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
30
```

Fig. 12.19 | Demostración de la conversión descendente y la información de tipos en tiempo de ejecución (parte 1 de 2).

1 Algunos compiladores requieren que se habilite la RTTI antes de poder usarla en un programa. Los compiladores que usamos para probar los ejemplos de este libro (GNU C++ 4.7, Visual C++ 2012 y Xcode 4.5 LLVM) habilitan la RTTI de manera predeterminada.

```
31     // procesa en forma polimórfica cada elemento en el vector empleados
32     for ( Empleado *empleadoPtr : empleados )
33     {
34        empleadoPtr->imprimir(); // imprime la información del empleado
35        cout << endl;
36
37        // intenta conversión descendente del apuntador
38        EmpleadoBaseMasComision *derivadaPtr =
39           dynamic_cast < EmpleadoBaseMasComision * >( empleadoPtr );
40
41        // determina si el elemento apunta a un EmpleadoBaseMasComision
42        if ( derivadaPtr != nullptr ) // true para una relación "es un"
43        {
44           double salarioBaseAnterior = derivadaPtr->obtenerSalarioBase();
45           cout << "salario base anterior: $" << salarioBaseAnterior << endl;
46           derivadaPtr->establecerSalarioBase( 1.10 * salarioBaseAnterior );
47           cout << "el nuevo salario base con aumento del 10% es: $"
48              << derivadaPtr->obtenerSalarioBase() << endl;
49        } // fin de if
50
51        cout << "obtuvo $" << empleadoPtr->ingresos() << "\n\n";
52     } // fin de for
53
54     // libera los objetos a los que apuntan los elementos del vector
55     for ( const Empleado *empleadoPtr : empleados )
56     {
57        // imprime el nombre de la clase
58        cout << "eliminando objeto de "
59           << typeid( *empleadoPtr ).name() << endl;
60
61        delete empleadoPtr;
62     } // fin de for
63  } // fin de main
```

```
empleado asalariado: John Smith
numero de seguro social: 111-11-1111
salario semanal: 800.00
obtuvo $800.00

empleado por comision: Sue Jones
numero de seguro social: 333-33-3333
ventas brutas: 10000.00; tarifa de comision: 0.06
obtuvo $600.00

con salario base empleado por comision: Bob Lewis
numero de seguro social: 444-44-4444
ventas brutas: 5000.00; tarifa de comision: 0.04; salario base: 300.00
salario base anterior: $300.00
el nuevo salario base con aumento del 10% es: $330.00
obtuvo $530.00

eliminando objeto de class EmpleadoAsalariado
eliminando objeto de class EmpleadoPorComision
eliminando objeto de class EmpleadoBaseMasComision
```

Fig. 12.19 | Demostración de la conversión descendente y la información de tipos en tiempo de ejecución (parte 2 de 2).

Determinar el tipo de un objeto con `dynamic_cast`

En este ejemplo, al encontrarnos con un objeto `EmpleadoBaseMasComision`, deseamos incrementar su salario base en un 10 por ciento. Como procesamos a los empleados de manera polimórfica, no podemos (con las técnicas que hemos aprendido hasta ahora) estar seguros de qué tipo de `Empleado` se está manipulando en cualquier momento dado. Esto crea un problema, ya que *debemos* identificar los empleados `EmpleadoBaseMasComision` al encontrarlos, para que puedan recibir el aumento del 10 por ciento en su salario. Para lograr esto, utilizamos el operador **`dynamic_cast`** (línea 39) para determinar si el tipo de cada objeto `Empleado` es `EmpleadoBaseMasComision`. Ésta es la operación de *conversión descendente* a la que hicimos referencia en la sección 12.3.3. En las líneas 38 y 39 se realiza una *conversión descendente dinámica* de `empleadoPtr`, del tipo `Empleado *` al tipo `EmpleadoBaseMasComision *`. Si `empleadoPtr` apunta a un objeto que *es un* objeto `EmpleadoBaseMasComision`, entonces la *dirección* de ese objeto se asigna al apuntador `comisionPtr` de la clase derivada; en caso contrario, se asigna `nullptr a derivadaPtr`. Cabe mencionar que aquí se *requiere* `dynamic_cast` en vez de `static_cast` para realizar la comprobación de tipos en el objeto subyacente; una conversión `static_cast` sólo convertiría el `Empleado *` a un `EmpleadoBaseMasComision *`, sin importar el tipo del objeto subyacente. Con una `static_cast`, el programa intentaría aumentar el salario base de *todos* los objetos `Empleado`, lo que produciría un comportamiento indefinido para cada objeto que no sea `EmpleadoBaseMasComision`.

Si el valor devuelto por el operador `dynamic_cast` en las líneas 38 a 39 *no es* `nullptr`, el objeto *es* del tipo correcto, y la instrucción `if` (líneas 42 a 49) realiza el procesamiento especial requerido para el objeto `EmpleadoBaseMasComision`. En las líneas 44, 46 y 48 se invocan las funciones `obtenerSalarioBase` y `establecerSalarioBase` de `EmpleadoBaseMasComision` para obtener y actualizar el salario del `empleado`.

Calcular los ingresos del `Empleado` *actual*

En la línea 51 se invoca a la función miembro `ingresos` en el objeto al que apunta `empleadoPtr`. Recuerde que `ingresos` se declara como `virtual` en la clase base, por lo que el programa invoca a la función `ingresos` del objeto de la clase derivada; otro ejemplo de *vinculación dinámica*.

Mostrar el tipo de un `Empleado`

En las líneas 55 a 62 se muestra el *tipo del objeto* de cada empleado y se utiliza el operador `delete` para desasignar la memoria dinámica a la que apunta cada elemento `vector`. El operador **`typeid`** (línea 59) devuelve una *referencia* a un objeto de la clase **`type_info`** que contiene la información acerca del tipo de su operando, incluyendo el nombre de ese tipo. Al invocarse, la función miembro **`name`** de `type_info` (línea 59) devuelve una cadena basada en apuntador que contiene el *nombre del tipo* (por ejemplo, `"class EmpleadoBaseMasComision"`) del argumento que se pasa a `typeid`. Para usar `typeid`, el programa debe incluir el encabezado **`<typeinfo>`** (línea 8).

Tip de portabilidad 12.1

La cadena devuelta por la función miembro `name` *de* `type_info` *puede variar de un compilador a otro.*

Errores de compilación que evitamos al usar `dynamic_cast`

Evitamos varios errores de compilación en este ejemplo al realizar una *conversión descendente* de un apuntador `Empleado` a un apuntador `EmpleadoBaseMasComision` (líneas 38 y 39). Si eliminamos el operador `dynamic_cast` de la línea 39 y tratamos de asignar el apuntador `Empleado` actual directamente al apuntador `derivadaPtr` de `EmpleadoBaseMasComision`, recibiremos un error de compilación. C++ *no* permite a un programa asignar un apuntador de la clase base a un apuntador de la clase derivada,

debido a que la relación *es-un* no se aplica; un `EmpleadoPorComision` *no* es un `EmpleadoBaseMasComision`. La relación *es-un* se aplica sólo entre la clase derivada y sus clases base, no viceversa.

De manera similar, si en las líneas 44, 46 y 48 se utilizara el apuntador de la clase base actual de `empleados` en vez de usar el apuntador `derivadaPtr` de la clase derivada para invocar a las funciones `obtenerSalarioBase` y `establecerSalarioBase` que sólo pertenecen a la clase derivada, recibiríamos un error de compilación en cada una de estas líneas. Como vimos en la sección 12.3.3, *no* está permitido tratar de invocar las *funciones que sólo pertenecen a la clase derivada* a través de un *apuntador de la clase base*. Aunque las líneas 44, 46 y 48 se ejecutan sólo si `comisionPtr` no es `nullptr` (es decir, si *puede* realizarse la conversión), *no podemos* tratar de invocar las funciones `obtenerSalarioBase` y `establecerSalarioBase` de la clase derivada `EmpleadoBaseMasComision` en el apuntador de la clase base `Empleado`. Recuerde que, al utilizar un apuntador de la clase base `Empleado`, sólo podemos invocar las funciones que se encuentran en la clase base `Empleado: ingresos`, `imprimir` y las funciones *obtener* y *establecer* de `Empleado`.

12.9 Conclusión

En este capítulo hablamos sobre el polimorfismo, que nos permite "programar en forma general" en vez de "programar en forma específica", y mostramos cómo esto hace a los programas más extensibles. Empezamos con un ejemplo sobre cómo el polimorfismo permitiría a un administrador de pantalla mostrar varios objetos "espaciales". Después demostramos cómo se pueden orientar los apuntadores de clases base y de clases derivadas a objetos de clases base y de clases derivadas. Dijimos que es natural orientar apuntadores de clase base a objetos de clase base, al igual que orientar apuntadores de clase derivada a objetos de clase derivada. También es natural orientar apuntadores de clase base a apuntadores de clase derivada, ya que un objeto de una clase derivada *es un* objeto de su clase base. El lector aprendió por qué es peligroso orientar apuntadores de clase derivada a objetos de clase base, y por qué el compilador no permite dichas asignaciones. Presentamos las funciones `virtual`, las cuales permiten llamar a las funciones apropiadas cuando se hace referencia a objetos en varios niveles de una jerarquía de herencia (en tiempo de ejecución) mediante apuntadores o referencias de clase base. A esto se le conoce como vinculación dinámica o postergada. Hablamos sobre los destructores `virtual`, y cómo aseguran que se ejecuten todos los destructores apropiados en una jerarquía de herencia en un objeto de clase derivada, cuando se elimina ese objeto mediante un apuntador o referencia a la clase base. Después hablamos sobre las funciones `virtual` puras y las clases abstractas (clases con una o más funciones `virtual` puras). También aprendió que las clases abstractas no se pueden utilizar para instanciar objetos, mientras que las clases concretas si se pueden usar. Después demostramos el uso de clases abstractas en una jerarquía de herencia. El lector aprendió cómo trabaja el polimorfismo "detrás de las cámaras" con *vtables* que el compilador crea. Hablamos sobre la información de tipos en tiempo de ejecución (RTTI) y la conversión dinámica para determinar el tipo de un objeto en tiempo de ejecución y actuar sobre ese objeto de manera acorde. También usamos el operador `typeid` para obtener un objeto `type_info` que contiene la información del tipo de un objeto dado.

En el siguiente capítulo hablaremos sobre muchas de las herramientas de E/S de C++ y demostraremos varios manipuladores de flujo que realizan diversas tareas de formato.

Resumen

Sección 12.1 Introducción

- El polimorfismo (pág. 518) nos permite "programar en forma general" en vez de "programar en forma específica".
- El polimorfismo nos permite escribir programas que procesen objetos de clases que sean parte de la misma jerarquía de clases, como si todos fueran objetos de la clase base de la jerarquía.

- Con el polimorfismo, podemos diseñar e implementar sistemas que sean fácilmente extensibles; pueden agregarse nuevas clases con poca (o ninguna) modificación a las porciones generales del programa. Las únicas partes de un programa que deben alterarse para dar cabida a nuevas clases son aquellas que requieren un conocimiento directo de las nuevas clases que agregamos a la jerarquía.

Sección 12.2 Introducción al polimorfismo: videojuego polimórfico

- Con el polimorfismo, una función puede ocasionar que ocurran distintas acciones, dependiendo del tipo del objeto en el que se invoca la función.
- Esto hace posible diseñar e implementar sistemas más extensibles. Los programas pueden escribirse para procesar objetos de tipos que tal vez no existían cuando el programa estaba en desarrollo.

Sección 12.3 Relaciones entre los objetos en una jerarquía de herencia

- C++ permite el polimorfismo: la habilidad de que los objetos de distintas clases relacionadas por la herencia respondan de manera distinta a la misma llamada a una función miembro.
- El polimorfismo se implementa a través de funciones `virtual` (pág. 526) y vinculación dinámica (pág. 527).
- Cuando se utiliza un apuntador o referencia de la clase base para llamar a una función `virtual`, C++ selecciona la función sobrescrita correcta en la clase derivada apropiada, asociada con el objeto.
- Si una función `virtual` se llama mediante la referencia a un objeto específico por su nombre y mediante el uso del operador punto de selección de miembros, la referencia se resuelve en tiempo de compilación (a esto se le conoce como vinculación estática; pág. 527); la función `virtual` que se llama es la que está definida para la clase de ese objeto específico.
- Las clases derivadas pueden sobrescribir a una función `virtual` de la clase base si es necesario, pero si no, se utiliza la implementación de la clase base.
- Debemos declarar el destructor de la clase base como `virtual` (pág. 532) si la clase contiene funciones `virtual`. Esto hace a todos los destructores de la clase derivada `virtual`, aun cuando no tengan el mismo nombre que el destructor de la clase base. Si un objeto en la jerarquía se destruye de manera explícita al aplicar el operador `delete` a un apuntador de la clase base que apunte a un objeto de la clase derivada, se llama al destructor para la clase apropiada. Después de que se ejecuta el destructor de una clase derivada, se ejecutan los destructores para todas las clases base de esa clase hacia arriba en la jerarquía.

Sección 12.4 Tipos de campos e instrucciones `switch`

- La programación polimórfica con funciones `virtual` puede eliminar la necesidad de la lógica de `switch`. El programador puede usar el mecanismo de funciones `virtual` para realizar la lógica equivalente de manera automática, evitando con ello los tipos de errores que se asocian comúnmente con la lógica de `switch`.

Sección 12.5 Clases abstractas y funciones `virtual` *puras*

- Las clases abstractas (pág. 533) se utilizan sólo como clases base, por lo que nos referimos a ellas como clases base abstractas (pág. 533). No se pueden instanciar objetos de una clase base abstracta.
- Las clases a partir de las cuales se instancian objetos se conocen como clases concretas (pág. 533).
- Una clase se hace abstracta al declarar una o más funciones `virtual` puras (pág. 534) con especificadores puros (`= 0`) en sus declaraciones.
- Si una clase se deriva de una clase con una función `virtual` pura y esa clase derivada no proporciona una definición para esa función `virtual` pura, entonces esa función `virtual` pura sigue siendo pura en la clase derivada. En consecuencia, la clase derivada es también una clase abstracta.
- Aunque no podemos instanciar objetos de clases base abstractas, podemos declarar apuntadores y referencias a objetos de clases base abstractas. Dichos apuntadores y referencias se pueden utilizar para permitir manipulaciones polimórficas de los objetos de clases derivadas que se instancian de clases derivadas concretas.

Sección 12.7 (Opcional) Polimorfismo, funciones virtuales y vinculación dinámica "detrás de las cámaras"

- La vinculación dinámica requiere que en tiempo de ejecución, la llamada a una función miembro virtual se dirija a la versión de la función `virtual` apropiada para esa clase. Una tabla de funciones `virtual`, conocida como *vtable* (pág. 550) se implementa como un arreglo que contiene apuntadores a funciones. Cada clase con funciones `virtual` tiene una *vtable*. Para cada función `virtual` en la clase, la *vtable* tiene una entrada que contiene un apuntador a función que apunta a la versión de la función `virtual` que se debe usar para un objeto de esa clase. La función `virtual` a utilizar para una clase específica podría ser la función definida en esa clase, o podría ser una función heredada ya sea de manera directa o indirecta de una clase base en un nivel más alto en la jerarquía.
- Cuando una clase base proporciona una función miembro `virtual`, las clases derivadas pueden sobrescribir a la función `virtual`, pero no tienen que sobrescribirla.
- Cada objeto de una clase con funciones `virtual` contiene un apuntador a la *vtable* para esa clase. Cuando se hace una llamada a una función desde un apuntador de la clase base a un objeto de la clase derivada, se obtiene el apuntador a la función apropiada en la *vtable* y se desreferencia para completar la llamada en tiempo de ejecución.
- Cualquier clase que tenga uno o más apuntadores `nullptr` en su *vtable* es una clase abstracta. Las clases sin apuntadores `nullptr` en su *vtable* son clases concretas.
- Se agregan nuevos tipos de clases a los sistemas con regularidad y se acomodan mediante la vinculación dinámica.

Sección 12.8 Caso de estudio: sistema de nómina mediante el uso de polimorfismo e información de tipos en tiempo de ejecución con conversión descendente, ***`dynamic_cast`, `typeid` y `type_info`***

- El operador `dynamic_cast` (pág. 554) comprueba el tipo del objeto al que apunta el apuntador, y determina si este tipo tiene una relación *es un* con el tipo al que se está convirtiendo el apuntador. De ser así, `dynamic_cast` devuelve la dirección del objeto. Si no, `dynamic_cast` devuelve `nullptr`.
- El operador `typeid` (pág. 556) devuelve una referencia a un objeto de la clase `type_info` (pág. 556) que contiene información acerca del tipo de su operando, incluyendo el nombre del tipo. Para usar `typeid`, el programa debe incluir el archivo de encabezado `<typeinfo>` (pág. 556).
- Al invocarse, la función miembro `name` de `type_info` (pág. 556) devuelve una cadena basada en apuntador que contiene el nombre del tipo que representa el objeto `type_info`.
- Los operadores `dynamic_cast` y `typeid` son parte de la característica de información de tipos en tiempo de ejecución (RTTI; pág. 554) de C++, la cual permite a un programa determinar el tipo de un objeto en tiempo de ejecución.

Ejercicios de autoevaluación

12.1 Complete las siguientes oraciones:

a) Tratar a un objeto de la clase base como un _____ puede provocar errores lógicos.
b) El polimorfismo ayuda a eliminar la lógica de _____.
c) Si una clase contiene al menos una función `virtual` pura, es una clase _____.
d) Las clases a partir de las cuales pueden instanciarse objetos se llaman clases _____.
e) El operador _____ se puede usar para realizar conversiones descendentes con los apuntadores de la clase base en forma segura.
f) El operador `typeid` devuelve una referencia a un objeto _____.
g) El _____ implica el uso de un apuntador o referencia de la clase base para invocar funciones `virtual` en objetos de la clase base y la clase derivada.
h) Las funciones que pueden sobrescribirse se declaran mediante la palabra clave _____.
i) Al proceso de convertir un apuntador de la clase base en un apuntador de la clase derivada se le conoce como _____.

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

a) Todas las funciones `virtual` en una clase base abstracta se deben declarar como funciones `virtual` puras.
b) Es peligroso tratar de hacer referencia a un objeto de la clase derivada con un manejador de la clase base.

c) Para hacer a una clase abstracta, se declara como `virtual`.
d) Si una clase base declara a una función `virtual` pura, una clase derivada debe implementar la función para convertirse en una clase concreta.
e) La programación polimórfica puede eliminar la necesidad de la lógica de `switch`.

Respuestas a los ejercicios de autoevaluación

12.1 a) objeto de la clase derivada. b) `switch`. c) abstracta. d) concretas. e) `dynamic_cast`. f) `type_info`. g) Polimorfismo. h) `virtual`. i) conversión descendente.

12.2 a) Falso. Una clase base abstracta puede incluir funciones virtuales con implementaciones. b) Falso. Es peligroso hacer referencia a un objeto de la clase base con un manejador de la clase derivada. c) Falso. Las clases nunca se declaran `virtual`. En vez de ello, una clase se hace abstracta al incluir por lo menos una función virtual pura en ella. d) Verdadero. e) Verdadero.

Ejercicios

12.3 ***(Programación en forma general)*** ¿Cómo es que el polimorfismo le permite programar "en forma general", en lugar de hacerlo "en forma específica"? Hable sobre las ventajas clave de la programación "en forma general".

12.4 ***(Comparación entre polimorfismo y lógica de `switch`)*** Hable sobre los problemas de programar con la lógica de `switch`. Explique por qué el polimorfismo puede ser una alternativa efectiva al uso de la lógica de `switch`.

12.5 ***(Comparación entre heredar la interfaz y la implementación)*** Explique la diferencia entre heredar la interfaz y heredar la implementación. ¿En qué difieren las jerarquías de herencia diseñadas para heredar la interfaz, de las jerarquías diseñadas para heredar la implementación?

12.6 ***(Funciones virtuales)*** ¿Qué son las funciones `virtual`? Describa una circunstancia en la que las funciones `virtual` serían apropiadas.

12.7 ***(Comparación entre vinculación dinámica y estática)*** Explique la diferencia entre la vinculación estática y la vinculación dinámica. Explique el uso de las funciones `virtual` y la *vtable* en la vinculación dinámica.

12.8 ***(Funciones virtuales)*** Explique la diferencia entre las funciones `virtual` y las funciones `virtual` puras.

12.9 ***(Clases base abstractas)*** Sugiera uno o más niveles de clases base abstractas para la jerarquía de `Figura` que vimos en este capítulo, y que se muestra en la figura 11.3. (El primer nivel es `Figura`, y el segundo nivel consiste en las clases `FiguraBidimensional` y `FiguraTridimensional`).

12.10 ***(Polimorfismo y extensibilidad)*** ¿Cómo promueve el polimorfismo la extensibilidad?

12.11 ***(Aplicación polimórfica)*** Se le ha pedido que desarrolle un simulador de vuelo que tenga salidas gráficas elaboradas. Explique por qué la programación polimórfica podría ser especialmente efectiva para un problema de esta naturaleza.

12.12 ***(Modificación al sistema de nómina)*** Modifique el sistema de nómina de las figuras 12.9 a 12.17 para incluir el miembro de datos `private` llamado `fechaNacimiento` en la clase `Empleado`. Use la clase `Fecha` de las figuras 10.6 y 10.7 para representar el cumpleaños de un empleado. Suponga que la nómina se procesa una vez al mes. Cree un `vector` de referencias `Empleado` para guardar los diversos objetos empleado. En un ciclo, calcule la nómina para cada `Empleado` (mediante el polimorfismo) y agregue una bonificación de $100.00 a la cantidad de pago de nómina de la persona, si el mes actual es el mes en el que ocurre el cumpleaños de ese `Empleado`.

12.13 ***(Jerarquía de herencia `Paquete`)*** Use la jerarquía de herencia `Paquete` creada en el ejercicio 11.9 para crear un programa que muestre la información de la dirección y que calcule los costos de envío para varios objetos `Paquete`. El programa debe contener un `vector` de apuntadores `Paquete` a objetos de las clases `PaqueteDosDias` y `PaqueteNocturno`. Itere a través del `vector` para procesar los objetos `Paquete` mediante el polimorfismo. Para cada `Paquete`, invoque a funciones *obtener* para obtener la información de las direcciones del emisor y del receptor, y después imprimir las dos direcciones como deben aparecer en las etiquetas de envío. Además, llame a la función miembro `calcularCosto` de cada `Paquete` e imprima el resultado. Lleve la cuenta del costo de envío total para todos los objetos `Paquete` en el `vector`, y muestre este total cuando termine el ciclo.

12.14 ***(Programa bancario polimórfico mediante el uso de la jerarquía `Cuenta`)*** Desarrolle un programa bancario polimórfico mediante el uso de la jerarquía `Cuenta` creada en el ejercicio 11.10. Cree un `vector` de apuntadores `Cuenta` a objetos `CuentaAhorros` y `CuentaCheques`. Para cada `Cuenta` en el `vector`, permita al usuario especificar un monto de dinero a retirar de la `Cuenta`, usando la función miembro `cargar`, y un monto de dinero a depositar en la `Cuenta` mediante el uso de la función miembro `abonar`. A medida que procese cada `Cuenta`, determine su tipo. Si una `Cuenta` es una `CuentaAhorros`, calcule el monto de interés que se debe a la `Cuenta` usando la función miembro `CalcularInteres`, y después agregue el interés al saldo actual mediante la función miembro `abonar`. Después de procesar una `Cuenta`, imprima el saldo de la cuenta actualizado que se obtiene al invocar a la función miembro `obtenerSaldo` de la clase base.

12.15 ***(Modificación al sistema de nómina)*** Modifique el sistema de nómina de las figuras 12.9 a 12.17 para incluir las subclases adicionales de `Empleado` llamadas `TrabajadorPorPiezas` y `TrabajadorPorHoras`. Un `TrabajadorPorPiezas` representa a un empleado cuyo sueldo se basa en el número de piezas de mercancía producidas. Un `TrabajadorPorHoras` representa a un empleado cuyo sueldo se basa en un salario por horas y en el número de horas trabajadas. Los trabajadores por horas reciben pago de tiempo extra (1.5 veces el salario por horas) por todas las horas trabajadas en exceso de 40 horas.

La clase `TrabajadorPorPiezas` debe contener las variables de instancia `private` llamadas `sueldo` (para almacenar el sueldo del empleado por pieza) y `piezas` (para almacenar el número de piezas producidas). La clase `TrabajadorPorHoras` debe contener las variables de instancia `private` llamadas `sueldo` (para almacenar el sueldo del empleado por hora) y `horas` (para almacenar las horas trabajadas). En la clase `TrabajadorPorPiezas`, proporcione una implementación concreta del método `ingresos` que calcule los ingresos del empleado, multiplicando el número de piezas producidas por el sueldo por cada pieza. En la clase `TrabajadorPorHoras`, proporcione una implementación completa del método `ingresos` que calcule los ingresos del empleado, multiplicando el número de horas trabajadas por el sueldo por hora. Si el número de horas trabajadas es mayor de 40, asegúrese de pagar al `TrabajadorPorHoras` las horas extras. Agregue un apuntador a un objeto de cada nueva clase al vector de apuntadores `Empleado` en `main`. Para cada `Empleado`, muestre su representación `string` y los ingresos.

Hacer la diferencia

12.16 ***(Clase abstracta `HuellaCarbono`: polimorfismo)*** Mediante el uso de una clase abstracta con sólo funciones virtuales puras, es posible especificar comportamientos similares para clases posiblemente dispares. Los gobiernos y las empresas en todo el mundo se están preocupando cada vez más por las huellas de carbono (emisiones anuales de dióxido de carbono hacia la atmósfera) de los edificios que queman diversos tipos de combustibles para calefacción, vehículos que queman combustibles para energía y otros usos similares. Muchos científicos culpan a estos gases de invernadero por el fenómeno conocido como calentamiento global. Cree tres clases pequeñas sin relación alguna mediante la herencia: las clases `Edificio`, `Auto` y `Bicicleta`. Dé a cada clase algunos atributos y comportamientos únicos apropiados que no tenga en común con otras clases. Escriba una clase abstracta llamada `HuellaCarbono` que sólo tenga un método virtual puro llamado `obtenerHuellaCarbono`. Haga que cada una de sus clases hereden de esa clase abstracta e implementen el método `obtenerHuellaCarbono` para calcular una huella de carbono apropiada para esa clase (visite sitios Web que expliquen cómo calcular las huellas de carbono). Escriba una aplicación que cree objetos de cada una de las tres clases, coloque apuntadores a esos objetos en un `vector` de apuntadores `HuellaCarbono` y luego itere a través del `vector`, invocando mediante el polimorfismo al método `obtenerHuellaCarbono` de cada objeto. Para cada objeto, imprima algo de información de identificación junto con la huella de carbono del objeto.

13

Entrada/salida de flujos: un análisis detallado

La conciencia... no aparece por sí misma cortada en pequeños pedazos... Un "río" o un "flujo" son las metáforas por las cuales se describe con más naturalidad.

—William James

Objetivos

En este capítulo aprenderá a:

- Usar la entrada/salida de flujos orientados a objetos en C++.
- Dar formato a la entrada y la salida.
- Conocer la jerarquía de la clase de E/S de flujos.
- Utilizar manipuladores de flujos.
- Controlar la justificación y el relleno de caracteres.
- Determinar el éxito o la falla de las operaciones de entrada/salida.
- Enlazar los flujos de salida a los flujos de entrada.

Plan general

13.1 Introducción

En este capítulo hablaremos sobre un rango de herramientas suficientes para realizar la mayoría de las operaciones de E/S comunes, y presentaremos las generalidades de las herramientas restantes. Anteriormente en el libro hablamos sobre algunas de estas características; ahora proporcionaremos un tratamiento más completo. Muchas de las características de E/S que veremos están orientadas a objetos. Este estilo de E/S hace uso de otras características de C++, como las referencias, la sobrecarga de funciones y la sobrecarga de operadores.

C++ utiliza la *E/S con seguridad de tipos*. Cada operación de E/S se ejecuta de una manera sensible al tipo de datos. Si se ha definido una función miembro de E/S para manejar un tipo de datos específico, entonces se hace una llamada a esa función miembro para manejar ese tipo de datos. Si no hay coincidencia entre el tipo de los datos actuales y una función para manejar ese tipo de datos, el compilador genera un error. Por ende, los datos inapropiados no pueden "infiltrarse" por el sistema (como puede ocurrir en C, con lo cual se permiten ciertos errores sutiles y raros).

Los usuarios pueden especificar cómo realizar operaciones de E/S para objetos de tipos definidos por el usuario, para lo cual sobrecargan el operador de inserción de flujo (<<) y el operador de extracción de flujo (>>). Esta **extensibilidad** es una de las características más valiosas de C++.

Observación de Ingeniería de Software 13.1

Use el estilo de E/S de C++ exclusivamente en programas de C++, aun cuando el estilo E/S de C esté disponible para los programadores de C++.

Tip para prevenir errores 13.1

La E/S en C++ es segura para los tipos.

Observación de Ingeniería de Software 13.2

C++ permite un tratamiento común de la E/S para los tipos predefinidos y los tipos definidos por el usuario. Estas características comunes facilitan el desarrollo y la reutilización del software.

13.2 Flujos

La E/S en C++ ocurre en forma de **flujos**, que son secuencias de bytes. En las operaciones de entrada, los bytes fluyen de un dispositivo (teclado, unidad de disco, conexión de red, etc.) a la memoria principal. En las operaciones de salida, los bytes fluyen de la memoria principal a un dispositivo (pantalla, impresora, unidad de disco, conexión de red, etcétera).

Una aplicación asocia un significado a los bytes. Éstos podrían representar caracteres, datos crudos, imágenes de gráficos, voz digital, video digital o cualquier otra información que pueda requerir una aplicación. Los mecanismos de E/S del sistema deben transferir bytes de los dispositivos a la memoria (y viceversa) en forma consistente y confiable. A menudo, dichas transferencias implican cierto movimiento mecánico, como la rotación de un disco o de una cinta, o la pulsación de teclas en un teclado. El tiempo que toman estas transferencias es por lo general mucho mayor que el tiempo que requiere el procesador para manipular los datos en forma interna. Por ende, las operaciones de E/S requieren un proceso cuidadoso de planeación y optimización para asegurar un rendimiento óptimo.

C++ proporciona herramientas de E/S de "bajo nivel" y de "alto nivel". Las herramientas de E/S de bajo nivel (**E/S sin formato**) especifican que se debe transferir cierto número de bytes de un dispositivo a la memoria, o de la memoria a un dispositivo. En dichas transferencias, el byte individual es el tema de interés. Dichas herramientas de bajo nivel proporcionan transferencias de alta velocidad y alto volumen, pero no son especialmente convenientes para los programadores.

Por lo general, los programadores prefieren una visión de la E/S a un nivel más alto (**E/S con formato**), en donde los bytes se agrupan en unidades significativas tales como enteros, números de punto flotante, caracteres, cadenas y tipos definidos por el usuario. Estas herramientas orientadas a los tipos son satisfactorias para la mayoría de las operaciones de E/S, exceptuando el procesamiento de archivos de alto volumen.

Tip de rendimiento 13.1

Use la E/S sin formato para el mejor rendimiento en el procesamiento de archivos de gran volumen.

Tip de portabilidad 13.1

La E/S sin formato no es portable entre todas las plataformas.

13.2.1 Comparación entre flujos clásicos y flujos estándar

En el pasado, las **bibliotecas de flujos clásicos** de C++ permitían la entrada y salida de objetos `char`. Como por lo general un `char` ocupa *un* byte, sólo puede representar un conjunto limitado de caracteres (como los del conjunto de caracteres ASCII que utiliza la mayoría de los lectores de este libro, u otros conjuntos de caracteres populares). Sin embargo, muchos lenguajes utilizan alfabetos que contienen más caracteres de los que puede representar un solo byte. El conjunto de caracteres ASCII no proporciona estos caracteres; el **conjunto de caracteres Unicode®** sí. Unicode es un conjunto de caracteres

internacional extenso, que representa la mayor parte de los lenguajes "comercialmente viables" del mundo, símbolos matemáticos y mucho más. Para obtener más información sobre Unicode, visite `www.unicode.org`.

C++ incluye las **bibliotecas de flujos estándar**, que permiten a los desarrolladores crear sistemas capaces de realizar operaciones de E/S con caracteres Unicode. Para este propósito, C++ incluye un tipo de carácter adicional llamado **`wchar_t`**, que entre otras cosas puede almacenar caracteres Unicode. El estándar de C++ también rediseñó las clases de flujos clásicos de C++, que sólo proporcionan objetos `char`, como plantillas de clase con especializaciones separadas para procesar caracteres de los tipos `char` y `wchar_t`, respectivamente. Nosotros utilizamos las especializaciones `char`. El tamaño del tipo `wchar_t` no está especificado por el estándar. Los nuevos tipos `char16_t` y `char32_t` de C++11 para representar caracteres Unicode se agregaron para proporcionar tipos de caracteres con tamaños especificados de manera explícita.

13.2.2 Encabezados de la biblioteca `iostream`

La biblioteca `iostream` de C++ proporciona cientos de herramientas de E/S. Varios encabezados contienen porciones de la interfaz de la biblioteca.

La mayoría de los programas de C++ incluyen el encabezado `<iostream>`, el cual declara los servicios básicos requeridos para todas las operaciones de E/S de flujos. El encabezado `<iostream>` define los objetos `cin`, `cout`, `cerr` y `clog`, que corresponden al flujo de entrada estándar, el flujo de salida estándar, el flujo de error estándar sin búfer y el flujo de error estándar con búfer, respectivamente. (En la sección 13.2.3 hablaremos sobre `cerr` y `clog`). Se proporcionan servicios de E/S sin formato y con formato.

El encabezado `<iomanip>` declara servicios útiles para realizar E/S con formato, con los denominados **manipuladores de flujos parametrizados**, tales como `setw` y `setprecision`.

El encabezado `<fstream>` declara servicios para el procesamiento de archivos. Utilizamos este encabezado en los programas de procesamiento de archivos del capítulo 14.

13.2.3 Clases y objetos de entrada/salida de flujos

La biblioteca `iostream` proporciona muchas plantillas para el manejo de las operaciones de E/S comunes. Por ejemplo, la plantilla de clase **`basic_istream`** soporta las operaciones de entrada de flujos, la plantilla de clase **`basic_ostream`** soporta las operaciones de salida de flujos, y la plantilla de clase **`basic_iostream`** soporta tanto operaciones de entrada de flujos como de salida de flujos. Cada plantilla tiene una especialización de plantilla predefinida que permite la E/S con objetos `char`. Además, la biblioteca `iostream` proporciona un conjunto de especificadores `typedef` que proporcionan alias para estas especializaciones de plantilla. El especificador **`typedef`** declara sinónimos (alias) para los tipos de datos. Algunas veces los programadores utilizan `typedef` para crear nombres de tipos más cortos o más legibles. Por ejemplo, la instrucción

```
typedef Carta *CartaPtr;
```

define un nombre de tipo adicional, `CartaPtr`, como un *sinónimo* para el tipo `Carta *`. Al crear un nombre mediante el uso de `typedef` *no* se crea un tipo de datos; sólo crea un nuevo nombre de tipo. En la sección 22.3 hablaremos sobre `typedef` con detalle. La definición `typedef istream` representa una especialización de `basic_stream<char>` que permite la entrada de objetos `char`. De manera similar, la definición `typedef ostream` representa una especialización de `basic_ostream<char>` que permite la salida de objetos `char`. Además, la definición `typedef iostream` representa una especialización de `basic_iostream<char>` que permite la entrada y salida de objetos `char`. A lo largo de este capítulo utilizaremos estas definiciones `typedef`.

Jerarquía de plantillas de E/S de flujos y sobrecarga de operadores

Las plantillas `basic_istream` y `basic_ostream` se derivan a través de la herencia simple de la plantilla base **`basic_ios`**.[1] La plantilla `basic_iostream` se deriva a través de la *herencia múltiple*[2] de las plantillas `basic_istream` y `basic_ostream`. El diagrama de clases de UML de la figura 13.1 sintetiza estas relaciones de herencia.

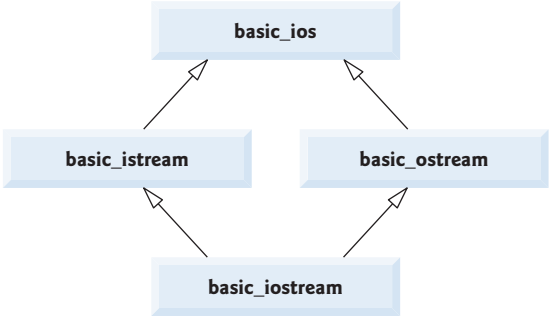


Fig. 13.1 | Porción de la jerarquía de plantillas de E/S de flujos.

La sobrecarga de operadores proporciona una notación conveniente para realizar operaciones de entrada/salida. El *operador de desplazamiento a la izquierda (<<)* se sobrecarga para designar la salida de flujos, y se conoce como el *operador de inserción de flujo*. El *operador de desplazamiento a la derecha (>>)* se sobrecarga para designar la entrada de flujos, y se conoce como el *operador de extracción de flujo*. Estos operadores se utilizan con los objetos de flujo estándar `cin`, `cout`, `cerr` y `clog`, y comúnmente con los objetos de flujo que usted cree en su propio código.

Los objetos de flujo estándar `cin`, `cout`, `cerr` *y* `clog`

El objeto predefinido `cin` es una instancia de `istream`, y se dice está "conectado a" (o unido a) *el dispositivo de entrada estándar*, que por lo general es el teclado. El operador de extracción de flujo (>>) que se utiliza en la siguiente instrucción hace que se introduzca un valor para la variable entera `calificacion` (suponiendo que `calificacion` se haya declarado como variable `int`) de `cin` a la memoria:

```
cin >> calificacion; // los datos "fluyen" en la dirección de las flechas
```

El compilador determina el tipo de datos de `calificacion` y selecciona el operador de extracción de flujo sobrecargado apropiado. Suponiendo que `calificacion` se haya declarado en forma apropiada, el operador de extracción de flujo no requiere información adicional sobre el tipo (como es el caso, por ejemplo, en la E/S estilo C). El operador >> se sobrecarga para introducir elementos de datos de los tipos fundamentales, cadenas y valores de apuntadores.

El objeto predefinido `cout` es una instancia de `ostream` y se dice que está "conectado a" el *dispositivo de salida estándar*, que por lo general es la pantalla. El operador de inserción de flujo (<<) que se utiliza en la siguiente instrucción hace que el valor de la variable `calificacion` se envíe de la memoria al dispositivo de salida estándar:

```
cout << calificacion; // los datos "fluyen" en la dirección de las flechas
```

El compilador determina el tipo de datos de `calificacion` (suponiendo que `calificacion` se haya declarado en forma apropiada) y selecciona el operador de inserción de flujo apropiado. El operador << se sobrecarga para imprimir elementos de datos de los tipos fundamentales, cadenas y valores de apuntadores.

---

1 En este capítulo hablaremos sobre las plantillas sólo en el contexto de las especializaciones de plantillas para la E/S de objetos `char`.

2 La herencia múltiple se discute en el capítulo 23, Other Topics.

El objeto predefinido `cerr` es una instancia de `ostream` y se dice que "está conectado a" el *dispositivo de error estándar*, que por lo general es la pantalla. Las operaciones de salida hacia el objeto `cerr` son **sin búfer**, lo cual implica que cada inserción de flujo en `cerr` hace que su salida aparezca *de inmediato*; esto es apropiado para notificar a un usuario oportunamente acerca de los errores.

El objeto predefinido `clog` es una instancia de la clase `ostream` y se dice que "está conectado a" el *dispositivo de error estándar*. Las salidas hacia `clog` son **con búfer**. Esto significa que cada inserción en `clog` podría hacer que su salida se contenga en un búfer (es decir, un área en memoria) hasta que éste se llene, o hasta que se vacíe. El uso de búfer es una técnica para mejorar el rendimiento de las operaciones de E/S que se describe en los cursos de sistemas operativos.

Plantillas de procesamiento de archivos

El procesamiento de archivos en C++ utiliza las plantillas de clases **`basic_ifstream`** (para la entrada de archivos), **`basic_ofstream`** (para la salida de archivos) y **`basic_fstream`** (para la entrada y salida de archivos). Al igual que con los flujos estándar, C++ proporciona un conjunto de definiciones `typedef` para trabajar con estas plantillas de clases. Por ejemplo, la definición `typedef` **`ifstream`** representa una especialización de `basic_ifstream<char>` que permite la entrada de objetos `char` desde un archivo. De manera similar, `typedef` **`ofstream`** representa una especialización de `basic_ofstream<char>` que permite la salida de objetos `char` hacia un archivo. Además, `typedef` **`fstream`** representa una especialización de `basic_fstream<char>` que permite la entrada y salida de objetos `char` desde, y hacia, un archivo. La plantilla `basic_ifstream` hereda de `basic_istream`, `basic_ofstream` hereda de `basic_ostream` y `basic_fstream` hereda de `basic_iostream`. El diagrama de clases de UML de la figura 13.2 sintetiza las diversas relaciones de herencia de las clases relacionadas con la E/S. La jerarquía de clases completa de E/S de flujos proporciona la mayoría de las herramientas que requieren los programadores. Consulte la referencia de las bibliotecas de clases de su sistema de C++ para obtener información adicional sobre el procesamiento de archivos.

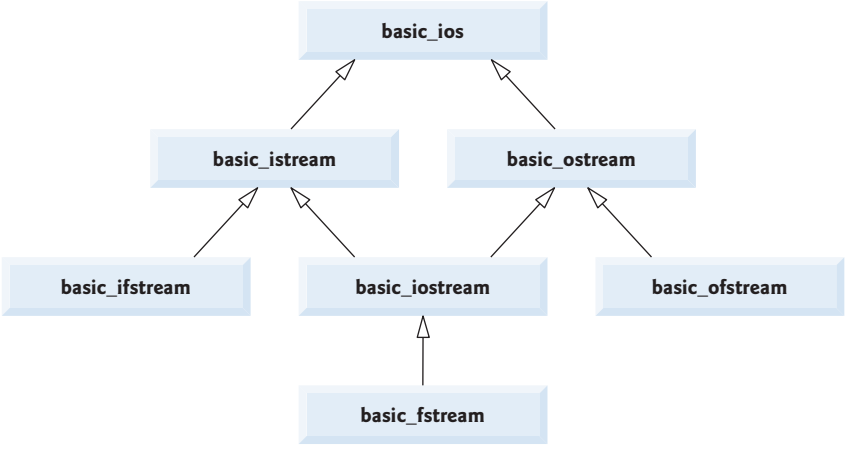


Fig. 13.2 | Parte de la jerarquía de plantillas de E/S de flujos, que muestra las plantillas principales de procesamiento de archivos.

13.3 Salida de flujos

La clase `ostream` proporciona las herramientas de salida con formato y sin formato. Las herramientas incluyen la salida de tipos de datos estándar con el operador de inserción de flujo (<<); la salida de caracteres mediante la función miembro `put`; la salida sin formato mediante la función miembro `write`; la

salida de enteros en los formatos decimal, octal y hexadecimal; la salida de valores de punto flotante con precisión variada, con puntos decimales forzados, en notación científica y en notación fija; la salida de datos justificados en campos con anchuras designadas; la salida de datos en campos rellenos con caracteres especificados y la salida de letras mayúsculas en notación científica y notación hexadecimal.

13.3.1 Salida de variables char *

C++ determina los tipos de datos de manera automática; una mejora sobre C. Pero esta característica algunas veces se "interpone en el camino". Por ejemplo, suponga que deseamos imprimir la dirección almacenada en un apuntador `char *`. El operador << se ha sobrecargado para imprimir datos de tipo `char *` como una *cadena estilo C con terminación nula*. Para imprimir la *dirección* podemos convertir el valor `char *` en `void *` (esto puede hacerse con cualquier variable apuntador). La figura 13.3 demuestra cómo imprimir una variable `char *` en los formatos de cadena y de dirección. La dirección se imprime aquí como un número hexadecimal (base 16); en general, la forma en que se imprimen las direcciones *depende de la implementación*. Para aprender más acerca de los números hexadecimales, vea el apéndice D. En las secciones 13.6.1 y 13.7.4 veremos más acerca de cómo controlar las bases de los números.

```
1  // Fig. 13.3: fig13_03.cpp
2  // Impresión de la dirección almacenada en una variable char *.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     const char *const palabra = "nuevamente";
9
10    // muestra el valor de char *, y luego muestra el valor de char *
11    // después de una static_cast a void *
12    cout << "El valor de la palabra es: " << palabra << endl
13       << "El valor de static_cast< const void * >( palabra ) es: "
14       << static_cast< const void * >( palabra ) << endl;
15 } // fin de main
```

```
El valor de la palabra es: nuevamente
El valor de static_cast< const void * >( palabra ) es: 0041675C
```

Fig. 13.3 | Impresión de la dirección almacenada en una variable `char *`.

13.3.2 Salida de caracteres mediante la función miembro put

Podemos usar la función miembro `put` para imprimir caracteres. Por ejemplo, la instrucción

```
cout.put( 'A' );
```

muestra un solo carácter `A`. Las llamadas a `put` se pueden poner en *cascada*, como en la instrucción

```
cout.put( 'A' ).put( '\n' );
```

que imprime la letra `A` seguida de un carácter de nueva línea. Al igual que con <<, la instrucción anterior se ejecuta de esta forma, debido a que el operador punto (`.`) asocia de izquierda a derecha, y la función miembro `put` devuelve una referencia al objeto `ostream` (`cout`) que recibió la llamada a `put`.

La función put también se puede llamar con una expresión numérica que represente un valor ASCII, como en la siguiente instrucción que también imprime A:

```
cout.put( 65 );
```

13.4 Entrada de flujos

Ahora vamos a considerar la entrada de flujos. La clase istream proporciona las herramientas de entrada con formato y sin formato. Por lo general, el operador de extracción de flujo (>>) omite los **caracteres de espacio en blanco** (como espacios, tabuladores y caracteres de nueva línea) en el flujo de entrada; más adelante veremos cómo modificar este comportamiento. Después de cada entrada, el operador de extracción de flujo devuelve una *referencia* al objeto flujo que recibió el mensaje de extracción (por ejemplo, cin en la expresión cin >> calificacion). Si se utiliza esa referencia como una condición (por ejemplo, en la condición de continuación de ciclo de una instrucción while), la función del operador de conversión void * sobrecargado del flujo se invoca de manera implícita para convertir la referencia en un valor de apuntador no nulo, o en el apuntador nulo con base en el éxito o fracaso de la última operación de entrada, respectivamente. Un apuntador no nulo se convierte en el valor bool true para indicar éxito, y el apuntador nulo se convierte en el valor bool false para indicar fracaso. Cuando se hace un intento de leer más allá del final de un flujo, el operador de conversión sobrecargado void * del flujo devuelve el *apuntador nulo* para indicar el *fin del archivo*.

Cada objeto flujo contiene un conjunto de **bits de estado** que se utilizan para controlar el estado del flujo (es decir, aplicar formato, establecer estados de error, etc.). El operador de conversión sobrecargado void * utiliza estos bits para determinar si debe devolver un apuntador no nulo o el apuntador nulo. La extracción de flujo hace que se establezca el bit **failbit** del flujo si se introducen datos del tipo incorrecto, y hace que se establezca el bit **badbit** del flujo si falla la operación. En las secciones 13.7 y 13.8 hablaremos sobre los bits de estado de un flujo con detalle, y posteriormente le mostraremos cómo evaluar estos bits después de una operación de E/S.

13.4.1 Funciones miembro get y getline

La función miembro **get** *sin argumentos* recibe como entrada *un* carácter del flujo designado (incluyendo caracteres de espacio en blanco y otros caracteres no gráficos, como la secuencia de teclas que representa el fin de archivo) y lo devuelve como el valor de la llamada a la función. Esta versión de get devuelve EOF cuando se encuentra el *fin del archivo* en el flujo.

Uso de las funciones miembro eof, get *y* put

La figura 13.4 demuestra el uso de las funciones miembro eof y get en el flujo de entrada cin, y la función miembro put en el flujo de salida cout. Si recuerda, en el capítulo 5 vimos que EOF se representa como un entero. El programa lee caracteres y los introduce en la variable int llamada caracter, para que podamos probar cada carácter introducido y ver si es EOF. El programa primero imprime el valor de cin.eof(), es decir, false (0 en la salida), para mostrar que *no* ha ocurrido el *fin de archivo* en cin. El usuario introduce una línea de texto y oprime *Intro* seguido del fin de archivo (<*Ctrl*>-*z* en sistemas Microsoft Windows, <*Ctrl*>-*d* en sistemas Linux y Mac). En la línea 15 se lee cada carácter, que en la línea 16 se envía como salida a cout mediante la función miembro put. Al encontrar el fin de archivo la instrucción while termina, y en la línea 20 se muestra el valor de cin.eof(), que ahora es true (1 en la salida), para mostrar que se ha establecido el fin de archivo en cin. Este programa utiliza la versión de la función miembro get de istream que no recibe argumentos y devuelve el carácter que se está introduciendo (línea 15). La función eof devuelve true sólo después de que el programa trata de leer más allá del último carácter en el flujo.

```
1  // Fig. 13.4: Fig13_04.cpp
2  // Las funciones miembro get, put y eof.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int caracter; // usa int, ya que char no puede representar EOF
9
10    // pide al usuario que introduzca una línea de texto
11    cout << "Antes de la entrada, cin.eof() es " << cin.eof() << endl
12       << "Escriba un enunciado seguido del fin de archivo:" << endl;
13
14    // usa get para leer cada carácter; usa put para mostrarlo
15    while ( ( caracter = cin.get() ) != EOF )
16       cout.put( caracter );
17
18    // muestra el caracter de fin de archivo
19    cout << "\nEOF en este sistema es: " << caracter << endl;
20    cout << "Despues de introducir EOF, cin.eof() es " << cin.eof() << endl;
21 } // fin de main
```

```
Antes de la entrada, cin.eof() es 0
Escriba un enunciado seguido del fin de archivo:
Prueba de las funciones miembro get y put
Prueba de las funciones miembro get y put
^Z

EOF en este sistema es: -1
Despues de introducir EOF, cin.eof() es 1
```

Fig. 13.4 | Funciones miembro `get`, `put` y `eof`.

La función miembro `get` con un argumento de referencia de carácter introduce el siguiente carácter del flujo de entrada (aun si es un *carácter de espacio en blanco*) y lo almacena en el argumento tipo carácter. Esta versión de `get` devuelve una referencia al objeto `istream` para el que se está invocando la función miembro `get`.

Una tercera versión de `get` recibe tres argumentos: un arreglo de caracteres, un límite de tamaño y un delimitador (con el valor predeterminado `'\n'`). Esta versión lee caracteres del flujo de entrada. Lee *un carácter menos* que el número máximo especificado de caracteres y termina, o se termina tan pronto como se lea el *delimitador*. Se inserta un carácter nulo para terminar la cadena de entrada en el arreglo de caracteres que el programa utiliza como búfer. El delimitador no se coloca en el arreglo de caracteres, sino que *permanece en el flujo de entrada* (el delimitador será el siguiente carácter que se lea). Así, el resultado de una segunda función `get` consecutiva es una línea vacía, a menos que el carácter delimitador se elimine del flujo de entrada (posiblemente con `cin.ignore()`).

Comparación entre `cin` y `cin.get`

La figura 13.5 compara la entrada mediante el uso de la extracción de flujo con `cin` (que lee caracteres hasta que se encuentra un carácter de espacio en blanco) y la entrada mediante el uso de `cin.get`. La llamada a `cin.get` (línea 22) *no* especifica un delimitador, por lo que se utiliza el carácter *predeterminado* `'\n'`.

```
1  // Fig. 13.5: Fig13_05.cpp
2  // Contraste entre la entrada de una cadena mediante cin y cin.get.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     // crea dos arreglos char, cada uno con 80 elementos
9     const int TAMANIO = 80;
10    char buffer1[ TAMANIO ];
11    char buffer2[ TAMANIO ];
12
13    // usa cin para introducir caracteres en bufer1
14    cout << "Escriba un enunciado:" << endl;
15    cin >> buffer1;
16
17    // muestra el contenido de bufer1
18    cout << "\nLa cadena leida con cin fue:" << endl
19       << buffer1 << endl << endl;
20
21    // usa cin.get para introducir caracteres en bufer2
22    cin.get( buffer2, TAMANIO );
23
24    // muestra el contenido de bufer2
25    cout << "La cadena leida con cin.get fue:" << endl
26       << buffer2 << endl;
27 } // fin de main
```

```
Escriba un enunciado:
Contraste entre la entrada de una cadena mediante cin y cin.get

La cadena leida con cin fue:
Contraste

La cadena leida con cin.get fue:
 entre la entrada de una cadena mediante cin y cin.get
```

Fig. 13.5 | Comparación de la entrada de una cadena mediante el uso de `cin` y mediante el uso de `cin.get`.

Uso de la función miembro `getline`

La función miembro **`getline`** opera de manera similar a la tercera versión de la función miembro `get` e *inserta un carácter nulo* después de la línea en el arreglo integrado de caracteres. La función `getline` elimina el delimitador del flujo (es decir, lee el carácter y lo descarta), pero *no* lo almacena en el arreglo de caracteres. El programa de la figura 13.6 demuestra el uso de la función miembro `getline` para introducir una línea de texto (línea 13).

```
1  // Fig. 13.6: Fig13_06.cpp
2  // Introducción de caracteres mediante la función miembro getline de cin.
3  #include <iostream>
4  using namespace std;
```

Fig. 13.6 | Introducción de datos tipo carácter con la función miembro `getline` de `cin` (parte 1 de 2).

```
5
6  int main()
7  {
8     const int TAMANIO = 80;
9     char buffer[ TAMANIO ]; // crea un arreglo de 80 caracteres
10
11    // introduce caracteres en bufer mediante la función getline de cin
12    cout << "Escriba un enunciado:" << endl;
13    cin.getline( buffer, TAMANIO );
14
15    // muestra el contenido de bufer
16    cout << "\nEl enunciado introducido es:" << endl << buffer << endl;
17 } // fin de main
```

```
Escriba un enunciado:
Uso de la funcion miembro getline

El enunciado introducido es:
Uso de la funcion miembro getline
```

Fig. 13.6 | Introducción de datos tipo carácter con la función miembro `getline` de `cin` (parte 2 de 2).

13.4.2 Funciones miembro peek, putback e ignore de istream

La función miembro **`ignore`** de `istream` lee y descarta un número designado de caracteres (el valor predeterminado es *uno*) o termina al momento de encontrar un delimitador designado (el delimitador predeterminado es `EOF`, que hace que `ignore` salte hasta el fin del archivo cuando lee datos del mismo).

La función miembro **`putback`** coloca el carácter anterior, obtenido por una operación `get` de un flujo de entrada, de vuelta a ese flujo. Esta función es útil para las aplicaciones que exploran un flujo de entrada en busca de un campo que empiece con un carácter específico. Cuando se introduce ese carácter, la aplicación devuelve el carácter al flujo, por lo que éste se puede incluir en los datos de entrada.

La función miembro **`peek`** devuelve el siguiente carácter de un flujo de entrada, pero no lo elimina del flujo.

13.4.3 E/S con seguridad de tipos

C++ ofrece la **E/S con seguridad de tipos**. Los operadores << y >> se sobrecargan para aceptar elementos de datos de tipos *específicos*. Si se procesan datos inesperados, se establecen varios bits de error, que el usuario puede evaluar para determinar si una operación de E/S tuvo éxito o fracasó. Si los operadores << y >> no se han sobrecargado para un tipo definido por el usuario y el programador intenta realizar operaciones de entrada o salida con el contenido de un objeto de ese tipo definido por el usuario, el compilador reporta un error. Esto permite al programa "permanecer con el control". En la sección 13.8 hablaremos acerca de estos estados de error.

13.5 E/S sin formato mediante el uso de read, write y gcount

La entrada/salida sin formato se lleva a cabo mediante las funciones miembro **`read`** y **`write`** de `istream` y `ostream`, respectivamente. La función miembro `read` introduce *bytes* en un arreglo integrado de caracteres en la memoria; la función miembro `write` envía bytes de salida desde un arreglo de caracteres. Estos bytes *no tienen ningún tipo de formato*. Se reciben como entrada o se envían de salida como bytes puros. Por ejemplo, la llamada

```
char buffer[] = "FELIZ CUMPLEANIOS";
cout.write( buffer, 10 );
```

imprime los primeros 10 bytes de `bufer` (incluyendo caracteres nulos, si los hay, que hagan que termine la salida con `cout` y <<). La llamada

```
cout.write( "ABCDEFGHIJKLMNOPQRSTUVWXYZ", 10 );
```

muestra los primeros 10 caracteres del alfabeto.

La función miembro `read` introduce un número designado de caracteres en un arreglo integrado de caracteres. Si se leen *menos* caracteres que el número designado, se establece el bit `failbit`. En la sección 13.8 le mostraremos cómo determinar si se ha establecido `failbit`. La función miembro **`gcount`** reporta el número de caracteres leídos por la última operación de entrada.

La figura 13.7 demuestra las funciones miembro `read` y `gcount` de `istream`, y la función miembro `write` de `ostream`. El programa introduce 20 caracteres (de una secuencia de entrada más larga) en el arreglo `bufer` con `read` (línea 13), determina el número de caracteres introducidos con `gcount` (línea 17) y envía como salida los caracteres en `bufer` con `write` (línea 17).

```
1  // Fig. 13.7: Fig13_07.cpp
2  // E/S sin formato mediante el uso de read, gcount y write.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     const int TAMANIO = 80;
9     char buffer[ TAMANIO ]; // crea un arreglo de 80 caracteres
10
11    // usa la función read para introducir caracteres en el bufer
12    cout << "Escriba un enunciado:" << endl;
13    cin.read( buffer, 20 );
14
15    // usa las funciones write y gcount para mostrar los caracteres del búfer
16    cout << endl << "El enunciado que escribio fue:" << endl;
17    cout.write( buffer, cin.gcount() );
18    cout << endl;
19 } // fin de main
```

```
Escriba un enunciado:
Uso de las funciones miembro read, write y gcount
El enunciado que escribio fue:
Uso de read, write
```

Fig. 13.7 | E/S sin formato mediante el uso de las funciones miembro `read`, `gcount` y `write`.

13.6 Introducción a los manipuladores de flujos

C++ proporciona varios **manipuladores de flujos** que realizan tareas de formato. Los manipuladores de flujos proporcionan herramientas para establecer las anchuras de los campos, establecer la precisión, establecer y quitar el formato de estado, establecer el carácter de relleno en los campos, vaciar flujos, insertar una nueva línea en el flujo de salida (y vaciar el flujo), insertar un carácter nulo en el flujo de salida y omitir el espacio en blanco en el flujo de entrada. Estas características se describen en las siguientes secciones.

13.6.1 Base de flujos integrales: dec, oct, hex y setbase

Los enteros se interpretan generalmente como valores decimales (base 10). Para cambiar la base en la que se interpretan los enteros en un flujo, inserte el manipulador **hex** para establecer la base en hexadecimal (base 16) o inserte el manipulador **oct** para establecer la base en octal (base 8). Inserte el manipulador **dec** para restablecer la base del flujo en decimal. Todos estos son manipuladores *pegajosos*.

La base de un flujo también se puede cambiar mediante el manipulador de flujos **setbase**, el cual recibe un argumento entero de 10, 8 o 16 para establecer la base en decimal, octal o hexadecimal, respectivamente. Debido a que `setbase` recibe un argumento, se conoce como *manipulador de flujo parametrizado*. Los manipuladores de flujo parametrizados como `setbase` requieren el encabezado `<iomanip>`. El valor de la base del flujo permanece igual hasta que se cambia de manera explícita; las opciones de `setbase` son pegajosas. La figura 13.8 demuestra los manipuladores de flujos `hex`, `oct`, `dec` y `setbase`. Para obtener más información sobre los números decimales, octales y hexadecimales, vea el apéndice D.

```
1  // Fig. 13.8: Fig13_08.cpp
2  // Uso de los manipuladores de flujos hex, oct, dec y setbase.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main()
8  {
9     int numero;
10
11    cout << "Escriba un numero decimal: ";
12    cin >> numero; // recibe el numero de entrada
13
14    // usa el manipulador de flujo hex para mostrar un número hexadecimal
15    cout << numero << " en hexadecimal es: " << hex
16       << numero << endl;
17
18    // usa el manipulador de flujo oct para mostrar un número octal
19    cout << dec << numero << " en octal es: "
20       << oct << numero << endl;
21
22    // usa el manipulador de flujo setbase para mostrar un número decimal
23    cout << setbase( 10 ) << numero << " en decimal es: "
24       << numero << endl;
25 } // fin de main
```

```
Escriba un numero decimal: 20
20 en hexadecimal es: 14
20 en octal es: 24
20 en decimal es: 20
```

Fig. 13.8 | Uso de los manipuladores de flujos `hex`, `oct`, `dec` y `setbase`.

13.6.2 Precisión de punto flotante (precision, setprecision)

Para controlar la **precisión** de los números de punto flotante (es decir, el número de dígitos a la derecha del punto decimal), podemos usar el manipulador de flujo `setprecision` o la función miembro **precision** de `ios_base`. Una llamada a uno de estos miembros establece la precisión para todas las

operaciones de salida subsecuentes, hasta la siguiente llamada para establecer la precisión. Una llamada a la función miembro `precision` sin argumento devuelve la opción de precisión actual (esto es lo que necesitamos usar para poder *restaurar la precisión original* en un momento dado, una vez que ya no sea necesaria una opción pegajosa). El programa de la figura 13.9 utiliza tanto la función miembro `precision` (línea 22) como el manipulador `setprecision` (línea 31) para imprimir una tabla que muestra la raíz cuadrada de 2, en donde la precisión varía de 0 a 9.

```
1  // Fig. 13.9: Fig13_09.cpp
2  // Control de la precisión de los valores de punto flotante.
3  #include <iostream>
4  #include <iomanip>
5  #include <cmath>
6  using namespace std;
7
8  int main()
9  {
10    double raiz2 = sqrt( 2.0 ); // calcula la raíz cuadrada de 2
11    int posiciones; // precisión, varía de 0 a 9
12
13    cout << "Raiz cuadrada de 2 con precisiones de 0 a 9." << endl
14       << "Precision establecida mediante la funcion miembro precision "
15       << "de ios_base:" << endl;
16
17    cout << fixed; // usa la notación de punto fijo
18
19    // muestra la raíz cuadrada usando la función precision de ios_base
20    for ( posiciones = 0; posiciones <= 9; ++posiciones )
21    {
22       cout.precision( posiciones );
23       cout << raiz2 << endl;
24    } // fin de for
25
26    cout << "\nPrecision establecida por el manipulador de flujo "
27       << "setprecision:" << endl;
28
29    // establece la precisión para cada dígito, y después muestra la raíz cuadrada
30    for ( posiciones = 0; posiciones <= 9; ++posiciones )
31       cout << setprecision( posiciones ) << raiz2 << endl;
32 } // fin de main
```

```
Raiz cuadrada de 2 con precisiones de 0 a 9.
Precision establecida mediante la funcion miembro precision de ios_base:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

Fig. 13.9 | Control de la precisión de los valores de punto flotante (parte 1 de 2).

```
Precision establecida por el manipulador de flujo setprecision:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

Fig. 13.9 | Control de la precisión de los valores de punto flotante (parte 2 de 2).

13.6.3 Anchura de campos (`width`, `setw`)

La función miembro **`width`** (de la clase base `ios_base`) establece la *anchura de campo* (es decir, el número de posiciones de caracteres en los que debe imprimirse un valor, o el número máximo de caracteres que deben introducirse) y *devuelve la anchura anterior*. Si los valores que se imprimen son menos que la anchura de campo, se insertan **caracteres de relleno** como **relleno** (**padding**). Un valor más ancho que la anchura designada no se truncará; se imprimirá el *número completo*. La función `width` sin argumento devuelve la configuración actual.

Error común de programación 13.1

La opción de anchura se aplica sólo para la siguiente inserción o extracción (es decir, la opción de anchura no es pegajosa); después de esto, la anchura se establece de manera implícita en 0 (es decir, la entrada y la salida se llevarán a cabo con las opciones predeterminadas*). Suponer que la opción de anchura se aplica a todas las operaciones de salida subsiguientes es un error lógico.*

Error común de programación 13.2

Cuando un campo no *es lo bastante ancho como para manejar las salidas, éstas se imprimen con la anchura necesaria, lo cual puede producir resultados confusos.*

La figura 13.10 demuestra el uso de la función miembro `width` en operaciones de entrada y de salida. Al introducir datos en un arreglo `char`, *se leerá un máximo de caracteres igual a uno menos la anchura*, ya que se toma en cuenta el carácter nulo que se va a colocar en la cadena de entrada. Recuerde que la extracción de flujo *termina* al encontrar *espacio en blanco a la derecha*. El manipulador de flujo `setw` también se puede usar para establecer la anchura de los campos. [*Nota:* cuando se pide al usuario la entrada en la figura 13.10, éste debe introducir una línea de texto y oprimir la tecla *Intro*, seguida del fin de archivo (<*Ctrl*>-*z* en sistemas Microsoft Windows, <*Ctrl*>-*d* en sistemas Linux y OS X)].

```
1  // Fig. 13.10: Fig13_10.cpp
2  // La función miembro width de la clase ios_base.
3  #include <iostream>
4  using namespace std;
5
```

Fig. 13.10 | La función miembro `width` de la clase `ios_base` (parte 1 de 2).

```
6  int main()
7  {
8     int valorAnchura = 4;
9     char enunciado[ 10 ];
10
11    cout << "Escriba un enunciado:" << endl;
12    cin.width( 5 ); // introduce sólo 5 caracteres de enunciado
13
14    // establece la anchura de campo y después muestra los caracteres con base
         en esa anchura
15    while ( cin >> enunciado )
16    {
17       cout.width( valorAnchura++ )
18       cout << enunciado << endl;
19       cin.width( 5 ); // introduce 5 caracteres más de enunciado
20    } // fin de while
21 } // fin de main
```

```
Escriba un enunciado:
Esta es una prueba de la funcion miembro width
Esta
  es
  una
  prue
    ba
     de
      la
     func
       ion
       miem
         bro
         width
```

Fig. 13.10 | La función miembro `width` de la clase `ios_base` (parte 2 de 2).

13.6.4 Manipuladores de flujos de salida definidos por el usuario

El programador puede crear sus propios manipuladores de flujos. La figura 13.11 muestra la creación y uso de los *nuevos* manipuladores de flujos no parametrizados `alarma` (líneas 8 a 11), `retornoCarro` (líneas 14 a 17), `tab` (líneas 20 a 23) y `finLinea` (líneas 27 a 30). Para los manipuladores de flujos de salida, el tipo de valor de retorno y el parámetro deben ser de tipo `ostream &`. Cuando en la línea 35 se inserta el manipulador `finLinea` en el flujo de salida, se hace una llamada a la función `finLinea` y en la línea 29 se imprime la secuencia de escape `\n` junto con el manipulador `flush` (que vacía el búfer de salida) al flujo de salida estándar `cout`. De manera similar, cuando en las líneas 35 a 44 se insertan los manipuladores `tab`, `alarma` y `retornoCarro` en el flujo de salida, se hacen llamadas a sus funciones correspondientes: `tab` (línea 20), `alarma` (línea 8) y `retornoCarro` (línea 14), que a su vez imprimen varias secuencias de escape.

```
1  // Fig. 13.11: Fig13_11.cpp
2  // Creación y prueba de manipuladores de flujos
3  // no parametrizados, definidos por el usuario.
4  #include <iostream>
5  using namespace std;
```

Fig. 13.11 | Manipuladores de flujos no parametrizados, definidos por el usuario (parte 1 de 2).

```
6
7  // manipulador alarma (usa la secuencia de escape \a)
8  ostream& alarma( ostream& salida )
9  {
10    return salida << '\a'; // emite el sonido del sistema
11 } // fin del manipulador alarma
12
13 // manipulador retornoCarro (usa la secuencia de escape \r)
14 ostream& retornoCarro( ostream& salida )
15 {
16    return salida << '\r'; // emite el retorno de carro
17 } // fin del manipulador retornoCarro
18
19 // manipulador tab (usa la secuencia de escape \t )
20 ostream& tab( ostream& salida )
21 {
22    return salida << '\t'; // emite el tabulador
23 } // fin del manipulador tab
24
25 // manipulador finLinea (usa la secuencia de escape \n y el
26 // manipulador de flujo flush para simular endl)
27 ostream& finLinea( ostream& salida )
28 {
29    return salida << '\n' << flush; // emite fin de línea parecido a endl
30 } // fin del manipulador finLinea
31
32 int main()
33 {
34    // usa los manipuladores tab y finLinea
35    cout << "Prueba del manipulador tab:" << finLinea
36       << 'a' << tab << 'b' << tab << 'c' << finLinea;
37
38    cout << "Prueba de los manipuladores retornoCarro y alarma:"
39       << finLinea << "..........";
40
41    cout << alarma; // usa el manipulador alarma
42
43    // usa los manipuladores retornoCarro y finLinea
44    cout << retornoCarro << "-----" << finLinea;
45 } // fin de main
```

```
Prueba del manipulador tab:
a       b       c
Prueba de los manipuladores retornoCarro y alarma:
-----.....
```

Fig. 13.11 | Manipuladores de flujos no parametrizados, definidos por el usuario (parte 2 de 2).

13.7 Estados de formato de flujos y manipuladores de flujos

Se pueden utilizar varios manipuladores de flujos para especificar los tipos de formato a realizar durante las operaciones de E/S de flujos. Los manipuladores de flujos controlan la configuración del formato de la salida. La figura 13.12 ilustra cada manipulador de flujo que controla un estado de formato de un flujo dado. Todos estos manipuladores pertenecen a la clase `ios_base`. En las siguientes secciones mostraremos ejemplos de la mayoría de estos manipuladores de flujos.

Manipulador	Descripción
`skipws`	*Omite los caracteres de espacio en blanco en un flujo de entrada.* Esta opción se restablece con el manipulador de flujo `noskipws`.
`left`	*Justifica la salida a la izquierda* en un campo. Si es necesario, aparecen caracteres de *relleno* a la *derecha.*
`right`	*Justifica la salida a la derecha* en un campo. Si es necesario, aparecen caracteres de relleno a la *izquierda.*
`internal`	Indica que el *signo* de un número debe *justificarse a la izquierda* en un campo, y que la *magnitud* del número se debe *justificar a la derecha* en ese mismo campo (es decir, deben aparecer caracteres de *relleno entre* el signo y el número).
`boolalpha`	Especifica que deben mostrarse *valores* `bool` como la palabra `true` o `false`. El manipulador `noboolalpha` establece el flujo de vuelta a mostrar valores `bool` como 1 (verdadero) y 0 (falso).
`dec`	Especifica que los enteros deben tratarse como valores *decimales* (base 10).
`oct`	Especifica que los enteros se deben tratar como valores *octales* (base 8).
`hex`	Especifica que los enteros se deben tratar como valores *hexadecimales* (base 16).
`showbase`	Especifica que la *base* de un número se debe imprimir *adelante* del mismo (un `0` a la izquierda para los valores octales; `0x` o `0X` a la izquierda para los valores hexadecimales). Esta opción se restablece con el manipulador de flujo `noshowbase`.
`showpoint`	Especifica que los números de punto flotante se deben imprimir con un *punto decimal*. Esto se usa generalmente con `fixed` para *garantizar* cierto número de dígitos a la *derecha* del punto decimal, aún y cuando sean ceros. Esta opción se restablece con el manipulador de flujo `noshowpoint`.
`uppercase`	Especifica que deben usarse *letras mayúsculas* (es decir, `X` y de la `A` a la `F`) en un entero *hexadecimal*, y que se debe usar la letra `E` *mayúscula* al representar un valor de punto flotante en *notación científica*. Esta opción se restablece con el manipulador de flujo `nouppercase`.
`showpos`	Especifica que a los números *positivos* se les debe anteponer un signo positivo (`+`). Esta opción se restablece con el manipulador de flujo `noshowpos`.
`scientific`	Especifica la salida de un valor de punto flotante en *notación científica*.
`fixed`	Especifica la salida de un valor de punto flotante en *notación de punto fijo*, con un número específico de dígitos a la *derecha* del punto decimal.

Fig. 13.12 | Manipuladores de flujo de formato de estado de `<iostream>`.

13.7.1 Ceros a la derecha y puntos decimales (`showpoint`)

El manipulador de flujo **`showpoint`** es una opción pegajosa que obliga a que un número de punto flotante se imprima con su *punto decimal* y *ceros a la derecha*. Por ejemplo, el valor de punto flotante `79.0` se imprime como `79` sin usar `showpoint`, y se imprime como `79.000000` (o con todos los ceros que se especifiquen mediante la *precisión* actual) usando `showpoint`. Para restablecer la opción de `showpoint`, hay que imprimir el manipulador de flujo **`noshowpoint`**. El programa de la figura 13.13 muestra cómo usar el manipulador de flujo `showpoint` para controlar la impresión de *ceros a la derecha* y *puntos decimales* para los valores de punto flotante. Recuerde que la *precisión predeterminada* de un número de punto flotante es 6. Cuando no se utilizan los manipuladores de flujo `fixed` o `scientific`, la precisión

representa el número de dígitos significativos a mostrar (es decir, el número total de dígitos a mostrar), *no* el número de dígitos a mostrar después del punto decimal.

```
1  // Fig. 13.13: Fig13_13.cpp
2  // Control de la impresión de ceros a la derecha y
3  // puntos decimales en valores de punto flotante.
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9     // muestra los valores double con formato de flujo predeterminado
10    cout << "Antes de usar showpoint" << endl
11       << "9.9900 se imprime como: " << 9.9900 << endl
12       << "9.9000 se imprime como: " << 9.9000 << endl
13       << "9.0000 se imprime como: " << 9.0000 << endl << endl;
14
15    // muestra el valor double después de showpoint
16    cout << showpoint
17       << "Despues de usar showpoint" << endl
18       << "9.9900 se imprime como: " << 9.9900 << endl
19       << "9.9000 se imprime como: " << 9.9000 << endl
20       << "9.0000 se imprime como: " << 9.0000 << endl;
21 } // fin de main
```

```
Antes de usar showpoint
9.9900 se imprime como: 9.99
9.9000 se imprime como: 9.9
9.0000 se imprime como: 9

Despues de usar showpoint
9.9900 se imprime como: 9.99000
9.9000 se imprime como: 9.90000
9.0000 se imprime como: 9.00000
```

Fig. 13.13 | Control de la impresión de ceros a la derecha y puntos decimales en los valores de punto flotante.

13.7.2 Justificación (`left`, `right` e `internal`)

Los manipuladores de flujos **`left`** y **`right`** permiten *justificar* los campos *a la izquierda* con caracteres de *relleno* a la *derecha*, o *justificarlos a la derecha* con caracteres de *relleno* a la *izquierda*, respectivamente. El carácter de relleno se especifica mediante la función miembro `fill` o el manipulador de flujo parametrizado `setfill` (que veremos en la sección 13.7.3). La figura 13.14 utiliza los manipuladores `setw`, `left` y `right` para justificar a la izquierda y a la derecha los datos enteros en un campo.

```
1  // Fig. 13.14: Fig13_14.cpp
2  // Justificación a la izquierda y a la derecha con los manipuladores de flujos
      left y right.
3  #include <iostream>
```

Fig. 13.14 | Justificación a la izquierda y a la derecha con los manipuladores de flujos `left` y `right` (parte 1 de 2).

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

int main()
{
   int x = 12345;

   // muestra el valor de x justificado a la derecha (predeterminado)
   cout << "La opcion predeterminada es justificado a la derecha:" << endl
      << setw( 10 ) << x;

   // usa el manipulador left para mostrar el valor de x justificado a la izquierda
   cout << "\n\nUso de std::left para justificar x a la izquierda:\n"
      << left << setw( 10 ) << x;

   // usa el manipulador right para mostrar el valor de x justificado a la derecha
   cout << "\n\nUso de std::right para justificar x a la derecha:\n"
      << right << setw( 10 ) << x << endl;
} // fin de main
```

```
La opcion predeterminada es justificado a la derecha:
    12345

Uso de std::left para justificar x a la izquierda:
12345

Uso de std::right para justificar x a la derecha:
    12345
```

Fig. 13.14 | Justificación a la izquierda y a la derecha con los manipuladores de flujos `left` y `right` (parte 2 de 2).

El manipulador de flujo **`internal`** indica que el *signo* de un número (o la *base*, cuando se utiliza el manipulador de flujo `showbase`) debe *justificarse a la izquierda* dentro de un campo, que la *magnitud* del número se debe *justificar a la derecha* y que los *espacios intermedios* deben *rellenarse* con el *carácter de relleno*. La figura 13.15 muestra el manipulador de flujo `internal` que especifica un espaciamiento interno (línea 10). Observe que **`showpos`** obliga a que se imprima el signo positivo (línea 10). Para restablecer la opción de `showpos`, hay que imprimir el manipulador de flujo **`noshowpos`**.

```
// Fig. 13.15: Fig13_15.cpp
// Impresión de un entero con espaciamiento interno y un signo positivo.
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   // muestra el valor con espaciamiento interno y signo positivo
   cout << internal << showpos << setw( 10 ) << 123 << endl;
} // fin de main
```

Fig. 13.15 | Impresión de un entero con espaciamiento interno y el signo positivo (parte 1 de 2).

```
+     123
```

Fig. 13.15 | Impresión de un entero con espaciamiento interno y el signo positivo (parte 2 de 2).

13.7.3 Relleno de caracteres (`fill`, `setfill`)

La **función miembro `fill`** especifica el *carácter de relleno* que se debe utilizar con los campos justificados; se utilizan *espacios* para rellenar de manera *predeterminada*. La función devuelve el carácter de relleno anterior. El **manipulador `setfill`** también establece el *carácter de relleno*. La figura 13.16 demuestra el uso de la función `fill` (línea 30) y el manipulador de flujo `setfill` (líneas 34 y 37) para establecer el carácter de relleno.

```
1  // Fig. 13.16: Fig13_16.cpp
2  // Uso de la función miembro fill y el manipulador de flujo setfill para cambiar
3  // el carácter de relleno para campos más grandes que el valor impreso.
4  #include <iostream>
5  #include <iomanip>
6  using namespace std;
7
8  int main()
9  {
10    int x = 10000;
11
12    // muestra x
13    cout << x << " impreso como int justificado a la derecha y a la izquierda\n"
14       << "y como hex con justificacion interna.\n"
15       << "Uso del caracter de relleno predeterminado (espacio):" << endl;
16
17    // muestra x con la base
18    cout << showbase << setw( 10 ) << x << endl;
19
20    // muestra x con justificación a la izquierda
21    cout << left << setw( 10 ) << x << endl;
22
23    // muestra x como hex con justificación interna
24    cout << internal << setw( 10 ) << hex << x << endl << endl;
25
26    cout << "Uso de varios caracteres de relleno:" << endl;
27
28    // muestra x usando caracteres de relleno (justificación a la derecha)
29    cout << right;
30    cout.fill( '*' );
31    cout << setw( 10 ) << dec << x << endl;
32
33    // muestra x usando caracteres de relleno (justificación a la izquierda)
34    cout << left << setw( 10 ) << setfill( '%' ) << x << endl;
35
36    // muestra x usando caracteres de relleno (justificación interna)
37    cout << internal << setw( 10 ) << setfill( '^' ) << hex
38       << x << endl;
39 } // fin de main
```

Fig. 13.16 | Uso de la función miembro `fill` y el manipulador de flujo `setfill` para modificar el carácter de relleno, cuando los campos son más grandes que los valores que se van a imprimir (parte 1 de 2).

```
1000 impreso como int justificado a la derecha y a la izquierda
y como hex con justificacion interna.
Uso del caracter de relleno predeterminado (espacio):
      10000
10000
0x     2710

Uso de varios caracteres de relleno:
*****10000
10000%%%%%
0x^^^^2710
```

Fig. 13.16 | Uso de la función miembro `fill` y el manipulador de flujo `setfill` para modificar el carácter de relleno, cuando los campos son más grandes que los valores que se van a imprimir (parte 2 de 2).

13.7.4 Base de flujos integrales (dec, oct, hex, showbase)

C++ proporciona los manipuladores de flujos **dec**, **hex** y **oct** para especificar que se van a mostrar enteros como valores decimales, hexadecimales y octales, respectivamente. Las inserciones de flujo usan la opción *predeterminada decimal* si no se utiliza uno de estos manipuladores. Con la extracción de flujo, los enteros con prefijo de `0` (cero) se tratan como valores *octales*, los enteros con el prefijo `0x` o `0X` se tratan como valores *hexadecimales*, y todos los demás enteros se tratan como valores *decimales*. Una vez que se especifica una base particular para un flujo, todos los enteros en ese flujo se procesan con esa base, hasta que se especifique una base distinta o cuando el programa termina.

El manipulador de flujo **showbase** obliga a que se imprima la *base* de un valor integral. Los números decimales se imprimen de manera predeterminada, los números octales se imprimen con un `0` a la izquierda, y los números hexadecimales se imprimen con `0x` o `0X` a la izquierda (como veremos en la sección 13.7.6, el manipulador de flujo `uppercase` determina qué opción se elige). La figura 13.17 demuestra el uso del manipulador de flujo `showbase` para obligar a un entero a imprimirse en los formatos decimal, octal y hexadecimal. Para restablecer la opción de `showbase`, hay que imprimir el manipulador de flujo **noshowbase**.

```
1  // Fig. 13.17: Fig13_17.cpp
2  // Manipulador de flujo showbase.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int x = 100;
9
10    // usa showbase para mostrar la base del número
11    cout << "Impresion de enteros, y a la derecha su base:" << endl
12       << showbase;
13
14    cout << x << endl; // imprime valor decimal
15    cout << oct << x << endl; // imprime valor octal
16    cout << hex << x << endl; // imprime valor hexadecimal
17 } // fin de main
```

Fig. 13.17 | El manipulador de flujo `showbase` (parte 1 de 2).

```
Impresion de enteros, y a la derecha su base:
100
0144
0x64
```

Fig. 13.17 | El manipulador de flujo `showbase` (parte 2 de 2).

13.7.5 Números de punto flotante; notación científica y fija (`scientific`, `fixed`)

Los manipuladores de flujos `scientific` y `fixed` controlan el formato de salida de los números de punto flotante. El manipulador de flujo **`scientific`** obliga a que la salida de un número de punto flotante se muestre en formato científico. El manipulador de flujo **`fixed`** obliga a que un número de punto flotante muestre un número específico de dígitos (según lo especificado por la función miembro `precision` o el manipulador de flujo `setprecision`) a la derecha del punto decimal. Sin usar otro manipulador, el valor del número-punto-flotante determina el formato de salida.

La figura 13.18 demuestra cómo mostrar números de punto-flotante en los formatos científico y fijo, usando los manipuladores de flujos `scientific` (línea 18) y `fixed` (línea 22). El formato exponencial en notación científica podría diferir de un compilador a otro.

```
1  // Fig. 13.18: Fig13_18.cpp
2  // Cómo mostrar los valores de punto flotante en los formatos
3  // predeterminado del sistema, científico y fijo.
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9     double x = 0.001234567;
10    double y = 1.946e9;
11
12    // muestra a x e y en el formato predeterminado
13    cout << "Mostrados en el formato predeterminado:" << endl
14       << x << '\t' << y << endl;
15
16    // muestra x e y en el formato científico
17    cout << "\nMostrados en el formato cientifico:" << endl
18       << scientific << x << '\t' << y << endl;
19
20    // muestra x e y en formato fijo
21    cout << "\nMostrados en formato fijo:" << endl
22       << fixed << x << '\t' << y << endl;
23 } // fin de main
```

```
Mostrados en el formato predeterminado:
0.00123457      1.946e+009

Mostrados en el formato cientifico:
1.234567e-003   1.946000e+009
```

Fig. 13.18 | Valores de punto flotante mostrados en los formatos predeterminado, científico y fijo (parte 1 de 2).

```
Mostrados en formato fijo:
0.001235      1946000000.000000
```

Fig. 13.18 | Valores de punto flotante mostrados en los formatos predeterminado, científico y fijo (parte 2 de 2).

13.7.6 Control de mayúsculas/minúsculas (uppercase)

El manipulador de flujo **uppercase** imprime una `X` o `E` mayúscula con valores hexadecimales enteros o con valores de punto flotante en notación científica, respectivamente (figura 13.19). El uso del manipulador de flujo `uppercase` también hace que todas las letras en un valor hexadecimal sean mayúsculas. De *manera predeterminada*, las letras para los valores hexadecimales y los exponentes en los valores de punto flotante en notación científica aparecen en *minúscula*. Para restablecer la opción de `uppercase`, hay que imprimir el manipulador de flujo **nouppercase**.

```
1  // Fig. 13.19: Fig13_19.cpp
2  // El manipulador de flujo uppercase.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     cout << "Impresion de letras mayusculas en exponentes de" << endl
9        << "notacion cientifica y valores hexadecimales:" << endl;
10
11    // usa std:uppercase para mostrar letras mayúsculas; usa std::hex y
12    // std::showbase para mostrar un valor hexadecimal y su base
13    cout << uppercase << 4.345e10 << endl
14       << hex << showbase << 123456789 << endl;
15 } // fin de main
```

```
Impresion de letras mayusculas en exponentes de
notacion cientifica y valores hexadecimales:
4.345E+010
0X75BCD15
```

Fig. 13.19 | El manipulador de flujo `uppercase`.

13.7.7 Especificación de formato booleano (boolalpha)

C++ proporciona el tipo de datos `bool`, cuyos valores pueden ser `false` o `true`, como una alternativa preferente al antiguo estilo de usar `0` para indicar `false` y un valor distinto de cero para indicar `true`. Una variable `bool` se imprime como `0` o `1` de *manera predeterminada*. Sin embargo, podemos usar el manipulador de flujo **boolalpha** para establecer el flujo de salida de manera que muestre los valores `bool` como las cadenas `"true"` y `"false"`. Use el manipulador de flujo **noboolalpha** para establecer el flujo de salida, de manera que muestre los valores `bool` como enteros (es decir, la opción predeterminada). El programa de la figura 13.20 demuestra estos manipuladores de flujos. En la línea 11 se muestra el valor `bool`, que en la línea 8 se establece en `true`, como un entero. En la línea 15 se utiliza el manipulador `boolalpha` para mostrar el valor `bool` como una cadena. Luego, en las líneas 18-19 se modifica el valor de `bool` y se usa el manipulador `noboolalpha`, por lo que en la línea 22 se puede mostrar el valor `bool` como un entero. En la línea 26 se utiliza el manipulador `boolalpha` para mostrar el valor `bool` como una cadena. Tanto `boolalpha` como `noboolalpha` son opciones pegajosas.

Buena práctica de programación 13.1

Al mostrar los valores `bool` *como* `true` *o* `false`*, en vez de mostrarlos como un valor distinto de cero o un* `0`*, respectivamente, los resultados de los programas se hacen más legibles.*

```
1  // Fig. 13.20: Fig13_20.cpp
2  // Los manipuladores de flujos boolalpha y noboolalpha.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     bool valorBooleano = true;
9
10    // muestra el valorBooleano verdadero predeterminado
11    cout << "valorBooleano es " << valorBooleano << endl;
12
13    // muestra el valorBooleano después de usar boolalpha
14    cout << "valorBooleano (despues de usar boolalpha) es "
15       << boolalpha << valorBooleano << endl << endl;
16
17    cout << "cambio de valorBooleano y uso de noboolalpha" << endl;
18    valorBooleano = false; // cambia valorBooleano
19    cout << noboolalpha << endl; // usa noboolalpha
20
21    // muestra el valorBooleano falso predeterminado después de usar noboolalpha
22    cout << "valorBooleano es " << valorBooleano << endl;
23
24    // muestra el valorBooleano después de usar boolalpha otra vez
25    cout << "valorBooleano (despues de usar boolalpha) es "
26       << boolalpha << valorBooleano << endl;
27 } // fin de main
```

```
valorBooleano es 1
valorBooleano (despues de usar boolalpha) es true

cambio de valorBooleano y uso de noboolalpha

valorBooleano es 0
valorBooleano (despues de usar boolalpha) es false
```

Fig. 13.20 | Los manipuladores de flujos `boolalpha` y `noboolalpha`.

13.7.8 Establecer y restablecer el estado de formato mediante la función miembro `flags`

En la sección 13.7 hemos estado usando manipuladores de flujos para modificar las características de formato de la salida. Ahora veremos cómo devolver el formato de un flujo de salida a su estado predeterminado después de haber aplicado varias manipulaciones. La función miembro **`flags`** sin un argumento devuelve las opciones de formato actuales como un tipo de datos **`fmtflags`** (de la clase `ios_base`), el cual representa el **estado del formato**. La función miembro `flags` con un argumento `fmtflags` establece el estado del formato según lo especificado por el argumento y devuelve las opciones de estado

anteriores. Las opciones iniciales del valor que devuelve `flags` podrían diferir de un sistema a otro. El programa de la figura 13.21 utiliza la función miembro `flags` para guardar el estado del formato original del flujo (línea 17), y después restaura las opciones de formato originales (línea 25).

```
1  // Fig. 13.21: Fig13_21.cpp
2  // La función miembro flags.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int valorEntero = 1000;
9     double valorDouble = 0.0947628;
10
11    // muestra el valor de flags, los valores int y double (formato original)
12    cout << "El valor de la variable flags es: " << cout.flags()
13       << "\nImpresion de int y double en formato original:\n"
14       << valorEntero << '\t' << valorDouble << endl << endl;
15
16    // usa la función flags de cout para guardar el formato original
17    ios_base::fmtflags formatoOriginal = cout.flags();
18    cout << showbase << oct << scientific; // cambia el formato
19
20    // muestra el valor de flags, los valores int y double (nuevo formato)
21    cout << "El valor de la variable flags es: " << cout.flags()
22       << "\nImpresion de int y double en un nuevo formato:\n"
23       << valorEntero << '\t'  << valorDouble << endl << endl;
24
25    cout.flags( formatoOriginal ); // restaura el formato
26
27    // muestra el valor de flags, los valores int y double (formato original)
28    cout << "El valor restaurado de la variable flags es: "
29       << cout.flags()
30       << "\nImpresion de los valores en su formato original otra vez:\n"
31       << valorEntero << '\t' << valorDouble << endl;
32 } // fin de main
```

```
El valor de la variable flags es: 513
Impresion de int y double en formato original:
1000    0.0947628

El valor de la variable flags es: 012011
Impresion de int y double en un nuevo formato:
01750   9.476280e-002

El valor restaurado de la variable flags es: 513
Impresion de los valores en su formato original otra vez:
1000    0.0947628
```

Fig. 13.21 | La función miembro `flags`.

13.8 Estados de error de los flujos

El estado de un flujo puede probarse a través de los bits en la clase `ios_base`. Anteriormente en el libro le indicamos que puede probar, por ejemplo, si una entrada fue exitosa. En la figura 13.22 le mostrare-

mos cómo probar estos bits de estado. En el código de uso industrial, es conveniente realizar pruebas similares en las operaciones de E/S.

```
1  // Fig. 13.22: Fig13_22.cpp
2  // Prueba de los estados de error.
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8     int valorEntero;
9
10    // muestra los resultados de las funciones de cin
11    cout << "Antes de una operacion de entrada incorrecta:"
12       << "\ncin.rdstate(): " << cin.rdstate()
13       << "\n    cin.eof(): " << cin.eof()
14       << "\n   cin.fail(): " << cin.fail()
15       << "\n    cin.bad(): " << cin.bad()
16       << "\n   cin.good(): " << cin.good()
17       << "\n\nEspera un entero, pero se introduce un caracter: ";
18
19    cin >> valorEntero; // escribe el valor tipo carácter
20    cout << endl;
21
22    // muestra los resultados de las funciones de cin después de una entrada
         incorrecta
23    cout << "Despues de una operacion de entrada incorrecta:"
24       << "\ncin.rdstate(): " << cin.rdstate()
25       << "\n    cin.eof(): " << cin.eof()
26       << "\n   cin.fail(): " << cin.fail()
27       << "\n    cin.bad(): " << cin.bad()
28       << "\n   cin.good(): " << cin.good() << endl << endl;
29
30    cin.clear(); // borra el flujo
31
32    // muestra los resultados de las funciones de cin después de borrar cin
33    cout << "Despues de cin.clear()" << "\ncin.fail(): " << cin.fail()
34       << "\ncin.good(): " << cin.good() << endl;
35 } // fin de main
```

```
Antes de una operacion de entrada incorrecta:
cin.rdstate(): 0
    cin.eof(): 0
   cin.fail(): 0
    cin.bad(): 0
   cin.good(): 1

Espera un entero, pero se introduce un caracter: A

Despues de una operacion de entrada incorrecta:
cin.rdstate(): 2
    cin.eof(): 0
   cin.fail(): 1
```

Fig. 13.22 | Prueba de los estados de error (parte 1 de 2).

```
   cin.bad(): 0
  cin.good(): 0

Despues de cin.clear()
cin.fail(): 0
cin.good(): 1
```

Fig. 13.22 | Prueba de los estados de error (parte 2 de 2).

El bit **eofbit** se establece para un flujo de entrada, al encontrar el *fin de archivo*. Un programa puede usar la función miembro **eof** para determinar si se ha encontrado el fin de archivo en un flujo después de un intento de extraer datos *más allá* del fin del flujo. La llamada

```
cin.eof()
```

devuelve `true` si se ha encontrado el fin de archivo en `cin`, y `false` en caso contrario.

El bit `failbit` se establece para un flujo cuando ocurre un *error de formato* en el mismo y no se introducen caracteres (por ejemplo, cuando tratamos de leer un *número* y el usuario introduce una *cadena*). Cuando ocurre dicho error, los caracteres *no* se pierden. La función miembro **fail** reporta si ha fallado una operación con un flujo. Por lo general, es posible recuperarse de dichos errores.

El bit `badbit` se establece para un flujo cuando ocurre un error que produce la *pérdida de datos*. La función miembro **bad** reporta si *falló* una operación con un flujo. Por lo general, no es posible recuperarse de dichas fallas graves.

El bit **goodbit** se establece para un flujo si no se establece *ninguno* de los bits `eofbit`, `failbit` o `badbit` para el flujo.

La función miembro **good** devuelve `true` si *todas* las funciones `bad`, `fail` y `eof` devuelven `false`. Las operaciones de E/S deben realizarse sólo en flujos "buenos".

La función miembro **rdstate** devuelve el *estado de error* del flujo. Por ejemplo, una llamada a `cout.rdstate` devolvería el estado del flujo, que entonces se podría evaluar mediante una instrucción `switch` que examine los bits `eofbit`, `badbit`, `failbit` y `goodbit`. La forma *preferente* de evaluar el estado de un flujo es mediante el uso de las funciones miembro `eof`, `bad`, `fail` y `good`; el uso de estas funciones no requiere que el programador esté familiarizado con los bits de estado específicos.

La función miembro **clear** se utiliza para *restaurar* el estado de un flujo a "bueno", de manera que la E/S pueda proceder en ese flujo. El argumento predeterminado para `clear` es `goodbit`, por lo que la instrucción

```
cin.clear();
```

limpia `cin` y establece el bit `goodbit` para el flujo. La instrucción

```
cin.clear( ios::failbit )
```

establece el bit `failbit`. Tal vez el programador desee hacer esto al realizar operaciones de entrada en `cin` con un tipo definido por el usuario y toparse con un problema. El nombre `clear` podría parecer inapropiado en este contexto, pero es correcto.

El programa de la figura 13.22 muestra las funciones miembro `rdstate`, `eof`, `fail`, `bad`, `good` y `clear`. Los valores actuales que se impriman podrían ser distintos de un compilador a otro.

La función miembro `operator!` de `basic_ios` devuelve `true` si se establece el bit `badbit`, si se establece el bit `failbit` o si se establecen *ambos*. La función miembro `operator void *` devuelve `false` (0) si se establece el bit `badbit`, si se establece el bit `failbit` o si se establecen ambos. Estas funciones son útiles en el procesamiento de archivos cuando se está evaluando una condición `true/false` bajo el control de una instrucción de selección o de repetición.

13.9 Enlazar un flujo de salida a un flujo de entrada

Por lo general, las aplicaciones interactivas implican un objeto `istream` para entrada y un objeto `ostream` para salida. Cuando aparece un mensaje de petición en la pantalla, el usuario responde introduciendo los datos apropiados. Obviamente, la petición necesita aparecer *antes* de que proceda la operación de entrada. Con el uso de búfer en la salida, las salidas aparecen sólo cuando se *llena* el búfer, cuando las salidas se *vacían* de manera explícita por el programa, o de manera automática al final del programa. C++ proporciona la función miembro **`tie`** para sincronizar (es decir, "enlazar entre sí") la operación de un objeto `istream` y un objeto `ostream` para asegurar que los resultados aparezcan *antes* de sus entradas subsiguientes. La llamada

```
cin.tie( &cout );
```

enlaza a `cout` (un objeto `ostream`) con `cin` (un objeto `istream`). En realidad, esta llamada específica es redundante, debido a que C++ realiza esta operación de manera automática para crear un entorno de entrada/salida estándar para el usuario. Sin embargo, el usuario podría enlazar otros pares `istream`/`ostream` de manera explícita. Para desenlazar un flujo de entrada (`flujoEntrada`) de un flujo de salida, utilice la llamada

```
flujoEntrada.tie( 0 );
```

13.10 Conclusión

En este capítulo sintetizamos la forma en que C++ realiza las operaciones de entrada/salida mediante el uso de flujos. El lector aprendió acerca de las clases y objetos de E/S de flujos, así como la jerarquía de clases de plantilla de E/S de flujos. Hablamos sobre las herramientas de salida con formato y sin formato de `ostream` que realizan las funciones `put` y `write`. Vimos ejemplos acerca del uso de las herramientas de entrada con formato y sin formato de `istream` realizadas por las funciones `eof`, `get`, `getline`, `peek`, `putback`, `ignore` y `read`. Después hablamos sobre los manipuladores de flujos y las funciones miembro que realizan tareas de formato: `dec`, `oct`, `hex` y `setbase` para mostrar enteros; `precisión` y `setprecision` para controlar la precisión de punto flotante; y `width` y `setw` para establecer la anchura de campo. También aprendió acerca de los manipuladores `iostream` para formato adicional y acerca de las funciones miembro: `showpoint` para mostrar el punto decimal y ceros a la derecha; `left`, `right` e `internal` para la justificación; `fill` y `setfill` para rellenar con caracteres; `scientific` y `fixed` para mostrar números de punto flotante en notación científica y fija; `uppercase` para el control de mayúsculas/minúsculas; `boolalpha` para especificar el formato booleano, y `flags` junto con `fmtflags` para restablecer el estado del formato.

En el siguiente capítulo aprenderá sobre el procesamiento de archivos, incluyendo cómo se almacenan los datos persistentes y cómo manipularlos.

Resumen

Sección 13.1 Introducción

- Las operaciones de E/S se realizan de una manera sensible al tipo de los datos.

Sección 13.2 Flujos

- En C++, las operaciones de E/S se realizan en flujos (pág. 564). Un flujo es una secuencia de bytes.
- Las herramientas de E/S de bajo nivel especifican que los bytes deben transferirse de dispositivo a memoria, o de memoria a dispositivo. La E/S de alto nivel se realiza con los bytes agrupados en unidades significativas tales como enteros, cadenas y tipos definidos por el usuario.

- C++ proporciona operaciones de E/S con formato y sin formato. Las transferencias de E/S sin formato (pág. 564) son rápidas, pero procesan datos puros que son difíciles de usar para las personas. La E/S con formato procesa los datos en unidades significativas, pero requiere un tiempo de procesamiento adicional que puede degradar el rendimiento.
- El encabezado `<iostream>` declara todas las operaciones de E/S con flujos (pág. 565).
- El encabezado `<iomanip>` declara los manipuladores de flujos parametrizados (pág. 565).
- El encabezado `<fstream>` declara las operaciones de procesamiento de archivos (pág. 567).
- La plantilla `basic_istream` (pág. 565) soporta las operaciones de entrada con flujos.
- La plantilla `basic_ostream` (pág. 565) soporta las operaciones de salida con flujos.
- La plantilla `basic_iostream` soporta las operaciones de entrada y de salida con flujos.
- Las plantillas `basic_istream` y `basic_ostream` se derivan de la plantilla `basic_ios` (pág. 565).
- La plantilla `basic_iostream` se deriva a través de las plantillas `basic_istream` y `basic_ostream`.
- El objeto `cin` de `istream` está enlazado al dispositivo de entrada estándar, que por lo general es el teclado.
- El objeto `cout` de `ostream` está enlazado al dispositivo de salida estándar, que por lo general es la pantalla.
- El objeto `cerr` de `ostream` está enlazado al dispositivo de error estándar, que por lo general es la pantalla. Las operaciones de salida con `cerr` no usan búfer (pág. 567); cada inserción en `cerr` aparece de inmediato.
- El objeto `clog` de `ostream` está enlazado al dispositivo de error estándar, que por lo general es la pantalla. Las operaciones de salida con `clog` usan búfer (pág. 567).
- El compilador de C++ determina los tipos de datos de manera automática para las operaciones de entrada y de salida.

Sección 13.3 Salida de flujos

- Las direcciones se despliegan en formato hexadecimal de manera predeterminada.
- Para imprimir una dirección en una variable apuntador, se convierte el apuntador a `void *`.
- La función miembro `put` imprime un carácter. Las llamadas a put se pueden hacer en cascada.

Sección 13.4 Entrada de flujos

- Las operaciones de entrada de flujos se realizan con el operador de extracción de flujo >>, que omite de manera automática los caracteres de espacio en blanco (pág. 569) en el flujo de entrada y devuelve `false` después de encontrar el fin de archivo.
- La extracción de flujo hace que se establezca el bit `failbit` (pág. 569) para los datos de entrada incorrectos, y que se establezca el bit `badbit` (pág. 569) si la operación falla.
- Se puede introducir una serie de valores mediante el uso del operador de extracción de flujo en el encabezado de un ciclo `while`. La extracción devuelve `0` al encontrarse con el fin de archivo o cuando ocurre un error.
- La función miembro `get` (pág. 569) sin argumentos introduce un carácter y devuelve ese carácter; se devuelve `EOF` al encontrar el fin de archivo en el flujo.
- La función miembro `get` con un argumento de referencia de carácter introduce el siguiente carácter del flujo de entrada y lo almacena en el argumento tipo carácter. Esta versión de `get` devuelve una referencia al objeto `istream` (pág. 565) para el que se está invocando la función miembro `get`.
- La función miembro `get` con tres argumentos [un arreglo de caracteres, un límite de tamaño y un delimitador (con el valor predeterminado de nueva línea)] lee caracteres del flujo de entrada hasta un máximo de caracteres equivalente al límite – 1, o hasta que se lee el delimitador. La cadena de entrada se termina con un carácter nulo. El delimitador no se coloca en el arreglo de caracteres, pero permanece en el flujo de entrada.
- La función miembro `getline` (pág. 571) opera como la función miembro `get` de tres argumentos. La función `getline` elimina el delimitador del flujo de entrada, pero no lo almacena en la cadena.
- La función miembro `ignore` (pág. 572) omite el número especificado de caracteres (el valor predeterminado es 1) en el flujo de entrada; termina al encontrar el delimitador especificado (el delimitador predeterminado es `EOF`).

- La función miembro `putback` (pág. 572) coloca el carácter anterior obtenido mediante `get` en un flujo, de vuelta a ese flujo.
- La función miembro `peek` (pág. 572) devuelve el siguiente carácter de un flujo de entrada, pero no lo extrae (elimina) del flujo.
- C++ ofrece la E/S con seguridad de tipos (pág. 572). Si los operadores << y >> procesan datos inesperados se establecen varios bits de error, que el usuario puede usar para determinar si una operación de E/S tuvo éxito o falló. Si el operador << no se ha sobrecargado para un tipo definido por el usuario, se reporta un error de compilación.

Sección 13.5 E/S sin formato mediante el uso de `read`, `write` *y* `gcount`

- La E/S sin formato se lleva a cabo con las funciones miembro `read` y `write` (pág. 572). Éstas envían o reciben cierto número de bytes hacia/desde la memoria, empezando en una dirección de memoria designada.
- La función miembro `gcount` (pág. 573) devuelve el número de caracteres introducidos por la operación `read` anterior en ese flujo.
- La función miembro `read` introduce un número especificado de caracteres en un arreglo de caracteres. El bit `failbit` se establece si se leen menos caracteres que el número especificado.

Sección 13.6 Introducción a los manipuladores de flujos

- Para cambiar la base en la que se imprimen los enteros, se utiliza el manipulador `hex` (pág. 574) para establecer la base en hexadecimal (base 16), o el manipulador `oct` (pág. 574) para establecer la base en octal (base 8). El manipulador `dec` (pág. 574) se utiliza para restablecer la base a decimal. La base permanece igual hasta que se cambia de manera explícita.
- El manipulador de flujo parametrizado `setbase` (pág. 574) también establece la base para las operaciones de salida con enteros. El manipulador `setbase` recibe un argumento entero de `10`, `8` o `16` para establecer la base.
- La precisión de punto flotante se puede controlar mediante el manipulador de flujo `setprecision`, o mediante la función miembro `precision` (pág. 574). Ambos establecen la precisión para todas las operaciones subsiguientes de salida, hasta la siguiente llamada para establecer la precisión. La función miembro `precision` sin argumento devuelve el valor de precisión actual.
- Los manipuladores parametrizados requieren la inclusión del archivo de encabezado `<iomanip>`.
- La función miembro `width` (pág. 576) establece la anchura de campo y devuelve la anchura anterior. Los valores de menor anchura que el campo se rellenan con caracteres de relleno (pág. 576). La opción de anchura de campo se aplica sólo para la siguiente inserción o extracción, después la entrada se realiza mediante el uso de las opciones predeterminadas. Los valores más anchos que un campo se imprimen en su totalidad. La función `width` sin argumento devuelve la opción de anchura actual. El manipulador `setw` también establece la anchura.
- Para las operaciones de entrada, el manipulador de flujo `setw` establece un tamaño máximo de cadena; si se introduce una cadena más grande, la línea más grande se divide en piezas que no sean mayores que el tamaño designado.
- Los programadores pueden crear sus propios manipuladores de flujos.

Sección 13.7 Estados de formato de flujos y manipuladores de flujos

- El manipulador de flujo `showpoint` (pág. 579) obliga a que un número de punto flotante se imprima con un punto decimal, y con el número de dígitos significativos especificado por la precisión.
- Los manipuladores de flujos `left` y `right` (pág. 580) hacen que los campos se justifiquen a la izquierda con caracteres de relleno a la derecha, o que se justifiquen a la derecha con caracteres de relleno a la izquierda.
- El manipulador de flujo `internal` (pág. 581) indica que el signo de un número (o la base, cuando se utiliza el manipulador de flujo `showbase`; pág. 583) debe justificarse a la izquierda dentro de un campo, su magnitud debe justificarse a la derecha y los espacios intermedios deben rellenarse con el carácter de relleno.
- La función miembro `fill` (pág. 582) especifica el carácter de relleno a usar con los manipuladores de flujos `left`, `right` e `internal` (el espacio es el valor predeterminado); se devuelve el carácter de relleno anterior. El manipulador de flujo `setfill` (pág. 582) también establece el carácter de relleno.

- Los manipuladores de flujos `oct`, `hex` y `dec` especifican que los enteros se van a tratar como valores octales, hexadecimales o decimales, respectivamente. Los valores predeterminados de las operaciones de salida con enteros están en decimal si no se establece ninguno de estos bits; las extracciones de flujo procesan los datos en la forma en que éstos se suministran.
- El manipulador de flujo `showbase` obliga a que se imprima la base de un valor integral.
- El manipulador de flujo `scientific` (pág. 584) se utiliza para imprimir un número de punto flotante en formato científico. El manipulador de flujo `fixed` (pág. 584) se utiliza para imprimir un número de punto flotante con la precisión especificada mediante la función miembro `precision`.
- El manipulador de flujo `uppercase` (pág. 579) imprime una `X` o `E` mayúscula para los enteros hexadecimales y los valores de punto flotante en notación científica, respectivamente. Los valores hexadecimales aparecen sólo en mayúsculas.
- La función miembro `flags` (pág. 586) sin argumento devuelve el estado del formato actual (pág. 586) como un valor `long`. La función `flags` con un argumento `long` establece el estado del formato especificado mediante el argumento.

Sección 13.8 Estados de error de los flujos

- El estado de un flujo se puede evaluar mediante los bits en la clase `ios_base`.
- El bit `eofbit` (pág. 589) se establece para un flujo de entrada, una vez que se encuentra el fin de archivo durante una operación de entrada. La función miembro `eof` (pág. 589) reporta si se ha establecido el bit `eofbit`.
- El bit `failbit` de un flujo se establece cuando ocurre un error de formato. La función miembro `fail` (pág. 589) reporta si ha fallado una operación de flujo; por lo general es posible recuperarse de dichos errores.
- El bit `badbit` de un flujo se establece cuando ocurre un error que provoca la pérdida de datos. La función miembro `bad` reporta si dicha operación de flujo falló. Dichas fallas graves por lo general son irrecuperables.
- La función miembro `good` (pág. 589) devuelve verdadero si todas las funciones `bad`, `fail` y `eof` devuelven `false`. Las operaciones de E/S deben realizarse sólo en los flujos "buenos".
- La función miembro `rdstate` (pág. 589) devuelve el estado de error del flujo.
- La función miembro `clear` (pág. 589) restaura el estado de un flujo a "bueno", para que puedan continuar las operaciones de E/S.

Sección 13.9 Enlazar un flujo de salida a un flujo de entrada

- C++ proporciona la función miembro `tie` (pág. 590) para sincronizar las operaciones con `istream` y `ostream`, para asegurar que los resultados aparezcan antes de las entradas subsiguientes.

Ejercicios de autoevaluación

13.1 *(Llene los espacios en blanco)* Complete los siguientes enunciados:

a) La entrada/salida en C++ ocurre en forma de _____ de bytes.
b) Los manipuladores de flujos que dan formato a la justificación son _____, _____ y _____.
c) La función miembro _____ se puede utilizar para establecer y restablecer el estado del formato.
d) La mayoría de los programas de C++ que realizan operaciones de E/S deben incluir el encabezado _____ que contiene las declaraciones requeridas para todas las operaciones de E/S de flujos.
e) Al utilizar manipuladores parametrizados, se debe incluir el encabezado _____.
f) El encabezado _____ contiene las declaraciones requeridas para el procesamiento de archivos.
g) La función miembro _____ de `ostream` se utiliza para realizar operaciones de salida sin formato.
h) Las operaciones de entrada están soportadas por la clase _____.
i) Las operaciones de salida del flujo de error estándar se dirigen a los objetos flujo _____ o _____.
j) Las operaciones de salida están soportadas por la clase _____.
k) El símbolo para el operador de inserción de flujo es _____.
l) Los cuatro objetos que corresponden a los dispositivos estándar en el sistema son _____, _____, _____ y _____.
m) El símbolo para el operador de extracción de flujo es _____.

n) Los manipuladores de flujos _____, _____ y _____ especifican que los enteros se deben mostrar en los formatos octal, hexadecimal y decimal, respectivamente.
o) El manipulador de flujo _____ hace que los números positivos se muestren con un signo positivo.

13.2 ***(Verdadero o falso)*** Indique si cada uno de los siguientes enunciados es *verdadero* o *falso*. En caso de ser *falso*, explique por qué.

a) La función miembro `flags` de un flujo con un argumento `long` establece la variable de estado `flags` con su argumento y devuelve su valor anterior.
b) El operador de inserción de flujo << y el operador de extracción de flujo >> se sobrecargan para manejar todos los tipos de datos estándar [incluyendo cadenas y direcciones de memoria (sólo el de inserción de flujo)] y todos los tipos de datos definidos por el usuario.
c) La función miembro `flags` de un flujo sin argumentos restablece el estado de formato del flujo.
d) El operador de extracción de flujo >> se puede sobrecargar con una función operador que reciba como argumentos una referencia `istream` y una referencia a un tipo definido por el usuario, y devuelva una referencia `istream`.
e) El operador de inserción de flujo << se puede sobrecargar con una función operador que reciba como argumentos una referencia `istream` y una referencia a un tipo definido por el usuario, y devuelva una referencia `istream`.
f) La entrada con el operador de extracción de flujo >> siempre omite los caracteres de espacio en blanco a la izquierda en el flujo de entrada, de manera predeterminada.
g) La función miembro `rdstate` de un flujo devuelve el estado actual del flujo.
h) Por lo general, el flujo `cout` está conectado a la pantalla.
i) La función miembro `good` de un flujo devuelve `true` si todas las funciones miembro `bad`, `fail` y `eof` devuelven `false`.
j) Por lo general, el flujo `cin` está conectado a la pantalla.
k) Si ocurre un error irrecuperable durante una operación de un flujo, la función miembro `bad` devolverá `true`.
l) La salida a `cerr` no usa búfer y la salida a `clog` tiene búfer.
m) El manipulador de flujo `showpoint` obliga a que los valores de punto flotante se impriman con los seis dígitos predeterminados de precisión, a menos que se haya modificado el valor de precisión, en cuyo caso los valores de punto flotante se imprimen con la precisión especificada.
n) La función miembro `put` de `ostream` imprime el número especificado de caracteres.
o) Los manipuladores de flujos `dec`, `oct` y `hex` sólo afectan a la siguiente operación de entrada con enteros.

13.3 ***(Escriba una instrucción en C++)*** Para cada uno de los siguientes enunciados, escriba una sola instrucción que realice la tarea indicada.

a) Imprimir la cadena `"Escriba su nombre: "`.
b) Usar un manipulador de flujo que haga que el exponente en la notación científica y las letras en los valores hexadecimales se impriman en mayúsculas.
c) Imprimir la dirección de la variable `miString` de tipo `char *`.
d) Usar un manipulador de flujo para asegurar que los valores de punto flotante se impriman en notación científica.
e) Imprimir la dirección en la variable `enteroPtr` de tipo `int *`.
f) Usar un manipulador de flujo de tal forma que, cuando se impriman valores enteros, se muestre la base entera para los valores octales y hexadecimales.
g) Imprimir el valor al que apunta `floatPtr` de tipo `float *`.
h) Usar una función miembro de flujo para establecer el carácter de relleno en `'*'` e imprimir en anchuras de campo mayores que los valores que se van a imprimir. Repita esta instrucción con un manipulador de flujo.
i) Imprimir los caracteres `'0'` y `'K'` en una instrucción con la función `put` de `ostream`.
j) Obtener el valor del siguiente carácter a introducir, sin extraerlo del flujo.
k) Introducir un solo carácter en la variable `valorChar` de tipo `char`, usando la función miembro `get` de `istream` en dos maneras distintas.
l) Introducir y descartar los siguientes seis caracteres en el flujo de entrada.
m) Usar la función miembro `read` de `istream` para introducir 50 caracteres en el arreglo `línea` tipo `char`.

n) Leer 10 caracteres y colocarlos en el arreglo de caracteres `nombre`. Dejar de leer caracteres al encontrar el delimitador `'.'`. No elimine el delimitador del flujo de entrada. Escriba otra instrucción que realice esta tarea y elimine el delimitador de la entrada.
o) Usar la función miembro `gcount` de `istream` para determinar el número de caracteres introducidos en el arreglo de caracteres `linea` mediante la última llamada a la función miembro `read` de `istream`, e imprimir ese número de caracteres, usando la función miembro `write` de `ostream`.
p) Imprimir `124`, `18.376`, `'Z'`, `1000000` y `"Cadena"` separados por espacios.
q) Mostrar la opción de precisión actual de `cout`.
r) Introducir un valor entero en la variable `int` llamada `meses`, y un valor de punto flotante en la variable `float` llamada `tasaPorcentaje`.
s) Imprimir `1.92`, `1.925` y `1.9258` separados por tabuladores y con `3` dígitos de precisión, usando un manipulador de flujo.
t) Imprimir el entero `100` en octal, hexadecimal y decimal, usando manipuladores de flujos y separado por tabuladores.
u) Imprimir el entero `100` en decimal, octal y hexadecimal separado por tabuladores, usando un manipulador de flujo para cambiar la base.
v) Imprimir `1234` justificado a la derecha en un campo de 10 dígitos.
w) Leer los caracteres en el arreglo de caracteres `linea` hasta encontrar el carácter `'z'`, hasta un límite de 20 caracteres (incluyendo un carácter nulo de terminación). No extraiga el carácter delimitador del flujo.
x) Usar las variables enteras `x` y `y` para especificar la anchura de campo y precisión utilizadas para mostrar el valor `double 87.4573`, y mostrar el valor.

13.4 ***(Buscar y corregir errores de código)*** Identifique el error en cada una de las siguientes instrucciones y explique cómo corregirlo.

a)
```
cout << "El valor de x <= y es: " << x <= y;
```
b) La siguiente instrucción debe imprimir el valor entero de `'c'`.
```
cout << 'c';
```
c)
```
cout << ""Una cadena entre comillas"";
```

13.5 ***(Mostrar resultados)*** Para cada uno de los siguientes incisos, muestre los resultados.

a)
```
cout << "12345" << endl;
cout.width( 5 );
cout.fill( '*' );
cout << 123 << endl << 123;
```
b)
```
cout << setw( 10 ) << setfill( '$' ) << 10000;
```
c)
```
cout << setw( 8 ) << setprecision( 3 ) << 1024.987654;
```
d)
```
cout << showbase << oct << 99 << endl << hex << 99;
```
e)
```
cout << 100000 << endl << showpos << 100000;
```
f)
```
cout << setw( 10 ) << setprecision( 2 ) << scientific << 444.93738;
```

Respuestas a los ejercicios de autoevaluación

13.1 a) flujos. b) `left`, `right` e `internal`. c) `flags`. d) `<iostream>`. e) `<iomanip>`. f) `<fstream>`. g) `write`. h) `istream`. i) `cerr` o `clog`. j) `ostream`. k) `<<`. l) `cin`, `cout`, `cerr` y `clog`. m) `>>`. n) `oct`, `hex` y `dec`. o) `showpos`.

13.2 a) Falso. La función miembro `flags` de un flujo con un argumento `fmtflags` establece la variable de estado `flags` con su argumento y devuelve las opciones de estado anteriores. b) Falso. Los operadores de inserción de flujo y de extracción de flujo no se sobrecargan para todos los tipos definidos por el usuario. El programador de una clase debe proporcionar de manera específica las funciones operador sobrecargadas, para sobrecargar los operadores de flujo y usarlos con cada tipo definido por el usuario que vaya a crear. c) Falso. La función miembro `flags` de un flujo sin argumentos devuelve las opciones de formato actuales como un tipo de datos

`fmtflags`, el cual representa el estado del formato. d) Verdadero. e) Falso. Para sobrecargar el operador de inserción de flujo <<, la función operador sobrecargada debe recibir como argumentos una referencia `ostream` y una referencia a un tipo definido por el usuario, y devolver una referencia `ostream`. f) Verdadero. g) Verdadero. h) Verdadero. i) Verdadero. j) Falso. El flujo `cin` está conectado a la entrada estándar de la computadora, que por lo general es el teclado. k) Verdadero. l) Verdadero. m) Verdadero. n) Falso. La función miembro `put` de `ostream` imprime su único argumento tipo carácter. o) Falso. Los manipuladores de flujos `dec`, `oct` y `hex` establecen el estado del formato de salida para los enteros con la base especificada, hasta que se cambia la base de nuevo o el programa termina.

13.3 a)
```
cout << "Escriba su nombre: ";
```
b)
```
cout << uppercase;
```
c)
```
cout << static_cast< void * >( myString );
```
d)
```
cout << scientific;
```
e)
```
cout << enteroPtr;
```
f)
```
cout << showbase;
```
g)
```
cout << *floatPtr;
```
h)
```
cout.fill( '*' );
cout << setfill( '*' );
```
i)
```
cout.put( 'O' ).put( 'K' );
```
j)
```
cin.peek();
```
k)
```
valorChar = cin.get();
cin.get( valorChar );
```
l)
```
cin.ignore( 6 );
```
m)
```
cin.read( linea, 50 );
```
n)
```
cin.get( nombre, 10, '.' );
cin.getline( nombre, 10, '.' );
```
o)
```
cout.write( linea, cin.gcount() );
```
p)
```
cout << 124 << ' ' << 18.376 << ' ' << "Z " << 1000000 << " String";
```
q)
```
cout << cout.precision();
```
r)
```
cin >> meses >> tasaPorcentaje;
```
s)
```
cout << setprecision( 3 ) << 1.92 << " " << 1.925 << " " << 1.9258;
```
t)
```
cout << oct << 100 << " " << hex << 100 << " " << dec << 100;
```
u)
```
cout << 100 << " " << setbase( 8 ) << 100 << " " << setbase( 16 ) << 100;
```
v)
```
cout << setw( 10 ) << 1234;
```
w)
```
cin.get( linea, 20, 'z' );
```
x)
```
cout << setw( x ) << setprecision( y ) << 87.4573;
```

13.4 a) *Error*: la precedencia del operador << es mayor que la de <=, lo cual hace que la instrucción se evalúe en forma incorrecta y también produce un error de compilación.
Corrección: coloque paréntesis alrededor de la expresión `x <= y`.

b) *Error:* en C++ los caracteres no se tratan como enteros pequeños, como en C.
Corrección: para imprimir el valor numérico para un carácter en el conjunto de caracteres de la computadora, el carácter debe convertirse en un valor entero, como en la siguiente instrucción:
```
cout << static_cast< int >( 'c' );
```

c) *Error:* los caracteres de comillas no se pueden imprimir en una cadena, a menos que se utilice una secuencia de escape.
Corrección: imprima la cadena:
```
cout << "\"Una cadena entre comillas\"";
```

13.5 a)
```
12345
**123
123
```
b)
```
$$$$$10000
```
c)
```
1024.988
```

d) `0143`
   `0x63`
e) `100000`
   `+100000`
f) `4.45e+002`

Ejercicios

13.6 ***(Escriba instrucciones en C++)*** Escriba una instrucción para cada uno de los siguientes incisos:
a) Imprimir el entero `40000` justificado a la izquierda en un campo de 15 dígitos.
b) Leer una cadena y colocarla en la variable tipo arreglo de caracteres llamada `estado`.
c) Imprimir `200` con y sin un signo.
d) Imprimir el valor decimal `100` en formato hexadecimal con el prefijo `0x`.
e) Leer caracteres en el arreglo `arregloChar` hasta encontrar el carácter `'p'`, hasta un límite de 10 caracteres (incluyendo el carácter nulo de terminación). Extraiga el delimitador del flujo de entrada, y descártelo.
f) Imprimir `1.234` en un campo de 9 dígitos con ceros a la izquierda.

13.7 ***(Introducir valores decimales, octales y hexadecimales)*** Escriba un programa para evaluar la introducción de valores enteros en los formatos decimal, octal y hexadecimal. Imprima cada entero leído por el programa en los tres formatos. Evalúe el programa con los siguientes datos de entrada: `10`, `010`, `0x10`.

13.8 ***(Imprimir valores de apuntadores como enteros)*** Escriba un programa que imprima valores de apuntadores, usando conversiones de tipos a todos los tipos de datos enteros. ¿Cuáles imprimen valores extraños? ¿Cuáles producen errores?

13.9 ***(Imprimir con anchuras de campos)*** Escriba un programa para evaluar los resultados de imprimir el valor entero `12345` y el valor de punto flotante `1.2345` en campos de diversos tamaños. ¿Qué ocurre cuando se imprimen los valores en campos que contengan menos dígitos que los valores?

13.10 ***(Redondeo)*** Escriba un programa que imprima el valor `100.453627` redondeado a la unidad, décima, centésima, milésima o diezmilésima más cercanas.

13.11 ***(Longitud de una cadena)*** Escriba un programa que reciba una cadena del teclado y determine la longitud de la cadena. Imprima la cadena en una anchura de campo que sea el doble de la longitud de la cadena.

13.12 ***(Conversión de Fahrenheit a Centígrados)*** Escriba un programa que convierta temperaturas Fahrenheit enteras, de `0` a `212` grados, a temperaturas en grados Centígrados de punto flotante, con `3` dígitos de precisión. Utilice la siguiente fórmula:

```
centigrados = 5.0 / 9.0 * ( fahrenheit - 32 );
```

para realizar el cálculo. Los resultados deben imprimirse en dos columnas justificadas a la derecha, y las temperaturas en grados Centígrados se les debe anteponer un signo, tanto para los valores positivos como negativos.

13.13 En ciertos lenguajes de programación, las cadenas se introducen entre comillas sencillas o dobles. Escriba un programa que lea las tres cadenas `suzy`, `"suzy"` y `'suzy'`. ¿Se ignoran las comillas simples y dobles, o se leen como parte de la cadena?

13.14 ***(Leer números telefónicos con el operador de extracción de flujo sobrecargado)*** En la figura 10.5 se sobrecargaron los operadores de extracción de flujo y de inserción de flujo para las operaciones de entrada y salida con objetos de la clase `NumeroTelefonico`. Vuelva a escribir el operador de extracción de flujo para realizar la siguiente comprobación de errores en la entrada. Se tendrá que volver a implementar la función `operator>>`.

Introduzca el número telefónico completo en un arreglo. Pruebe que se haya introducido el número apropiado de caracteres. Debe haber un total de 14 caracteres leídos para un número telefónico de la forma `(800) 555-1212`. Use la función miembro `clear` de `ios_base` para establecer el bit `failbit` para la entrada inapropiada.

g) El código de área y el intercambio no empiezan con `0` o `1`. Pruebe el primer dígito de las porciones del código de área y del intercambio del número telefónico para asegurar que ninguna empiece con `0` o `1`. Use la función miembro `clear` de `ios_base` para establecer el bit `failbit` para una entrada incorrecta.

h) El dígito intermedio de un código de área solía limitarse a `0` o `1` (aunque esto ha cambiado recientemente). Pruebe el dígito intermedio para un valor de `0` o `1`. Use la función miembro `clear` de `ios_base` para establecer el bit `failbit` para una entrada incorrecta. Si ninguna de las operaciones anteriores provoca que se establezca el bit `failbit` para una entrada incorrecta, copie las tres partes del número telefónico en los miembros `codigoArea`, `intercambio` y `linea` del objeto `NumeroTelefonico`. Si se estableció `failbit` en la entrada, haga que el programa imprima un mensaje de error y termine, en vez de imprimir el número telefónico.

13.15 *(Clase `Punto`)* Escriba un programa que realice cada una de las siguientes acciones:

a) Crear una clase `Punto` definida por el usuario que contenga los datos miembro privados enteros `coordenadaX` y `coordenadaY`, y declarar las funciones de los operadores sobrecargados de inserción de flujo y extracción de flujo como funciones `friend` de la clase.

b) Definir las funciones de los operadores de inserción de flujo y extracción de flujo. La función del operador de extracción de flujo debe determinar si los datos introducidos son válidos y, en caso contrario, debe establecer el bit `failbit` para indicar una entrada incorrecta. El operador de inserción de flujo no debe mostrar el punto después de que ocurra un error en la entrada.

c) Escriba una función `main` que pruebe la entrada y salida de la clase `Punto` definida por el usuario, usando los operadores sobrecargados de extracción de flujo y de inserción de flujo.

13.6 *(Clase `Complejo`)* Escriba un programa que realice cada una de las siguientes acciones:

a) Crear una clase `Complejo` definida por el usuario que contenga los miembros de datos enteros `real` e `imaginario`, y que declare las funciones de los operadores sobrecargados de inserción de flujo y extracción de flujo como funciones `friend` de la clase.

b) Definir las funciones de los operadores de inserción de flujo y de extracción de flujo. La función del operador de extracción de flujo debe determinar si los datos introducidos son válidos y, en caso contrario, debe establecer el bit `failbit` para indicar una entrada incorrecta. La entrada deberá establecerse de manera que sea de la siguiente forma:

```
3 + 8i
```

c) Los valores pueden ser negativos o positivos, y es posible que no se proporcione uno de los dos valores, en cuyo caso los miembros de datos apropiados se deberán establecer en `0`. El operador de inserción de flujo no deberá mostrar el punto si ocurrió un error de entrada. Para los valores imaginarios negativos, debe imprimirse un signo negativo en vez de un signo positivo.

d) Escriba una función `main` que pruebe la entrada y salida de la clase `Complejo` definida por el usuario, usando los operadores sobrecargados de extracción de flujo y de inserción de flujo.

13.17 *(Imprimir una tabla de valores ASCII)* Escriba un programa que utilice una instrucción `for` para imprimir una tabla de valores ASCII para los caracteres en el conjunto de caracteres ASCII de `33` a `126`. El programa debe imprimir el valor decimal, valor octal, valor hexadecimal y valor de carácter para cada carácter. Use los manipuladores de flujos `dec`, `oct` y `hex` para imprimir los valores enteros.

13.18 *(Carácter nulo de terminación de cadenas)* Escriba un programa para mostrar que la función miembro `getline` y la función miembro `get` con tres argumentos de `istream` terminan la cadena de entrada con un carácter nulo de terminación de cadenas. Además, muestre que `get` deja el carácter delimitador en el flujo de entrada, mientras que `getline` extrae el carácter delimitador y lo descarta. ¿Qué ocurre con los caracteres en el flujo que no se leen?

14

Procesamiento de archivos

Una gran memoria no hace a un filósofo; cualquier cosa que sea más que un diccionario se le puede llamar gramática.

—John Henry, Cardenal Newman

Sólo puedo suponer que un documento "No archivar" se archiva en un archivo "No archivar".

—Senador Frank Church
Audiencia del subcomité de inteligencia del Senado, 1975

Objetivos

En este capítulo aprenderá a:

- Crear, leer, escribir y actualizar archivos.
- Procesar archivos secuenciales.
- Procesar archivos de acceso aleatorio.
- Utilizar operaciones de E/S sin formato de alto rendimiento.
- Conocer las diferencias entre los datos con formato y el procesamiento de archivos de datos puros.
- Crear un programa para procesar transacciones, usando el procesamiento de archivos de acceso aleatorio.
- Comprender el concepto de la serialización de objetos.

Plan general

14.1 Introducción

El almacenamiento de datos en memoria es *temporal*. Los **archivos** se utilizan para la **persistencia de los datos**; la retención *permanente* de los datos. Las computadoras almacenan archivos en **dispositivos de almacenamiento secundario** como discos duros, CD, DVD, unidades flash y cintas magnéticas. En este capítulo explicaremos cómo construir programas en C++ para crear, actualizar y procesar archivos de datos. Consideraremos los *archivos secuenciales* y los *archivos de acceso aleatorio*. (En el capítulo 21, en inglés en el sitio web, puede comparar el procesamiento de archivos de *datos con formato* y el procesamiento de archivos de *datos puros*, también se examinan las técnicas para introducir y enviar datos a flujos `string` en vez de archivos).

14.2 Archivos y flujos

C++ considera a cada archivo como una *secuencia de bytes* (figura 14.1). Cada archivo termina con un **marcador de fin de archivo** o con un número de bytes específico que se registra en una estructura de datos administrativa, mantenida por el sistema operativo. Cuando se *abre* un archivo, se crea un objeto y se asocia un flujo a ese objeto. En el capítulo 13 vimos que los objetos `cin`, `cout`, `cerr` y `clog` se crean cuando se incluye `<iostream>`. Los flujos asociados con estos objetos proporcionan canales de comunicación entre un programa y un archivo o dispositivo específico. Por ejemplo, el objeto `cin` (objeto flujo de entrada estándar) permite a un programa introducir datos desde el teclado o desde otros dispositivos, el objeto `cout` (objeto flujo de salida estándar) permite a un programa enviar datos a la pantalla o a otros dispositivos, y los objetos `cerr` y `clog` (objetos flujo de error estándar) permiten a un programa enviar mensajes de error a la pantalla o a otros dispositivos.

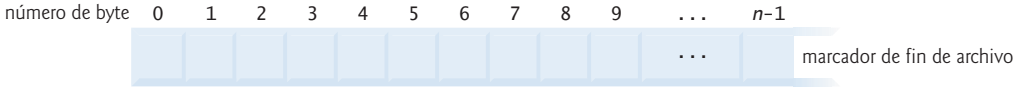


Fig. 14.1 | La manera en que C++ ve a un archivo de *n* bytes.

Plantillas de clases para procesar archivos

Para llevar a cabo el procesamiento de archivos en C++, se deben incluir los encabezados `<iostream>` y `<fstream>`. El encabezado `<fstream>` incluye las definiciones para las plantillas de clases de flujos

`basic_ifstream` (para las operaciones de entrada con archivos), `basic_ofstream` (para las operaciones de salida con archivos) y `basic_fstream` (para las operaciones de entrada *y* salida con archivos). Cada plantilla de clase tiene una especialización de plantilla predefinida que permite las operaciones de E/S con valores `char`. Además, la biblioteca `<fstream>` proporciona alias `typedef` para estas especializaciones de plantilla. Por ejemplo, la definición `typedef istream` representa a una especialización de `basic_ifstream` que permite la entrada de valores `char` desde un archivo. De manera similar, `typedef ofstream` representa una especialización de `basic_ofstream` que permite enviar valores `char` a archivos. Además, `typedf fstream` representa una especialización de `basic_fstream` que permite introducir valores `char` desde (*y* enviarlos hacia) archivos.

Estas plantillas se derivan de las plantillas de clases `basic_istream, basic_ostream` y `basic_iostream`, respectivamente. Por ende, todas las funciones miembro, operadores y manipuladores que pertenecen a estas plantillas (y que describimos en el capítulo 13) también se pueden aplicar a los flujos de archivos. En la figura 14.2 se sintetizan las relaciones de herencia de las clases de E/S que hemos visto hasta este punto.

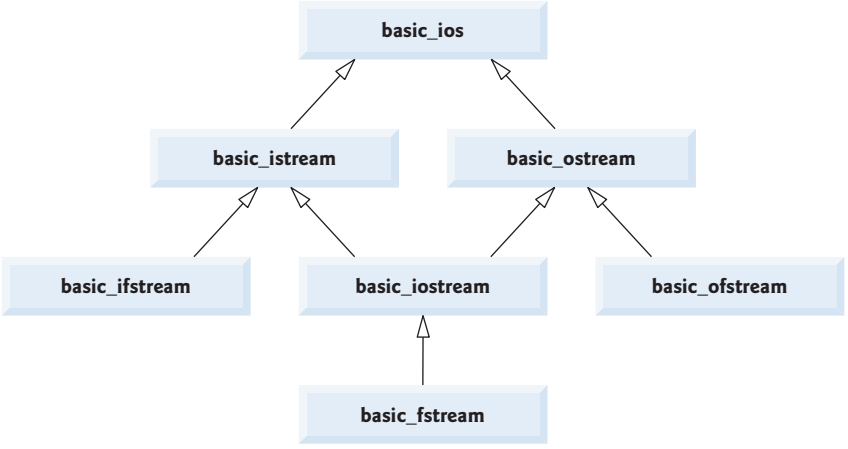


Fig. 14.2 | Porción de la jerarquía de plantillas de E/S de flujos.

14.3 Creación de un archivo secuencial

C++ no impone una estructura sobre un archivo. Por ende, en un archivo de C++ no existe un concepto tal como el de un "registro". El programador debe estructurar los archivos de manera que cumplan con los requerimientos de la aplicación. En el siguiente ejemplo veremos cómo se puede imponer una estructura de registro simple sobre un archivo.

La figura 14.3 crea un archivo secuencial que podría utilizarse en un sistema de cuentas por cobrar, para ayudar a administrar el dinero que deben los clientes de crédito a una empresa. Para cada cliente, el programa obtiene su número de cuenta, nombre y saldo (es decir, el monto que el cliente debe a la empresa por los bienes y servicios recibidos en el pasado). Los datos que se obtienen para cada cliente constituyen un *registro* para ese cliente. El número de cuenta sirve como la *clave de registro*; es decir, el programa crea y da mantenimiento a los registros del archivo en orden por número de cuenta. Este programa supone que el usuario introduce los registros en orden por número de cuenta. En un sistema de cuentas por cobrar completo, se debe incluir una herramienta para ordenar datos, para que el usuario pueda introducir los registros en *cualquier* orden; después los registros se *ordenan* y escriben en el archivo.

```
1  // Fig. 14.3: Fig14_03.cpp
2  // Creación de un archivo secuencial.
3  #include <iostream>
4  #include <string>
5  #include <fstream> // contiene tipos de procesamiento de flujos de archivos
6  #include <cstdlib> // prototipo de la función exit
7  using namespace std;
8
9  int main()
10 {
11    // el constructor de ofstream abre el archivo
12    ofstream archivoClientesSalida( "clientes.txt", ios::out );
13
14    // sale del programa si no puede crear el archivo
15    if ( !archivoClientesSalida ) // operador ! sobrecargado
16    {
17       cerr << "No se pudo abrir el archivo" << endl;
18       exit( EXIT_FAILURE );
19    } // fin de if
20
21    cout << "Escriba la cuenta, nombre y saldo." << endl
22       << "Escriba fin de archivo para terminar la entrada.\n? ";
23
24    int cuenta; // el número de cuenta
25    string nombre; // el nombre del propietario de la cuenta
26    double saldo; // el saldo de la cuenta
27
28    // lee la cuenta, nombre y saldo de cin, y después los coloca en el archivo
29    while ( cin >> cuenta >> nombre >> saldo   )
30    {
31       archivoClientesSalida << cuenta << ' ' << nombre << ' ' << saldo << endl;
32       cout << "? ";
33    } // fin de while
34 } // fin de main
```

```
Escriba la cuenta, nombre y saldo.
Escriba fin de archivo para terminar la entrada.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z
```

Fig. 14.3 | Creación de un archivo secuencial.

Abrir un archivo

En la figura 14.3 se escriben datos en un archivo, por lo que abrimos el archivo para salida mediante la creación de un objeto `ofstream`. Se pasan dos argumentos al constructor del objeto: el **nombre de archivo** y el **modo de apertura de archivo** (línea 12). Para un objeto `ofstream`, el modo de apertura de archivo puede ser **`ios::out`** para *enviar* datos a un archivo, o **`ios::app`** para *adjuntar* datos al final de un archivo (sin modificar los datos que ya estén en el archivo). Como `ios::out` es el predeterminado, no se requiere el segundo argumento del constructor en la línea 12. Los archivos existentes que se abren con el modo `ios::out` se **truncan**: se *descartan* todos los datos en el archivo. Si el archivo especificado *no* existe todavía, entonces el objeto `ofstream` *crea* el archivo, usando ese nombre de archivo. Antes de

C++11, el nombre de archivo se especificaba como una cadena basada en apuntador; a partir de C++11, también puede especificarse como un objeto `string`.

Tip para prevenir errores 14.1

Tenga cuidado al abrir un archivo existente en modo de salida (`ios::out`), en especial cuando desee preservar el contenido del archivo, que se descartará sin ninguna advertencia.

En la línea 12 se crea un objeto `ofstream` llamado `archivoClientesSalida`, asociado con el archivo `clientes.txt` que se abre en modo de salida. Los argumentos `"clientes.txt"` e `ios::out` se pasan al constructor de `ofstream`, el cual abre el archivo (esto establece una "línea de comunicación" con el archivo). De manera *predeterminada*, los objetos `ofstream` se abren en modo de *salida*, por lo que en la línea 12 se podría haber utilizado la instrucción alterna

```
ofstream archivoClientesSalida( "clientes.txt" );
```

para abrir `clientes.txt` en modo de salida. En la figura 14.4 se listan los modos de apertura de archivos. Estos modos pueden combinarse, como veremos en la sección 14.8.

Modo	Descripción
`ios::app`	*Adjunta* toda la salida al final del archivo.
`ios::ate`	Abre un archivo en modo de salida y se desplaza hasta el final del archivo (por lo general se utiliza para adjuntar datos a un archivo). Los datos se pueden escribir en *cualquier* parte del archivo.
`ios::in`	Abre un archivo en modo de *entrada*.
`ios::out`	Abre un archivo en modo de *salida*.
`ios::trunc`	Descarta el contenido del archivo (también es la acción predeterminada para `ios::out`).
`ios::binary`	Abre un archivo en modo de entrada o salida binaria (es decir, que no es texto).

Fig. 14.4 | Modos de apertura de archivos.

Abrir un archivo mediante la función miembro **open**

Se puede crear un objeto `ofstream` *sin necesidad* de abrir un archivo específico; en este caso, se puede adjuntar después un archivo al objeto. Por ejemplo, la instrucción

```
ofstream archivoClientesSalida;
```

crea un objeto `ofstream` que no está asociado todavía con un archivo. La función miembro **open** de `ofstream` abre un archivo y lo adjunta a un objeto `ofstream` *existente*, como se muestra a continuación:

```
outClientFile.open( "clients.txt", ios::out );
```

Tip para prevenir errores 14.2

Algunos sistemas operativos nos permiten abrir el mismo archivo varias veces al mismo tiempo. Evite hacer esto, ya que puede provocar problemas leves.

Probar si se abrió un archivo con éxito

Después de crear un objeto `ofstream` y *tratar* de abrirlo, el programa prueba si la operación de apertura tuvo éxito. La instrucción `if` en las líneas 15 a 19 utiliza la función miembro `operator!` *sobrecargada* de `ios` para determinar si la operación `open` tuvo éxito. La condición devuelve un valor `true` si se establece

el bit `failbit` o el bit `badbit` (vea el capítulo 13) para el flujo en la operación `open`. Ciertos posibles errores son: tratar de abrir un archivo *no existente* en modo de lectura, tratar de abrir un archivo en modo de lectura o escritura de un directorio para el que no se tiene permiso de acceso, y abrir un archivo en modo de escritura cuando no hay espacio disponible en disco.

Si la condición indica un intento fallido de abrir el archivo, en la línea 17 se imprime el mensaje de error `"No se pudo abrir el archivo"`, y en la línea 18 se invoca a la función `exit` para terminar el programa. El argumento para `exit` se devuelve al entorno desde el que se invocó el programa. Al pasar `EXIT_SUCCESS` (también definido en `<cstdlib>`) a `exit` se indica que el programa terminó en forma *normal*; cualquier otro valor (en este caso, `EXIT_FAILURE`) indica que el programa terminó debido a un *error*.

El operador `void *` sobrecargado

Otra función miembro *sobrecargada* de `ios` (`operator void*`) convierte el flujo en un apuntador, para que se pueda evaluar como `0` (es decir, el apuntador nulo) o un número distinto de cero (es decir, cualquier otro valor de apuntador). Cuando se usa el valor de un apuntador como una condición, C++ interpreta un apuntador nulo en una condición como el valor `bool false` e interpreta un apuntador no nulo como el valor `bool true`. Si se establecieron los bits `failbit` o `badbit` para el flujo, se devuelve `0` (`false`). La condición en la instrucción `while` de las líneas 29 a 33 invoca a la función miembro `operator void *` en `cin` de manera *implícita*. La condición permanece como `true`, siempre y cuando no se haya establecido el bit `failbit` o el bit `badbit` para `cin`. Al introducir el indicador de *fin de archivo* se establece el bit `failbit` para `cin`. La función `operator void *` se puede utilizar para probar un objeto de entrada para el fin de archivo, pero también es posible llamar a la función miembro `eof` de manera explícita en el objeto de entrada.

Procesamiento de datos

Si en la línea 12 se abrió el archivo con éxito, el programa empieza a procesar los datos. En las líneas 21 y 22 se pide al usuario que introduzca varios campos para cada registro, o el indicador de fin de archivo cuando esté completa la entrada de datos. En la figura 14.5 se listan las combinaciones de teclado para introducir el fin de archivo en varios sistemas computacionales.

Sistema computacional	Combinación de teclado
UNIX/Linux/Mac OS X	<*Ctrl-d*> (en una línea por sí solo)
Microsoft Windows	<*Ctrl-z*> (algunas veces va seguido de *Intro*)

Fig. 14.5 | Combinaciones de teclas de fin de archivo.

En la línea 29 se extrae cada conjunto de datos y se determina si se introdujo el fin de archivo. Al encontrar el fin de archivo, o cuando se introducen datos incorrectos, `operator void *` devuelve el apuntador nulo (que se convierte en el valor `bool false`) y la instrucción `while` termina. El usuario introduce el fin de archivo para informar al programa que ya no procese más datos. El indicador de fin de archivo se establece cuando el usuario introduce la combinación de teclas de fin de archivo. La instrucción `while` itera hasta que se establece el indicador de fin de archivo (o se introducen datos incorrectos).

En la línea 31 se escribe un conjunto de datos en el archivo `clientes.txt`, usando el operador de inserción de flujo `<<` y el objeto `archivoClientesSalida` asociado con el archivo al principio del programa. Los datos se pueden recuperar mediante un programa diseñado para leer el archivo (vea la sección 14.4). El archivo creado en la figura 14.3 es simplemente un *archivo de texto*, por lo que puede verse en cualquier editor de texto.

Cerrar un archivo
Una vez que el usuario introduce el indicador de fin de archivo, `main` termina. Esto invoca de manera implícita a la función destructor del objeto `archivoClientesSalida`, que *cierra* el archivo `clientes.txt`. También podemos cerrar el objeto `ofstream` de manera *explícita*, usando la función miembro `close` como se indica a continuación:

```
archivoClientesSalida.close();
```

Tip para prevenir errores 14.3
Cierre siempre un archivo tan pronto como ya no lo necesite en un programa.

La ejecución de ejemplo
En la ejecución de ejemplo para el programa de la figura 14.3, el usuario introduce información para cinco cuentas y después indica que la entrada de datos está completa, al introducir el fin de archivo (se muestra `^Z` para Microsoft Windows). Esta ventana de diálogo *no* muestra cómo aparecen los registros de datos en el archivo. En la siguiente sección se muestra cómo crear un programa que lea este archivo e imprima su contenido para verificar que el programa haya creado el archivo con éxito.

14.4 Cómo leer datos de un archivo secuencial

Los archivos almacenan datos de manera que éstos se puedan *recuperar* para procesarlos cuando sea necesario. En la sección anterior demostramos cómo crear un archivo para acceso secuencial. Ahora veremos cómo *leer* datos secuencialmente desde un archivo. En la figura 14.6 se leen y muestran los registros del archivo de datos `clientes.txt` que creamos usando el programa de la figura 14.3. Al crear un objeto `ifstream`, se abre un archivo en modo de *entrada*. El constructor de `ifstream` puede recibir el nombre del archivo y el modo de apertura del mismo como argumentos. En la línea 15 se crea un objeto `ifstream` llamado `archivoClientesEntrada`, y se asocia con el archivo `clientes.txt`. Los argumentos entre paréntesis se pasan a la función constructor de `ifstream`, la cual abre el archivo y establece una "línea de comunicación" con el mismo.

Buena práctica de programación 14.1
Si el contenido de un archivo no se debe modificar, use `ios::in` para abrirlo solamente *en modo de entrada. Esto evita la modificación accidental del contenido del archivo, y es otro ejemplo del principio del menor privilegio.*

```
1  // Fig. 14.6: Fig14_06.cpp
2  // Cómo leer e imprimir un archivo secuencial.
3  #include <iostream>
4  #include <fstream> // flujo de archivo
5  #include <iomanip>
6  #include <string>
7  #include <cstdlib>
8  using namespace std;
9
10 void outputLine( int, const string &, double ); // prototipo
```

Fig. 14.6 | Cómo leer e imprimir un archivo secuencial (parte 1 de 2).

```
11
12 int main()
13 {
14    // el constructor de ifstream abre el archivo
15    ifstream archivoClientesEntrada( "clientes.txt", ios::in );
16
17    // sale del programa si ifstream no pudo abrir el archivo
18    if ( !archivoClientesEntrada )
19    {
20       cerr << "No se pudo abrir el archivo" << endl;
21       exit( EXIT_FAILURE );
22    } // fin de if
23
24    int cuenta; // el número de cuenta
25    string nombre; // el nombre del propietario de la cuenta
26    double saldo; // el saldo de la cuenta
27
28    cout << left << setw( 10 ) << "Cuenta" << setw( 13 )
29       << "Nombre" << "Saldo" << endl << fixed << showpoint;
30
31    // muestra cada registro en el archivo
32    while ( archivoClientesEntrada >> cuenta >> nombre >> saldo )
33       imprimirLinea( cuenta, nombre, saldo );
34 } // fin de main
35
36 // muestra un solo registro del archivo
37 void imprimirLinea( int cuenta, const string &nombre, double saldo )
38 {
39    cout << left << setw( 10 ) << cuenta << setw( 13 ) << nombre
40       << setw( 7 ) << setprecision( 2 ) << right << saldo << endl;
41 } // fin de la función imprimirLinea
```

```
Cuenta   Nombre        Saldo
100      Jones         24.98
200      Doe          345.67
300      White          0.00
400      Stone        -42.16
500      Rich         224.62
```

Fig. 14.6 | Cómo leer e imprimir un archivo secuencial (parte 2 de 2).

Abrir un archivo en modo de entrada

Los objetos de la clase `ifstream` se abren en modo de *entrada* de manera predeterminada, por lo que la instrucción

```
ifstream archivoClientesEntrada( "clientes.txt" );
```

abre `clientes.txt` en modo de entrada. Al igual que un objeto `ofstream`, es posible crear un objeto `ifstream` sin abrir un archivo específico, ya que se le puede adjuntar un archivo más adelante.

Asegurar que se abrió el archivo

El programa utiliza la condición `!archivoClientEntrada` para determinar si el archivo se abrió con éxito antes de tratar de obtener datos del mismo.

Leer datos del archivo

En la línea 32 se lee un conjunto de datos (es decir, un registro) del archivo. Después de que se ejecuta la línea 32 la primera vez, `cuenta` tiene el valor `100`, `nombre` tiene el valor `"Jones"` y `saldo` tiene el valor `24.98`. Cada vez que se ejecuta la línea 32, lee otro registro del archivo y lo coloca en las variables `cuenta`, `nombre` y `saldo`. En la línea 33 se muestran los registros, usando la función `imprimirLinea` (líneas 37 a 41), la cual utiliza manipuladores de flujos parametrizados para dar formato a los datos que se van a mostrar. Al llegar al fin del archivo, la *llamada implícita a* `operator void *` *en la condición* `while` devuelve el apuntador nulo (que se convierte en el valor `bool false`), el destructor de `ifstream` cierra el archivo y el programa termina.

Apuntadores de posición del archivo

Para obtener datos secuencialmente de un archivo, por lo general los programas empiezan a leer desde el principio del archivo y leen todos los datos en forma consecutiva, hasta encontrar los datos deseados. Tal vez sea necesario procesar el archivo secuencialmente varias veces (desde el principio del mismo) durante la ejecución de un programa. Tanto `istream` como `ostream` proporcionan funciones miembro para *reposicionar el* ***apuntador de posición del archivo*** (el número de byte del siguiente byte en el archivo que se va a leer o escribir). Estas funciones miembro son **`seekg`** ("seek get", "buscar obtener") para `istream` y **`seekp`** ("seek put", "buscar colocar") para `ostream`. Cada objeto `istream` tiene un "*apuntador obtener*", el cual indica el número de byte en el archivo a partir del cual va a ocurrir la siguiente *entrada*, y cada objeto `ostream` tiene un "*apuntador colocar*", el cual indica el número de byte en el archivo en el que se debe colocar la siguiente *salida*. La instrucción

```
archivoClientesEntrada.seekg( 0 );
```

reposiciona el apuntador de posición del archivo y lo coloca al *principio* del mismo (ubicación `0`) adjunto a `archivoClientesEntrada`. El argumento para `seekg` es comúnmente un entero `long`. Se puede especificar un segundo argumento para indicar la **dirección de búsqueda**, que puede ser **`ios::beg`** (la opción predeterminada) para un posicionamiento relativo al *inicio* de un flujo, **`ios::cur`** para un posicionamiento relativo a la *posición actual* en un flujo, o **`ios::end`** para un posicionamiento relativo al *final* de un flujo. El apuntador de posición del archivo es un valor entero que especifica la ubicación en el archivo como un número de bytes desde la ubicación inicial del archivo (a ésta también se le conoce como el **desplazamiento** desde el inicio del archivo). Algunos ejemplos de cómo posicionar el apuntador "*obtener*" de posición del archivo son:

```
// se posiciona en el n-ésimo byte de objetoArchivo (asumiendo ios::beg)
objetoArchivo.seekg( n );

// se posiciona n bytes hacia adelante en objetoArchivo
objetoArchivo.seekg( n, ios::cur );

// se posiciona n bytes hacia atrás desde el final de objetoArchivo
objetoArchivo.seekg( n, ios::end );

// se posiciona al final de objetoArchivo
objetoArchivo.seekg( 0, ios::end );
```

Se pueden realizar las mismas operaciones mediante la función miembro `seekp` de `ostream`. Las funciones miembro **`tellg`** y **`tellp`** se proporcionan para devolver las posiciones actuales de los apuntadores "*obtener*" y "*colocar*", respectivamente. La siguiente instrucción asigna el valor del apuntador "*obtener*" de posición del archivo a la variable `ubicacion` de tipo `long`:

```
ubicacion = objetoArchivo.tellg();
```

Programa de consulta de crédito

En la figura 14.7 se permite a un gerente de créditos mostrar la información de la cuenta para los clientes con saldos en cero (es decir, los clientes que no deben dinero a la empresa), saldos de crédito (negativos) (es decir, son clientes a quienes les debe dinero la empresa) y saldos de débito (positivos) (es decir, los clientes que deben dinero a la empresa por los bienes y servicios recibidos en el pasado). El programa muestra un menú y permite al gerente de créditos introducir una de tres opciones para obtener la información de crédito. La opción 1 produce una lista de cuentas con saldos en cero. La opción 2 produce una lista de cuentas con saldos de crédito. La opción 3 produce una lista de cuentas con saldos de débito. La opción 4 termina la ejecución del programa. Si se introduce una opción inválida, se muestra el indicador para que el usuario introduzca otra opción. En las líneas 64 y 65 se permite al programa leer desde el principio del archivo, una vez que se lee el fin de archivo.

```
1  // Fig. 14.7: Fig14_07.cpp
2  // Programa de solicitud de crédito.
3  #include <iostream>
4  #include <fstream>
5  #include <iomanip>
6  #include <string>
7  #include <cstdlib>
8  using namespace std;
9
10 enum TipoSolicitud { SALDO_CERO = 1, SALDO_CREDITO, SALDO_DEBITO, TERMINAR };
11 int obtenerSolicitud();
12 bool debeMostrar( int, double );
13 void imprimirLinea( int, const string &, double );
14
15 int main()
16 {
17    // el constructor de ifstream abre el archivo
18    ifstream archivoClientesSalida( "clientes.txt", ios::in );
19
20    // sale del programa si ifstream no pudo abrir el archivo
21    if ( !archivoClientesSalida )
22    {
23       cerr << "No se pudo abrir el archivo" << endl;
24       exit( EXIT_FAILURE );
25    } // fin de if
26
27    int cuenta; // el número de cuenta
28    string nombre; // el nombre del propietario de la cuenta
29    double saldo; // el saldo de la cuenta
30
31    // obtiene la solicitud del usuario (por ejemplo, saldo en cero, de crédito
         o débito)
32    int solicitud = obtenerSolicitud();
33
34    // procesa la solcitud del usuario
35    while ( solicitud != TERMINAR )
36    {
37       switch ( solicitud )
38       {
```

Fig. 14.7 | Programa de solicitud de crédito (parte 1 de 4).

```
39          case SALDO_CERO:
40             cout << "\nCuentas con saldos en cero:\n";
41             break;
42          case SALDO_CREDITO:
43             cout << "\nCuentas con saldos de credito:\n";
44             break;
45          case SALDO_DEBITO:
46             cout << "\nCuentas con saldos de debito:\n";
47             break;
48       } // fin de switch
49
50       // lee la cuenta, el nombre y el saldo del archivo
51       archivoClientesSalida >> cuenta >> nombre >> saldo;
52
53       // muestra el contenido del archivo (hasta eof)
54       while ( !archivoClientesSalida.eof() )
55       {
56          // muestra el registro
57          if ( debeMostrar( solicitud, saldo ) )
58             imprimirLinea( cuenta, nombre, saldo );
59
60          // lee la cuenta, el nombre y el saldo del archivo
61          archivoClientesSalida >> cuenta >> nombre >> saldo;
62       } // fin de while interior
63
64       archivoClientesSalida.clear(); // restablece eof para la siguiente entrada
65       archivoClientesSalida.seekg( 0 ); // se reposiciona al inicio del archivo
66       solicitud = obtenerSolicitud();
         // obtiene una solicitud adicional del usuario
67    } // fin de while exterior
68
69    cout << "Fin de ejecucion." << endl;
70 } // fin de main
71
72 // obtiene la solicitud del usuario
73 int obtenerSolicitud()
74 {
75    int solicitud; // solicitud del usuario
76
77    // muestra las opciones de solicitud
78    cout << "\nEscriba la opcion"  << endl
79       << " 1 - Listar cuentas con saldos en cero" << endl
80       << " 2 - Listar cuentas con saldos de credito" << endl
81       << " 3 - Listar cuentas con saldos de debito" << endl
82       << " 4 - Finalizar ejecucion" << fixed << showpoint;
83
84    do // introduce la solicitud del usuario
85    {
86       cout << "\n? ";
87       cin >> solicitud;
88    } while ( solicitud < SALDO_CERO && solicitud > TERMINAR );
89
90    return solicitud;
91 } // fin de la función obtenerSolicitud
```

Fig. 14.7 | Programa de solicitud de crédito (parte 2 de 4).

```
92
93 // determina si se va a mostrar el registro dado
94 bool debeMostrar( int tipo, double saldo )
95 {
96    // determina si se van a mostrar los saldos en cero
97    if ( tipo == SALDO_CERO && saldo == 0 )
98       return true;
99
100   // determina si se van a mostrar los saldos de crédito
101   if ( tipo == SALDO_CREDITO && saldo < 0 )
102      return true;
103
104   // determina si se van a mostrar los saldos de débito
105   if ( tipo == SALDO_DEBITO && saldo > 0 )
106      return true;
107
108   return false;
109 } // fin de la función debeMostrar
110
111 // muestra un solo registro del archivo
112 void imprimirLinea( int cuenta, const string &nombre, double saldo )
113 {
114    cout << left << setw( 10 ) << cuenta << setw( 13 ) << nombre
115       << setw( 7 ) << setprecision( 2 ) << right << saldo << endl;
116 } // fin de la función imprimirLinea
```

```
Escriba la opcion
 1 - Listar cuentas con saldos en cero
 2 - Listar cuentas con saldos de credito
 3 - Listar cuentas con saldos de debito
 4 - Finalizar ejecucion
? 1

Cuentas con saldos en cero:
300       White          0.00

Escriba la opcion
 1 - Listar cuentas con saldos en cero
 2 - Listar cuentas con saldos de credito
 3 - Listar cuentas con saldos de debito
 4 - Finalizar ejecucion
? 2

Cuentas con saldos de credito:
400       Stone        -42.16

Escriba la opcion
 1 - Listar cuentas con saldos en cero
 2 - Listar cuentas con saldos de credito
 3 - Listar cuentas con saldos de debito
 4 - Finalizar ejecucion
? 3

Cuentas con saldos de debito:
100       Jones         24.98
200       Doe          345.67
500       Rich         224.62
```

Fig. 14.7 | Programa de solicitud de crédito (parte 3 de 4).

```
Escriba la opcion
 1 - Listar cuentas con saldos en cero
 2 - Listar cuentas con saldos de credito
 3 - Listar cuentas con saldos de debito
 4 - Finalizar ejecucion
? 4
Fin de ejecucion.
```

Fig. 14.7 | Programa de solicitud de crédito (parte 4 de 4).

14.5 Actualización de archivos secuenciales

Los datos a los que se dan formato y se escriben en un archivo secuencial, como se muestra en la figura 14.3, no se pueden modificar sin el riesgo de destruir otros datos en el archivo. Por ejemplo, si hay que cambiar el nombre "`White`" por "`Worthington`", el nombre anterior no se puede sobrescribir sin que se corrompa el archivo. El registro para `White` se escribió en el archivo de la siguiente manera:

```
300 White 0.00
```

Si se vuelve a escribir este registro, empezando en la misma ubicación en el archivo y usando el nuevo nombre más largo, el registro sería:

```
300 Worthington 0.00
```

El nuevo registro contiene seis caracteres más que el original. Por lo tanto, los caracteres más allá de la segunda "`o`" en "`Worthington`" sobrescribirían el inicio del siguiente registro secuencial en el archivo. El problema es que, en el modelo de entrada/salida con formato que utiliza el operador de inserción de flujo << y el operador de extracción de flujo >>, los campos (y por ende, los registros) pueden variar en cuanto a su tamaño. Por ejemplo, los valores `7`, `14`, `-117`, `2074` y `27383` son todos `int`, los cuales almacenan el mismo número de bytes de "datos puros" de manera interna (por lo general, cuatro bytes en los equipos de 32 bits y ocho bytes en los equipos de 64 bits). Sin embargo, estos enteros se convierten en campos de distintos tamaños, dependiendo de sus valores actuales, cuando se imprimen como texto con formato (secuencias de caracteres). Por lo tanto, el modelo de entrada/salida con formato no se utiliza comúnmente para actualizar registros *en su posición en el archivo*. En las secciones 14.6 a 14.10 le mostraremos cómo realizar actualizaciones en la posición del archivo con los registros de longitud fija.

Dicha actualización puede realizarse de una manera complicada. Por ejemplo, para modificar el nombre anterior, los registros antes de `300 White 0.00` en un archivo secuencial podrían *copiarse* en un nuevo archivo, después se escribiría el registro actualizado en el nuevo archivo, y luego se copiarían los registros después de `300 White 0.00` en el nuevo archivo. Luego podríamos eliminar el archivo anterior y cambiar el nombre del nuevo archivo. Para ello se requiere procesar *todos* los registros en el archivo, para actualizar tan sólo un registro. Si se van a actualizar muchos registros del archivo en una sola pasada, esta técnica puede ser aceptable.

14.6 Archivos de acceso aleatorio

Hasta ahora hemos visto cómo crear archivos secuenciales y buscar en ellos para localizar información. Los archivos secuenciales son inapropiados para las **aplicaciones de acceso instantáneo**, en las que un registro específico se debe localizar inmediatamente. Las aplicaciones comunes de acceso instantáneo son los sistemas de reservación de aerolíneas, sistemas bancarios, sistemas de punto de venta, cajeros automáticos y otros tipos de **sistemas de procesamiento de transacciones** que requieran acceso rápido a datos específicos. Un banco podría tener cientos de miles (o incluso millones) de otros clientes, y a pesar de ello, cuando un cliente utiliza un cajero automático, el programa comprueba la cuenta de ese

cliente en unos cuantos segundos o menos, para ver si tiene suficientes fondos. Este tipo de acceso instantáneo es posible mediante los **archivos de acceso aleatorio**. Los registros individuales de un archivo de acceso aleatorio se pueden utilizar de manera directa (y rápida), sin tener que buscar en otros registros.

Como hemos dicho, C++ no impone una estructura sobre un archivo. Por lo tanto, la aplicación que desee utilizar archivos de acceso aleatorio debe crearlos. Se puede utilizar una variedad de técnicas. Tal vez el método más sencillo sea requerir que todos los registros en un archivo sean de la *misma longitud fija*. Al utilizar registros de longitud fija y con el mismo tamaño, es más fácil para el programa calcular (como una función del tamaño de registro y la clave de registro) la ubicación exacta de cualquier registro relativo al inicio del archivo. Pronto veremos cómo esto facilita el *acceso inmediato* a registros específicos, incluso en archivos extensos.

En la figura 14.8 se ilustra la forma en que C++ ve a un archivo de acceso aleatorio, compuesto de registros de longitud fija (en este caso, cada registro tiene 100 bytes de longitud). Un archivo de acceso aleatorio es como un ferrocarril con muchos carros del mismo tamaño; algunos están vacíos y otros llenos.

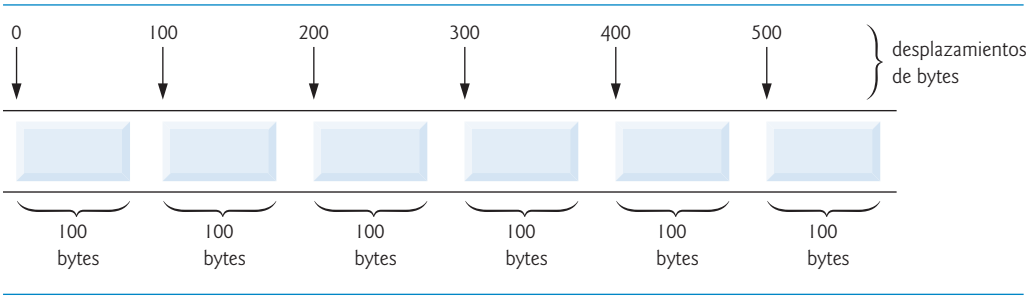


Fig. 14.8 | Forma en que C++ ve a un archivo de acceso aleatorio.

Se pueden insertar datos en un archivo de acceso aleatorio sin destruir otros datos en el archivo. Los datos almacenados con anterioridad también se pueden actualizar o eliminar, sin necesidad de volver a escribir el archivo completo. En las siguientes secciones explicaremos cómo crear un archivo de acceso aleatorio, introducir datos en el archivo, leerlos tanto en forma secuencial como aleatoria, actualizarlos y eliminar los que ya no sean necesarios.

14.7 Creación de un archivo de acceso aleatorio

La función miembro `write` de `ostream` envía un número fijo de bytes al flujo especificado, empezando en una ubicación específica en memoria. Cuando el flujo se asocia con un archivo, la función `write` escribe los datos *en la ubicación en el archivo especificado mediante el apuntador "colocar" de posición del archivo*. La función miembro **`read`** de `istream` recibe un número fijo de bytes del flujo especificado y los coloca en un área en memoria, que empieza en una dirección especificada. Si el flujo se asocia con un archivo, la función `read` introduce bytes en la ubicación en el archivo especificado por el apuntador "obtener" de posición del archivo.

Escritura de bytes mediante la función miembro `write` *de* `ostream`

Al escribir el entero `numero` en un archivo, en vez de usar la instrucción

```
archivoSalida << numero;
```

que para un entero de cuatro bytes podría imprimir desde un dígito hasta 11 (10 dígitos más un signo, cada uno de los cuales requiere un solo byte de almacenamiento), podemos usar la instrucción

```
archivoSalida.write( reinterpret_cast< const char * >( &numero ),
  sizeof( numero ) );
```

que siempre escribe la versión *binaria* de los *cuatro* bytes del número entero (en un equipo con enteros de cuatro bytes). La función `write` trata a su primer argumento como un grupo de bytes, al ver el objeto en memoria como un `const char *`, el cual es un apuntador a un byte. Empezando desde esa ubicación, la función `write` envía como salida el número de bytes especificados por su segundo argumento; un entero de tipo `size_t`. Como veremos, la función `read` de `istream` se puede utilizar después para leer los cuatro bytes y colocarlos de vuelta en la variable entera `numero`.

Conversión entre los tipos de apuntadores con el operador `reinterpret_cast`

Por desgracia, la mayoría de los apuntadores que pasamos a la función `write` como el primer argumento *no* son de tipo `const char *`. Para enviar como salida objetos de otros tipos, debemos convertir los apuntadores a esos objetos al tipo `const char *`; en caso contrario, el compilador no compilará las llamadas a la función `write`. C++ proporciona el operador `reinterpret_cast` para casos como éste en el que un apuntador de un tipo debe convertirse en un tipo de apuntador *no relacionado*. Sin un operador `reinterpret_cast`, la instrucción `write` que envía como salida el entero `numero` no se compilará, ya que el compilador *no* permite pasar un apuntador de tipo `int *` (el tipo devuelto por la expresión `&numero`) a una función que espera un argumento de tipo `const char *`; en lo que respecta al compilador, estos tipos son *incompatibles*.

Una operación `reinterpret_cast` se lleva a cabo en *tiempo de compilación*, y *no* cambia el valor del objeto al cual apunta su operando. En vez de ello, solicita que el compilador reinterprete el operando como el tipo de destino (especificado en los signos < y > que siguen después de la palabra clave `reinterpret_cast`). En la figura 14.11 utilizamos a `reinterpret_cast` para convertir un apuntador `DatosCliente` en un `const char *`, que reinterpreta un objeto `DatosCliente` como bytes que se van a enviar a un archivo. Los programas de procesamiento de archivos de acceso aleatorio raras veces escriben un solo campo en un archivo. Por lo general, escriben un objeto de una clase a la vez, como demostramos en los siguientes ejemplos.

Tip para prevenir errores 14.4

Es fácil utilizar `reinterpret_cast` *para realizar manipulaciones peligrosas que podrían conducir a errores graves en tiempo de ejecución.*

Tip de portabilidad 14.1

El uso de `reinterpret_cast` *es dependiente del compilador, y puede hacer que los programas se comporten de manera diferente en distintas plataformas. Use este operador sólo si es absolutamente necesario.*

Tip de portabilidad 14.2

Un programa que lee datos sin formato (escritos por `write`*) debe compilarse y ejecutarse en un sistema compatible con el programa que escribió los datos, ya que distintos sistemas pueden representar los datos internos de manera diferente.*

Programa de procesamiento de crédito

Considere el siguiente enunciado del problema:

> *Cree un programa de procesamiento de crédito que sea capaz de almacenar a lo más 100 registros de longitud fija para una empresa que puede tener hasta 100 clientes. Cada registro debe consistir de un número de cuenta que actúe como la clave de registro, un apellido paterno, un primer nombre y un saldo. El programa debe ser capaz de actualizar una cuenta, insertar una nueva, eliminar una cuenta e insertar todos los registros de las cuentas en un archivo de texto con formato para imprimirlo.*

En las siguientes secciones presentaremos las técnicas para crear este programa de procesamiento de crédito. La figura 14.11 ilustra cómo abrir un archivo de acceso aleatorio, definir el formato de los registros usando un objeto de la clase `DatosCliente` (figuras 14.9 y 14.10) y escribir datos en el disco, en formato *binario*. Este programa inicializa los 100 registros del archivo `credito.dat` con objetos *vacíos*, usando la función `write`. Cada objeto vacío contiene `0` para el número de cuenta, cadenas vacías para el apellido paterno y el primer nombre, y `0.0` para el saldo. Cada registro se inicializa con la cantidad de espacio vacío en donde se almacenarán los datos de la cuenta.

```
1  // Fig. 14.9: DatosCliente.h
2  // Definición de la clase DatosCliente, utilizada en las figuras 14.11 a 14.14.
3  #ifndef DATOSCLIENTE_H
4  #define DATOSCLIENTE_H
5
6  #include <string>
7
8  class DatosCliente
9  {
10 public:
11    // constructor predeterminado de DatosCliente
12    DatosCliente( int = 0, const std::string & = "",
13       const std::string & = "", double = 0.0 );
14
15    // funciones de acceso para numeroCuenta
16    void establecerNumeroCuenta( int );
17    int gobtenerNumeroCuenta() const;
18
19    // funciones de acceso para apellidoPaterno
20    void establecerApellidoPaterno( const std::string & );
21    std::string obtenerApellidoPaterno() const;
22
23    // funciones de acceso para primerNombre
24    void establecerPrimerNombre( const std::string & );
25    std::obtenerPrimerNombre() const;
26
27    // funciones de acceso para el saldo
28    void establecerSaldo( double );
29    double obtenerSaldo() const;
30 private:
31    int numeroCuenta;
32    char apellidoPaterno[ 15 ];
33    char primerNombre[ 10 ];
34    double saldo;
35 }; // fin de la clase DatosCliente
36
37 #endif
```

Fig. 14.9 | Archivo de encabezado de la clase `DatosCliente`.

```
1  // Fig. 14.10: DatosCliente.cpp
2  // La clase DatosCliente almacena la información de crédito del cliente.
3  #include <string>
```

Fig. 14.10 | La clase `DatosCliente` representa la información de crédito de un cliente (parte 1 de 3).

```
4  #include "DatosCliente.h"
5  using namespace std;
6
7  // constructor predeterminado de DatosCliente
8  DatosCliente::DatosCliente( int valorNumeroCuenta, const string
   &apellidoPaterno,
9     const string &primerNombre, double valorSaldo )
10   : numeroCuenta( valorNumeroCuenta ), saldo( valorSaldo )
11 {
12    establecerApellidoPaterno( valorApellidoPaterno );
13    establecerPrimerNombre( valorPrimerNombre );
14 } // fin del constructor de DatosCliente
15
16 // obtiene el valor del número de cuenta
17 int DatosCliente::obtenerNumeroCuenta() const
18 {
19    return numeroCuenta;
20 } // fin de la función obtenerNumeroCuenta
21
22 // establece el valor del número de cuenta
23 void DatosCliente::establecerNumeroCuenta( int valorNumeroCuenta )
24 {
25    numeroCuenta = valorNumeroCuenta; // debe validar
26 } // fin de la función establecerNumeroCuenta
27
28 // obtiene el valor del apellido paterno
29 string DatosCliente::obtenerApellidoPaterno() const
30 {
31    return apellidoPaterno;
32 } // fin de la función obtenerApellidoPaterno
33
34 // establece el valor del apellido paterno
35 void DatosCliente::establecerApellidoPaterno( const string
   &cadenaApellidoPaterno )
36 {
37    // copia a lo más 15 caracteres de la cadena a apellidoPaterno
38    int longitud = cadenaApellidoPaterno.size();
39    longitud = ( longitud < 15 ? llongitud : 14 );
40    cadenaApellidoPaterno.copy( apellidoPaterno, longitud );
41    apellidoPaterno[ longitud ] = '\0';
      // adjunta un carácter nulo a apellidoPaterno
42 } // fin de la función establecerApellidoPaterno
43
44 // obtiene el valor del primer nombre
45 string DatosCliente::obtenerPrimerNombre() const
46 {
47    return primerNombre;
48 } // fin de la función obtenerPrimerNombre
49
50 // establece el valor del primer nombre
51 void DatosCliente::establecerPrimerNombre( const string &cadenaPrimerNombre )
52 {
53    // copia a lo más 10 caracteres de la cadena a primerNombre
54    int longitud = cadenaPrimerNombre.size();
```

Fig. 14.10 | La clase `DatosCliente` representa la información de crédito de un cliente (parte 2 de 3).

```
55    longitud = ( longitud < 10 ? longitud : 9 );
56    cadenaPrimerNombre.copy( primerNombre, longitud );
57    primerNombre[ longitud ] = '\0'; // adjunta un carácter nulo a primerNombre
58 } // fin de la función establecerPrimerNombre
59
60 // obtiene el valor del saldo
61 double DatosCliente::obtenerSaldo() const
62 {
63    return saldo;
64 } // fin de la función obtenerSaldo
65
66 // establece el valor del saldo
67 void DatosCliente::establecerSaldo( double valorSaldo )
68 {
69    saldo = valorSaldo;
70 } // fin de la función establecerSaldo
```

Fig. 14.10 | La clase `DatosCliente` representa la información de crédito de un cliente (parte 3 de 3).

Los objetos de la clase `string` *no tienen tamaño uniforme*, sino que utilizan la memoria asignada en forma dinámica para dar cabida a las cadenas de varias longitudes. Este programa debe mantener registros de longitud fija, por lo que la clase `DatosCliente` almacena el primer nombre y el apellido del cliente en arreglos `char` de longitud fija (declarados en la figura 14.9, líneas 32 y 33). Las funciones miembro `establecerApellidoPaterno` (figura 14.10, líneas 35 a 42) y `establecerPrimerNombre` (figura 14.10, líneas 51 a 58) copian los caracteres de un objeto `string` en el arreglo `char` correspondiente. Considere la función `establecerApellidoPaterno`. En la línea 38 se invoca a la función miembro `string` llamada **`size`** para obtener la longitud de `cadenaApellidoPaterno`. En la línea 39 se asegura que `longitud` sea menor de 15 caracteres, y después en la línea 40 se copian `longitud` caracteres de `valorApellidoPaterno` al arreglo `char` llamado `apellidoPaterno` mediante el uso de la función miembro `string` llamada `copy`. La función miembro `establecerPrimerNombre` realiza los mismos pasos para el primer nombre.

Abrir un archivo para salida en modo binario

En la figura 14.11, la línea 11 crea un objeto `ofstream` para el archivo `credito.dat`. El segundo argumento para el constructor (`ios::out | ios::binary`) indica que vamos a abrir el archivo para salida en *modo binario*, lo cual es *requerido* si debemos escribir *registros de longitud fija*. Los modos múltiples de apertura de archivos se combinan al separar cada modo de apertura del siguiente mediante el operador `|`, que se conoce como *operador OR inclusivo a nivel de bits* (en el capítulo 22 veremos este operador con detalle). En las líneas 24 y 25 se escribe el objeto `clienteEnBlanco` (que se construyó con argumentos predeterminados en la línea 20) en el archivo `credito.dat` asociado con el objeto `ofstream` llamado `creditoSalida`. Recuerde que el operador `sizeof` devuelve el tamaño en bytes del objeto contenido entre paréntesis (vea el capítulo 8). El primer argumento para la función `write` en la línea 24 debe ser de tipo `const char *`. Sin embargo, el tipo de datos de `&clienteEnBlanco` es `DatosCliente *`. Para convertir `&clienteEnBlanco` en `const char *`, en la línea 24 se utiliza el operador de conversión `reinterpret_cast`, por lo que la llamada a `write` se compila sin generar un error de compilación.

```
1 // Fig. 14.11: Fig14_11.cpp
2 // Creación de un archivo de acceso aleatorio.
3 #include <iostream>
```

Fig. 14.11 | Creación de un archivo de acceso aleatorio con 100 registros en blanco, en forma secuencial (parte 1 de 2).

```
#include <fstream>
#include <cstdlib>
#include "DatosCliente.h" // definición de la clase DatosCliente
using namespace std;

int main()
{
   ofstream creditoSalida( "credito.dat", ios::out | ios::binary );

   // sale del programa si ofstream no pudo abrir el archivo
   if ( !creditoSalida )
   {
      cerr << "No se pudo abrir el archivo." << endl;
      exit( EXIT_FAILURE );
   } // fin de if

   DatosCliente clienteEnBlanco;
   // el constructor pone en ceros cada miembro de datos

   // escribe 100 registros en blanco en el archivo
   for ( int i = 0; i < 100; ++i )
      creditoSalida.write( reinterpret_cast< const char * >( &clienteEnBlanco),
         sizeof( DatosCliente ) );
} // fin de main
```

Fig. 14.11 | Creación de un archivo de acceso aleatorio con 100 registros en blanco, en forma secuencial (parte 2 de 2).

14.8 Cómo escribir datos al azar a un archivo de acceso aleatorio

En la figura 14.12 se escriben datos en el archivo `credito.dat` y se utiliza la combinación de las funciones `seekp` y `write` de `fstream` para almacenar datos en ubicaciones *exactas* en el archivo. La función `seekp` establece el apuntador "*colocar*" de posición del archivo en una posición específica en el archivo, y después `write` escribe los datos. En la línea 6 se incluye el encabezado `DatosCliente.h` definido en la figura 14.9, para que el programa pueda utilizar objetos `DatosCliente`.

```
// Fig. 14.12: Fig14_12.cpp
// Escritura en un archivo de acceso aleatorio.
#include <iostream>
#include <fstream>
#include <cstdlib>
#include "DatosCliente.h" // definición de la clase DatosCliente
using namespace std;

int main()
{
   int numeroCuenta;
   string apellidoPaterno;
   string primerNombre;
   double saldo;

   fstream creditoSalida( "credito.dat", ios::in | ios::out | ios::binary );
```

Fig. 14.12 | Escritura en un archivo de acceso aleatorio (parte 1 de 3).

```
17
18    // sale del programa si fstream no puede abrir el archivo
19    if ( !creditoSalida )
20    {
21       cerr << "No se pudo abrir el archivo." << endl;
22       exit( EXIT_FAILURE );
23    } // fin de if
24
25    cout << "Escriba el numero de cuenta (de 1 a 100, 0 para terminar la
             entrada)\n?  ";
26
27    // requiere que el usuario especifique el número de cuenta
28    DatosCliente cliente;
29    cin >> numeroCuenta;
30
31    // el usuario introduce información, la cual se copia en el archivo
32    while ( numeroCuenta > 0 && numeroCuenta <= 100 )
33    {
34       // el usuario introduce el apellido paterno, primer nombre y saldo
35       cout << "Escriba apellido paterno, primer nombre y saldo\n? ";
36       cin >> apellidoPaterno;
37       cin >> primerNombre;
38       cin >> saldo;
39
40       // establece los valores de numeroCuenta, apellidoPaterno, primerNombre y
            saldo del registro
41       cliente.establecerNumeroCuenta( numeroCuenta );
42       cliente.establecerApellidoPaterno( apellidoPaterno );
43       cliente.establecerPrimerNombre( primerNombre );
44       cliente.establecerSaldo( saldo );
45
46       // busca la posición en el archivo del registro especificado por el usuario
47       creditoSalida.seekp( ( cliente.obtenerNumeroCuenta() - 1 ) *
48          sizeof( DatosCliente ) );
49
50       // escribe la información especificada por el usuario en el archivo
51       creditoSalida.write( reinterpret_cast< const char * >( &cliente ),
52          sizeof( DatosCliente ) );
53
54       // permite al usuario escribir otro número de cuenta
55       cout << "Escriba el numero de cuenta\n? ";
56       cin >> numeroCuenta;
57    } // fin de while
58 } // fin de main
```

```
Escriba el numero de cuenta (de 1 a 100, 0 para terminar la entrada)
? 37
Escriba apellido paterno, primer nombre y saldo
? Barker Doug 0.00
Escriba el numero de cuenta
? 29
Escriba apellido paterno, primer nombre y saldo
? Brown Nancy -24.54
Escriba el numero de cuenta
? 96
```

Fig. 14.12 | Escritura en un archivo de acceso aleatorio (parte 2 de 3).

```
Escriba apellido paterno, primer nombre y saldo
? Stone Sam 34.98
Escriba el numero de cuenta
? 88
Escriba apellido paterno, primer nombre y saldo
? Smith Dave 258.34
Escriba el numero de cuenta
? 33
Escriba apellido paterno, primer nombre y saldo
? Dunn Stacey 314.33
Escriba el numero de cuenta
? 0
```

Fig. 14.12 | Escritura en un archivo de acceso aleatorio (parte 3 de 3).

Abrir un archivo para entrada y salida en modo binario

En la línea 16 se utiliza el objeto `creditoSalida` de `fstream` para abrir el archivo `credito.dat` existente. El archivo se abre para entrada y salida en *modo binario* mediante la combinación de los modos de apertura de archivos `ios::in`, `ios::out` e `ios::binary`. Al abrir el archivo `credito.dat` existente de esta forma aseguramos que este programa pueda manipular los registros que el programa de la figura 14.11 escribe en el archivo, en vez de crearlo desde cero.

Posicionamiento del apuntador de posición del archivo

En las líneas 47 y 48 se posiciona el apuntador "*colocar*" de posición del archivo para el objeto `creditoSalida` en la posición de byte calculada por

```
( cliente.obtenerNumeroCuenta() - 1 ) * sizeof( DatosCliente )
```

Puesto que el número de cuenta está entre 1 y 100, se resta 1 del número de cuenta al calcular la posición de byte del registro. Así, para el registro 1, el apuntador de posición del archivo se establece en el byte 0 del archivo.

14.9 Cómo leer de un archivo de acceso aleatorio en forma secuencial

En las secciones anteriores, creamos un archivo de acceso aleatorio y escribimos datos en ese archivo. En esta sección, desarrollaremos un programa que lee el archivo en forma secuencial e imprime sólo esos registros que contienen datos. Estos programas producen un beneficio adicional. Vea si puede determinar cuál es; lo revelaremos al final de esta sección.

La función `read` de `istream` introduce un número especificado de bytes, desde la posición actual en el flujo especificado, hasta un objeto. Por ejemplo, en las líneas 31 y 32 de la figura 14.13 se lee el número de bytes especificado por `sizeof(DatosCliente)` del archivo asociado con el objeto `creditoEntrada` de `ifstream` y se almacenan los datos en el registro `cliente`. La función `read` requiere un primer argumento de tipo `char *`. Como `&cliente` es de tipo `DatosCliente *`, `&cliente` debe convertirse en `char *` utilizando el operador de conversión `reinterpret_cast`.

```
1  // Fig. 14.13: Fig14_13.cpp
2  // Lectura secuencial de un archivo de acceso aleatorio.
3  #include <iostream>
```

Fig. 14.13 | Lectura secuencial de un archivo de acceso aleatorio (parte 1 de 3).

```
4  #include <iomanip>
5  #include <fstream>
6  #include <cstdlib>
7  #include "DatosCliente.h" // definición de la clase DatosCliente
8  using namespace std;
9
10 void imprimirLinea( ostream&, const DatosCliente & ); // prototipo
11
12 int main()
13 {
14    ifstream creditoEntrada( "credito.dat", ios::in | ios::binary );
15
16    // sale del programa si ifstream no puede abrir el archivo
17    if ( !creditoEntrada )
18    {
19       cerr << "No se pudo abrir el archivo." << endl;
20       exit( EXIT_FAILURE );
21    } // fin de if
22
23    // imprimir encabezados de columnas
24    cout << left << setw( 10 ) << "Cuenta" << setw( 16 )
25       << "Apellido" << setw( 11 ) << "Nombre" << left
26       << setw( 10 ) << right << "Saldo" << endl;
27
28    DatosCliente cliente; // crea un registro
29
30    // lee el primer registro del archivo
31    creditoEntrada.read( reinterpret_cast< char * >( &cliente ),
32       sizeof( DatosCliente ) );
33
34    // lee todos los registros del archivo
35    while ( creditoEntrada && !creditoEntrada.eof() )
36    {
37       // muestra un registro
38       if ( cliente.obtenerNumeroCuenta() != 0 )
39          imprimirLinea( cout, cliente );
40
41       // lee el siguiente registro del archivo
42       creditoEntrada( reinterpret_cast< char * >( &cliente )
43          sizeof( DatosCliente ) );
44    } // fin de while
45 } // fin de main
46
47 // muestra un solo registro
48 void imprimirLinea( ostream &salida, const DatosCliente &registro )
49 {
50    salida << left << setw( 10 ) << registro.obtenerNumeroCuenta()
51       << setw( 16 ) << registro.obtenerApellidoPaterno()
52       << setw( 11 ) << registro.obtenerPrimerNombre()
53       << setw( 10 ) << setprecision( 2 ) << right << fixed
54       << showpoint << registro.obtenerSaldo() << endl;
55 } // fin de la función imprimirLinea
```

Fig. 14.13 | Lectura secuencial de un archivo de acceso aleatorio (parte 2 de 3).

```
Cuenta    Apellido      Nombre        Saldo
29        Brown         Nancy         -24.54
33        Dunn          Stacey        314.33
37        Barker        Doug            0.00
88        Smith         Dave          258.34
96        Stone         Sam            34.98
```

Fig. 14.13 | Lectura secuencial de un archivo de acceso aleatorio (parte 3 de 3).

En la figura 14.13 se lee cada registro en el archivo `credito.dat` de manera secuencial, se comprueba cada registro para determinar si contiene datos y se muestran los resultados con formato para los registros que contienen datos. La condición en la línea 35 utiliza la función miembro `eof` de `ios` para determinar cuándo se llega al fin del archivo, y hace que la ejecución de la instrucción `while` termine. Además, si ocurre un error al leer del archivo, el ciclo termina debido a que `creditoEntrada` se evalúa como `false`. Los datos recibidos del archivo se imprimen mediante la función `imprimirLinea` (líneas 48 a 55), que recibe dos argumentos: un objeto `ostream` y una estructura `datosCliente` a imprimir. El tipo de parámetro `ostream` es interesante, ya que cualquier objeto `ostream` (como `cout`) o cualquier objeto de una clase derivada de `ostream` (como un objeto de tipo `ofstream`) puede suministrarse como argumento. Esto significa que se puede utilizar la *misma* función, por ejemplo, para enviar datos al flujo de salida estándar y a un flujo de archivo, sin tener que escribir funciones separadas.

¿Qué hay acerca de ese beneficio adicional que prometimos? Si examina la ventana de resultados, observará que los registros se listan *en orden* (por número de cuenta). Ésta es una consecuencia de la forma en que almacenamos estos registros en el archivo, usando las técnicas de acceso directo. El ordenamiento mediante las técnicas de acceso directo es relativamente rápido. *La velocidad se logra al hacer el archivo lo bastante grande como para que pueda contener todos los posibles registros que se podrían crear*. Desde luego que esto significa que el archivo podría estar ocupado *escasamente* la mayor parte del tiempo, produciendo como resultado un desperdicio del almacenamiento. Éste es un ejemplo de la *concesión entre espacio y tiempo*: al utilizar *grandes cantidades de espacio*, podemos desarrollar un *algoritmo de ordenamiento mucho más rápido*. Por fortuna, la continua reducción en el precio de las unidades de almacenamiento ha provocado que esto no sea tan problemático.

14.10 Caso de estudio: un programa para procesar transacciones

Ahora presentaremos un programa para procesar transacciones de un tamaño considerable (figura 14.14), en el que se utiliza un archivo de acceso aleatorio para lograr un procesamiento de acceso instantáneo. El programa mantiene la información de las cuentas de un banco. Actualiza las cuentas existentes, agrega nuevas cuentas, elimina cuentas y almacena un listado con formato de todas las cuentas actuales en un archivo de texto. Asumimos que se ha ejecutado el programa de la figura 14.11 para crear el archivo `credito.dat` y que se ha ejecutado el programa de la figura 14.12 para insertar los datos iniciales. En la línea 25 se abre el archivo `credito.dat` mediante la creación de un objeto `fstream` para escritura y lectura en formato binario.

```
1  // Fig. 14.14: Fig14_14.cpp
2  // Este programa lee un archivo de acceso aleatorio en forma secuencial,
3  // actualiza los datos escritos anteriormente en el archivo, crea datos
4  // para colocarlos en el archivo, y elimina los datos previamente almacenados.
```

Fig. 14.14 | Programa de cuentas bancarias (parte 1 de 6).

```
5  #include <iostream>
6  #include <fstream>
7  #include <iomanip>
8  #include <cstdlib>
9  #include "DatosCliente.h" // definición de la clase DatosCliente
10 using namespace std;
11
12 int escribirOpcion();
13 void crearArchivoTexto( fstream& );
14 void actualizarRegistro( fstream& );
15 void nuevoRegistro( fstream& );
16 void eliminarRegistro( fstream& );
17 void imprimirLinea( ostream&, const DatosCliente & );
18 int obtenerCuenta( const char * const );
19
20 enum Opciones { IMPRIMIR = 1, ACTUALIZAR, NUEVO, ELIMINAR, TERMINAR };
21
22 int main()
23 {
24    // abre el archivo para leer y escribir
25    fstream creditoEntSal( "credito.dat", ios::in | ios::out | ios::binary );
26
27    // sale del programa si fstream no puede abrir el archivo
28    if ( !creditoEntSal )
29    {
30       cerr << "No se pudo abrir el archivo." << endl;
31       exit ( EXIT_FAILURE );
32    } // fin de if
33
34    int opcion; // almacena la opción del usuario
35
36    // permite al usuario especificar una acción
37    while ( ( opcion = escribirOpcion() ) != TERMINAR )
38    {
39       switch ( opcion )
40       {
41          case IMPRIMIR: // crea un archivo de texto a partir del archivo de registros
42             crearArchivoTexto( creditoEntSal );
43             break;
44          case ACTUALIZAR: // actualiza el registro
45             actualizarRegistro( creditoEntSal );
46             break;
47          case NUEVO: // crea un registro
48             nuevoRegistro( creditoEntSal );
49             break;
50          case ELIMINAR: // elimina un registro existente
51             eliminarRegistro( reditoEntSal );
52             break;
53          default: // muestra un error si el usuario no selecciona una opción válida
54             cerr << "Opcion incorrecta" << endl;
55             break;
56       } // fin de switch
57
```

Fig. 14.14 | Programa de cuentas bancarias (parte 2 de 6).

```
58       creditoEntSal.clear(); // restablece el indicador de fin de archivo
59    } // fin de while
60 } // fin de main
61
62 // permite al usuario introducir la opción del menú
63 int escribirOpcion()
64 {
65    // muestra las opciones disponibles
66    cout << "\nEscriba su opcion" << endl
67       << "1 - almacenar un archivo de texto con formato de las cuentas" << endl
68       << "    llamado \"imprimir.txt\" para imprimirlo" << endl
69       << "2 - actualizar una cuenta" << endl
70       << "3 - agregar una nueva cuenta" << endl
71       << "4 - eliminar una cuenta" << endl
72       << "5 - fin del programa\n? ";
73
74    int opcionMenu;
75    cin >> opcionMenu; // introduce la selección del menú que hizo el usuario
76    return opcionMenu;
77 } // fin de la función escribirOpcion
78
79 // crea un archivo de texto con formato para imprimirlo
80 void crearArchivoTexto( fstream &leerDelArchivo )
81 {
82    // crea un archivo de texto
83    ofstream archivoImprimirSalida( "imprimir.txt", ios::out );
84
85    // sale del programa si ofstream no puede crear el archivo
86    if ( !archivoImprimirSalida )
87    {
88       cerr << "No se pudo crear el archivo." << endl;
89       exit( EXIT_FAILURE );
90    } // fin de if
91
92    // imprime los encabezados de las columnas
93    archivoImprimirSalida << left << setw( 10 ) << "Cuenta" << setw( 16 )
94       << "Apellido" << setw( 11 ) << "Nombre" << right
95       << setw( 10 ) << "Saldo" << endl;
96
97    // establece el apuntador de posición del archivo en el inicio de leerDelArchivo
98    leerDelArchivo.seekg( 0 );
99
100   // lee el primer registro del archivo de registros
101   DatosCliente cliente;
102   leerDelArchivo.read( reinterpret_cast< char * >( &cliente ),
103      sizeof( DatosCliente ) );
104
105   // copia todos los registros del archivo de registros al archivo de texto
106   while ( !leerDelArchivo.eof() )
107   {
108      // escribe un solo registro en el archivo de texto
109      if ( cliente.obtenerNumeroCuenta() != 0 ) // omite los registros vacíos
110         imprimirLinea( archivoImprimirSalida, cliente );
```

Fig. 14.14 | Programa de cuentas bancarias (parte 3 de 6).

```
111
112       // lee el siguiente registro del archivo de registros
113       leerDelArchivo.read( reinterpret_cast< char * >( &cliente ),
114          sizeof( DatosCliente ) );
115    } // fin de while
116 } // fin de la función crearArchivoTexto
117
118 // actualiza el saldo en el registro
119 void actualizarRegistro( fstream &actualizarArchivo )
120 {
121    // obtiene el número de la cuenta que se va a actualizar
122    int numeroCuenta = obtenerCuenta( "Escriba la cuenta que se debe actualizar" );
123
124    // desplaza el apuntador de posición del archivo al registro correcto en el archivo
125    actualizarArchivo.seekg( ( numeroCuenta - 1 ) * sizeof( DatosCliente ) );
126
127    // lee el primer registro del archivo
128    DatosCliente cliente;
129    actualizarArchivo.read( reinterpret_cast< char * >( &cliente  ),
130       sizeof( DatosCliente ) );
131
132    // actualiza el registro
133    if ( cliente.obtenerNumeroCuenta() != 0 )
134    {
135       imprimirLinea( cout, cliente ); // muestra el registro
136
137       // solicita al usuario que especifique la transacción
138       cout << "\nEscriba el cargo (+) o pago (-): ";
139       double transaction; // cargo o pago
140       cin >> transaction;
141
142       // actualiza el saldo del registro
143       double saldoAnterior = cliente.obtenerSaldo();
144       cliente.establecerSaldo( saldoAnterior + transaccion );
145       imprimirLinea( cout, cliente ); // muestra el registro
146
147       // desplaza el apuntador de posición del archivo al registro correcto en el archivo
148       actualizarArchivo.seekp( ( numeroCuenta - 1 ) * sizeof( DatosCliente ) );
149
150       // escribe el registro actualizado sobre el registro anterior en el archivo
151       actualizarArchivo.write( reinterpret_cast< const char * >( &cliente ),
152          sizeof( DatosCliente ) );
153    } // fin de if
154    else // muestra un error si la cuenta no existe
155       cerr << "La cuenta #" << numeroCuenta
156          << " no tiene informacion." << endl;
157 } // fin de la función actualizarRegistro
158
159 // crea e inserta un registro
160 void nuevoRegistrod( fstream &insertarEnArchivo )
161 {
162    // obtiene el número de cuenta que se debe crear
163    int numeroCuenta = obtenerCuenta( "Escriba el nuevo numero de cuenta" );
```

Fig. 14.14 | Programa de cuentas bancarias (parte 4 de 6).

```
164
165    // desplaza el apuntador de posición del archivo al registro correcto en el
          archivo
166    insertarEnArchivo.seekg( ( numeroCuenta - 1 ) * sizeof( DatosCliente ) );
167
168    // lee un registro del archivo
169    DatosCliente cliente;
170    insertarEnArchivo.read( reinterpret_cast< char * >( &cliente ),
171       sizeof( DatosCliente ) );
172
173    // crea un registro, si no es que ya existe
174    if ( cliente.obtenerNumeroCuenta() == 0 )
175    {
176       string apellidoPaterno;
177       string primerNombre;
178       double saldo;
179
180       // el usuario introduce el apellido paterno, primer nombre y saldo
181       cout << "Escriba apellido paterno, primer nombre y saldo\n? ";
182       cin >> setw( 15 ) >> apellidoPaterno;
183       cin >> setw( 10 ) >> primerNombre;
184       cin >> saldo;
185
186       // usa los valores para llenar los valores de la cuenta
187       cliente.establecerApellidoPaterno( apellidoPaterno );
188       cliente.establecerPrimerNombre( primerNombre  );
189       cliente.establecerSaldo( saldo );
190       cliente.establecerNumeroCuenta( numeroCuenta );
191
192       // desplaza el apuntador de posición del archivo al registro correcto
             en el archivo
193       insertarEnArchivo.seekp( ( numeroCuenta - 1 ) * sizeof( DatosCliente ) );
194
195       // inserta el registro en el archivo
196       insertarEnArchivo.write( reinterpret_cast< const char * >( &cliente ),
197          sizeof( DatosCliente ) );
198    } // fin de if
199    else // muestra un error si la cuenta ya existe
200       cerr << "La cuenta #" << numeroCuenta
201          << " ya contiene información." << endl;
202 } // fin de la función nuevoRegistro
203
204 // elimina un registro existente
205 void eliminarRegistro( fstream &eliminarDelArchivo )
206 {
207    // obtiene el número de cuenta que debe eliminar
208    int numeroCuenta = obtenerCuenta( "Escriba la cuenta a eliminar" );
209
210    // desplaza el apuntador de posición del archivo al registro correcto en el archivo
211    eliminarDelArchivo.seekg( ( numeroCuenta - 1 ) * sizeof( DatosCliente ) );
212
213    // lee el registro del archivo
214    DatosCliente cliente;
215    eliminarDelArchivo.read( reinterpret_cast< char * >( &cliente ),
216       sizeof( DatosCliente ) );
```

Fig. 14.14 | Programa de cuentas bancarias (parte 5 de 6).

```
217
218    // elimina el registro, si es que existe en el archivo
219    if ( cliente.obtenerNumeroCuenta() != 0 )
220    {
221       DatosCliente clienteEnBlanco; // crea un registro en blanco
222
223       // desplaza el apuntador de posición del archivo al
             registro correcto en el archivo
224       eliminarDelArchivo.seekp( ( numeroCuenta - 1 ) *
225          sizeof( DatosCliente ) );
226
227       // reemplaza el registro existente con uno en blanco
228       eliminarDelArchivo.write(
229          reinterpret_cast< const char * >( &clienteEnBlanco ),
230          sizeof( DatosCliente ) );
231
232       cout << "La cuenta #" << numeroCuenta << " se elimino.\n";
233    } // fin de if
234    else // muestra un error si el registro no existe
235       cerr << "La cuenta #" << numeroCuenta << " esta vacia.\n";
236 } // fin de eliminarRegistro
237
238 // muestra un solo registro
239 void imprimirLinea( ostream &salida, const DatosCliente &registro )
240 {
241    salida << left << setw( 10 ) << registro.obtenerNumeroCuenta()
242       << setw( 16 ) << registro.obtenerApellidoPaterno()
243       << setw( 11 ) << registro.obtenerPrimerNombre()
244       << setw( 10 ) << setprecision( 2 ) << right << fixed
245       << showpoint << registro.obtenerSaldo() << endl;
246 } // fin de la función imprimirLinea
247
248 // obtiene el valor del número de cuenta del usuario
249 int obtenerCuenta( const char * const indicador )
250 {
251    int numeroCuenta;
252
253    // obtiene el valor del número de cuenta
254    do
255    {
256       cout << indicador << " (1 - 100): ";
257       cin >> numeroCuenta;
258    } while ( numeroCuenta < 1 || numeroCuenta > 100 );
259
260    return numeroCuenta;
261 } // fin de la función obtenerCuenta
```

Fig. 14.14 | Programa de cuentas bancarias (parte 6 de 6).

El programa tiene cinco opciones (la opción 5 es para terminar el programa). La opción 1 llama a la función `crearArchivoTexto` para almacenar una lista con formato de toda la información de las cuentas en un archivo de texto llamado `imprimir.txt`, el cual se puede imprimir. La función `crearArchivoTexto` (líneas 80 a 116) recibe un objeto `fstream` como un argumento a utilizar para introducir datos desde el archivo `credito.dat`. La función `crearArchivoTexto` invoca a la función miembro `read` de `istream` (líneas 102 y 103) y utiliza las técnicas de acceso a archivos secuenciales de la figura 14.13 para introducir datos desde `credito.dat`. La función `imprimirLinea`, que vimos en

la sección 14.9, se utiliza para escribir los datos en el archivo `imprimir.txt`. Observe que la función `crearArchivoTexto` utiliza la función miembro `seekg` de `istream` (línea 98) para asegurar que el apuntador de posición del archivo esté en el inicio del archivo. Después de seleccionar la opción 1, el archivo `imprimir.txt` contiene lo siguiente:

```
Cuenta    Apellido        Nombre          Saldo
29        Brown           Nancy          -24.54
33        Dunn            Stacey         314.33
37        Barker          Doug             0.00
88        Smith           Dave           258.34
96        Stone           Sam             34.98
```

La opción 2 llama a `actualizarRegistro` (líneas 119 a 157) para actualizar una cuenta. Esta función sólo actualiza un registro *existente*, por lo que primero determina si el registro especificado está *vacío*. En las líneas 129 y 130 se leen datos y se colocan en el objeto `cliente`, usando la función miembro `read` de `istream`. Después, en la línea 133 se compara el valor devuelto por `obtenerNumeroCuenta` del objeto `cliente` con cero, para determinar si el registro contiene información. Si este valor es cero, en las líneas 155 y 156 se imprime un mensaje de error que indica que el registro está vacío. Si el registro contiene información, en la línea 135 se muestra el registro mediante la función `imprimirLinea`, en la línea 140 se introduce el monto de la transacción y en las líneas 143 a 152 se calcula el nuevo saldo y se vuelve a escribir el registro en el archivo. Una ejecución típica para la opción 2 es

```
Escriba la cuenta que se debe actualizar (1 - 100): 37
37        Barker          Doug             0.00

Escriba el cargo (+) o pago (-): +87.99
37        Barker          Doug            87.99
```

La opción 3 llama a la función `nuevoRegistro` (líneas 160 a 202) para agregar una nueva cuenta al archivo. Si el usuario escribe un número de cuenta para una cuenta *existente*, `nuevoRegistro` muestra un mensaje de error indicando que la cuenta existe (líneas 200 y 201). Esta función agrega una nueva cuenta de la misma forma que el programa de la figura 14.12. Una ejecución típica para la opción 3 es

```
Escriba el nuevo numero de cuenta (1 - 100): 22
Escriba apellido paterno, primer nombre y saldo
? Johnston Sarah 247.45
```

La opción 4 llama a la función `eliminarRegistro` (líneas 205 a 236) para eliminar un registro del archivo. En la línea 208 se pide al usuario que introduzca el número de cuenta. Sólo puede eliminarse un registro *existente*, por lo que si la cuenta especificada está vacía, en la línea 235 se muestra un mensaje de error. Si la cuenta existe, en las líneas 221 a 230 se reinicializa esa cuenta al copiar un registro vacío (`clienteEnBlanco`) al archivo. En la línea 232 se muestra un mensaje para informar al usuario que se ha eliminado el registro. Una ejecución típica para la opción 4 es

```
Escriba la cuenta a eliminar (1 - 100): 29
La cuenta #29 se elimino.
```

14.11 Serialización de objetos

En este capítulo y en el capítulo 13 se presentó el estilo orientado a objetos de las operaciones de entrada/salida. Sin embargo, nuestros ejemplos se concentraron en la E/S de los tipos fundamentales, en vez de los objetos de tipos definidos por el usuario. En el capítulo 10 mostramos cómo realizar operaciones de entrada y salida con objetos, mediante la sobrecarga de operadores. Realizamos operaciones de entrada con objetos mediante la sobrecarga del operador de extracción de flujo (>>) para el objeto `istream` apropiado. Realizamos operaciones de salida con objetos mediante la sobrecarga del operador de inserción de flujo (<<) para el objeto `ostream` apropiado. En ambos casos, sólo se realizaron operaciones de entrada o salida con los miembros de datos de un objeto y, en cada caso, se hicieron con un formato significativo sólo para los objetos de ese tipo específico. Las funciones miembro de un objeto *no* se envían o reciben con los datos de un objeto; en vez de ello, *una copia de las funciones miembro de la clase permanece disponible en forma interna y es compartida por todos los objetos de la clase*.

Cuando se escriben los miembros de datos de un objeto en un archivo en disco, se *pierde* la información del tipo del objeto. Sólo se almacenan los valores de los atributos del objeto, y no la información del tipo en el disco. Si el programa que lee estos datos conoce el tipo del objeto al que éstos corresponden, puede leer los datos y colocarlos en un objeto de ese tipo, como hicimos en nuestros ejemplos con archivos de acceso aleatorio.

Cuando almacenamos objetos de distintos tipos en el mismo archivo, ocurre un problema interesante. ¿Cómo podemos diferenciarlos (o sus colecciones de miembros de datos) al leerlos e introducirlos en un programa? El problema es que, por lo general, los objetos *no* tienen campos de tipos (vimos este problema en el capítulo 12).

Una metodología utilizada por varios lenguajes de programación es lo que se conoce como **serialización de objetos**. Un **objeto serializado** es un objeto que se representa como una secuencia de bytes que incluye los *datos* del objeto, así como información acerca de su *tipo* y de los *tipos de datos almacenados en el mismo*. Una vez que se escribe un objeto serializado en un archivo, se puede leer del mismo y **deserializarse**; es decir, se pueden utilizar la información del tipo y los bytes que representan al objeto y sus datos para *recrearlo* en memoria. C++ *no* proporciona un mecanismo de serialización integrado; sin embargo, existen bibliotecas de C++ de código fuente abierto y de terceros que soportan la serialización de objetos. Las Bibliotecas Boost de C++ de código fuente abierto (`www.boost.org`) ofrecen soporte para serializar objetos en los formatos de texto, binario y en lenguaje de marcado extensible (XML) (`www.boost.org/libs/serialization/doc/index.html`).

14.12 Conclusión

En este capítulo presentamos varias técnicas de procesamiento de archivos para manipular datos persistentes. Vimos una introducción a las diferencias entre los flujos basados en caracteres y basados en bytes, y a varias plantillas de clases de procesamiento de archivos en el encabezado `<fstream>`. Después, el lector aprendió a utilizar el procesamiento de archivos secuenciales para manipular los registros almacenados en orden, mediante el campo clave de registro. También aprendió a utilizar archivos de acceso aleatorio para obtener y manipular "al instante" registros de longitud fija. Presentamos un ejemplo práctico considerable sobre el procesamiento de transacciones, que utiliza un archivo de acceso aleatorio para realizar un procesamiento con "acceso instantáneo". Por último, vimos los conceptos básicos de la serialización de objetos. En el capítulo 7 presentamos las clases `array` y `vector` de la Biblioteca estándar. En el siguiente capítulo aprenderá sobre las demás estructuras predefinidas de datos de la Biblioteca estándar (conocidas como contenedores), así como sobre los fundamentos de los iteradores, que se utilizan para manipular elementos contenedores.

Resumen

Sección 14.1 Introducción

- Los archivos se utilizan para la persistencia de los datos (pág. 600): la retención permanente de datos.
- Las computadoras almacenan archivos en dispositivos de almacenamiento secundario (pág. 600), como discos duros, CD, DVD, memoria flash y cintas magnéticas.

Sección 14.2 Archivos y flujos

- C++ ve a cada archivo simplemente como una secuencia de bytes.
- Cada archivo termina con un marcador de fin de archivo (pág. 600), o en un número de byte específico registrado en una estructura de datos administrativa, mantenida por el sistema.
- Al abrir un archivo, se crea un objeto y se asocia un flujo a ese objeto.
- Para realizar el procesamiento de archivos en C++, se deben incluir los encabezados `<iostream>` y `<fstream>`.
- El encabezado `<fstream>` (pág. 600) incluye las definiciones para las plantillas de clases de flujos `basic_ifstream` (operaciones de entrada con archivos), `basic_ostream` (operaciones de salida con archivos) y `basic_fstream` (operaciones de entrada y salida con archivos).
- Cada plantilla de clase tiene una especialización de plantilla predefinida que permite operaciones de E/S con caracteres. La biblioteca `<fstream>` proporciona alias con `typedef` para estas especializaciones de plantillas. La definición `typedef ifstream` representa una especialización de `basic_ifstream` que permite operaciones de entrada con valores `char` desde un archivo. La definición `typedef ofstream` representa una especialización de `basic_ofstream` que permite operaciones de salida con valores `char` hacia archivos. La definición `typedef fstream` (pág. 600) representa una especialización de `basic_fstream` que permite operaciones de entrada/salida con valores `char` desde/hacia archivos.
- Las plantillas de procesamiento de archivos se derivan de las plantillas de clases `basic_istream`, `basic_ostream` y `basic_iostream`, respectivamente. Por ende, todas las funciones miembro, operadores y manipuladores que pertenecen a estas plantillas también se pueden aplicar a los flujos de archivos.

Sección 14.3 Creación de un archivo secuencial

- C++ no impone una estructura sobre un archivo, por lo que debemos estructurar los archivos para que cumplan con los requerimientos de la aplicación.
- Un archivo se puede abrir en modo de salida cuando se crea un objeto `ofstream`. Se pasan dos argumentos al constructor del objeto: el nombre del archivo (pág. 602) y el modo de apertura del archivo (pág. 602).
- Para un objeto `ofstream` (pág. 602), el modo de apertura del archivo puede ser `ios::out` (pág. 602) para enviar datos a un archivo, o `ios::app` (pág. 602) para adjuntar datos al final de un archivo. Los archivos existentes que se abren con el modo `ios::out` se truncan (pág. 603). Si el archivo especificado no existe aún, entonces el objeto `ofstream` crea el archivo, usando ese nombre de archivo.
- De manera predeterminada, los objetos `ofstream` se abren en modo de salida.
- Un objeto `ofstream` se puede crear sin abrir un archivo específico; se puede adjuntar un archivo al objeto más adelante, con la función miembro `open` (pág. 603).
- La función miembro `operator!` de `ios` determina si un flujo se abrió en forma correcta. Este operador se puede utilizar en una condición que devuelva un valor verdadero si se establece el bit `failbit` o el bit `badbit` para el flujo en la operación de apertura.
- La función miembro `operator void *` de `ios` convierte un flujo en un apuntador, para que se pueda comparar con `0`. Cuando se utiliza el valor de un apuntador como una condición, un apuntador nulo representa `false` y un apuntador no nulo representa `true`. Si se ha establecido el bit `failbit` o el bit `badbit` para el flujo, se devuelve `0` (`false`).
- Al introducir el indicador de fin de archivo, se establece el bit `failbit` para `cin`.
- La función `operator void *` se puede utilizar para evaluar un objeto de entrada para el fin de archivo, en vez de llamar a la función miembro `eof` de manera explícita en el objeto de entrada.
- Cuando se hace una llamada al destructor de un objeto flujo, se cierra el flujo correspondiente. También se puede cerrar el flujo de manera explícita, mediante la función miembro `close` del flujo.

Sección 14.4 Cómo leer datos de un archivo secuencial

- Los archivos almacenan datos, para que puedan obtenerse y procesarse cuando sea necesario.
- Al crear un objeto `ifstream`, se abre un archivo en modo de entrada. El constructor de `ifstream` puede recibir el nombre del archivo y el modo de apertura del mismo como argumentos.
- Debemos abrir un archivo en modo de entrada solamente si el contenido del archivo no se debe modificar.
- Los objetos de la clase `ifstream` se abren en modo de entrada de manera predeterminada.
- Un objeto `ifstream` se puede crear sin tener que abrir un archivo específico; se le puede adjuntar un archivo más adelante.
- Para obtener los datos secuencialmente de un archivo, los programas generalmente empiezan a leer desde el inicio del archivo, y leen todos los datos en forma consecutiva hasta encontrar los datos deseados.
- Las funciones miembro para reposicionar el apuntador de posición del archivo (pág. 607) son `seekg` ("buscar obtener"; pág. 607) para `istream` y `seekp` ("buscar colocar"; pág. 607) para `ostream`. Cada objeto `istream` tiene un "apuntador obtener", el cual indica el número de byte en el archivo desde el que se va a realizar la siguiente operación de entrada, y cada objeto `ostream` tiene un "apuntador colocar", el cual indica el número de byte en el archivo en el que se deben colocar los datos de la siguiente operación de salida.
- El argumento para `seekg` (pág. 607) es un entero largo. Se puede especificar un segundo argumento para indicar la dirección de búsqueda (pág. 607), que puede ser `ios::beg` (el valor predeterminado; pág. 607) para un posicionamiento relativo al inicio de un flujo, `ios::cur` (pág. 607) para un posicionamiento relativo a la posición actual en un flujo, o `ios::end` (pág. 607) para un posicionamiento relativo al final de un flujo.
- El apuntador de posición del archivo (pág. 607) es un valor entero que especifica la ubicación en el archivo como un número de bytes a partir de la ubicación inicial del archivo [es decir, el desplazamiento (pág. 607) desde el inicio del archivo].
- Las funciones miembro `tellg` (pág. 607) y `tellp` (pág. 607) se proporcionan para devolver las ubicaciones actuales de los apuntadores "obtener" y "colocar", respectivamente.

Sección 14.5 Actualización de archivos secuenciales

- Los datos a los que se da formato y se escriben en un archivo secuencial no se pueden modificar sin el riesgo de destruir otros datos en el archivo. El problema es que los registros pueden variar en tamaño.

Sección 14.6 Archivos de acceso aleatorio

- Los archivos secuenciales son inapropiados para las aplicaciones de acceso instantáneo (pág. 611), en los que un registro específico se debe localizar de inmediato.
- El acceso instantáneo se logra mediante los archivos de acceso aleatorio (pág. 612). Se puede acceder a los registros individuales de un archivo de acceso aleatorio en forma directa (y rápida), sin tener que buscar en otros registros.
- El método más sencillo para dar formato a los archivos para un acceso aleatorio es requerir que todos los registros en un archivo tengan la misma longitud fija. Al utilizar registros de longitud fija que tengan el mismo tamaño, es más fácil para el programa calcular (como una función del tamaño del registro y la clave de registro) la ubicación exacta de cualquier registro, relativa al inicio del archivo.
- Se pueden insertar datos en un archivo de acceso aleatorio sin destruir los demás datos en el archivo.
- Los datos almacenados con anterioridad se pueden actualizar o eliminar sin tener que volver a escribir el archivo completo.

Sección 14.7 Creación de un archivo de acceso aleatorio

- La función miembro `write` de `ostream` envía como salida un número fijo de bytes (empezando en una ubicación específica en memoria) al flujo especificado. La función `write` escribe los datos en la ubicación en el archivo especificada por el apuntador "colocar" de posición del archivo.
- La función miembro `read` de `istream` (pág. 612) introduce un número fijo de bytes del flujo especificado hacia un área en la memoria, empezando en una dirección especificada. Si el flujo está asociado con un archivo, la función `read` introduce bytes en la ubicación en el archivo especificada por el apuntador "obtener" de posición del archivo.

- La función `write` trata su primer argumento como un grupo de bytes, al ver el objeto en memoria como un valor `const char *`, el cual es un apuntador a un byte (recuerde que un `char` es un byte). Empezando a partir de esa ubicación, la función `write` envía a la salida el número de bytes especificado por su segundo argumento. Después, se puede utilizar la función `read` de `istream` para leer los bytes y colocarlos de vuelta en la memoria.
- El operador `reinterpret_cast` (pág. 613) convierte un apuntador de un tipo a un tipo de apuntador no relacionado.
- Una operación con `reinterpret_cast` se realiza en tiempo de compilación y no modifica el valor del objeto al que apunta su operando.
- Un programa que lee datos sin formato debe compilarse y ejecutarse en un sistema compatible con el programa que escribió los datos; distintos sistemas pueden representar internamente los datos de una forma diferente.
- Los objetos de la clase `string` no tienen un tamaño uniforme, sino que utilizan la memoria asignada en forma dinámica para dar cabida a las cadenas de varias longitudes.

Sección 14.8 Cómo escribir datos al azar a un archivo de acceso aleatorio

- Para combinar varios modos de apertura de archivos, hay que separar cada modo de apertura de los otros mediante el operador OR inclusivo a nivel de bits (|).
- La función miembro `size` de `string` (pág. 616) obtiene la longitud de un objeto `string`.
- El modo de apertura de archivos `ios::binary` (pág. 616) indica que un archivo debe abrirse en modo binario.

Sección 14.9 Cómo leer de un archivo de acceso aleatorio en forma secuencial

- La función `read` de `istream` introduce un número especificado de bytes, desde la posición actual del flujo especificado, hacia un objeto.
- Una función que recibe un parámetro `ostream` puede recibir cualquier objeto `ostream` (como `cout`) o cualquier objeto de una clase derivada de `ostream` (como un objeto de tipo `stream`) como argumento. Esto significa que se puede utilizar la misma función, por ejemplo, para realizar operaciones de salida en el flujo de salida estándar y en un flujo de archivos, sin tener que especificar funciones separadas.

Sección 14.11 Serialización de objetos

- Cuando se envían miembros de datos de un objeto a un archivo en disco, perdemos la información del tipo de ese objeto. Sólo almacenamos los valores de los atributos de los objetos, no la información del tipo, en el disco. Si el programa que lee estos datos conoce el tipo del objeto al que corresponden, puede leer los datos y colocarlos en un objeto de ese tipo.
- Un objeto serializado (pág. 628) es un objeto que se representa como una secuencia de bytes que incluye los datos del objeto, así como información acerca del tipo del objeto y los tipos de datos almacenados en él. Un objeto serializado puede leerse del archivo y deserializarse (pág. 628).
- Las Bibliotecas Boost de código fuente abierto proporcionan soporte para los objetos (pág. 628) en los formatos de texto, binario y en lenguaje de marcado extensible (XML).

Ejercicios de autoevaluación

14.1 *(Llene los espacios en blanco)* Complete los siguientes enunciados:

a) La función miembro _____ de los flujos de archivos `fstream`, `ifstream` y `ofstream` cierra un archivo.
b) La función miembro _____ de `istream` lee un carácter del flujo especificado.
c) La función miembro _____ de los flujos de archivos `fstream`, `ifstream` y `ostream` abre un archivo.
d) La función miembro _____ de `istream` se utiliza comúnmente cuando se leen datos de un archivo en aplicaciones de acceso aleatorio.
e) Las funciones miembro _____ y _____ de `istream` y `ostream` establecen el apuntador de posición del archivo en una ubicación específica de un flujo de entrada o salida, respectivamente.

14.2 ***(Verdadero o falso)*** Indique si cada uno de los siguientes enunciados es *verdadero* o *falso*. En caso de ser *falso*, explique por qué.

a) La función miembro `read` no puede utilizarse para leer datos del objeto de entrada `cin`.
b) El programador debe crear los objetos `cin`, `cout`, `cerr` y `clog` de manera explícita.
c) Un programa debe llamar a la función `close` de manera explícita para cerrar un archivo asociado con un objeto `ifstream`, `ofstream` o `fstream`.
d) Si el apuntador de posición del archivo apunta a una ubicación en un archivo secuencial que no sea el inicio del archivo, éste se debe cerrar y volver a abrir para leer datos desde el inicio del mismo.
e) La función miembro `write` de `ostream` puede escribir en el flujo de salida estándar `cout`.
f) Los datos en archivos secuenciales siempre se actualizan sin sobrescribir los datos aledaños.
g) Es innecesario buscar en todos los registros de un archivo de acceso aleatorio para encontrar un registro específico.
h) Los registros en los archivos de acceso aleatorio deben ser de una longitud uniforme.
i) Las funciones miembro `seekp` y `seekg` deben buscar en forma relativa al inicio de un archivo.

14.3 Asuma que cada uno de los siguientes enunciados se aplica al mismo programa.

a) Escriba una instrucción que abra el archivo `maestant.dat` en modo de entrada; use un objeto `ifstream` llamado `maestAntEntrada`.
b) Escriba una instrucción que abra el archivo `trasn.dat` en modo de entrada; use un objeto `ifstream` llamado `transaccionEnt`.
c) Escriba una instrucción que abra el archivo `maestnuevo.dat` en modo de salida (y de creación); use el objeto `ofstream maestNuevSalida`.
d) Escriba una instrucción que lea un registro del archivo `maestant.dat`. El registro consiste en el entero `numeroCuenta`, la cadena `nombre` y el valor de punto flotante `saldoActual`; use el objeto `ifstream maestAntEntrada`.
e) Escriba una instrucción que lea un registro del archivo `trans.dat`. El registro consiste en el entero `numCuenta` y el valor de punto flotante `montoDolares`; utilice el objeto `ifstream transaccionEnt`.
f) Escriba una instrucción que escriba un registro en el archivo `maestnuev.dat`. El registro consiste en el entero `numCuenta`, la cadena `nombre` y el valor de punto flotante `saldoActual`; utilice el objeto `ofstream maestNuevSalida`.

14.4 Busque el (los) error(es) y muestre cómo corregirlo(s) en cada uno de los siguientes enunciados.

a) El archivo `porpagar.dat` al que hace referencia el objeto `ofstream porPagarSalida` no se ha abierto.

```
porPagarSalida << cuenta << empresa << monto << endl;
```

b) La siguiente instrucción debe leer un registro del archivo `porpagar.dat`. El objeto `ifstream porPagar-Entrada` hace referencia a este archivo, y el objeto `ifstream porCobrarEntrada` hace referencia al archivo `porcobrar.dat`.

```
porCobrarEntrada >> cuenta >> empresa >> monto;
```

c) El archivo `herramientas.dat` debe abrirse para agregarle datos al archivo sin descartar los datos actuales.

```
ofstream herramientasSalida( "herramientas.dat", ios::out );
```

Respuestas a los ejercicios de autoevaluación

14.1 a) `close`. b) `get`. c) `open`. d) `read`. e) `seekg`, `seekp`.

14.2 a) Falso. La función `read` puede leer de cualquier objeto flujo de entrada derivado de `ifstream`.
b) Falso. Estos cuatro flujos se crean de manera automática para el programador. El encabezado `<iostream>` debe incluirse en un archivo para usarlos. Este encabezado incluye declaraciones para cada objeto flujo predefinido.

c) Falso. Los archivos se cerrarán cuando se ejecuten los destructores para los objetos `ifstream`, `ofstream` o `fstream`, cuando los objetos flujo quedan fuera de alcance, o antes de que termine la ejecución del programa, pero es una buena práctica de programación cerrar todos los archivos en forma explícita mediante `close`, una vez que ya no sean necesarios.
d) Falso. La función miembro `seekg` se puede utilizar para reposicionar el apuntador "obtener" o "colocar" de posición del archivo en el inicio del archivo.
e) Verdadero.
f) Falso. En la mayoría de los casos, los registros de archivos secuenciales no son de una longitud uniforme. Por lo tanto, es posible que al actualizar un registro se sobrescriban otros datos.
g) Verdadero.
h) Falso. Por lo general, los registros en un archivo de acceso aleatorio son de longitud uniforme.
i) Falso. Es posible buscar desde el inicio del archivo, desde el final del archivo y desde la posición actual en el archivo.

14.3 a)
```
ifstream transaccionEnt( "maestant.dat", ios::in );
```
b)
```
ifstream transaccionEnt( "trans.dat", ios::in );
```
c)
```
ofstream maestNuevSalida( "maestnuev.dat", ios::out );
```
d)
```
maestNuevEntrada >> numeroCuenta >> nombre >> saldoActual;
```
e)
```
transaccionEnt >> numCuenta >> montoDolares;
```
f)
```
maestNuevEntrada << numCuenta << " " << nombre << " " << saldoActual;
```

14.4 a) *Error:* el archivo `porpagar.dat` no se ha abierto antes de tratar de enviar datos al flujo.
Corrección: use la función `open de ostream` para abrir `porpagar.dat` en modo de salida.
b) *Error:* se está utilizando el objeto `istream` incorrecto para leer un registro del archivo llamado `porpagar.dat`.
Corrección: use el objeto `porPagarEnt` de `istream` para hacer referencia a `porpagar.dat`.
c) *Error:* el contenido del archivo se descarta, debido a que se abre en modo de salida (`ios::out`).
Corrección: para agregar datos al archivo, ábralo para actualizar (`ios::ate`) o para adjuntar (`ios::app`).

Ejercicios

14.5 ***(Llene los espacios en blanco)*** Complete los siguientes enunciados:
a) Las computadoras almacenan grandes cantidades de datos en dispositivos de almacenamiento secundario, tales como _____.
b) Los objetos de flujo estándar declarados por el encabezado <iostream> son _____, _____, _____ y _____.
c) La función miembro _____ de `ostream` envía un carácter al flujo especificado.
d) La función miembro _____ de `ostream` se utiliza por lo general para escribir datos en un archivo con acceso aleatorio.
e) La función miembro _____ de `istream` reposiciona el apuntador de posición del archivo en un archivo.

14.6 ***(Asociar archivos)*** El ejercicio 14.3 pide al lector que escriba una serie de instrucciones individuales. En realidad, estas instrucciones forman el núcleo de un tipo importante de programa para procesar archivos: un programa para asociar archivos. En el procesamiento comercial de datos, es común tener varios archivos en cada sistema de aplicaciones. Por ejemplo, en un sistema de cuentas por cobrar hay generalmente un archivo maestro, el cual contiene información detallada acerca de cada cliente, como su nombre, dirección, número telefónico, saldo deudor, límite de crédito, términos de descuento, acuerdos contractuales y posiblemente un historial condensado de compras recientes y pagos en efectivo.

A medida que ocurren las transacciones (es decir, a medida que se generan las ventas y llegan los pagos en efectivo), se introducen en un archivo. Al final de cada periodo de negocios (un mes para algunas compañías, una semana para otras y un día en algunos casos), el archivo de transacciones (llamado `trans.dat` en el ejercicio 14.3) se aplica al archivo maestro (llamado `maestant.dat` en el ejercicio 14.3) para actualizar el registro de compras y pagos de cada cuenta. Durante una actualización, el archivo maestro se vuelve a escribir como un nuevo archivo (`maestnue.dat`), el cual se utiliza al final del siguiente periodo de negocios para empezar de nuevo el proceso de actualización.

Los programas para asociar archivos deben tratar con ciertos problemas que no existen en programas de un solo archivo. Por ejemplo, no siempre ocurre una asociación. Si un cliente en el archivo maestro no ha realizado compras ni pagos en efectivo en el periodo actual de negocios, no aparecerá ningún registro para este cliente en el archivo de transacciones. De manera similar, un cliente que haya realizado compras o pagos en efectivo podría haberse mudado recientemente a esta comunidad, y tal vez la compañía no haya tenido la oportunidad de crear un registro maestro para este cliente.

Use las instrucciones del ejercicio 14.3 como base para escribir un programa completo para asociar archivos de cuentas por cobrar. Utilice el número de cuenta en cada archivo como la clave de registro para fines de asociar los archivos. Suponga que cada archivo es un archivo secuencial con registros almacenados en orden ascendente, por número de cuenta.

Cuando ocurra una coincidencia (es decir, que aparezcan registros con el mismo número de cuenta en el archivo maestro y en el archivo de transacciones), sume el monto en dólares del archivo de transacciones al saldo actual en el archivo maestro, y escriba el registro de `maestnuev.dat`. (Suponga que las compras se indican mediante montos positivos en el archivo de transacciones, y los pagos mediante montos negativos.) Cuando haya un registro maestro para una cuenta específica, pero no haya un registro de transacciones correspondiente, simplemente escriba el registro maestro en `maestnue.dat`. Cuando haya un registro de transacciones pero no un registro maestro correspondiente, imprima el mensaje de error `"Hay un registro de transacciones no asociado para ese numero de cliente ..."` (utilice el número de cuenta del registro de transacciones).

14.7 ***(Datos de prueba para asociar archivos)*** Después de escribir el programa del ejercicio 14.6, escriba un programa simple para crear ciertos datos de prueba para verificar el programa. Utilice los siguientes datos de cuentas de ejemplo:

Archivo maestro Número de cuenta	Nombre	Saldo
100	Alan Jones	348.17
300	Mary Smith	27.19
500	Sam Sharp	0.00
700	Suzy Green	–14.22

Archivo de transacciones Número de cuenta	Monto de la transacción
100	27.14
300	62.11
400	100.56
900	82.17

14.8 ***(Prueba de asociación de archivos)*** Ejecute el programa del ejercicio 14.6, usando los archivos de datos de prueba creados en el ejercicio 14.7. Imprima el nuevo archivo maestro. Compruebe que las cuentas se hayan actualizado en forma correcta.

14.9 ***(Mejora al programa para asociar archivos)*** Es común tener varios registros de transacciones con la misma clave de registro, ya que un cliente específico podría realizar varias compras y pagos en efectivo durante un periodo de negocios. Vuelva a escribir su programa de asociación de archivos de cuentas por cobrar del ejercicio 14.6 para proveer la posibilidad de manejar varios registros de transacciones con la misma clave de registro. Modifique los datos de prueba del ejercicio 14.7 para incluir los siguientes registros de transacciones adicionales:

Número de cuenta	Monto en dólares
300	83.89
700	80.78
700	1.53

14.10 Escriba una serie de instrucciones que realicen cada una de las siguientes acciones. Suponga que hemos definido la clase `Persona` que contiene los siguientes miembros de datos `private`:

```
char apellidoPaterno[ 15 ];
char primerNombre[ 10 ];
int edad;
int id;
```

y las funciones miembro `public`

```
// funciones de acceso para id
void establecerId( int );
int obtenerId() const;

// funciones de acceso para apellidoPaterno
void establecerApellidoPaterno( const string & );
string obtenerApellidoPaterno() const;

// funciones de acceso para primerNombre
void establecerPrimerNombre( const string & );
string obtenerPrimerNombre() const;

// funciones de acceso para edad
void establecerEdad( int );
int obtenerEdad() const;
```

Suponga también que cualquier archivo de acceso aleatorio se ha abierto en forma apropiada.

a) Inicialice el archivo `nombreedad.dat` con 100 registros que almacenen los valores `apellidoPaterno = "noasignado"`, `primerNombre = ""` y `edad = 0`.
b) Introduzca 10 apellidos paternos, primeros nombres y edades, y escríbalos en el archivo.
c) Actualice un registro que ya contenga información. Si el registro no contiene información, informe al usuario que `"No hay informacion"`.
d) Elimine un registro que contenga información, reinicializando ese registro específico.

14.11 ***(Inventario de ferretería)*** Usted es el propietario de una ferretería y necesita llevar un inventario que le pueda indicar los distintos tipos de herramientas que tiene, cuántas de ellas tiene a la mano y el costo de cada una. Escriba un programa que inicialice el archivo de acceso aleatorio `ferreteria.dat` con 100 registros vacíos, que le permita introducir los datos relacionados con cada herramienta, listar todas sus herramientas, eliminar un registro para una herramienta que ya no tenga y que le permita actualizar *cualquier* información en el archivo. El número de identificación de cada herramienta deberá ser el número de registro. Utilice la siguiente información para empezar su archivo:

Registro #	Nombre de la herramienta	Cantidad	Costo
3	Lijadora eléctrica	7	57.98
17	Martillo	76	11.99
24	Serrucho	21	11.00
39	Podadora de césped	3	79.50
56	Sierra eléctrica	18	99.99
68	Destornillador	106	6.99
77	Mazo	11	21.50
83	Llave inglesa	34	7.50

14.12 ***(Generador de palabras de números telefónicos)*** Los teclados telefónicos estándar contienen los dígitos del 0 al 9. Cada uno de los números del 2 al 9 tiene tres letras asociadas, como se indica en la siguiente tabla:

Dígito	Letras	Dígito	Letras
2	A B C	6	M N O
3	D E F	7	P Q R S
4	G H I	8	T U V
5	J K L	9	W X Y Z

A muchas personas se les dificulta memorizar números telefónicos, por lo que utilizan la correspondencia entre los dígitos y las letras para desarrollar palabras de siete letras que corresponden a sus números telefónicos. Por ejemplo, una persona cuyo número telefónico sea 686-3767 podría utilizar la correspondencia indicada en la tabla anterior para desarrollar la palabra de siete letras "NUMEROS".

Las empresas intentan con frecuencia obtener números telefónicos que sean fáciles de recordar para sus clientes. Si una empresa puede anunciar una palabra simple para que sus clientes la marquen, entonces sin duda esa empresa recibirá unas cuantas llamadas más. Cada palabra de siete dígitos corresponde a muchas palabras separadas de siete letras. Por desgracia, la mayoría de estas palabras representan yuxtaposiciones irreconocibles de letras. Sin embargo, es posible que el dueño de una carpintería se complazca en saber que el número telefónico de su taller, 683-2537, corresponde a "MUEBLES". Un veterinario con el número telefónico 627-2682 se complacería en saber que ese número corresponde a las letras "MASCOTA".

Escriba un programa que, dado un número de siete dígitos, escriba en un archivo todas las combinaciones posibles de palabras de siete letras que corresponden a ese número. Hay 2,187 (3 elevado a la séptima potencia) combinaciones posibles. Evite los números telefónicos con los dígitos 0 y 1.

14.13 ***(Operador `sizeof`)*** Escriba un programa que utilice el operador `sizeof` para determinar los tamaños en bytes de los diversos tipos de datos de su sistema computacional. Escriba los resultados en el archivo `tamdatos.dat`, de manera que pueda imprimir los resultados más tarde. Estos resultados se deberán mostrar en un formato de dos columnas, con el nombre del tipo en la columna izquierda y el tamaño de ese tipo en la columna derecha, como se muestra a continuación:

```
char                  1
unsigned char         1
short int             2
unsigned short int    2
int                   4
unsigned int          4
long int              4
unsigned long int     4
float                 4
double                8
long double          10
```

[*Nota:* los tamaños de los tipos de datos integrados en su computadora podrían ser distintos de los antes listados].

Hacer la diferencia

14.14 ***(Explorador de phishing)*** El phishing es una forma de robo de identidad, en la que mediante un correo electrónico, el emisor se hace pasar por una fuente confiable, e intenta adquirir información privada, como nombres de usuario, contraseñas, números de tarjetas de crédito y número de seguro social. Los correos electrónicos de phishing que afirman provenir de bancos, compañías de tarjetas de crédito, sitios de subastas, redes sociales y sistemas de pago en línea populares pueden tener una apariencia bastante legítima. A menudo, estos mensajes fraudulentos proveen vínculos a sitios Web falsificados, en donde se pide al usuario que introduzca información confidencial.

Visite Security Extra (`www.securityextra.com/`), `www.snopes.com` y otros sitios Web en donde encontrará listas de las principales estafas de phishing. Visite también el sitio Web del Grupo de trabajo anti-phishing

```
www.antiphishing.org/
```

y el sitio Web de Investigaciones cibernéticas del FBI

```
www.fbi.gov/cyberinvest/cyberhome.htm
```

en donde encontrará información sobre las estafas más recientes y cómo protegerse a sí mismo.

Cree una lista de 30 palabras, frases y nombres de compañías que se encuentren con frecuencia en los mensajes de phishing. Asigne el valor de un punto a cada una, con base en una estimación de la probabilidad de que aparezca en un mensaje de phishing (por ejemplo, un punto si es poco probable, dos puntos si es algo probable, o tres puntos si es muy probable). Escriba un programa que explore un archivo de texto en busca de estos términos y frases. Para cada ocurrencia de una palabra clave o frase dentro del archivo de texto, agregue el valor del punto asignado al total de puntos para esa palabra o frase. Para cada palabra clave o frase encontrada, imprima en pantalla una línea con la palabra o frase, el número de ocurrencias y el total de puntos. Después muestre el total de puntos para todo el mensaje. ¿Asigna su programa un total de puntos alto a ciertos mensajes reales de correo electrónico de phishing que haya recibido? ¿Asigna un total de puntos alto a ciertos correos electrónicos legítimos que haya recibido?

Nota al lector

Los capítulos 15, 16 y 17
se encuentran, en español en el sitio web del libro

Los capítulos 18, 19, 20, 21, 22 y 23
se encuentran, en inglés en el sitio web del libro

A

Precedencia y asociatividad de operadores

Los operadores se muestran en orden decreciente de precedencia, de arriba hacia abajo (figura A.1).

Operador	Tipo	Asociatividad

`::`
`::`
`()`
	binario de resolución de alcance
	unario de resolución de alcance
	paréntesis de agrupamiento *[vea la precaución en la figura 2.10 con respecto a los paréntesis de agrupamiento].*
		izquierda a derecha

`()`
`[]`
`.`
`->`
`++`
`--`
`typeid`
`dynamic_cast < tipo >`
`static_cast< tipo >`
`reinterpret_cast< tipo >`
`const_cast< tipo >`
	llamada a función
	subíndice de arreglo
	selección de miembro mediante un objeto
	selección de miembro mediante un apuntador
	unario de incremento postfijo
	unario de decremento postfijo
	información de tipos en tiempo de ejecución
	conversión con comprobación de tipos en tiempo de ejecución
	conversión con comprobación de tipos en tiempo de compilación
	conversión para conversiones no estándar
	eliminar el calificador `const`
		izquierda a derecha

`++`
`--`
`+`
`-`
`!`
`~`
`sizeof`
`&`
`*`
`new`
`new[]`
`delete`
`delete[]`
	unario de incremento prefijo
	unario de decremento prefijo
	unario de suma
	unario de resta
	unario de negación lógica
	unario de complemento a nivel de bits
	determinar el tamaño en bytes
	dirección
	desreferencia
	asignación dinámica de memoria
	asignación dinámica de arreglo
	desasignación dinámica de memoria
	desasignación dinámica de arreglo
		derecha a izquierda

`( tipo )`	unario de conversión estilo C	derecha a izquierda

Fig. A.1 | Tabla de precedencia y asociatividad de operadores (parte 1 de 2).

Operador	Tipo	Asociatividad
`.*` `->*`	apuntador a miembro mediante un objeto apuntador a miembro mediante un apuntador	izquierda a derecha

`*`
`/`
`%`
	multiplicación
	división
	módulo
		izquierda a derecha

`+` `-`	suma resta	izquierda a derecha
`<<` `>>`	desplazamiento a la izquierda a nivel de bits desplazamiento a la derecha a nivel de bits	izquierda a derecha

`<`
`<=`
`>`
`>=`
	menor que relacional
	menor o igual que relacional
	mayor que relacional
	mayor o igual que relacional
		izquierda a derecha

`==` `!=`	es igual a relacional no es igual a relacional	izquierda a derecha
`&`	AND a nivel de bits	izquierda a derecha
`^`	OR exclusivo a nivel de bits	izquierda a derecha
`|`	OR inclusivo a nivel de bits	izquierda a derecha
`&&`	AND lógico	izquierda a derecha
`||`	OR lógico	izquierda a derecha
`?:`	ternario condicional	derecha a izquierda

`=`
`+=`
`-=`
`*=`
`/=`
`%=`
`&=`
`^=`
`|=`
`<<=`
`>>=`
	asignación
	asignación de suma
	asignación de resta
	asignación de multiplicación
	asignación de división
	asignación de módulo
	AND de asignación a nivel de bits
	OR de asignación exclusivo a nivel de bits
	OR de asignación inclusivo a nivel de bits
	desplazamiento a la izquierda de asignación a nivel de bits
	desplazamiento a la derecha de asignación a nivel de bits
		derecha a izquierda

`,`	coma	izquierda a derecha

Fig. A.1 | Tabla de precedencia y asociatividad de operadores (parte 2 de 2).

B

Conjunto de caracteres ASCII

Conjunto de caracteres ASCII

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	‘
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\’	]	^	_	’	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Fig. B.1 | Conjunto de caracteres ASCII.

Los dígitos a la izquierda de la tabla son los dígitos izquierdos de los equivalentes decimales (0-127) de los códigos de caracteres, y los dígitos en la parte superior de la tabla son los dígitos derechos de los códigos de caracteres. Por ejemplo, el código de carácter para la “F” es 70, mientras que para el “&” es 38.

C

Tipos fundamentales

En la figura C.1 se listan los tipos fundamentales de C++. El Documento del estándar de C++ no proporciona el número exacto de bytes requeridos para almacenar variables de estos tipos en memoria. Sin embargo, el Documento del estándar de C++ sí indica cómo se relacionan los requerimientos de memoria para los tipos fundamentales entre sí. Por orden creciente de requerimientos de memoria, los tipos enteros con signo son `signed char`, `short int`, `int`, `long int` y `long long int`. Esto significa que un `short int` debe proporcionar cuando menos tanto espacio de almacenamiento como un `signed char`; un `int` debe proporcionar cuando menos tanto espacio de almacenamiento como un `short int`; un `long int` debe proporcionar cuando menos tanto espacio de almacenamiento como un `int`, y un `long long int` debe proporcionar cuando menos tanto espacio de almacenamiento como un `long int`. Cada tipo entero con signo tiene su correspondiente tipo entero sin signo con los mismos requerimientos de memoria. Los tipos sin signo no pueden representar valores negativos, pero pueden representar aproximadamente el doble de valores positivos que sus tipos con signo asociados. Por orden creciente de requerimientos de memoria, los tipos de punto flotante son `float`, `double` y `long double`. Al igual que los tipos enteros, un `double` debe proporcionar cuando menos tanto espacio de almacenamiento como un `float`, y un `long double` debe proporcionar cuando menos tanto espacio de almacenamiento como un `double`.

Tipos integrales	Tipos de punto flotante
`bool`	`float`
`char`	`double`
`signed char`	`long double`
`unsigned char`	
`short int`	
`unsigned short int`	
`int`	
`unsigned int`	
`long int`	
`unsigned long int`	
`long long int`	
`unsigned long long int`	
`char16_t`	
`char32_t`	
`wchar_t`	

Fig. C.1 | Tipos fundamentales de C++.

Los tamaños y rangos exactos de valores para los tipos fundamentales dependen de la implementación. Los archivos de encabezado `<climits>` (para los tipos integrales) y `<cfloat>` (para los tipos de punto flotante) especifican los rangos de valores soportados en el sistema del programador.

El rango de valores que soporta un tipo depende del número de bytes que se utilizan para representar a ese tipo. Por ejemplo, considere un sistema con valores `int` de 4 bytes (32 bits). Para el tipo `int` con signo, los valores no negativos están en el rango de 0 a 2 147 483 647 ($2^{31} - 1$). Los valores negativos están en el rango de –1 a –2 147 483 647 ($-2^{31} - 1$). Esto da un total de 2^{32} valores posibles. Un valor `unsigned int` en el mismo sistema utilizaría el mismo número de bits para representar datos, pero no representaría ningún valor negativo. Esto produce valores en el rango de 0 a 4 294 967 295 ($2^{32}-1$). En el mismo sistema, un `short int` no podría utilizar más de 32 bits para representar sus datos, y un `long int` debe utilizar cuando menos 32 bits.

C++ proporciona el tipo de datos `bool` para las variables que sólo pueden contener los valores `true` y `false`. C++11 introdujo los tipos `long long int` y `unsigned long long int`; por lo general para valores enteros de 64 bits (aunque el estándar no requiere esto). C++11 también introdujo los nuevos tipos de caracteres `char16_t` y `char32_t` para representar caracteres Unicode.

D

Sistemas numéricos

He aquí sólo los números ratificados.
—William Shakespeare

Objetivos

En este apéndice, usted aprenderá a:

- Comprender los conceptos básicos acerca de los sistemas numéricos, como base, valor posicional y valor simbólico.
- Trabajar con los números representados en los sistemas numéricos binario, octal y hexadecimal.
- Abreviar los números binarios como octales o hexadecimales.
- Convertir los números octales y hexadecimales en binarios.
- Realizar conversiones hacia y desde números decimales y sus equivalentes en binario, octal y hexadecimal.
- Comprender el funcionamiento de la aritmética binaria y la manera en que se representan los números binarios negativos, utilizando la notación de complemento a dos.

Plan general

D.1 Introducción

En este apéndice presentaremos los sistemas numéricos clave que utilizan los programadores de C++, especialmente cuando trabajan en proyectos de software que requieren de una estrecha interacción con el hardware a nivel de máquina. Entre los proyectos de este tipo están los sistemas operativos, el software de redes computacionales, los compiladores, sistemas de bases de datos y aplicaciones que requieren de un alto rendimiento.

Cuando escribimos un entero, como 227 o –63 en un programa de C++, se asume que el número está en el **sistema numérico decimal** (**base 10**). Los **dígitos** en el sistema numérico decimal son 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9. El dígito más bajo es el 0 y el más alto es el 9 (uno menos que la base, 10). En su interior, las computadoras utilizan el **sistema numérico binario** (**base 2**). Este sistema numérico sólo tiene dos dígitos: 0 y 1. El dígito más bajo es el 0 y el más alto es el 1 (uno menos que la base, 2).

Como veremos, los números binarios tienden a ser mucho más extensos que sus equivalentes decimales. Los programadores que trabajan en lenguajes ensambladores y en lenguajes de alto nivel como C++, que les permiten llegar hasta el nivel de máquina, encuentran que es complicado trabajar con números binarios. Por eso existen otros dos sistemas numéricos, el **sistema numérico octal** (**base 8**) y el **sistema numérico hexadecimal** (**base 16**), que son populares debido a que permiten abreviar los números binarios de una manera conveniente.

En el sistema numérico octal, los dígitos utilizados son del 0 al 7. Debido a que tanto el sistema numérico binario como el octal tienen menos dígitos que el sistema numérico decimal, sus dígitos son los mismos que sus correspondientes en decimal.

El sistema numérico hexadecimal presenta un problema, ya que requiere de 16 dígitos: el dígito más bajo es 0 y el más alto tiene un valor equivalente al 15 decimal (uno menos que la base, 16). Por convención utilizamos las letras de la A a la F para representar los dígitos hexadecimales que corresponden a los valores decimales del 10 al 15. Por lo tanto, en hexadecimal podemos tener números como el 876, que consisten solamente de dígitos similares a los decimales; números como 8A55F que consisten de dígitos y letras; y números como FFE que consisten solamente de letras. En ocasiones un número hexadecimal puede coincidir con una palabra común como FACE o FEED (en inglés); esto puede parecer extraño para los programadores acostumbrados a trabajar con números. Los dígitos de los sistemas numéricos binario, octal, decimal y hexadecimal se sintetizan en las figuras D.1 y D.2.

Cada uno de estos sistemas numéricos utilizan la **notación posicional**: cada posición en la que se escribe un dígito tiene un **valor posicional** distinto. Por ejemplo, en el número decimal 937 (el 9, el 3 y el 7 se conocen como **valores simbólicos**) decimos que el 7 se escribe en la posición de las unidades; el 3, en la de las decenas; y el 9, en la de las centenas. Observe que cada una de estas posiciones es una potencia de la base (10) y que estas potencias empiezan en 0 y aumentan de 1 en 1 a medida que nos desplazamos hacia la izquierda por el número (figura D.3).

Dígito binario	Dígito octal	Dígito decimal	Dígito hexadecimal
0	0	0	0
1	1	1	1
	2	2	2
	3	3	3
	4	4	4
	5	5	5
	6	6	6
	7	7	7
		8	8
		9	9
			A (valor de 10 en decimal)
			B (valor de 11 en decimal)
			C (valor de 12 en decimal)
			D (valor de 13 en decimal)
			E (valor de 14 en decimal)
			F (valor de 15 en decimal)

Fig. D.1 | Dígitos de los sistemas numéricos binario, octal, decimal y hexadecimal.

Atributo	Binario	Octal	Decimal	Hexadecimal
Base	2	8	10	16
Dígito más bajo	0	0	0	0
Dígito más alto	1	7	9	F

Fig. D.2 | Comparación de los sistemas numéricos binario, octal, decimal y hexadecimal.

Valores posicionales en el sistema numérico decimal			
Dígito decimal	9	3	7
Nombre de la posición	Centenas	Decenas	Unidades
Valor posicional	100	10	1
Valor posicional como potencia de la base (10)	10^2	10^1	10^0

Fig. D.3 | Valores posicionales en el sistema numérico decimal.

Para números decimales más extensos, las siguientes posiciones a la izquierda serían: de millares (10 a la tercera potencia), de decenas de millares (10 a la cuarta potencia), de centenas de millares (10 a la quinta potencia), de los millones (10 a la sexta potencia), de decenas de millones (10 a la séptima potencia), y así sucesivamente.

En el número binario 101, el 1 más a la derecha se escribe en la posición de los unos, el 0 se escribe en la posición de los dos y el 1 de más a la izquierda se escribe en la posición de los cuatros. Observe que cada una de estas posiciones es una potencia de la base (base 2) y que estas potencias empiezan en 0 y aumentan de 1 en 1 a medida que nos desplazamos hacia la izquierda por el número (figura D.4). Por lo tanto, $101 = 2^2 + 2^0 = 4 + 1 = 5$.

Valores posicionales en el sistema numérico binario			
Dígito binario	1	0	1
Nombre de la posición	Cuatros	Dos	Unos
Valor posicional	4	2	1
Valor posicional como potencia de la base (2)	2^2	2^1	2^0

Fig. D.4 | Valores posicionales en el sistema numérico binario.

Para números binarios más extensos, las siguientes posiciones a la izquierda serían la posición de los ochos (2 a la tercera potencia), la posición de los dieciséis (2 a la cuarta potencia), la posición de los treinta y dos (2 a la quinta potencia), la posición de los sesenta y cuatros (2 a la sexta potencia), y así sucesivamente.

En el número octal 425, decimos que el 5 se escribe en la posición de los unos, el 2 se escribe en la posición de los ochos y el 4 se escribe en la posición de los sesenta y cuatros. Observe que cada una de estas posiciones es una potencia de la base (base 8) y que estas potencias empiezan en 0 y aumentan de 1 en 1 a medida que nos desplazamos hacia la izquierda por el número (figura D.5).

Valores posicionales en el sistema numérico octal			
Dígito octal	4	2	5
Nombre de la posición	Sesenta y cuatros	Ochos	Unos
Valor posicional	64	8	1
Valor posicional como potencia de la base (8)	8^2	8^1	8^0

Fig. D.5 | Valores posicionales en el sistema numérico octal.

Para números octales más extensos, las siguientes posiciones a la izquierda sería la posición de los quinientos doces (8 a la tercera potencia), la posición de los cuatro mil noventa y seis (8 a la cuarta potencia), la posición de los treinta y dos mil setecientos sesenta y ochos (8 a la quinta potencia), y así sucesivamente.

En el número hexadecimal 3DA, decimos que la A se escribe en la posición de los unos, la D se escribe en la posición de los dieciséis y el 3 se escribe en la posición de los doscientos cincuenta y seis. Observe que cada una de estas posiciones es una potencia de la base (base 16) y que estas potencias empiezan en 0 y aumentan de 1 en 1 a medida que nos desplazamos hacia la izquierda por el número (figura D.6).

Para números hexadecimales más extensos, las siguientes posiciones a la izquierda serían la posición de los cuatro mil noventa y seis (16 a la tercera potencia), la posición de los sesenta y cinco mil quinientos treinta y seis (16 a la cuarta potencia), y así sucesivamente.

Valores posicionales en el sistema numérico hexadecimal			
Dígito hexadecimal	3	D	A
Nombre de la posición	Doscientos cincuenta y seis	Dieciséis	Unos
Valor posicional	256	16	1
Valor posicional como potencia de la base (16)	16^2	16^1	16^0

Fig. D.6 | Valores posicionales en el sistema numérico hexadecimal.

D.2 Abreviatura de los números binarios como números octales y hexadecimales

En computación, el uso principal de los números octales y hexadecimales es para abreviar representaciones binarias demasiado extensas. La figura D.7 muestra que los números binarios extensos pueden expresarse más concisamente en sistemas numéricos con bases mayores que en el sistema numérico binario.

Número decimal	Representación binaria	Representación octal	Representación hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Fig. D.7 | Equivalentes en decimal, binario, octal y hexadecimal.

Una relación especialmente importante que tienen tanto el sistema numérico octal como el hexadecimal con el sistema binario es que las bases de los sistemas octal y hexadecimal (8 y 16, respectivamente) son potencias de la base del sistema numérico binario (base 2). Considere el siguiente número

binario de 12 dígitos y sus equivalentes en octal y hexadecimal. Vea si puede determinar cómo esta relación hace que sea conveniente el abreviar los números binarios en octal o hexadecimal. Las respuestas siguen después de los números.

```
Número binario    Equivalente en octal    Equivalente en hexadecimal
100011010001      4321                    8D1
```

Para ver cómo el número binario se convierte fácilmente en octal, sólo divida el número binario de 12 dígitos en grupos de tres bits consecutivos, empezando desde la derecha, y escriba esos grupos por encima de los dígitos correspondientes del número octal, como se muestra a continuación:

```
100     011     010     001
4       3       2       1
```

Observe que el dígito octal que escribió debajo de cada grupo de tres bits corresponde precisamente al equivalente octal de ese número binario de 3 dígitos que se muestra en la figura D.7.

El mismo tipo de relación puede observarse al convertir números de binario a hexadecimal. Divida el número binario de 12 dígitos en grupos de cuatro bits consecutivos, empezando desde la derecha, y escriba esos grupos por encima de los dígitos correspondientes del número hexadecimal, como se muestra a continuación:

```
1000    1101    0001
8       D       1
```

Observe que el dígito hexadecimal que escribió debajo de cada grupo de cuatro bits corresponde precisamente al equivalente hexadecimal de ese número binario de 4 dígitos que se muestra en la figura D.7.

D.3 Conversión de números octales y hexadecimales a binarios

En la sección anterior vimos cómo convertir números binarios a sus equivalentes en octal y hexadecimal, formando grupos de dígitos binarios y simplemente volviéndolos a escribir como sus valores equivalentes en dígitos octales o hexadecimales. Este proceso puede utilizarse en forma inversa para producir el equivalente en binario de un número octal o hexadecimal.

Por ejemplo, el número octal 653 se convierte en binario simplemente escribiendo el 6 como su equivalente binario de 3 dígitos 110, el 5 como su equivalente binario de 3 dígitos 101 y el 3 como su equivalente binario de 3 dígitos 011 para formar el número binario de 9 dígitos 110101011.

El número hexadecimal FAD5 se convierte en binario simplemente escribiendo la F como su equivalente binario de 4 dígitos 1111, la A como su equivalente binario de 4 dígitos 1010, la D como su equivalente binario de 4 dígitos 1101 y el 5 como su equivalente binario de 4 dígitos 0101, para formar el número binario de 16 dígitos 1111101011010101.

D.4 Conversión de un número binario, octal o hexadecimal a decimal

Como estamos acostumbrados a trabajar con el sistema decimal, a menudo es conveniente convertir un número binario, octal o hexadecimal en decimal para tener una idea de lo que "realmente" vale el número. Nuestros diagramas en la sección D.1 expresan los valores posicionales en decimal. Para convertir un número en decimal desde otra base, multiplique el equivalente en decimal de cada dígito por su valor

posicional y sume estos productos. Por ejemplo, el número binario 110101 se convierte en el número 53 decimal, como se muestra en la figura D.8.

Conversión de un número binario en decimal						
Valores posicionales:	32	16	8	4	2	1
Valores simbólicos:	1	1	0	1	0	1
Productos:	1*32=32	1*16=16	0*8=0	1*4=4	0*2=0	1*1=1
Suma:	= 32 + 16 + 0 + 4 + 0s + 1 = 53					

Fig. D.8 | Conversión de un número binario en decimal.

Para convertir el número 7614 octal en el número 3980 decimal utilizamos la misma técnica, esta vez utilizando los valores posicionales apropiados para el sistema octal, como se muestra en la figura D.9.

Conversión de un número octal en decimal				
Valores posicionales:	512	64	8	1
Valores simbólicos:	7	6	1	4
Productos:	7*512=3584	6*64=384	1*8=8	4*1=4
Suma:	= 3584 + 384 + 8 + 4 = 3980			

Fig. D.9 | Conversión de un número octal en decimal.

Para convertir el número AD3B hexadecimal en el número 44347 decimal utilizamos la misma técnica, esta vez empleando los valores posicionales apropiados para el sistema hexadecimal, como se muestra en la figura D.10.

Conversión de un número hexadecimal en decimal				
Valores posicionales:	4096	256	16	1
Valores simbólicos:	A	D	3	B
Productos:	A*4096=40960	D*256=3328	3*16=48	B*1=11
Suma:	= 40960 + 3328 + 48 + 11 = 44347			

Fig. D.10 | Conversión de un número hexadecimal en decimal.

D.5 Conversión de un número decimal a binario, octal o hexadecimal

Las conversiones de la sección D.4 siguen naturalmente las convenciones de la notación posicional. Las conversiones de decimal a binario, octal o hexadecimal también siguen estas convenciones.

Suponga que queremos convertir el número 57 decimal en binario. Empezamos escribiendo los valores posicionales de las columnas de derecha a izquierda, hasta llegar a una columna cuyo valor posicional sea mayor que el número decimal. Como no necesitamos esa columna, podemos descartarla. Por lo tanto, primero escribimos:

Valores posicionales:	64	32	16	8	4	2	1

Luego descartamos la columna con el valor posicional de 64, dejando:

```
Valores posicionales:   32   16   8   4   2   1
```

A continuación, empezamos a trabajar desde la columna más a la izquierda y nos vamos desplazando hacia la derecha. Dividimos 57 entre 32 y observamos que hay un 32 en 57, con un residuo de 25, por lo que escribimos 1 en la columna de los 32. Dividimos 25 entre 16 y observamos que hay un 16 en 25, con un residuo de 9, por lo que escribimos 1 en la columna de los 16. Dividimos 9 entre 8 y observamos que hay un 8 en 9 con un residuo de 1. Las siguientes dos columnas producen el cociente de cero cuando se divide 1 entre sus valores posicionales, por lo que escribimos 0 en las columnas de los 4 y de los 2. Por último, 1 entre 1 es 1, por lo que escribimos 1 en la columna de los 1. Esto nos da:

```
Valores posicionales:   32   16   8   4   2   1
Valores simbólicos:     1    1    1   0   0   1
```

y por lo tanto, el 57 decimal es equivalente al 111001 binario.

Para convertir el número decimal 103 en octal, empezamos por escribir los valores posicionales de las columnas hasta llegar a una columna cuyo valor posicional sea mayor que el número decimal. Como no necesitamos esa columna, podemos descartarla. Por lo tanto, primero escribimos:

```
Valores posicionales:   512   64   8   1
```

Luego descartamos la columna con el valor posicional de 512, lo que nos da:

```
Valores posicionales:   64   8   1
```

A continuación, empezamos a trabajar desde la columna más a la izquierda y nos vamos desplazando hacia la derecha. Dividimos 103 entre 64 y observamos que hay un 64 en 103 con un residuo de 39, por lo que escribimos 1 en la columna de los 64. Dividimos 39 entre 8 y observamos que el 8 cabe cuatro veces en 39 con un residuo de 7, por lo que escribimos 4 en la columna de los 8. Por último, dividimos 7 entre 1 y observamos que el 1 cabe siete veces en 7 y no hay residuo, por lo que escribimos 7 en la columna de los 1. Esto nos da:

```
Valores posicionales:   64   8   1
Valores simbólicos:     1    4   7
```

y por lo tanto, el 103 decimal es equivalente al 147 octal.

Para convertir el número decimal 375 en hexadecimal, empezamos por escribir los valores posicionales de las columnas hasta llegar a una columna cuyo valor posicional sea mayor que el número decimal. Como no necesitamos esa columna, podemos descartarla. Por consecuencia, primero escribimos:

```
Valores posicionales:   4096   256   16   1
```

Luego descartamos la columna con el valor posicional de 4096, lo que nos da:

```
Valores posicionales:   256   16   1
```

A continuación, empezamos a trabajar desde la columna más a la izquierda y nos vamos desplazando hacia la derecha. Dividimos 375 entre 256 y observamos que 256 cabe una vez en 375 con un residuo de 119, por lo que escribimos 1 en la columna de los 256. Dividimos 119 entre 16 y observamos que el 16 cabe siete veces en 119 con un residuo de 7, por lo que escribimos 7 en la columna de los 16. Por último, dividimos 7 entre 1 y observamos que el 1 cabe siete veces en 7 y no hay residuo, por lo que escribimos 7 en la columna de los 1. Esto produce:

```
Valores posicionales:   256   16   1
Valores simbólicos:     1     7    7
```

y por lo tanto, el 375 decimal es equivalente al 177 hexadecimal.

D.6 Números binarios negativos: notación de complemento a dos

La discusión en este apéndice se ha enfocado hasta ahora en números positivos. En esta sección explicaremos cómo las computadoras representan números negativos mediante el uso de la **notación de complementos a dos**. Primero explicaremos cómo se forma el complemento a dos de un número binario y después mostraremos por qué representa el valor negativo de dicho número binario.

Considere una máquina con enteros de 32 bits. Suponga que se ejecuta la siguiente instrucción:

```
int valor = 13;
```

La representación en 32 bits de `valor` es:

```
00000000 00000000 00000000 00001101
```

Para formar el negativo de `valor`, primero formamos su **complemento a uno** aplicando el **operador de complemento a nivel de bits** (~) de C++:

```
complementoAUnoDeValor = ~valor;
```

Internamente, `~valor` es ahora valor con cada uno de sus bits invertidos; los unos se convierten en ceros y los ceros en unos, como se muestra a continuación:

```
valor:
00000000 00000000 00000000 00001101
~valor  (es decir, el complemento a uno de valor):
11111111 11111111 11111111 11110010
```

Para formar el complemento a dos de `valor`, simplemente sumamos 1 al complemento a uno de `valor`. Por lo tanto:

```
El complemento a dos de valor es:
11111111 11111111 11111111 11110011
```

Ahora, si esto de hecho es igual a –13, deberíamos poder sumarlo al 13 binario y obtener como resultado 0. Comprobemos esto:

```
 00000000 00000000 00000000 00001101
+11111111 11111111 11111111 11110011
-----------------------------------
 00000000 00000000 00000000 00000000
```

El bit de acarreo que sale de la columna que está más a la izquierda se descarta y evidentemente obtenemos 0 como resultado. Si sumamos el complemento a uno de un número a ese mismo número, todos los dígitos del resultado serían iguales a 1. La clave para obtener un resultado en el que todos los dígitos sean cero es que el complemento a dos es uno más que el complemento a 1. La suma de 1 hace que el resultado de cada columna sea 0 y se acarrea un 1. El acarreo sigue desplazándose hacia la izquierda hasta que se descarta en el bit que está más a la izquierda, con lo que todos los dígitos del número resultante son iguales a cero.

En realidad, las computadoras realizan una suma como:

```
x = a - valor;
```

mediante la suma del complemento a dos de `valor` con `a`, como se muestra a continuación:

```
x = a + (~valor + 1);
```

Suponga que a es 27 y que `valor` es 13 como en el ejemplo anterior. Si el complemento a dos de `valor` es en realidad el negativo de éste, entonces al sumar el complemento de dos de `valor` con a se produciría el resultado de 14. Comprobemos esto:

```
a (es decir, 27)      00000000 00000000 00000000 00011011
+(~valor + 1)        +11111111 11111111 11111111 11110011
                     -----------------------------------
                      00000000 00000000 00000000 00001110
```

lo que ciertamente da como resultado 14.

Resumen

- Cuando escribimos un entero como 19, 227 o –63 en un programa de C++, suponemos que el número se encuentra en el sistema numérico decimal (base 10). Los dígitos en el sistema numérico decimal son 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9. El dígito más bajo es el 0 y el más alto es el 9 (uno menos que la base, 10).
- Las computadoras utilizan el sistema numérico binario (base 2). Este sistema numérico sólo tiene dos dígitos: 0 y 1. El dígito más bajo es el 0 y el más alto es el 1 (uno menos que la base de 2).
- El sistema numérico octal (base 8) y el sistema numérico hexadecimal (base 16) son populares principalmente debido a que permiten abreviar los números binarios de una manera conveniente.
- Los dígitos que se utilizan en el sistema numérico octal son del 0 al 7.
- El sistema numérico hexadecimal presenta un problema, ya que requiere de 16 dígitos: el dígito más bajo es 0 y el más alto tiene un valor equivalente al 15 decimal (uno menos que la base de 16). Por convención utilizamos las letras de la A a la F para representar los dígitos hexadecimales que corresponden a los valores decimales del 10 al 15.
- Cada uno de estos sistemas numéricos utilizan la notación posicional: cada posición en la que se escribe un dígito tiene un distinto valor posicional.
- Una relación especialmente importante que tienen tanto el sistema numérico octal como el hexadecimal con el sistema binario es que sus bases (8 y 16, respectivamente) son potencias de la base del sistema numérico binario (base 2).
- Para convertir un número octal en binario, sustituya cada dígito octal con su equivalente binario de tres dígitos.
- Para convertir un número hexadecimal en binario, simplemente sustituya cada dígito hexadecimal con su equivalente binario de cuatro dígitos.
- Como estamos acostumbrados a trabajar con el sistema decimal, es conveniente convertir un número binario, octal o hexadecimal en decimal para tener una idea de lo que "realmente" vale el número.
- Para convertir un número en decimal desde otra base, multiplique el equivalente en decimal de cada dígito por su valor posicional y sume estos productos.
- Las computadoras representan números negativos mediante el uso de la notación de complementos a dos.
- Para formar el negativo de un valor en binario, primero formamos su complemento a uno aplicando el operador de complemento a nivel de bits (~) de C++: Esto invierte los bits del valor. Para formar el complemento a dos de un valor, simplemente sumamos uno al complemento a uno de ese valor.

Ejercicios de autoevaluación

D.1 Las bases de los sistemas numéricos decimal, binario, octal y hexadecimal son _____, _____, _____ y _____, respectivamente.

D.2 En general, las representaciones en decimal, octal y hexadecimal de un número binario dado contienen (más/menos) dígitos de los que contiene el número binario.

D.3 (*Verdadero/falso*) Una de las razones populares de utilizar el sistema numérico decimal es que forma una notación conveniente para abreviar números binarios, en la que simplemente se sustituye un dígito decimal por cada grupo de cuatro dígitos binarios.

D.4 La representación (octal/hexadecimal/decimal) de un valor binario grande es la más concisa (de las alternativas dadas).

D.5 (*Verdadero/falso*) El dígito de mayor valor en cualquier base es uno más que la base.

D.6 (*Verdadero/falso*) El dígito de menor valor en cualquier base es uno menos que la base.

D.7 El valor posicional del dígito que se encuentra más a la derecha en cualquier número, ya sea binario, octal, decimal o hexadecimal es siempre _____.

D.8 El valor posicional del dígito que está a la izquierda del dígito que se encuentra más a la derecha en cualquier número, ya sea binario, octal, decimal o hexadecimal es siempre igual a _____.

D.9 Complete los valores que faltan en esta tabla de valores posicionales para las cuatro posiciones que están más a la derecha en cada uno de los sistemas numéricos indicados:

decimal	1000	100	10	1
hexadecimal	...	256	...	...
binario	...	...	...	...
octal	512	...	8	...

D.10 Convierta el número binario 110101011000 en octal y en hexadecimal.

D.11 Convierta el número hexadecimal FACE en binario.

D.12 Convierta el número octal 7316 en binario.

D.13 Convierta el número hexadecimal 4FEC en octal. (*Sugerencia:* primero convierta el número 4FEC en binario y después convierta el número resultante en octal).

D.14 Convierta el número binario 1101110 en decimal.

D.15 Convierta el número octal 317 en decimal.

D.16 Convierta el número hexadecimal EFD4 en decimal.

D.17 Convierta el número decimal 177 en binario, en octal y en hexadecimal.

D.18 Muestre la representación binaria del número decimal 417. Después muestre el complemento a uno de 417 y el complemento a dos del mismo número.

D.19 ¿Cuál es el resultado cuando se suma el complemento a dos de un número con ese mismo número?

Respuestas a los ejercicios de autoevaluación

D.1 10, 2, 8, 16.

D.2 Menos.

D.3 Falso. El hexadecimal hace esto.

D.4 Hexadecimal.

D.5 Falso. El dígito de mayor valor en cualquier base es uno menos que la base.

D.6 Falso. El dígito de menor valor en cualquier base es cero.

D.7 1 (la base elevada a la potencia de cero).

D.8 La base del sistema numérico.

D.9 Complete la tabla que se muestra a continuación:

decimal	1000	100	10	1
hexadecimal	4096	256	16	1
binario	8	4	2	1
octal	512	64	8	1

D.10 6530 octal; D58 hexadecimal.

D.11 1111 1010 1100 1110 binario.

D.12 111 011 001 110 binario.

D.13 0 100 111 111 101 100 binario; 47754 octal.

D.14 2 + 4 + 8 + 32 + 64 = 110 decimal.

D.15 7 + 1 * 8 + 3 * 64 = 7 + 8 + 192 = 207 decimal.

D.16 4 + 13 * 16 + 15 * 256 + 14 * 4096 = 61396 decimal.

D.17 177 decimal
en binario:

```
256 128 64 32 16 8 4 2 1
128 64 32 16 8 4 2 1
(1*128)+(0*64)+(1*32)+(1*16)+(0*8)+(0*4)+(0*2)+(1*1)
10110001
```

en octal:

```
512 64 8 1
64 8 1
(2*64)+(6*8)+(1*1)
261
```

en hexadecimal:

```
256 16 1
16 1
(11*16)+(1*1)
(B*16)+(1*1)
B1
```

D.18 Binario:

```
512 256 128 64 32 16 8 4 2 1
256 128 64 32 16 8 4 2 1
(1*256)+(1*128)+(0*64)+(1*32)+(0*16)+(0*8)+(0*4)+(0*2)+(1*1)
110100001
```

Complemento a uno: 001011110
Complemento a dos: 001011111
Comprobación: Número binario original + su complemento a dos:

```
110100001
001011111
---------
000000000
```

D.19 Cero.

Ejercicios

D.20 Algunas personas argumentan que muchos de nuestros cálculos se realizarían más fácilmente en el sistema numérico de base 12 que en el sistema numérico de base 10 (decimal), ya que el 12 puede dividirse por muchos más números que el 10 (por la base 10). ¿Cuál es el dígito de menor valor en la base 12? ¿Cuál podría ser el símbolo con mayor valor para un dígito en la base 12? ¿Cuáles son los valores posicionales de las cuatro posiciones más a la derecha de cualquier número en el sistema numérico de base 12?

D.21 Complete la siguiente tabla de valores posicionales para las cuatro posiciones más a la derecha en cada uno de los sistemas numéricos indicados:

decimal	1000	100	10	1
base 6	...	...	6	...
base 13	...	169	...	...
base 3	27	...	...	...

D.22 Convierta el número binario 100101111010 en octal y en hexadecimal.

D.23 Convierta el número hexadecimal 3A7D en binario.

D.24 Convierta el número hexadecimal 765F en octal. (*Sugerencia:* primero conviértalo en binario y después convierta el número resultante en octal).

D.25 Convierta el número binario 1011110 en decimal.

D.26 Convierta el número octal 426 en decimal.

D.27 Convierta el número hexadecimal FFFF en decimal.

D.28 Convierta el número decimal 299 en binario, en octal y en hexadecimal.

D.29 Muestre la representación binaria del número decimal 779. Después muestre el complemento a uno de 779 y el complemento a dos del mismo número.

D.30 Muestre el complemento a dos del valor entero –1 en una máquina con enteros de 32 bits.

E

Preprocesador

Sostén el bien; defínelo en forma adecuada.
—Alfred, Lord Tennyson

Le he encontrado un argumento; pero no estoy obligado a encontrarle una comprensión.
—Samuel Johnson

Un buen símbolo es el mejor argumento, y un misionario para persuadir a miles.
—Ralph Waldo Emerson

Las condiciones son fundamentalmente sólidas.
—Herbert Hoover [Diciembre 1929]

Objetivos

En este apéndice, usted aprenderá a:

- Utilizar `#include` para desarrollar programas extensos.
- Utilizar `#define` para crear macros y macros con argumentos.
- Comprender la compilación condicional.
- Mostrar mensajes de error durante la compilación condicional.
- Utilizar aserciones para probar si los valores de las expresiones son correctos.

Plan general

E.1 Introducción

En este capítulo presentamos el **preprocesador**. El preprocesamiento ocurre antes de compilar un programa. Algunas acciones posibles son la inclusión de otros archivos en el archivo que se va a compilar, la definición de **constantes simbólicas** y **macros**, la **compilación condicional** de código del programa y la **ejecución condicional de directivas del preprocesador**. Todas las directivas del preprocesador empiezan con #, y sólo pueden aparecer espacios en blanco antes de una directiva del preprocesador en una línea. Las directivas del preprocesador no son instrucciones de C++, por lo que no terminan con punto y coma (;). Las directivas del preprocesador se procesan por completo antes de que empiece la compilación.

Error común de programación E.1

Colocar un punto y coma al final de una directiva del preprocesador puede producir una variedad de errores, dependiendo del tipo de directiva del preprocesador.

Observación de Ingeniería de Software E.1

Muchas características del preprocesador (en especial las macros) son más apropiadas para los programadores de C que para los programadores de C++. Los programadores de C++ deben estar familiarizados con el preprocesador, ya que podrían tener la necesidad de trabajar con código heredado de C.

E.2 La directiva de preprocesamiento `#include`

Hemos utilizado la **directiva de preprocesamiento `#include`** a lo largo de este libro. Esta directiva hace que se incluya una copia de un archivo especificado en lugar de la directiva. Las dos formas de la directiva `#include` son:

```
#include <nombrearchivo>
#include "nombrearchivo"
```

La diferencia entre ellas es la ubicación en la que el preprocesador busca el archivo a incluir. Si el nombre del archivo está encerrado entre los signos < y > (la forma que se utiliza para los archivos de encabezado de la biblioteca estándar), el preprocesador busca el archivo especificado de una manera dependiente de la implementación, por lo general a través de directorios previamente designados. Si el nombre de archivo está encerrado entre comillas, el preprocesador busca primero en el mismo directorio en el que se va a compilar el archivo, y después en la misma forma dependiente de la implementación que para un nombre de archivo encerrado entre los signos < y >. Este método se utiliza comúnmente para incluir archivos de encabezado definidos por el programador.

La directiva #include se utiliza para incluir archivos de encabezado estándar, como <iostream> y <iomanip>. La directiva #include también se utiliza con los programas que consisten de varios archivos de código fuente que se van a compilar en conjunto. Por lo general, se crea y se incluye un archivo de encabezado que contiene declaraciones y definiciones comunes para los archivos separados del programa. Algunos ejemplos de dichas declaraciones y definiciones son: clases, estructuras, uniones, enumeraciones y prototipos de funciones, constantes y objetos de flujo (por ejemplo, cin).

E.3 La directiva de preprocesamiento #define: constantes simbólicas

La **directiva de preprocesamiento #define** crea **constantes simbólicas** (constantes representadas como símbolos) y macros (operaciones definidas como símbolos). El formato de esta directiva es:

```
#define identificador texto-de-reemplazo
```

Cuando aparece esta línea en un archivo, todas las ocurrencias subsiguientes (excepto las que están dentro de una cadena) de *identificador* en ese archivo se reemplazarán por *texto-de-reemplazo* antes de que se compile el programa. Por ejemplo,

```
#define PI 3.14159
```

reemplaza todas las ocurrencias subsiguientes de la constante simbólica PI con la constante numérica 3.14159. Las constantes simbólicas nos permiten crear un nombre para una constante, y utilizar el nombre a lo largo del programa. Más adelante, si la constante necesita modificarse en el programa, se puede modificar una sola vez en la directiva del preprocesador #define; y cuando el programa se vuelva a compilar, todas las ocurrencias de la constante en el programa se modificarán. [*Nota:* todo lo que está a la derecha del nombre de la constante simbólica reemplaza a esta constante simbólica. Por ejemplo, #define PI = 3.14159 hace que el preprocesador reemplace cada ocurrencia de PI con = 3.14159. Dicho reemplazo es la causa de muchos ligeros errores lógicos y de sintaxis]. También es un error redefinir una constante simbólica con un nuevo valor sin primero eliminar la primera definición. Observe que en C++ se prefieren las variables const en C++ en vez de las constantes simbólicas. Las variables constantes tienen un tipo de datos específico y son visibles por nombre para un depurador. Una vez que se reemplaza una constante simbólica con su texto de reemplazo, sólo el texto de reemplazo está visible para el depurador. Una desventaja de las variables const es que podrían requerir una ubicación de memoria del tamaño de su tipo de datos; las constantes simbólicas no requieren memoria adicional.

Error común de programación E.2

Si se utilizan constantes simbólicas en un archivo que no sea el archivo en el que están definidas, se produce un error de compilación (a menos que se incluyan de un archivo de encabezado, mediante #include).

Buena práctica de programación E.1

Al utilizar nombres significativos para las constantes simbólicas, los programas se vuelven más autodocumentados.

E.4 La directiva de preprocesamiento #define: macros

[*Nota:* esta sección se incluye para beneficio de los programadores de C++ que necesiten trabajar con código heredado de C. En C++, las macros se pueden reemplazar comúnmente por las plantillas y funciones en línea]. Una macro es una operación definida en una directiva del preprocesador #define. Al igual que con las constantes simbólicas, el ***identificador de macros*** se reemplaza con el ***texto de reemplazo*** antes de compilar el programa. Las macros se pueden definir con o sin ***argumentos***. Una macro

sin argumentos se procesa como una constante simbólica. En una macro con argumentos, éstos se reemplazan en el *texto-de-reemplazo*, y después la macro se expande (es decir, el *texto-de-reemplazo* reemplaza el identificador de la macro y la lista de argumentos en el programa). No hay comprobación de tipos de datos para los argumentos de una macro. Ésta se utiliza simplemente para sustituir el texto.

Considere la siguiente definición de una macro con un argumento para el área de un círculo:

```
#define AREA_CIRCULO( x ) ( PI * ( x ) * ( x ) )
```

En cualquier parte del archivo en donde aparezca `AREA_CIRCULO( y )`, el valor de `y` se sustituye por `x` en el texto de reemplazo, la constante simbólica `PI` se reemplaza por su valor (definido con anterioridad) y la macro se expande en el programa. Por ejemplo, la instrucción

```
area = AREA_CIRCULO( 4 );
```

se expande a

```
area = ( 3.14159 * ( 4 ) * ( 4 ) );
```

Como la expresión sólo contiene constantes, en tiempo de compilación el valor de la expresión se puede evaluar, y el resultado se asigna a `area` en tiempo de ejecución. Los paréntesis alrededor de cada `x` en el texto de reemplazo y alrededor de toda la expresión obligan a que se utilice el orden de evaluación apropiado cuando el argumento de la macro es una expresión. Por ejemplo, la instrucción

```
area = AREA_CIRCULO( c + 2 );
```

se expande a

```
area = ( 3.14159 * ( c + 2 ) * ( c + 2 ) );
```

lo cual se evalúa en forma correcta, ya que los paréntesis obligan a que se utilice el orden de evaluación apropiado. Si se omiten los paréntesis, la expansión de la macro sería

```
area = 3.14159 * c + 2 * c + 2;
```

lo cual se evalúa de manera incorrecta como

```
area = ( 3.14159 * c ) + ( 2 * c ) + 2;
```

debido a las reglas de precedencia de los operadores.

Error común de programación E.3

Olvidar encerrar los argumentos de una macro entre paréntesis en el texto de reemplazo es un error.

La macro `AREA_CIRCULO` se podría definir como una función. La función `areaCirculo`, como en

```
double areaCirculo( double x ) { return 3.14159 * x * x; }
```

realiza el mismo cálculo que `AREA_CIRCULO`, pero la función `areaCirculo` tiene asociada la sobrecarga de la llamada a una función. Las ventajas de `AREA_CIRCULO` son que las macros insertan código directamente en el programa (evitando la sobrecarga de las funciones) y el programa mantiene su legibilidad, debido a que `AREA_CIRCULO` se define por separado y tiene un nombre significativo. Una desventaja es que el argumento se evalúa dos veces. Además, cada vez que aparece una macro en un programa, se expande. Si la macro es extensa, esto produce un aumento en el tamaño del programa. Por ende, hay una concesión entre la velocidad de ejecución y el tamaño del programa (si el espacio en disco es poco). Hay

que tener en cuenta que se prefieren las funciones `inline` (vea el capítulo 6) para obtener el rendimiento de las macros y los beneficios de las funciones relacionados con la ingeniería de software.

Tip de rendimiento E.1

Algunas veces las macros se pueden utilizar para reemplazar la llamada a una función con el código `inline` antes del tiempo de ejecución. Esto elimina la sobrecarga de una llamada a una función. Las funciones en línea son preferibles a las macros, ya que ofrecen los servicios de comprobación de tipos de las funciones.

A continuación se muestra la definición de una macro con dos argumentos para el área de un rectángulo:

```
#define AREA_RECTANGULO( x, y )  ( ( x ) * ( y ) )
```

En cualquier parte del programa en donde aparezca `AREA_RECTANGULO( a, b )`, los valores de `a` y `b` se sustituyen en el texto de reemplazo de la macro, y la macro se expande en lugar de su nombre. Por ejemplo, la instrucción

```
areaRect = AREA_RECTANGULO( a + 4, b + 7 );
```

se expande a

```
areaRect = ( ( a + 4 ) * ( b + 7 ) );
```

El valor de la expresión se evalúa y se asigna a la variable `areaRect`.

El texto de reemplazo para una macro o constante simbólica es por lo general cualquier texto en la línea después del identificador en la directiva `#define`. Si el texto de reemplazo para una macro o constante simbólica es más extenso que el resto de la línea, debemos colocar una barra diagonal inversa (\) al final de cada línea de la macro (excepto la última línea), con lo cual indicamos que el texto de reemplazo continúa en la siguiente línea.

Las constantes simbólicas y las macros se pueden descartar utilizando la **directiva de preprocesamiento `#undef`**. La directiva `#undef` elimina la definición de una constante simbólica o nombre de macro. El alcance de una constante simbólica o macro es desde su definición, hasta que quede indefinida con `#undef` o hasta llegar al final del archivo. Una vez indefinido, un nombre puede redefinirse con `#define`.

Observe que las expresiones con efectos secundarios (por ejemplo, que se modifiquen los valores de las variables) no deben pasarse a una macro, ya que los argumentos de la macro pueden llegar a evaluarse más de una vez.

Error común de programación E.4

A menudo, las macros reemplazan un nombre que no estaba destinado a ser un uso de la macro, sino que simplemente se escribe igual. Esto puede provocar errores de compilación y de sintaxis excepcionalmente misteriosos.

E.5 Compilación condicional

La **compilación condicional** nos permite controlar la ejecución de las directivas del preprocesador y la compilación del código del programa. Cada una de las directivas del preprocesador condicionales evalúa una expresión entera constante que determinará si el código se va a compilar o no. Las expresiones de conversión de tipos, las expresiones `sizeof` y las constantes de enumeración no se pueden evaluar en las directivas del preprocesador, ya que todas son determinadas por el compilador y el preprocesamiento ocurre antes de la compilación.

La construcción de preprocesador condicional es muy parecida a la estructura de selección `if`. Considere el siguiente código de preprocesador:

```
#ifndef NULL
   #define NULL 0
#endif
```

este código determina si la constante simbólica `NULL` ya se encuentra definida. La expresión `#ifndef NULL` incluye el código hasta `#endif` si `NULL` no está definida, y omite el código si `NULL` está definida. Cada construcción `#if` termina con `#endif`. Las directivas `#ifdef` y `#ifndef` son abreviaciones para `#if defined(`*nombre*`)` y `#if !defined(`*nombre*`)`. Podemos evaluar una construcción del preprocesador condicional que conste de varias partes mediante el uso de las directivas `#elif` (el equivalente de `else if` en una estructura `if`) y `#else` (el equivalente de `else` en una estructura `if`).

Durante el desarrollo del programa, comúnmente los programadores encuentran que es útil "comentar" porciones extensas de código para evitar que se compile. Si el código contiene comentarios estilo C, no se pueden utilizar los signos `/*` y `*/` para realizar esta tarea, debido a que al encontrar el `*/` se terminaría el comentario. En vez de ello, podemos usar la siguiente construcción del preprocesador:

```
#if 0
   código que no se debe compilar
#endif
```

Para permitir que el código se compile, simplemente reemplazamos el valor `0` en la construcción anterior con el valor `1`.

La compilación condicional se utiliza comúnmente como una ayuda para la depuración. A menudo se utilizan instrucciones de salida para imprimir valores de variables y confirmar el flujo de control. Estas instrucciones de salida se pueden encerrar en directivas del preprocesador condicionales, de manera que las instrucciones se compilen sólo hasta que se complete el proceso de depuración. Por ejemplo,

```
#ifdef DEBUG
   cerr << "Variable x = " << x << endl;
#endif
```

hace que se compile la instrucción `cerr` en el programa, si se ha definido la constante simbólica `DEBUG` antes de la directiva `#ifdef DEBUG`. Esta constante simbólica generalmente se establece mediante un compilador de línea de comandos, o a través de las opciones en el IDE (por ejemplo, Visual Studio) y no por una definición `#define` explícita. Cuando se completa la depuración, la directiva `#define` se elimina del archivo de código fuente y las instrucciones de salida que se insertaron para fines de depuración se ignoran durante la compilación. En programas más extensos, podría ser conveniente definir varias constantes simbólicas distintas que controlen la compilación condicional en secciones separadas del archivo de código fuente.

Error común de programación E.5

Al insertar instrucciones de salida compiladas en forma condicional para fines de depuración, en ubicaciones en donde C++ espere una sola instrucción, se pueden provocar errores lógicos y de sintaxis. En este caso, la instrucción compilada en forma condicional debe encerrarse en una instrucción compuesta. Así, cuando se compile el programa con instrucciones de depuración, el flujo de control del programa no se alterará.

E.6 Las directivas de preprocesamiento `#error` y `#pragma`

La **directiva `#error`**

```
#error tokens
```

imprime un mensaje dependiente de la implementación, incluyendo los *tokens* especificados en la directiva. Los tokens son secuencias de caracteres separados por espacios. Por ejemplo,

```
#error 1 - Error fuera de rango
```

contiene seis tokens. Por ejemplo, en un compilador de C++ popular, cuando se procesa una directiva `#error`, los tokens en la directiva se muestran como un mensaje de error, el preprocesamiento se detiene y el programa no se compila.

La **directiva `#pragma`**

```
#pragma tokens
```

provoca una acción definida por la implementación. Una directiva `#pragma` que no sea reconocida por la implementación se ignora. Por ejemplo, un compilador de C++ específico podría reconocer directivas `#pragma` que nos permitan aprovechar las capacidades específicas de ese compilador. Para obtener más información acerca de `#error` y `#pragma`, consulte la documentación de su implementación de C++.

E.7 Los operadores # y

Los operadores del procesador `#` y `##` están disponibles en C++ y en C de ANSI/ISO. El operador `#` hace que un token del texto de reemplazo se convierta en una cadena encerrada entre comillas. Considere la siguiente definición de una macro:

```
#define HOLA( x ) cout << "Hola, " #x << endl;
```

Cuando aparece `HOLA(Juan)` en un archivo del programa, se expande a

```
cout << "Hola, " "Juan" << endl;
```

La cadena `"Juan"` reemplaza a `#x` en el texto de reemplazo. Las cadenas separadas por espacios en blanco se concatenan durante el preprocesamiento, por lo que la instrucción anterior es equivalente a

```
cout << "Hola, Juan" << endl;
```

Observe que se debe utilizar el operador `#` en una macro con argumentos, debido a que el operando de `#` hace referencia a un argumento de la macro.

El operador `##` concatena dos tokens. Considere la siguiente definición de una macro:

```
cout << "Hola, Juan" << endl;
#define CONCATTOKEN( x, y )  x ## y
```

Cuando aparece `CONCATTOKEN` en el programa, sus argumentos se concatenan y se utilizan para reemplazar la macro. Por ejemplo, `CONCATTOKEN(O, K)` se reemplaza por `OK` en el programa. El operador `##` debe tener dos operandos.

E.8 Constantes simbólicas predefinidas

Hay seis **constantes simbólicas predefinidas** (figura E.1). Los identificadores para cada una de estas constantes empiezan y (excepto por `__cplusplus`) terminan con *dos* guiones bajos. Estos identificadores y el operador del preprocesador `defined` (sección E.5) no se pueden utilizar en directivas `#define` o `#undef`.

Constante simbólica	Descripción
`__LINE__`	El número de línea de la línea actual de código fuente (una constante entera).
`__FILE__`	El presunto nombre del archivo de código fuente (una cadena).
`__DATE__`	La fecha de compilación del archivo de código fuente (una cadena de la forma `"Mmm dd aaaa"`, tal como `"Ago 19 2002"`).
`__STDC__`	Indica si el programa se conforma al estándar ANSI/ISO de C. Contiene el valor 1 si hay una conformidad completa, y está indefinida en caso contrario.
`__TIME__`	La hora de compilación del archivo de código fuente (una literal de cadena de la forma `"hh:mm:ss"`).
`__cplusplus`	Contiene el valor `199711L` (la fecha en que se aprobó el estándar ISO de C++) si el archivo va a ser compilado por un compilador de C++, y está indefinida en caso contrario. Permite establecer un archivo para que se compile como C o C++.

Fig. E.1 | Las constantes simbólicas predefinidas.

E.9 Aserciones

La **macro `assert`** (definida en el archivo de encabezado **`<cassert>`**) prueba el valor de una expresión. Si el valor de la expresión es `0` (falso), entonces `assert` imprime un mensaje de error y llama a la función **`abort`** (de la biblioteca de herramientas generales: `<cstdlib>`) para terminar la ejecución del programa. Ésa es una herramienta útil de depuración para evaluar si una variable tiene un valor correcto. Por ejemplo, suponga que la variable `x` nunca debe ser mayor que `10` en un programa. Se puede utilizar una aserción para evaluar el valor de `x` e imprimir un mensaje de error si el valor de `x` es incorrecto. La instrucción sería

```
assert( x <= 10 );
```

Si `x` es mayor que `10` al llegar a la instrucción anterior en un programa, se imprime un mensaje de error que contiene el número de línea y el nombre del archivo, y el programa termina. Así, el programador se puede concentrar en esta área del código para encontrar el error. Si la constante simbólica `NDEBUG` está definida, se ignorarán las aserciones subsiguientes. Por ende, cuando ya no sean necesarias las aserciones (es decir, cuando se complete la depuración), insertamos la línea

```
#define NDEBUG
```

en el archivo del programa, en vez de eliminar cada aserción en forma manual. Al igual que con la constante simbólica `DEBUG`, `NDEBUG` se establece a menudo mediante opciones de la línea de comandos del compilador, o a través de una opción en el IDE.

La mayoría de los compiladores de C++ incluyen ahora el manejo de excepciones. Los programadores de C++ prefieren utilizar excepciones en vez de aserciones. Pero las aserciones aún son valiosas para los programadores de C++ que trabajan con código heredado de C.

E.10 Conclusión

En este apéndice vimos la directiva `#include`, que se utiliza para desarrollar programas extensos. También aprendió acerca de la directiva `#define`, que se utiliza para crear macros. Presentamos la compilación condicional, cómo mostrar mensajes de error y utilizar aserciones.

Resumen

Sección E.2 La directiva de preprocesamiento `#include`

- Todas las directivas de preprocesamiento empiezan con # y se procesan antes de que se compile el programa.
- Sólo pueden aparecer caracteres de espacio en blanco antes de una directiva del preprocesador en una línea.
- La directiva `#include` incluye una copia del archivo especificado. Si el nombre de archivo va encerrado entre comillas, el preprocesador empieza a buscar el archivo a incluir en el mismo directorio en el que se va a compilar el archivo. Si el nombre del archivo va encerrado entre los signos < y >, la búsqueda se realiza de una manera definida por la implementación.

Sección E.3 La directiva de preprocesamiento `#define`*: constantes simbólicas*

- La directiva del preprocesador `#define` se utiliza para crear constantes simbólicas y macros.
- Una constante simbólica es un nombre para una constante.

Sección E.4 La directiva de preprocesamiento `#define`*: macros*

- Una macro es una operación definida en una directiva del preprocesador `#define`. Las macros se pueden definir con o sin argumentos.
- El texto de reemplazo para una macro o constante simbólica es cualquier texto restante en la línea después del identificador (y, si la hay, la lista de argumentos de la macro) en la directiva `#define`. Si el texto de reemplazo para una macro o constante simbólica es demasiado extenso como para caber en una sola línea, se coloca una barra diagonal inversa (\) al final de la línea, indicando que el texto de reemplazo continúa en la siguiente línea.
- Las constantes simbólicas y las macros se pueden descartar mediante el uso de la directiva del preprocesador `#undef`. La directiva `#undef` elimina la definición de la constante simbólica o nombre de la macro.
- El alcance de una constante simbólica o macro es a partir de su definición, hasta que se elimine su definición con `#undef` o al llegar al final del programa.

Sección E.5 Compilación condicional

- La compilación condicional nos permite controlar la ejecución de las directivas del preprocesador y la compilación del código del programa.
- Las directivas del preprocesador condicionales evalúan expresiones enteras constantes. Las expresiones de conversión de tipos, las expresiones `sizeof` y las constantes de enumeración no se pueden evaluar en las directivas del preprocesador.
- Cada construcción `#if` termina con `#endif`.
- Las directivas `#ifdef` y `#ifndef` se proporcionan como abreviaciones para `#if defined(`*nombre*`)` y `#if !defined(`*nombre*`)`.
- Una construcción del preprocesador condicional que consiste de varias partes se evalúa con las directivas `#elif` y `#else`.

Sección E.6 Las directivas de preprocesamiento `#error` *y* `#pragma`

- La directiva `#error` imprime un mensaje dependiente de la implementación que incluye los tokens especificados en la directiva, y termina el preprocesamiento y la compilación.
- La directiva `#pragma` provoca una acción definida por la implementación. Si la implementación no reconoce esta directiva, se ignora.

Sección E.7 Los operadores # y ##

- El operador # hace que el siguiente token del texto de reemplazo se convierta en una cadena encerrada entre comillas. El operador # debe utilizarse en una macro con argumentos, debido a que el operando de # debe ser un argumento de la macro.
- El operador ## concatena dos tokens. El operador ## debe tener dos operandos.

Sección E.8 Constantes simbólicas predefinidas

- Hay seis constantes simbólicas predefinidas. La constante `__LINE__` es el número de línea de la línea actual en el código fuente (un entero). La constante `__FILE__` es el presunto nombre del archivo (una cadena). La constante `__DATE__` es la fecha de compilación del archivo de código fuente (una cadena). La constante `__TIME__` es la hora de compilación del archivo de código fuente (una cadena). Observe que cada una de las constantes simbólicas predefinidas empieza (y, con la excepción de `__cplusplus`, termina) con dos guiones bajos.

Sección E.9 Aserciones

- La macro `assert` (definida en el archivo de encabezado `<cassert>`) prueba el valor de una expresión. Si el valor de la expresión es `0` (falso), entonces `assert` imprime un mensaje de error y llama a la función `abort` para terminar la ejecución del programa.

Ejercicios de autoevaluación

E.1 Complete los siguientes enunciados:

a) Cada directiva del preprocesador debe empezar con _____.
b) La construcción de compilación condicional puede extenderse para evaluar múltiples casos, utilizando las directivas _____ y _____.
c) La directiva _____ crea macros y constantes simbólicas.
d) Sólo pueden aparecer caracteres _____ antes de una directiva del preprocesador en una línea.
e) La directiva _____ descarta los nombres de las constantes simbólicas y las macros.
f) Las directivas _____ y _____ se proporcionan como notación abreviada para `#if defined(`*nombre*`)` y `#if !defined(`*nombre*`)`.
g) _____ nos permite controlar la ejecución de las directivas del preprocesador y la compilación del código del programa.
h) La macro _____ imprime un mensaje y termina la ejecución del programa si el valor de la expresión con la que se evalúa la macro es `0`.
i) La directiva _____ inserta un archivo en otro archivo.
j) El operador _____ concatena sus dos argumentos.
k) El operador _____ convierte su operando en una cadena.
l) El carácter _____ indica que el texto de reemplazo para una constante simbólica o macro continúa en la siguiente línea.

E.2 Escriba un programa para imprimir los valores de las constantes simbólicas predefinidas `__LINE__`, `__FILE__`, `__DATE__` y `__TIME__` que se listan en la figura E.1.

E.3 Escriba una directiva del preprocesador para realizar cada una de las siguientes acciones:

a) Definir la constante simbólica `SI` para que tenga el valor `1`.
b) Definir la constante simbólica `NO` para que tenga el valor `0`.
c) Incluir el archivo de encabezado `comun.h`. Este encabezado se encuentra en el mismo directorio que el archivo que se va a compilar.
d) Si la constante simbólica `TRUE` está definida, eliminar su definición y volverla a definir como `1`. No utilice `#ifdef`.
e) Si la constante simbólica `TRUE` está definida, eliminar su definición y volverla a definir como `1`. Utilice la directiva del preprocesador `#ifdef`.
f) Si la constante simbólica `ACTIVO` no es igual a `0`, definir la constante simbólica `INACTIVO` como `0`. En caso contrario, definir `INACTIVO` como `1`.
g) Definir la macro `VOLUMEN_CUBO` que calcula el volumen de un cubo (recibe un argumento).

Respuestas a los ejercicios de autoevaluación

E.1 a) `#`. b) `#elif`, `#else`. c) `#define`. d) de espacio en blanco. e) `#undef`. f) `#ifdef`, `#ifndef`. g) La compilación condicional. h) `assert`. i) `#include`. j) `##`. k) `#`. l) `\`.

E.2 (Vea a continuación).

```
// ejE_02.cpp
// Solución al Ejercicio de autoevaluación E.2.
#include <iostream>
using namespace std;

int main()
{
   cout << "__LINE__ = " << __LINE__ << endl
        << "__FILE__ = " << __FILE__ << endl
        << "__DATE__ = " << __DATE__ << endl
        << "__TIME__ = " << __TIME__ << endl
        << "__cplusplus = " << __cplusplus << endl;
}  // fin de main
```

```
__LINE__ = 9
__FILE__ = c:\cpp4e\cap19\ej19_02.CPP
__DATE__ = Jul 17 2002
__TIME__ = 09:55:58
__cplusplus = 199711L
```

E.3 a) `#define SI 1`

b) `#define NO 0`

c) `#include "common.h"`

d)
```
#if defined(TRUE)
    #undef TRUE
    #define TRUE 1
#endif
```

e)
```
#ifdef TRUE
    #undef TRUE
    #define TRUE 1
#endif
```

f)
```
#if ACTIVO
    #define INACTIVO 0
#else
    #define INACTIVO 1
#endif
```

g) `#define VOLUMEN_CUBO( x )  ( ( x ) * ( x ) * ( x ) )`

Ejercicios

E.4 Escriba un programa que defina una macro con un argumento para calcular el volumen de una esfera. El programa debe calcular el volumen para las esferas cuyos radios se encuentren en el rango de 1 a 10, y debe imprimir los resultados en formato tabular. La fórmula para el volumen de una esfera es

```
( 4.0 / 3 ) * p * r3
```

en donde π es `3.14159`.

E.5 Escriba un programa que produzca los siguientes resultados:

```
La suma de x y y es 13
```

El programa debe definir la macro `SUMA` con dos argumentos, `x` y `y`, y debe utilizar `SUMA` para producir los resultados.

E.6 Escriba un programa que utilice la macro `MINIMO2` para determinar el menor de dos valores numéricos. Debe recibir los valores como entrada mediante el teclado.

E.7 Escriba un programa que utilice la macro `MINIMO3` para determinar el menor de tres valores numéricos. La macro `MINIMO3` debe utilizar la macro `MINIMO2` definida en el ejercicio E.6 para determinar el menor número. Debe recibir los valores como entrada mediante el teclado.

E.8 Escriba un programa que utilice la macro `IMPRIMIR` para imprimir un valor de cadena.

E.9 Escriba un programa que utilice la macro `IMPRIMIRARREGLO` para imprimir un arreglo de enteros. La macro debe recibir el arreglo y el número de elementos en el arreglo como argumentos.

E.10 Escriba un programa que utilice la macro `SUMARARREGLO` para sumar los valores en un arreglo numérico. La macro debe recibir el arreglo y el número de elementos en el arreglo como argumentos.

E.11 Vuelva a escribir las soluciones a los ejercicios E.4 a E.10 como funciones `inline`.

E.12 Para cada una de las siguientes macros, identifique los posibles problemas (si los hay) cuando el preprocesador expanda las macros:

a) `#define SQR( x ) x * x`
b) `#define SQR( x ) ( x * x )`
c) `#define SQR( x ) ( x ) * ( x )`
d) `#define SQR( x ) ( ( x ) * ( x ) )`

Índice

Símbolos

Numéricos

A

D

E

F

M

P

Q

R

S

T

u

v

W

X

Y

Z

Una introducción completa y autorizada del código activo DEITEL® a C++, la programación orientada a objetos (POO) y el diseño orientado a objetos (DOO) con UML™

Bienvenido a la programación con C++, uno de los lenguajes de programación orientada a objetos más populares.

Este libro utiliza las tecnologías de computación de vanguardia, y su base es el reconocido *método de código activo* de Deitel, donde los conceptos se presentan en el contexto de programas funcionales completos, seguidos de ejecuciones de ejemplo, en lugar de hacerlo a través de fragmentos separados de código.

A lo largo del texto encontrará recuadros con tips de programación, para prevenir errores, y de rendimiento, así como buenas prácticas de programación, que resultan de gran utilidad.

Estándar C++11

Entre las características clave del nuevo estándar C++11 que se presenta en esta nueva edición destacan las siguientes:

- ***Cumple con el nuevo estándar C++11.*** Con una extensa cobertura de las nuevas características de C++11.
- ***Apuntadores inteligentes***. Los apuntadores inteligentes ayudan a evitar los errores de administración de memoria dinámica al proveer una funcionalidad adicional a la de los apuntadores integrados.
- ***Cobertura anticipada de los contenedores, iteradores y algoritmos de la Biblioteca Estándar, mejorada con capacidades de C++11.*** La gran mayoría de las necesidades de los programadores en cuanto a estructuras de datos pueden satisfacerse mediante la *reutilización* de las capacidades de la Biblioteca Estándar.

Este libro se adhiere al estándar de codificación segura en C++ de CERT.

Para mayor información visite:
www.pearsonenespañol.com/deitel

ISBN 978-607-32-2739-1
90000
9 786073 227391

Visítenos en:
www.pearsonenespañol.com